Practical Applications of Industrial Optimization: From High-Speed Embedded Controllers to Large Discrete Utility Systems

Jonathan David Currie

School of Engineering

February 2014

Supervised by:
Associate Professor David I. Wilson
Professor Brent R. Young

A thesis submitted to Auckland University of Technology in fulfilment of the requirements for the degree of Doctor of Philosophy.

Abstract

Optimization of large-scale industrial systems requires not only state-of-the-art numerical algorithms, but also accurate tailor-made underlying models to ensure the solution is both sensible and useful. The combination of setting up a rigorous optimization solver together with building a high-fidelity model can cause the typical industrial user to become overwhelmed with formulating one or both of these steps, resulting in poor performance and/or a suboptimal solution. This work addresses the problem by developing a high-level framework for modelling and solving industrially significant optimization problems. The framework allows the user to concentrate on their domain specialization, while the framework automatically tailors the optimization problem by exploiting structural features within the user's model. To illustrate the benefit of this approach, two widely varying industrial optimization problems are investigated: Online optimization within an embedded predictive controller and large-scale steam utility system operational optimization.

Within the first chosen example, an embedded model predictive controller, an optimal control problem must be solved at each sample in order to calculate the next control move(s). In a traditional linear predictive controller, this requires solving online a quadratic programming problem which, even for modest problems with relatively short prediction horizons, can involve tens of decision variables and hundreds of linear constraints. On an embedded platform, such as a microcontroller, solving a problem of this size *online* requires substantial computational power together with a large amount of dynamic memory, both of which are highly constrained on typical hardware. To overcome the hurdle, this work introduces the jMPC Toolbox, a high-level MATLAB framework for describing, tuning, simulating and generating embedded predictive controllers. Furthermore, the `quad_wright` and `quad_mehrotra` interior-point quadratic programming solvers have been developed, which are specifically tailored to solve modestly-sized online optimization problems within a model predictive controller on embedded hardware. Together, these two contributions allow an embedded predictive controller with an online optimization solver capable of over 10kHz sampling rates to be built, verified and deployed to modest embedded hardware in less than ten seconds. A case study demonstrates the effectiveness of the approach applied to an unstable, nonlinear laboratory-scale helicopter, while benchmarks against literature show for the problems of interest that the `quad_mehrotra` solver is the best in class.

The second chosen example, steam utility systems, are designed to supply the heating, mechanical and electrical demands of an on-site process system, such as an oil refinery, paper mill, chemical process plant or a variety of other energy intensive industries. Steam is used as the working fluid within the utility system, and is generated by boilers or recovered from waste heat, which is then used to supply the heating requirements of the process, or used to drive steam turbines to supply mechanical and electrical loads. In addition, gas turbines provide modern utility systems with co-generation potential, allowing the system to export excess electricity if economically viable. However, due to the discrete nature of a utility system where equipment can be switched in and out of service, steam flows redistributed, and where zero-flow conditions are normal, optimizing the operation of a utility system requires a rigorous model based on thermodynamics and state-of-the-art numerical algorithms. To address this problem, a second MATLAB framework, the OPTI Toolbox, has been developed which provides a suite of state-of-the-art open-source optimization algorithms suitable for solving the discrete optimization problems that arise from operational optimization. Furthermore, to tailor the utility system model to the optimizer, a symbolic mixed integer nonlinear modelling strategy is developed to approximate a rigorous simulator model, combining regressions from literature, industrial experience and process specific knowledge, resulting in an efficient model for optimization. Multiple case studies are presented to demonstrate the efficiency of the approach, including the operational optimization of an industrial petrochemical utility system. Each of the case studies encompass a range of operating conditions and superstructures, noting the framework correctly solves for the global optimum for all problems in less than 5 seconds, matches the solution from an equivalent rigorous thermodynamic model and provides industrially significant CAPEX-free economic savings.

While the jMPC and OPTI Toolboxes target substantially different ends of the industrial optimization spectrum in terms of physical size and dynamic response, this work shows that the common approach of abstracting the optimization problem via a higher-level framework, together with exploiting problem specific characteristics, allows high-speed and robust solutions to be obtained to industrially significant problems. Moreover, in both examples the complexities of the model and the interface to the optimizer are hidden, allowing the user to focus directly on the problem at hand, yet still obtain best-in-class performance.

Acknowledgements

I firstly wish to thank my academic supervisors, Associate Professor David I Wilson and Professor Brent R Young, for offering me this project. Both David and Brent have provided countless hours of advice, knowledge and insight, and it is due to their invaluable assistance that the project has been a success. I am especially grateful to David's enthusiasm and support, specifically within the fields of numerical analysis, automatic control and process modelling.

In addition, I wish to thank AUT University for providing me both undergraduate and postgraduate scholarships, which has allowed me to dedicate a significant part of the last 8 years to my study and research. Furthermore, a special mention is due to all the engineering academic and administration staff at AUT who have without exception been incredibly supportive of my research. I especially appreciate the time spent with Mark Beckerleg, John Collins, David White and Roy Nates when discussing questions related to my research, as well around my decision to pursue postgraduate study.

I also wish to thank my parents, Diana Donald and Richard Currie, for the emotional and financial support through this period of my life. In addition, special thanks is due for the assistance in proof-reading, an area I find challenging.

Also, considerable recognition is due to Dr. Nick Depree, who taught me the foundations of chemical engineering needed to undertake this project. Without his input and advice, the utility system section would not have been possible.

Financial support for this project was provided by the Vice Chancellor's Doctoral Scholarship, the Industrial Information and Control Centre (I^2C^2) and the Engineering Research Institute (ERI), all of which enabled me to undertake this research, and which is gratefully acknowledged.

List of Publications

Currie, J., Prince-Pike, A., and Wilson, D. Auto-Code Generation for Fast Embedded Model Predictive Controllers. In *19th International Conference on Mechatronics and Machine Vision in Practice* (Auckland, New Zealand, 28–30 November 2012), pp. 122–128.

Currie, J., Prince-Pike, A., and Wilson, D. A Cost Effective High-Speed Auto-Coded Embedded Model Predictive Controller. *International Journal of Intelligent Systems Technologies and Applications 13*, 1/2 (2014).

Currie, J., Wilson, D., Yu, W., and Young, B. Rethinking the Modelling of Energy And Utility Systems. In *9th World Congress of Chemical Engineering* (Seoul, Korea, 18–23 August 2013). Invited Keynote.

Currie, J., and Wilson, D. I. A Model Predictive Control Toolbox Intended for Rapid Prototyping. In *16th Electronics New Zealand Conference (ENZCon 2009)* (Dunedin, New Zealand, 18–20 November 2009), Tim Molteno, Ed., pp. 7–12.

Currie, J., and Wilson, D. I. Lightweight Model Predictive Control Intended for Embedded Applications. In *9th International Symposium on Dynamics and Control of Process Systems (DYCOPS)* (Leuven, Belgium, 5–7 July 2010), pp. 264–269.

Currie, J., and Wilson, D. I. Interpolated Model Predictive Control: Having Your Cake and Eating it Too. In *Australian and New Zealand Annual Chemical Engineering Conference, Chemeca* (Sydney Australia, 18–21 September 2011).

Currie, J., and Wilson, D. I. OPTI: Lowering the Barrier Between Open Source Optimizers and the Industrial MATLAB User. In *Foundations of Computer-Aided Process Operations* (Savannah, Georgia, USA, 8–11 January 2012), Nick Sahinidis and Jose Pinto, Eds.

Currie, J., and Wilson, D. I. Rigorously Modelling Steam Utility Systems for Mixed Integer Optimization. In *10th International Power and Energy Conference* (Ho Chi Minh City, Vietnam, 12–14 December 2012), IEEE, pp. 526–531.

Currie, J., and Wilson, D. I. The Efficient Modelling of Steam Utility Systems. In *Australian and New Zealand Annual Chemical Engineering Conference, Chemeca* (Wellington, New Zealand, 23–26 September 2012), Engineers Australia.

Currie, J., Wilson, D. I., Depree, N., Young, B., Azmanai, S., and Karim, L. Steam Utility Systems are not "Business As Usual" for Chemical Plant Simulators. In *American Institute of Chemical Engineers (AIChE) Spring Meeting* (Chicago, USA, 13–17 March 2011), pp. 63d:1–6.

Lim, H., Currie, J., and Wilson, D. Validating a Thermodynamic Model of the Otahuhu B Combined Cycle Gas Turbine Power Station. In *Australian and New Zealand Annual Chemical Engineering Conference, Chemeca* (Brisbane, Australia, 29 Sept–2 October 2013), Engineers Australia.

Contents

Attestation of Authorship

I hereby declare that this submission is my own work and that, to the best of my knowledge and belief, it contains no material previously published or written by another person (except where explicitly defined in the acknowledgements), nor material which to a substantial extent has been submitted for the award of any other degree or diploma of a university or other institution of higher learning.

Jonathan Currie ______________________________

List of Figures

List of Tables

Nomenclature

This work considers two quite varied industrial problems and therefore domain specific nomenclature is adopted for each problem.

Table 1: Model predictive control nomenclature (Chapters 3-4)

Symbol	Description
$x, \mathbf{x}$	State Vector
$\Delta\mathbf{x}$	State Increment Vector
$u, \mathbf{u}$	Control Input(s)
$\Delta u, \Delta\mathbf{u}$	Control Input Increment(s)
$y, \mathbf{y}$	Plant Output(s)
$y^*, \mathbf{y}^*$	Reference/Setpoint(s)
$\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D}$	State Space Model Matrices
N_p	Prediction Horizon Length (samples)
N_c	Control Horizon Length (samples)
k	Current sample/iteration
$\mathbf{\Phi}, \mathbf{F}$	Prediction Matrices
$\mathbf{H}$	Quadratic Program Hessian Matrix
$\mathbf{f}$	Quadratic Program Gradient Vector
$\mathbf{M}$	Linear Inequality Constraint Matrix
$\mathbf{b}$	Linear Inequality Constraint Right Hand Side
$\mathbf{l}_\mathrm{b}, \mathbf{u}_\mathrm{b}$	Rectangular Bound Vectors
$\mathbf{Q}$	Setpoint Deviation Weighting Matrix
$\mathbf{R}$	Input Increment Weighting Matrix
$\mathcal{J}$	Objective Function Result
$\mathbf{z}$	Primal Variable Vector
$\boldsymbol{\lambda}, \mathbf{l}, \mathbf{u}$	Dual Variable Vector
$\mathbf{t}, \mathbf{g}, \mathbf{s}$	Slack Variable Vector
α	Step-Size Scaling Factor
μ	Complementarity Gap
σ	Centering Parameter
ϕ	Feasibility Parameter
ϵ_r, ϵ_c	Convergence Tolerances
$\mathbf{r}_1$	Dual Residual Vector
$\mathbf{r}_2$	Primal Residual Vector

In addition to the above symbols, the model predictive control chapters will use the following notation for mathematical equations.

Table 2: Model predictive control notation.

Notation	Description
$\mathbf{X}$	Matrix (bold and capitalized)
$\mathbf{x}$	Column vector (bold, lower case)
x	Scalar (italic, lower case)

For the steam utility modelling and optimization chapters, the nomenclature is modified to match that from literature.

Table 3: Steam utility system modelling nomenclature (Chapters 5-7)

Symbol	Property	Unit
Q	Heat Duty	kW
W	Mechanical Work/Electrical Power	kW
M	Mass Flow Rate [1]	kg/s
H	Specific Enthalpy [1]	kJ/kg
S	Specific Entropy [1]	kJ/(kg K)
C_p	Heat Capacity at Constant Pressure [1]	kJ/(kg K)
T	Temperature	°C
P	Pressure	bar
X	Quality	-
η	Efficiency	-
F	Fraction	-
R	Gas Constant (8.3145)	J/(mol K)
Z	Compressibility Factor	-
C	Cost	$
b	Binary Variable	
α, β, γ	Model Specific Regression Parameters	

[1] A mass basis (kg) is used for water and steam thermodynamic calculations, while a molar basis (kmol) is used for fuel-gas thermodynamic calculations.

Chapter 1

Introduction

1.1 Optimization in Industrial Applications

Mathematical optimization, also known as Operations Research, is becoming increasingly ubiquitous in today's world of narrow commercial margins and competitive markets. Whether applied by economists to portfolio optimization, engineers to power flow or managers to employee scheduling, the focus remains the same: Minimize cost or risk while maximizing efficiency or performance, given the resources currently available and specified operational constraints. When applied to real-world practical optimizations problems, the term industrial optimization is used, however this term also carries a certain amount of baggage. This is because real-world problems are often large and unwieldy due to the complex physical relationships between elements within the system. It is for this reason that optimization of real systems is more difficult than textbook mathematical problems, and as such requires state-of-the-art optimization solvers tailored for the particular problem to obtain robust solutions within an acceptable time frame.

In our experience within an academic-based industrial consulting group, (I^2C^2, `i2c2.aut.ac.nz`), we see a strong reluctance by industrial engineers to use optimization on a regular and routine basis. The interesting observation is that this reluctance still persists even when the client is competent using the basics of optimization, and is confident with the underlying process models. We believe that the expense of robust high-quality optimization code, coupled with the difficulty in setting up the problem (GAMS (General Algebraic Modeling System)/AMPL (A Mathematical Programming Language) etc notwithstanding) explains much of this reluctance. What has changed recently in our dealings with industrial clients, primarily in the process, oil and gas and utility industries, is their comfort and familiarity with software tools, so that they are now prepared to rely on software such as MATLAB. While there could well be more sophisticated prototyping environments from a computer science perspective (such as Python or Ruby), or even environments more cost effective (such as Scilab or Octave), or environments that have garnered domain specific loyalty (such as R amongst statisticians), nothing approaches the market presence or commercial support of MATLAB for such a wide diverse group comprised of scientists and engineers.

We regularly talk to competent engineers and technologists whose primary interest and expertise is their own domain. They are not optimization specialists, nor particularly comfortable in sophisticated coding environments, but they do want to solve their particular optimization problem, and typically in a hurry. We term this group the 'industrial MATLAB user', who are using MATLAB on a Windows PC to solve industrial modelling and optimization problems. We recognised that this group was substantially under-utilizing the enormous potential in the open-source optimization community, which for the industrial user was either too hard to compile, too hard to use, or too time consuming to set it up. We also realised that a framework was required that could provide the 'glue' between the problem in MATLAB and the optimization solver, automatically converting the high-level problem into a low-level format suitable for the solver algorithm to understand, and also exploiting structure in the problem to speed up solving it.

To illustrate the optimization framework developed, this work investigates two apparently distinct industrial optimization problems: Firstly the solving of a quadratic program within a high-speed embedded controller for dynamic systems, and secondly, the operational optimization of large combined heat and power systems. These two areas are deliberately chosen to emphasize physical differences, such as process speed and plant size, and they therefore represent a large spectrum of problems that could be encountered in an industrial environment. However we say "apparently distinct" because as will be shown in this work, they are actually quite similar when viewed from an optimization point of view. For each problem, this work will aim to exploit both the structure of the problem, as well as how to formulate the optimization problem, which when a solver is tailored to solve, can result in substantial performance improvements.

A contribution of this work is to provide a framework within which the optimization problem can be described, using a high level description such as that within MATLAB, while still exploiting the structure of the problem. This allows the user to focus on modelling their particular system, without worrying about the implementation details within a particular optimization solver. To achieve this aim requires the development of a framework which can pre-process an optimization problem into a form suitable for the targeted solver, as well as applying problem specific simplifications such as the identification of redundant expressions, automatic derivative generation and generating an efficient formulation. Using this framework, problems from physical size extremes of the industrial optimization spectrum can be described using a common format, then solved using vastly differing techniques which are tailored for each particular problem.

In the case of the predictive control algorithm, this work focuses on the quadratic program solver, simultaneously accelerating the solution speed of the algorithm, while reducing the memory requirement. To achieve this, we must bound the problems of interest with respect to size, density, and structure, as well as investigate the numerical conditioning of the problem matrices. This thesis aims to develop both algorithmic modifications to the established primal-dual solver, as well as developing problem conditioning and warm-starting heuristics which will enable the algorithm to be deployed on modest embedded hardware. Furthermore, in order to keep the implementation as high-level as possible, an auto-code generator will take a high level control description and generate a tailored solver based on the problem properties, which is suitable for compilation with any standard ANSI C compiler.

With respect to the second problem, this work focuses on both developing a succinct symbolic modelling language for describing industrial utility systems, as well as a suite of optimizer-ready part-load models suitable for predicting off-design performance, based on literature regressions and rigorous thermodynamic models. The modelling language will allow algebraic models to be described natively within MATLAB, allowing for problem identification, simplification and compilation into a format tailored to the selected optimizer, thereby increasing the performance and robustness of the optimizer. This technique again combines a high-level problem description with a common intermediate format which is then tailored depending on the problem entered and the solver chosen.

In both cases substantial performance improvements are observed over standard generic optimization routines, with no low-level modifications required by the user. This approach 'lowers-the-bar' to the design and implementation of an optimization model, and allows non-optimization specialists access to obtaining economically significant results for real-world optimization problems [68].

1.2 The Spectrum of Industrial Optimization

As stated by Patrick Bangert in his book *Optimization for Industrial Problems*, [243], "Industrial optimization lies on the crossroads between mathematics, computer science, engineering and management". This renders the subject multi-disciplinary, requiring knowledge of a range of specialist areas to be able to generate an optimization problem that not only encapsulates the key problem properties but also when solved, provides a *practically* feasible solution and not just a mathematically feasible one. Moreover, the design and setup of the optimization problem must be able to be completed in a succinct manner, without obtuse constructs or exhaustive manual data entry, to ensure the problem is maintained to current operating conditions. Furthermore, for an optimized solution to be meaningful, it must also be relevant, and therefore capable of being solved within an acceptable time frame on hardware common in an industrial plant.

For this work we will investigate two significant industrial optimization problems, which as stated in the previous section, encompass a wide spectrum of problems that can be expected within a typical industrial environment.

1.2.1 Embedded Model Predictive Control

Model Predictive Control, or MPC, is an advanced control strategy suitable for constrained control of multivariable systems [105, 223, 271]. MPC is an acronym used to describe several similar control strategies such as Receding Horizon Control, Generalized Predictive Control (GPC) [75], the related Predictive Functional Control (PFC) and Identification and Command (IDCOM) [276, 275], and the original Dynamic Matrix Control (DMC), developed by Shell Oil engineers Cutler and Ramakar [74].

Given that the MPC algorithm can not only handle multivariable systems naturally, but also that operating constraints can be placed on inputs, outputs or states, the MPC algorithm has proved to be highly successful in industrial applications across the world, with more than 4600 implementations reported in the early survey by Qin and Badgewell [263]. As described by this paper, MPC was originally developed to fulfill the specialized control needs of petroleum refineries, however this technology is now found in other areas such as chemical, food processing and has recently moved into the automotive and aerospace industries. This success places MPC as one of the most successful advanced control algorithms in common use today [9, 334], however this success rests almost exclusively on a robust internal optimization solver.

In all traditional MPC algorithms (which excludes *explicit* MPC, described later in Section 2.2.2), an optimization problem must be solved online, at each sample, in order to calculate the next control move(s). This optimization problem is a result of the control move formulation (see Chapter 3) as well as the system constraints included in the controller, and due to both this constrained property and the problem size, must be solved numerically. This presents a significant challenge for the MPC controller: It must solve the optimization problem within the sample time allowed in order to apply the next control move(s) in time, and therefore the sample rate of the controller is often limited by the speed of solving the optimization problem. It is for this reason that MPC has been typically applied to petrochemical refineries and chemical plants, where the slower time constants of the systems to control permit slower sample rates, giving the optimizer time to converge.

The upside to this extra computational load is that the MPC controller is *optimal*, and, given the inherent prediction within its formulation, can out-perform most other industrial controllers in terms of minimizing reference deviation, constraint violation, and overall variability. This excellent control performance has not gone unnoticed, with multiple other industrial and commercial users recently investigating the applicability of the algorithm to high-speed processes such as unmanned vehicles, medical device control, and other small agile systems, as reviewed later in Section 2.2.4.1. The challenge of their applications has been two-fold: Firstly a number of these application areas require low-power, small and lightweight electronics, excluding the computing power of the desktop PC, and secondly, as the physical size of system to control decreases, sampling rates often have to increase to match faster time constants and thus maintain control. The result is that we are applying memory and computationally intensive MPC to small high-speed systems, which is then implemented on embedded hardware limited by available power, size and weight.

Existing research into this area of 'embedded MPC' has primarily focused on two distinct areas. The first area retains the online optimization step, but uses either advanced hardware or novel algorithms to accelerate the optimization solver, and thus the achievable sampling rate. Advanced hardware implementations include embedded MPC on Field Programmable Gate Arrays (FPGAs) which allow parts of the solver to be parallelized, such as common mathematical operations [186]. Alternative hardware implementations include using Digital Signal Processors (DSPs) targeted with an assembler level implementation to fully exploit the hardware resources available [331]. From an algorithm perspective, recent work has explored existing mathematical algorithms applied to MPC, such as the fast-gradient method [348], as well as exploring opportunities within existing MPC solvers [338].

The second area of existing research aims to reduce or remove the online optimization step altogether by applying a parametric step which precalculates all possible control moves [8], known as explicit MPC. Each of these areas is described in more detail later in Chapter 2, including a review of current literature.

What has been overlooked we believe, is that an efficient, hand-coded implementation in C of an online quadratic programming solver tailored for both MPC problems, and traditional embedded hardware (such as microcontrollers) can be just as effective, if not more practical, for real implementations of MPC on embedded hardware. In this way the optimization solver is not limited to one hardware target (assembler is target specific), or specialty hardware (FPGAs are inherently difficult to program, as described in Section A.4.1); nor is it limited to a highly memory intensive explicit formulation, but remains suitably general for practical implementation, whilst at the same time achieving sampling rates exceeding 10kHz on modest embedded hardware. In order to bound the problems of interest for this work, we consider problems with an upper limit of 40 decision variables and 320 constraints, with the nominal problem size consisting of 20 decision variables and 160 constraints. This problem size encompasses a significant proportion of the possible embedded control problems in literature, as reviewed in Section 2.2.4, as well as opens the door for larger control problems with more inputs or outputs, or longer horizons for increased stability.

This targeted application of industrial optimization will cover the small, high-speed section of the spectrum, and will include the development of a new implementation of two quadratic programming solvers, together with a framework which can deploy high-speed, memory efficient MPC controllers to a range of common embedded hardware.

1.2.2 Large-Scale Energy Optimization

In contrast to the small high-speed controllers from the last section, the second investigation into industrial optimization will target the opposite end of the spectrum, physically large and slow-reacting energy systems. For this work the energy system is defined as a plant designed to supply the heating, mechanical and electrical demands of an on-site process plant, also known as a utility system. These systems use steam as the working fluid which is generated using boilers or heat recovery steam generators, then used to drive steam turbines (whether back pressure or condensing) to supply mechanical or electrical loads, or simply the steam itself is used to supply heat to process users. In addition, gas turbines are often used for on-site cogeneration, also known as Combined Heat and Power (CHP) systems, allowing the utility system the ability to export electricity when economically sensible to generate more than the site requirement. A simple hypothetical steam utility system is shown in Figure 1.1, indicating the major pieces of equipment present in a typical system. Note the utility system considered in this work is nonlinear, discrete (contains integer/binary variables) and algebraic (i.e. steady-state optimization only) and thus represents a very different problem from the previous section.

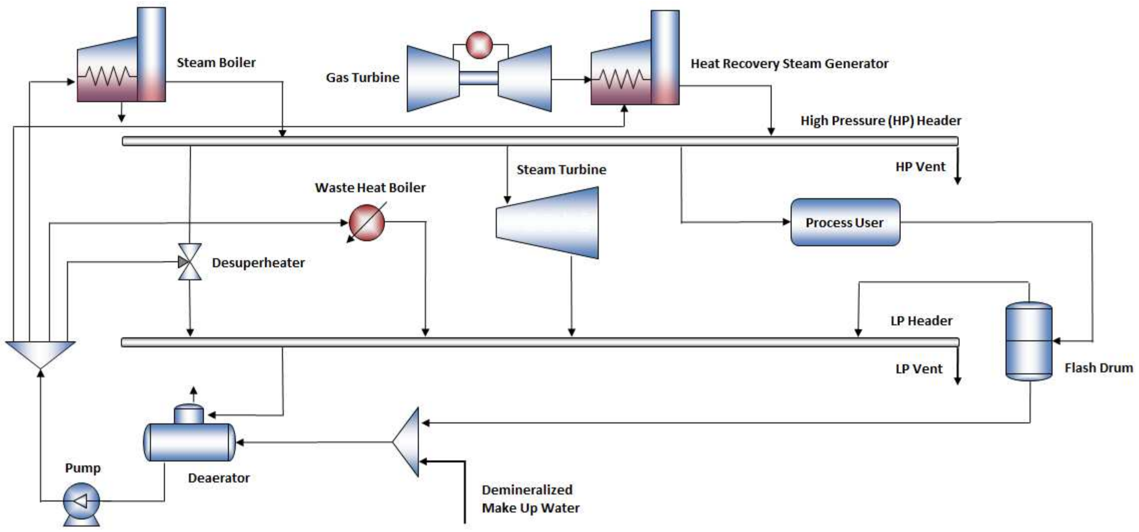


Figure 1.1: An example steam utility system with steam and water material streams shown, along with common utility equipment such as boilers, turbines, headers, users and condensate recovery

The operation of a utility system represents the largest managed operating cost for the hydrocarbon industry [92], yet most interesting, from both a control and operational point of view, it typically takes a back seat to production [89]. Historically, with flat energy prices and constant production policies, many sites probably found a reasonable operating point eventually, even if only by trial and error. However, with the development of the smart grid and subsequent possibilities for online electricity trading, and the move to flexible production policies means that the efficient use of energy is now crucial, complex, multi-faceted [3] and with substantial potential for operational savings [90, 80].

Given the opportunities to export electricity, together with varying process demands, the optimal operation of a utility system is a non-intuitive decision of which equipment to run, how to generate the required steam, how much power to generate, and how to meet the required energy demands. Most industrial utility systems are built with electrical back-ups of key mechanical drivers (i.e. steam turbines), therefore exploiting this redundancy provides extra degrees of freedom to optimize the operational expenditure (OPEX), and thus reduce operating costs. However deciding what equipment to run introduces binary variables, which when coupled with the discrete nature of a utility system and nonlinear (and non-convex) energy balances, results in a Mixed Integer Nonlinear Optimization Program (MINLP). This problem is non-deterministically polynomial ($\mathcal{NP}$-complete or $\mathcal{NP}$-hard, depending on the problem), meaning there is no known algorithm that can deterministically solve it in polynomial time [52].

With recent developments in both convex MINLP solvers (convex in the sense the relaxed nonlinear program is convex) such as BONMIN [42], as well as non-convex deterministically global MINLP solvers, such as BARON [304], it is now reasonable to expect to solve the utility system operational optimization problem not only quickly, but also robustly. However building a utility system model within a process simulator (as done in [71] or described in [89]) and wrapping it in one of these optimizers is either unlikely to work, or as shown in Section 6.3.2, is slow and typically converges to a suboptimal point. Therefore a more advanced formulation is required in order to leverage the structure of the problem, and enable the optimizer to take advantage of mathematical characteristics present in the model.

A common utility system formulation in literature is the reduction of the optimization problem to a set of linear constraints with a linear objective which is then solved as a mixed integer linear program [5, 134, 149, 197, 213]. While existing literature has predominantly focused on the *synthesis* optimization problem, whereby the complete design of a utility system is being considered, those mentioned all linearize the optimization problem by breaking energy balances across the system, resulting in fixed header temperatures. When considering operational optimization where the available OPEX savings are limited (typically 1 to 5% [92]), this inaccuracy will lead to either an unrealistic savings or an unimplementable operating point, and possibly a damaging condition known as 'steam hammer' [311]. Therefore to describe modelling a utility system using linear expressions alone is not adequate for the operational optimization task.

To enable optimization of these large industrial systems, a formulation which retains the nonlinearities is required. It must however also describe the model in a way the optimizer can exploit, for example via analytical derivatives. Furthermore, by identifying features such as linear, bilinear and quadratic expressions, the problem can be reduced to a form which a tailored optimizer (e.g. quadratic program or second-order cone solver) can solve much more efficiently than via a general nonlinear optimizer. To achieve this aim, this work develops two new tools for describing and solving optimization problems: SymBuilder for generating algebraic formulations of general nonlinear models, and OPTI, which provides a suite of open-source optimization solvers for solving the resulting problem. These two tools enable large industrial utility systems to be modelled, validated and optimized within a few seconds, and provide economically significant results.

1.3 Research Questions and The Thesis Contribution

This work aims to both lower the barrier for the application of optimization to general industrial problems, and improve the results achievable by tailoring the optimization solvers to exploit structure within the problems. Two industrial optimization problems that superficially look very different are used to show that common tools and methods can be used to improve the performance of solving both problems, regardless of differences in scale or system speed. To complete this aim, the following questions are posed, then critiqued later in Chapter 8:

1. Can we fabricate a linear embedded MPC controller that costs less than US$200, is capable of sampling rates exceeding 5kHz, fits entirely within on-chip memory and robustly solves the optimal control moves in reduced precision hardware?
2. What is a suitable high-level framework that can describe, tune and simulate a model predictive controller, and can also generate a hand-optimized, high-speed ANSI C controller suitable for embedding into a range of low-cost hardware?
3. By using a rapid-prototyping language such as MATLAB, what is a framework design that can simplify the structure of a system of equations and generate an efficient, optimizer-friendly representation? In addition, what is the design of this representation so that a common problem format can be defined, regardless of the optimization solver being used?
4. Can we formulate an operational steam utility model that is thermodynamically rigorous, yet still remain tailored for an optimization solver, so as to robustly solve industrially significant problems in less than a second, and return physically realisable results?

We believe the framework-based approach to be a significant contribution of this work, whereby a user of our framework is able to complete an entire design, beginning to end, within a single environment. This framework concept allows a common methodology, a common way of thinking and a common set of tools and functions, allowing the user to focus on the design, rather than the intricacies of the optimization/control problem. The significance of this initiative is demonstrated by the suite of new tools developed, one of which is now used internationally by several thousand researchers and industrialists. The most successful is the OPTI Toolbox [64], a MATLAB toolbox collecting together a library of open source optimization solvers, as described in Section 6.2. The toolbox provides a common problem definition object which allows linear, quadratic or nonlinear problems, including mixed integer variants, to be described within the same object and then solved with any compatible optimization solver. The toolbox automatically handles all low-level tasks such as identifying the problem entered, conversions required to match the specified solver, approximating derivatives and options tuning, as well as the final collection and processing of results.

In addition, the SymBuilder algebraic framework is supplied as part of the OPTI Toolbox, which extends the symbolic capabilities of MATLAB to succinctly model optimization problems, including generation of full analytical first and second derivatives, sparsity patterns and structure identification. Furthermore, two interfaces have been developed to deterministic global optimization solvers, providing MATLAB users with the first opportunity to prove global optimality to general nonlinear problems. Together, these tools are in use by over 4000 registered individual users worldwide, which demonstrates that providing this high-level framework within MATLAB is of significance to the operations research community.

A further package developed was the JSteam modelling library [62], as described in Section 5.3. The library contains two robust thermodynamic engines, together with a suite of utility system unit operation models. JSteam is used to construct utility system unit operation model approximations, by providing a rigorous thermodynamic platform from which regressed models can be generated for later optimization. Together with the Excel interface, JSteam has been successfully commercialized by a company spawned from this research, and is now sold to companies and universities around the world, further reinforcing the significance of this work.

The final package written is the jMPC Toolbox [63], described in Section 3.5.2, which encompasses the solvers and algorithms developed within model predictive control research within this work. Predominantly open source, the toolbox has been downloaded by over 250 users internationally and is used for teaching at a number of leading universities. The quadratic programming solvers developed are currently the best in class for the problems they are designed to solve, and the helicopter case study (Section 4.6) has been profiled by Quanser, the manufacturer of the laboratory equipment used [266].

Together, these three software packages encapsulate the academic contributions of this work. This includes the formulation of a high-speed quadratic programming solver for embedded MPC and an extension to existing literature utility system unit operation models. In addition, a framework for describing symbolic optimization problems within MATLAB has been developed, which automatically exploits structure when solving and allows generation of global optimization solver algebraic descriptions. Finally a modelling methodology for exploiting the structure present in utility system models has been described, allowing high-speed optimization of industrially significant systems. In all contributions of this work, metrics such as cost, sampling rate, problem size and solution speed are reported to give the reader a sense of scale, as well as provide a benchmark for future comparisons. We also believe these metrics are important to bound the scope of this research.

1.4 Thesis Outline

This chapter has introduced the two industrial optimization problems that will be considered within this work: The development of a quadratic programming solver suitable for high-speed embedded model predictive control, and secondly the mixed integer modelling and optimization of steam utility systems. The reasons why these areas are significant has been explained, and the research questions detailed.

Chapter 2 reviews existing studies into both model predictive control and utility system optimization, as well as general industrial optimization. Alternate algorithms and formulations are discussed, reasons are given as to why these are not applicable to this work, and gaps identified which this work aims to fill.

Chapter 3 introduces the model predictive algorithm and the formulation of the quadratic program. A path-following primal-dual quadratic programming solver is developed, including algorithmic modifications to improve the efficiency of the algorithm for solving problems that result from the MPC control law.

Chapter 4 describes the embedded MPC implementation, and includes the development of an auto-coding framework used for generating a highly efficient quadratic programming solver and MPC controller. Two case studies are presented to validate the algorithms, the first a processor in the loop implementation and the second embedded MPC control of a laboratory helicopter.

Chapter 5 develops a library of thermodynamic routines together with a suite of utility system unit operations models suitable for rigorously describing all common equipment. Regressions from literature are adapted and incorporated to formulate a set of optimization ready part-load unit operation models.

Chapter 6 introduces the OPTI Toolbox and SymBuilder framework for modelling utility systems. An optimization methodology is described, and several case studies from both literature and industry are presented.

Chapter 7 describes whitebox optimization and the development of interfaces for two deterministically global optimization solvers. Case studies from Chapter 6 are re-run and a comparison of global and local solutions given.

Chapter 8 critiques the research questions, gives conclusions from this work, and suggests recommendations for future development opportunities.

Chapter 2

Literature Review

2.1 Industrial Optimization

Applying mathematical optimization techniques to increase the efficiency/production of industrial systems is an established discipline, generally referred to as Operations Research (OR) or from a business perspective, Management Science (MS). The origin of operations research can be traced back to England during World War II, where A. P. Rowe formed teams to carry out "operational researches" on the communication system within a British radar station [270]. The technique was soon adopted by English physicist P. Blackett, who succeeded in convincing management at the time that a scientific approach to managing complex operations was not only worthwhile, but could also save lives during the war. Examples of his studies included improving the survival odds of naval convoys, which were constantly being sunk by Nazi German U-boats, as well as minimizing the armour plating of bombers in order to reduce weight (and thus increase payload), and still have a high success rate of returning to base [47].

The Americans also recognised the benefit of this scientific approach to warfare, initially implementing it within the US Navy's Mine Warfare Operations Research Group. It made its biggest impact however in the Antisubmarine Warfare Operations Group. Led by American physicist Philip Morse, the group was tasked with addressing the problem of Nazi German U-boats attacking transatlantic shipping [270, 296]. Significantly, many regard the contribution Morse and his team made to the American war effort as an important factor in winning the war [49], earning Morse the title of the Founder of Operations Research in the USA.

Post World War II it was soon realised that OR was equally applicable to civilian problems, and specifically problems within the manufacturing industry such as scheduling, inventory control and resource allocation [270]. A major contribution to the field of OR at the time was the Simplex algorithm, which could efficiently solve linear programs [26, 77]. As stated by Rajopal with respect to OR, "George Dantzig, who in 1947 developed the simplex algorithm for Linear Programming (LP), provided the single most important impetus for this growth", indicating the impact that rigorous mathematical methods had on solving OR problems [270].

With the advent of digital computers, solving large industrial optimization problems using OR techniques became possible. This not only enabled large linear optimization problems to be solved, but allowed other mathematical techniques developed years earlier to be applied to industrially significant problems. An example is the multivariable Newton-Rhapson method for nonlinear equations, which forms the basis for many modern optimization algorithms. Furthermore, new industrially significant applications allowed researchers to revisit well-known problems such as the travelling salesman problem and realise it could be solved as an integer linear programming problem. This prompted researchers such as Ralph Gomory to develop methods to simplify integer programming problems by using cutting planes, which successively refine an optimization problem using linear inequalities [114].

To optimize an industrial system, Figure 2.1 illustrates the 7-step OR process as described by Rajopal in [270]. The process begins with an orientation step where

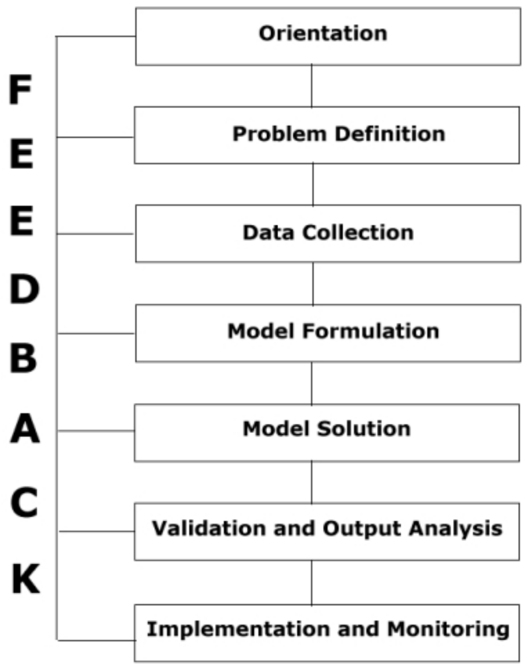


Figure 2.1: The Operations Research approach (Figure 1 in [270])

typically a multidisciplinary team is formed. The requirement of a multidisciplinary team is a key feature of operations research, which combines the domain specialty knowledge from a range of specialists, as described in Section 1.2. The second step is recognised as the most difficult, whereby the objective function is defined, based on identifying goals for the process targeted (such as minimizing resources and/or maximizing profit). The third step requires the collection of historical process data, so that informed decisions can be made on future requirements. The fourth step combines all information collected so far into a model suitable for optimization. The fifth and sixth steps solve the resulting optimization problem, then validate the results to ensure they are both physically realisable and sensible. If problems are identified in this step, then typically the model from step four will need to be revisited. The final step implements the study findings within the process, and monitors them to ensure the expected outcomes are realised.

This approach to industrial optimization is now well known and appreciated, with a multitude of case studies reported in the literature. Examples include aircrew scheduling for an international airline [288, 289], supply chain planning in the paper industry [251], crude oil blending problems in the petrochemical industry [282] and manufacturing under tooling constraints [95].

In order to formulate and solve the optimization problem in steps 4 and 5 of Figure 2.1, most users will opt for a modelling and optimization framework on a desktop computer or high-performance cluster. This allows the process model to be described in a language designed for optimization problems, and then solved with powerful optimization solvers that can exploit the large amount of memory and processing power of a modern computer. Within the commercial realm of modelling and optimization packages, the five main contenders are GAMS [286], AIMMS [284], AMPL [101], IBM ILOG CPLEX Optimization Studio [145] and FICO's Xpress Optimization Suite [93]. CPLEX and Xpress both provide the modelling environment and their own solver, while the remaining packages rely on third party external optimization solvers for solving the model. All of these tools are designed purely for solving mathematical optimization problems that result from operational, scheduling and planning studies. More general commercial tools that also include the ability to solve optimization problems, but are generally regarded as modelling platforms, include MATLAB [205], Mathematica [335] and MAPLE [196]. In addition, Microsoft Excel is often as a modelling package as it contains a range of optimization solvers via the Solver add-in.

Within the open-source/free community a number of packages exist which can model and solve problems from the industrial optimization field. These include Modelica [20], Python and the associated NumPy/SciPy modules [158, 236], Octave [91], Scilab [292], Julia [34] and open-source addons to commercial platforms such as OpenSolver [202] for Excel and YALMIP [189] and OPTI Toolbox [68] for MATLAB.

This summary concludes a brief review of the literature leading to the formation of the definition of industrial optimization, and the techniques and tools used to solve problems that arise when undertaking such a study. The following sections will review the literature within the two detailed areas of industrial optimization targeted within this work: Embedded model predictive control and the optimal operation of utility systems.

2.2 Model Predictive Control

Model Predictive Control, or MPC, is perhaps the most successful advanced control strategy in use today [9], with thousands of industrial applications across a range of process industries [263]. It is, however, an anomaly in that unlike most other control algorithms, such as the Linear Quadratic Regulator (LQR) or Gaussian (LQG) [160], MPC is the result of original development in industry, rather than academia. This is a result of control theory not fulfilling the needs of control practice at the time, together with a considerable gap between the academic and industrial control engineers.

As described by Qin and Badgwell in [263], the predecessor to MPC was LQG [159], first described by Rudolf Kalman, an electrical engineering academic. The algorithm had many interesting properties, including the handling of multivariable systems implicitly, optimal state estimation and being self stabilizing due to its infinite prediction horizon. Interestingly however, it ultimately failed to have a strong impact on the industrial process control community. The reasons were twofold: The controller was unable to handle constraints [105, 276], and the reluctance from control engineers to adopt the technology, due to the culture at the time [263]. It was to be nearly 15 years before the next major control technology was to appear, and this time two technologies would be independently developed in industry.

The first MPC-type application was described by Richalet et al. in [276], which they later called Model Predictive Heuristic Control (MPHC) [277]. Jacques Richalet was a French engineer who specialized in applied mathematics, regarded by many as the ‘grandfather of predictive control’. He formed the process engineering consulting company ADERSA in 1968, and commercialized an MPC-type product called IDCOM, Identification and Command. As with traditional MPC, detailed in the next section, IDCOM used an internal dynamic model, implemented a quadratic performance objective over a finite prediction horizon and allowed constraints on both inputs and outputs, although solved via a heuristic iterative algorithm. The concept of a finite prediction horizon meant the control problem could be solved iteratively at each time step, by only considering a finite number of future time steps (i.e. the prediction horizon) to control over. This technique allows the controller to take into account the transient behaviour of the system, as well limit the size of the control problem to solve at each sample. This idea is also known as receding horizon control, as the finite prediction horizon would shift one step into the future at each sample. The concept of a finite prediction horizon underpins most practical predictive control algorithms, and thus was a significant feature within IDCOM. However within IDCOM there are several differences compared to traditional MPC, including that it used a finite impulse response (FIR) model, so that system inputs immediately effected system outputs, together with the heuristic solution strategy. IDCOM still exists today as Predictive Functional Control (PFC), and as the underlying algorithm in Honeywell's Profit controller, known as Robust Model Predictive Control Technology (RMPCT) [131].

Independently of Richalet, two Shell Oil engineers, Cutler and Ramakar, developed their own MPC-type technology which they called Dynamic Matrix Control (DMC) [74]. An initial application is reported in 1973, but because a description of the algorithm was not published until 1979 [73], Richalet earned the honour of the first MPC description. However it is DMC (and its successors) which has had the most significant impact on industry [223], due to its huge adoption by the oil and gas industry [262]. The fundamental DMC algorithm uses a step response model together with a quadratic objective over a finite prediction horizon. However, unlike QDMC (described next), it was an unconstrained controller which solved the control moves using a least squares methods. Despite this difference it was an optimal controller, and provided the platform for rapid developments in the area of predictive control. In a companion paper to the Cutler and Ramakar paper, Prett and Gillette [254] described a modified version of the DMC algorithm applied to a Fluid Catalytic Cracking Unit (FCCU), which included the ability to handle nonlinearities and constraints via a technique known as 'time variant constraints'.

As stated by Qin and Badgwell in [263], "The original IDCOM and DMC algorithms provided excellent control of unconstrained multivariable processes. Constraint handling, however, was still somewhat ad hoc." This comment referred to the heuristic method used by IDCOM to solve its constrained optimal control problem, as well as to the time variant constraints within the modified DMC method. These were added as required when the process came close to a constraint and therefore were not rigorous in their application. The solution was proposed by Cutler, Morshedi and Haydel in [72], where the DMC problem could be posed as a quadratic program which included the quadratic performance as the objective, and the input and output constraints as linear constraints. The controller was termed Quadratic DMC (QDMC), and was described in detail by a later paper by Garcìa [104] in which an application to a 3×3 multivariable pyrolysis furnace was described.

A further technology that evolved out of the finite prediction horizon concept was that of Generalized Predictive Control (GPC), [75], or long range predictive control. As summarized by Morari and Lee in [223], GPC was designed as an extension to adaptive control and disturbance modelling played a key role in its algorithm. This was opposed to the design of DMC which was purely deterministic in its formulation. As stated by Morari and Lee themselves, GPC has failed to establish itself in the industrial world, due in part to difficulty applying the algorithm to multivariable problems, although it is still the subject of academic research. The lines are however now blurred between GPC and MPC, with many authors using the terms interchangeably and in recent times, generally referred to the same class of controller [287].

The QDMC algorithm forms the underlying algorithm for what is termed the 'traditional MPC controller' within the remainder of this work, and is generally regarded as the predecessor to modern MPC. The following subsections will detail the traditional MPC algorithm, the problems associated with the traditional algorithm, and lastly expand into the area of embedded MPC and its challenges.

2.2.1 Traditional MPC

Traditional MPC is defined within this work as a discrete model predictive controller which utilizes a linear state-space (or transfer function) model of the process to be controlled. The model is used to predict the future response of the process over what is known as the prediction horizon (a finite set of future samples), in order to be able to optimize the control moves. At each sample the controller optimizes online a sequence of current and future control moves over the control horizon, in order to bring the model's predicted output(s) to follow a reference (setpoint). It then implements just the first control move. The algorithm repeats at each sample, where both the prediction and control horizons move one step into the future, hence the name *receding horizon* control.

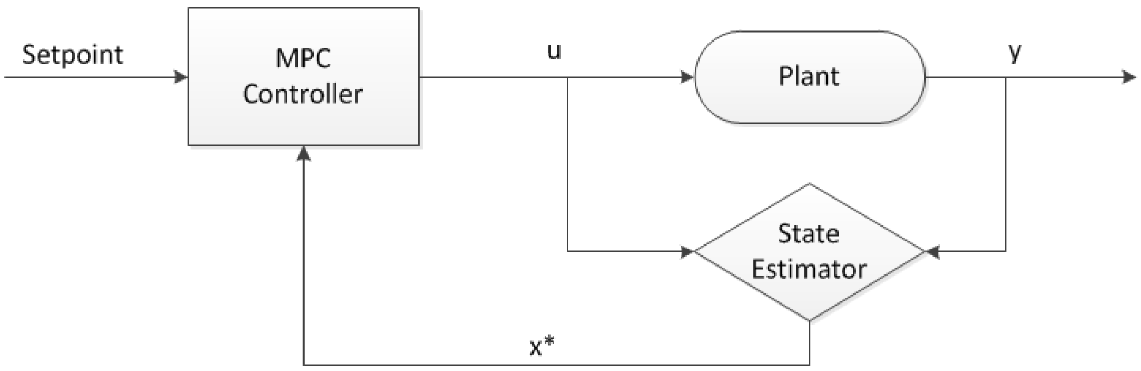


Figure 2.2: The standard form of the MPC algorithm.

The general structure of the MPC algorithm is illustrated in Figure 2.2, while it is described in more detail in Section 3.2.1. Further algorithm details can be found in the review paper by Manfred Morari [221], as well as the early survey paper by Garcìa et al. [105].

Given that MPC has seen both a huge surge in research (as any journal search of 'predictive control' will show), and a large uptake in industrial applications as reported by Qin and Badgwell, it is therefore surprising to note that until recently it has not been utilized outside the process control community [86]. This can be explained by the computational expense of the algorithm, because solving the optimization problem online (such as done with QDMC) requires significant computation resources. This therefore limits the applicability of the algorithm to processes with relatively slow time constants, as is typical in process control applications [66].

To apply the control benefits of model predictive control to smaller, faster processes such as those within the automotive, medical and aerospace industries, two changes need to occur. Firstly, optimization algorithm improvements are required to solve the online optimization faster, and secondly MPC has to be implemented on small, low-power embedded hardware suitable for deploying to mobile platforms. It is worth noting that these two requirements do not complement each other; Low-power embedded hardware is often severely restrained in computational power and memory, and this then lowers the peak achievable performance of algorithms implemented on the hardware.

2.2.2 Explicit MPC

Explicit MPC retains the same underlying principles of MPC, however it aims to remove the online optimization step. It does this by explicitly calculating all possible future control moves, as a function of the current state and reference vectors, by posing the MPC problem as a multiparametric programming problem. This reduces the online optimization step to a table lookup, of the form

$$u(x) = \begin{cases} F_1 x + g_1, & \text{if } H_1 x \leq k_1 \\ \quad\vdots & \quad\vdots \\ F_M x + g_M, & \text{if } H_M x \leq k_M \end{cases} \tag{2.1}$$

which is significantly faster to solve than a full quadratic program online. The full algorithm details, including formulation of the multiparametric MPC problem, are detailed in the survey paper of explicit MPC by Alessio and Bemporad [8].

Given that explicit MPC reduces the computational cost of MPC, it could easily be thought of as a solution to this industrial optimization problem. However, as with most strategies there is a downside and for explicit MPC this is the memory requirement. Within the explicit MPC formulation, every possible combination of constraints active at the solution (problem partitions) must be formulated and included in Equation 2.1, which for a multiparametric quadratic program, is 2^q, where q is the number of constraints [8]. For the problems of interest within this work, the number of constraints is typically greater than 50 even for small problems with modest horizons, and therefore the memory requirement is important. An example of explicit MPC is given in [156], where for an MPC problem with a control horizon of 2 and 4 decision variables, 62MB is required to store the explicit formulation. This amount of memory would be unrealistic on the embedded hardware looked at for this work (see Section 4.2, noting typical embedded memory of the low-power, portable embedded systems considered is less than 256KB), and together with such a short horizon, is unlikely to offer the required control performance.

It is for this reason that recent research has focused on *suboptimal* explicit MPC, whereby suboptimal solutions can be obtained by relaxing the KKT constraints [31], offline searching for the most common partitions [239] or even simply merging areas where the affine gain is the same [110]. Each of these methods attempts to minimize the number of partitions stored in order to reduce memory requirements, at a cost of optimality. Whether this suboptimal solution severely affects the control performance will depend on both the control problem and the approximations made. However the fact remains that in order for explicit MPC to be deployed on modest embedded hardware with reasonable horizons, it is no longer a strictly optimal controller.

2.2.3 First-Order Methods

A recent important re-discovery within the control community is that of first-order methods, also known as gradient methods, for solving the online quadratic optimization problem resulting from a linear MPC controller [280]. Traditional gradient methods solve strongly convex optimization problems of the form

$$\min_{\mathbf{x}} f(\mathbf{x}) \tag{2.2}$$

$$\text{subject to: } \mathbf{x} \in \mathcal{C} \tag{2.3}$$

where f is a real convex function and $\mathcal{C}$ is a closed convex set (further requirements, including Lipschitz Continuity are described in [280]). A traditional gradient method solves this minimization problem by stepping along the anti-gradient

$$\mathbf{x}_{k+1} = \mathbf{x}_k - h_i \nabla f(\mathbf{x}_k) \tag{2.4}$$

where h_i is the step-size and is chosen so that it satisfies the relaxation sequence

$$f(\mathbf{x}_{k+1}) \leq f(\mathbf{x}_k), \quad \forall k \geq 0 \tag{2.5}$$

which for traditional strategies is typically implemented as a constant. An interesting property of the algorithm allows it to be applied to constrained problems as well, by replacing the gradient ∇f with a gradient mapping. This is defined in Equations 10a-10c in [280].

While the algorithm described so far is suitable for solving the convex quadratic programs within an MPC controller, a modification by Yurii Nesterov in [234] transforms the algorithm into the fast gradient method. The modification relaxes the requirement in Equation 2.5, and instead replaces it with an estimate sequence. Using this strategy allows the algorithm to be proved to be globally linearly converging, as well as the ability to calculate *a-priori* the number of solver iterations required to meet a specified tolerance. In practice, as long as the condition number of the quadratic program Hessian is sufficiently small, then tens of iterations are sufficient to find a suitable solution [280].

The fast-gradient method was further explored in [170] with respect to embedded model predictive control. The authors exploited the structure of the optimization problem to efficiently determine the gradient, which was then used to determine the memory and computational effort of the algorithm. An interesting conclusion by the authors was that if the model state-space matrix $\mathbf{A}$ were unstable (any eigenvalue falls outside the unit circle), then the computational effort to solve the optimization problem using the fast-gradient method would increase exponentially with the prediction horizon.

The fast-gradient method is only applicable to input-constrained systems, so its practical use is limited. To solve this, the Alternating Direction Method of Multipliers (ADMM) uses an augmented Lagrangian approach to solve both input and state constrained MPC problems [171]. The method solves the optimization problem by iterating over two sub-problems: An outer augmented problem and an inner problem solved via the fast-gradient method. The authors prove that for asymptotically stable systems the computational effort required to solve the modified method increases only with $O(N^{0.5})$, where N is the control horizon. However this attractive property does not hold with unstable systems which still increase exponentially with the horizon. Further work has also been carried out parallelizing the ADMM algorithm in [172], providing an additional speed-up to this algorithm.

For stable control problems, the fast-gradient method is particularly attractive because it is not only fast but also certifiable for critical control applications. This is based on the deterministic upper bound on time required by the algorithm to converge to a specified tolerance, which allows implementation of a controller in time-critical systems, as described in [112, 279, 281]. However given that the speed advantage inherent in its algorithm is only applicable to stable, well conditioned systems, the fast-gradient method is, like explicit MPC, only suitable for a class of control problems.

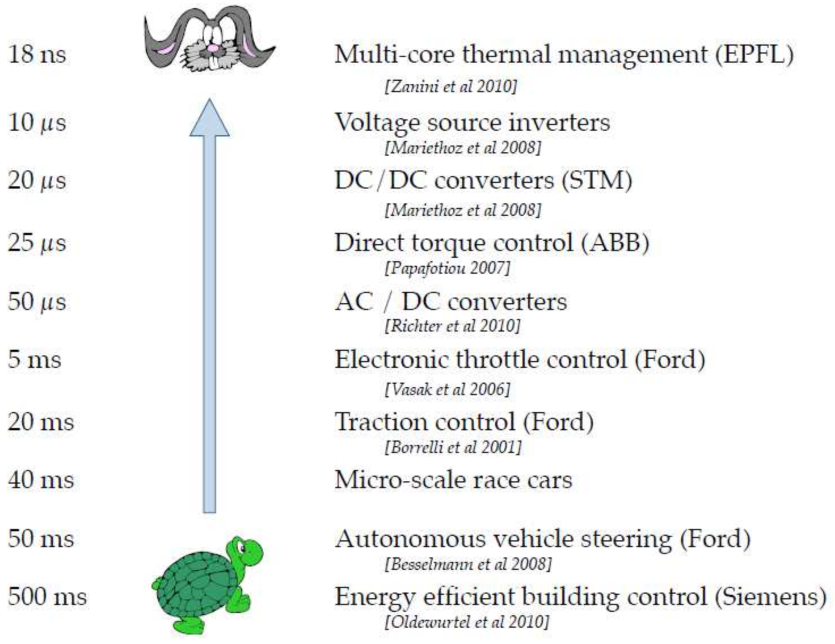
2.2.4 Embedded Model Predictive Control

Embedded MPC aims to implement an MPC-type algorithm on embedded hardware, such as a Digital Signal Processor (DSP), Microcontroller/Microprocessor Unit (MCU) or Field Programmable Gate Array (FPGA). This implementation enables the controller to be applied to low-power, low-cost and mobile systems where a large, power-intensive and expensive desktop computer would not be suitable. The complication here is that the systems where an embedded controller are required are often high-speed, due in part to being smaller in physical size and therefore smaller time constants can be expected.

2.2.4.1 High-Speed Applications

To illustrate the current state-of-the-art embedded MPC, consider the slide in Figure 2.3 taken from [222]. This slide is from a 2013 presentation by Manfred Morari on fast model predictive control, which reports applications from his research centre. The times reported are the implemented sampling rates of various embedded MPC controllers, indicating sampling rates from 2Hz up to 50MHz.

It is worth mentioning that the majority of these applications utilize some form of explicit MPC, which are common in literature and can achieve impressive sampling rates. However, as mentioned, applications of explicit MPC are highly memory intensive, and therefore limited to application on specialized hardware, which is not the intention of this work. In addition to the applications mentioned above, further applications of high-speed MPC include those from Unmanned Aerial Vehicles (UAVs) [238], aerospace [125], spacecraft [195], insulin delivery [320] and a multitude of other applications. A brief survey of embedded MPC technologies is presented below, comparing the main technologies used.



18 ns	Multi-core thermal management (EPFL) [Zanini et al 2010]
10 μs	Voltage source inverters [Mariethoz et al 2008]
20 μs	DC / DC converters (STM) [Mariethoz et al 2008]
25 μs	Direct torque control (ABB) [Papafotiou 2007]
50 μs	AC / DC converters [Richter et al 2010]
5 ms	Electronic throttle control (Ford) [Vasak et al 2006]
20 ms	Traction control (Ford) [Borrelli et al 2001]
40 ms	Micro-scale race cars
50 ms	Autonomous vehicle steering (Ford) [Besselmann et al 2008]
500 ms	Energy efficient building control (Siemens) [Oldewurtel et al 2010]

Figure 2.3: Embedded MPC applications reported by the Automatic Control Laboratory, ETH Zürich (Presentation slide in [222]).

2.2.4.2 Field Programmable Gate Arrays

A field programmable gate array is an Integrated Circuit (IC) which allows the user to program and configure the digital hardware inside the device. The FPGA is an evolution of the family of Programmable Logic Devices (PLD), and nowadays contain millions of programmable discrete logic elements, such as Look Up Tables (LUT), flip flops and electronic gates [176]. Being a hardware circuit rather than software executed instruction by instruction, an FPGA is inherently parallel and this enables it to execute multiple calculation paths simultaneously. Furthermore, most modern FPGAs include a significant amount of block RAM, as well as a number of hardware multipliers, allowing numerical algorithms to be implemented and run.

Traditional applications of FPGAs are within the digital signal processing realm, where a constant supply of information is processed by the device, such as audio or video. This feature of the data allows an FPGA designer to pipeline their algorithm, accelerating execution by providing parallel execution routes. Common applications of FPGAs include medical image processing, computer vision, speech recognition and radio astronomy, all of which have a large set of data available for processing in parallel, as well as algorithms that facilitate the use of parallel programming. However, the use of FPGAs for sequential tasks, such as optimization algorithms is limited, due in part to clocking rates of FPGAs being insignificant, and limited scope for parallelization within the algorithm itself.

A successful implementation of traditional MPC on a modest FPGA was described by Ling et al. in [187], where the authors used the commercial (and expensive) Handel-C compiler to automatically convert a C based MPC algorithm into a Hardware Description Language (HDL) controller. Using their implementation and processor in the loop implementation showed their controller had a sampling interval of 10ms, using a sequential algorithm. The same authors explored parallelization of a Multiply-Accumulate (MAC) function in [186], where two multipliers were run in parallel. Using this implementation, together with a more powerful FPGA, the implementation achieved a sampling interval of 3ms. The authors also went on to compare two quadratic programming methods implemented on an FPGA in [182], noting that the interior point method was reported as superior.

Another implementation of traditional MPC on an FPGA was described by Wills et al. in [333], where a hand-coded VHDL implementation of MPC was reported. This implementation was not only hand-tuned, but also highly parallelized, and resulted in a minimum sampling interval of 30μs when applied to control of a damped resonant system. The authors used an interior-point method implemented completely within an Altera Stratix III, which was clocked at 70MHz and used a custom 23bit floating point number format.

In a further implementation, Jerez et al. implemented a parallelized sparse quadratic programming solver on a Xilinx Virtex 6 FPGA in [155]. The algorithm was hand-coded in VHDL, and utilized a deeply pipelined implementation in order to minimize latency and therefore maximize performance. Using this approach, the paper reports a maximum throughput for a rotating antenna MPC problem of 2.5μs, with a latency of 85μs.

In addition to traditional MPC, explicit MPC has been implemented very successfully on FPGAs, due in part to easy parallelization of the binary tree-search for implementing the control action. An application which targeted an Application Specific Integrated Circuit (ASIC), effectively a high-speed non-reprogrammable FPGA, was described by Johansen et al in [156]. In this work they described a small explicit MPC controller which achieved a 1μs sample interval with a 200MHz clock, representing one of the first 1MHz MPC applications. Since then many authors have adopted FPGAs for implementing explicit MPC controllers, with [6, 298] as two examples.

With respect to first-order methods, FPGAs have also been exploited to implement MHz sampling rates, as described in [153, 154]. In this work, a fixed-point number system is used together with auto-coded parallelized implementations of the fast-gradient and ADMM algorithms.

2.2.4.3 Co-Processor Designs

An alternative approach to the purely digital hardware approach is to split the computational load and utilize a co-processor to accelerate computationally intensive sections, while a traditional sequential processor handles control logic. This approach is taken in [320], where the authors propose a System-on-a-Chip (SOC) MPC controller, which utilizes an application specific matrix-coprocessor and limited resource host controller. The matrix-processor is designed with a 16bit logarithmic number system Arithmetic Logic Unit (ALU), which is tailored for solving the matrix problems which arise when solving the online optimization problem.

Koh applied an FPGA soft-processor design in his Master's work [173], implementing both the soft-processor and co-processor within the FPGA logic. This strategy is quite powerful because both the design of the host processor, and the co-processor can be completely customized. It is however much more resource intensive, as the soft-processor typically consumes a large number of logic elements and available block RAM.

2.2.4.4 Microprocessor Implementations

The final area reviewed is MPC implemented directly on a traditional microcontroller/microprocessor or Digital Signal Processor (DSP). This approach has several advantages, the most significant being that the control algorithm requires little modification from the development algorithm on a desktop computer. This is opposed to hardware designs, such as FPGAs or co-processors, which require substantial redevelopment into a hardware description language.

As described later in Section 4.2.1, microcontrollers, microprocessors and DSPs are all primarily sequential devices, processing software instructions one-by-one. A microcontroller is the simplest device of the three, because it typically includes all peripherals (timers, interface logic, memory controllers) on-chip, while a microprocessor, such as the one in a desktop PC, requires external components such as RAM, a hard disk drive (HDD), external BIOS IC and so forth. This allows the microprocessor to pack more computing power into its integrated circuit, at the cost of requiring multiple external components. The advantage is however that modern microprocessors may contain 2 or more computing cores, and thus allow parallelized execution.

The DSP is in a different category again, because it is designed for high-throughput numerical calculations, such as those now targeted by FPGAs. It is however primarily a sequential device, although many common numerical calculations such as multiply-accumulate may be pipelined and therefore parallelized. It remains as a specialized device that requires substantial algorithm redevelopment in order to leverage the benefits of its architecture, although not to the same degree as an FPGA (unless written in assembler).

An early implementation of MPC which utilized a modified Newton's method for solving the online optimization problem was described by Bleris and Kothare in [38]. In this work they used a phyCORE development board which included a 32bit Motorola MPC555 microcontroller, and also, significantly, a built-in hardware 64bit floating point unit. This platform allowed the authors to implement their controller in a processor-in-the-loop environment that would leverage the full precision typical on a PC. Numerical roundoff was therefore not significant within this implementation. This work resulted in a sampling rate that approached 1Hz once the control horizon exceeded 5 samples.

A DSP implementation of traditional MPC is described in [332], where the authors apply MPC to an active noise and vibration control system. In this work the controller is implemented on an Analog Devices ADSP-21262 200MHz 32bit floating point DSP, noting the algorithm is completely written in assembler. This is a significant undertaking and limits the implementation to only a small family of hardware, however the performance payoff is that the sampling interval ranges from 30μs to 140μs for horizons from 4 to 12.

In later work Zometa et al. [348] applied linear MPC using the fast gradient method to control a two-wheeled robot using the LEGO-NXT hardware. The controller is implemented on the NXT controller, which utilizes a 48MHz 32bit Atmel AT91SAM microcontroller, noting however this processor does not have a hardware floating point unit. The authors implemented their controller with a control horizon of 20 to an 8 state, 2 input and 3 output model of the robot and achieved a sampling interval of 4ms.

A recent implementation from ETH Zürich is the FORCES framework [83] which, as described in the next subsection and profiled later in Section 4.4.5, is an auto-code generator for efficient interior-point methods, including second-order cone problems. Targeted primarily at MPC problems, such as in [84], the framework can also generate solvers for general optimization problems. In [84] the authors describe two embedded implementations of MPC, the first on a 1.6GHz Intel ATOM Z530 and the second on a 500MHz ARM Cortex A8, noting both are at the high end of embedded processor technology. Using the FORCES framework, an MPC controller was designed to solve the chain of masses problem [327] with varying numbers of masses. On the Intel processor, solution times ranged from 2ms to 90ms while on the ARM processor the same problems solved in 60ms to 3210ms.

2.2.4.5 Auto-Coded MPC

When deploying an MPC controller to an embedded target some authors have opted to develop an automatic coding framework which reduces the otherwise substantial manual implementation work required. The framework is tasked with converting a high-level description of the MPC controller into a low-level form, typically ANSI C, suitable for implementation within common embedded hardware. The benefits of automatic coding of MPC controllers include rapid re-tuning, less coding errors (once the framework is proven) and typically faster code, because techniques such as loop-unrolling are much easier implemented when automated.

An early implementation of auto-coded MPC is described by Richards et al. in [278], where the authors used Simulink to implement the complete controller, including an interior point solver. The advantage of using Simulink over MATLAB was that by using the MathWork's Real-Time Workshop, the Simulink model could be automatically converted to embedded C-code, including code optimizations targeted at specific processor architecture. The focus in this work was on flight control where a verifiable control solution is required. By utilizing Simulink together with the rigorous code generation technology provided by the MathWorks, steps were made towards this goal.

For the generation of explicit MPC controllers the MATLAB Multiparametric (MPT) Toolbox from ETH Zürich is the most comprehensive [127]. Now in its third major release, the Toolbox has been in development for over a decade and includes functionality for describing, tuning and simulating explicit MPC controllers, together with automatic code generation when using the MATLAB Embedded Coder. Further developments of the MPT Toolbox for natively generating C explicit controllers, as well as Programmable Logic Controller (PLC) forms are described in [178].

One of the first tools for automatically generating convex optimization solvers was CVXGEN, a code generator for convex optimization [210], based on the CVX framework described in [116]. While this tool was not designed directly for MPC, the MPC optimization problem can be succinctly described in the CVXGEN code generator, and therefore suitable solvers can be generated. The tool is predominantly web-based, however the generated code also comes with a MATLAB interface which allows the resulting solver to be called directly from MATLAB. The authors report speedups of up to 10,000 times over traditional solvers when using automatically generated solvers, which is achieved by fully unrolling the algorithm, i.e. there are no loops and the code is completely 'flat'. The downside to this approach, however, is that the resulting code is very large, and scales poorly with problem size [84].

To automatically generate first-order solvers for parametric convex programs, FiOrdOs is a MATLAB Toolbox developed by Fabian Ullmann at ETH Zürich [312]. As with CVXGEN, the tool is not targeted at MPC controllers directly, rather its primary design focus is on generating code for embedded platforms. The toolbox allows generation of both the gradient and fast-gradient solvers, as well as Lagrange relaxation solvers for inequality or equality constrained problems.

For generating MPC controllers which utilize a first-order solver, Zometa et al. described the Python framework μAO-MPC in [349]. The tool is designed to generate embedded MPC controllers using either Nesterov's method or an augmented Lagrangian approach for solving the online optimization problem, and targets memory and performance constrained embedded systems. Using their framework when applied to the classic Cessna Citation MPC problem [192], the authors report a required sample interval of 9ms to achieve a relatively low accuracy (≈ 0.1) on a 168MHz ARM Cortex-M4 with floating point unit.

As described in the previous section, the FORCES framework also allows the generation of embedded MPC controllers that retain the traditional interior-point solver [82]. For a comparison of performance between FORCES and the method proposed in this work, see Section 4.4.5.

Within the field of nonlinear MPC (NMPC), as described in the next section, the auto-code generator ACADO [132] is a MATLAB and C++ framework for generating embeddable nonlinear optimal controllers, including MPC controllers. The authors report on an implementation of an NMPC controller using ACADO to a continuously stirred reactor model on a desktop PC in [133], indicating that sampling rates of less than 1ms are possible. The framework utilizes either CVXGEN, or qpOASES for solving the online optimization problem, which is used within a sequential quadratic program approach to solve the resulting NLP.

2.2.5 Nonlinear MPC

The field of nonlinear MPC is beyond the scope of this work, however a brief survey is presented for completeness. Nonlinear MPC allows a nonlinear process model to be used within the controller which more accurately represents process dynamics. This is opposed to a linear dynamic model which has been used by all methods described so far within this review. The issues with using a nonlinear model are twofold; Firstly, forming the process prediction requires the use of an integrator (whether fixed or variable step), and secondly, the resulting online optimization problem is a general nonlinear problem, meaning a full NLP solver is required. Both of these problems typically restrict NMPC applications to slow chemical processes, as described in [261].

As described by Allgöwer et al. in [11], many process systems are inherently nonlinear and when coupled with tighter operating constraints, linear models with linear controllers are not sufficient for adequate control performance. To overcome this problem, several approaches to implement efficient nonlinear MPC have been proposed, such as described in [10, 36, 215, 223, 272]. Typical methods discretize the ordinary differential equations (or differential algebraic equations) describing the process model via direct collocation or collocation based on a polynomial form. This process generates an algebraic representation suitable for solving directly with a nonlinear equation solver. Alternative methods use single or multiple shooting to solve the dynamic optimization problem, or recently, Real-Time Iteration (RTI), such as within the ACADO Toolkit, which is a Newton-type framework that reduces the problem to a single quadratic program [133].

The inherent complexity of solving a large dynamic optimization problems renders NMPC beyond the scope of implementation on the modest embedded hardware we are considering in this work, however techniques such as RTI may provide a suitable platform for future work.

2.2.6 MPC Summary

Within the embedded MPC field it is clear that there has been a surge in recent activity, as demonstrated by the multitude of auto-coding frameworks developed and papers published in the past two years. Three methods of solving the optimization problem resulting from a linear MPC controller have been identified as offline, via explicit MPC, online, via first-order methods and thirdly a traditional quadratic programming solver.

It is now commonly accepted that the explicit MPC method is limited to small-scale systems based on its memory demand, which may grow exponentially in the number of states, inputs or prediction horizon [84, 171, 280]. As the embedded systems targeted for this work will be memory constrained (an upper limit of 256KB, see Section 4.2), and also that the systems of interest will require horizons greater than 3 or 4 samples, explicit MPC will not be pursued within this work, because the controllers simply will not fit for the problems of interest. It is however acknowledged that impressive sampling rates exceeding 1MHz are achievable using this approach, but it is also noted that sample rate alone is not the only measure of an efficient controller.

With regard to first-order methods, while attractive because they are very simple and therefore computationally efficient, as described in [170], it is noted that the approach does not lend itself well to unstable systems, and requires a large increase in computational complexity as the horizons are increased. As this research is deliberately not focused on any particular class of system (so long as they are linear), the optimization solver must be efficient and implementable regardless of the stability of the system to be controlled.

For this reason the traditional approach of implementing a quadratic programming solver to solve the online optimization problem will be pursued. This is identified as the most computationally expensive method, but also as the most general, allowing application to a range of control problems both large and small. Our belief is that optimality does not have to be sacrificed in order to still achieve high-speed MPC, nor for it to be implemented on an embedded platform. Therefore this work will retain the online optimization step using a quadratic program, as was done within QDMC, but as stated in the introduction, tailor the solver for the problems of interest. This approach is reinforced by two recent competitive frameworks for generating high-speed interior-point quadratic programming solvers, CVXGEN and FORCES, both of which have garnered significant industrial and academic interest. Furthermore, as shown in Section 4.4.5, there is still room for improvement on the techniques described in both of these packages, and especially for the problems of interest within this work.

A comparison of active-set versus interior-point algorithms for solving quadratic programs is presented later in Sections 3.3.1 and 3.4.

The next section will survey the literature around the second industrial optimization problem investigated within this work, namely the optimization of large-scale industrial steam utility systems.

2.3 Utility System Optimization

Industrial utility systems such as steam or electricity form an integral and economically important part of most processing plants, supplying the demands for heat and power of the process undertaken [314]. Utility energy is the largest managed operating cost for the hydrocarbon process industry [92], yet from both a control and operational point of view, it typically takes a back seat to production [89].

Within the utility system, cogeneration is a typical modern feature where steam turbines and/or gas turbines can be used to provide both mechanical and electrical power, either for the plant itself or if economically viable, to sell energy back to the electrical grid. However the cost of fuel to generate steam or run a gas turbine can be significant (hundreds of millions of dollars per year as in our recent industrial work), as can be purchasing power from the local supplier [69, 315].

Searching for the most efficient way to operate a utility system given varying electricity and fuel prices, changes in process demands and equipment availability is a complex problem [80, 90], and the optimal selection of equipment is not always directly obvious. This provides an excellent opportunity for a rigorous utility system model to be built which can not only provide run time information but also forms the basis for optimization.

Optimization with a cost function including real economics (power and fuel tariffs, power and fuel costs, etc) allows an advanced solver to find the operating point which can reduce running costs of the utility system by millions of dollars per year. Examples are detailed in [80, 89, 90, 92, 94] which show industrial applications of a steam utility optimization package which typically result in savings of 1-5% of the total annual utility cost.

2.3.1 Pinch Technology

Utility system modelling and optimization has been explored by a number of authors with perhaps the most well known being Pinch Technology by Linhoff [188]. Still in use today as the core technology of KBC's SuperTarget software [163], Pinch Technology was one of the first commercially viable technologies to focus primarily on process utility systems and the optimization of these systems. This strategy allows an engineer to graphically optimize a heat exchanger network using two curves, representing the hot and cold sides of the network. Where the curves touch is known as the 'Pinch' point, and represents the optimum matching of heat exchanger steam and process flows. Although Pinch is still a valid technology, its use as a tool for optimizing an entire process utility system (i.e. not just the heat exchangers) may not determine the true optimum because it will not take into account constraints or variable efficiencies [94].

2.3.2 Total Site Analysis

Total Site Analysis (TSA) was an extension to Pinch developed by Dhole and Linhoff [79]. By superimposing the steam levels (HP, MP, LP, etc) onto curves of the total site energy sources and sinks (referred to as Total Site Profiles), the optimum selection of flows between each pressure level and therefore throughout the steam turbine network could be established. This algorithm was further developed in [212] where Mavromatis introduced the challenge of the optimal selection of steam turbines (multiple small, one large, dual stage, etc) to meet the design requirement from the TSA. This work introduced the varying efficiency problem inherent in steam turbines: the efficiency of a steam turbine varies nonlinearly with power production, so that the calculation of the optimum flows must take this problem into account. However, as discussed below, this approach does not include all the basic elements of an industrial process utility system, thus does not represent a complete optimization of a utility system.

2.3.3 R Curve

An alternative method from the early 1980s is the R-Curve proposed by Kenney [166]. This was another graphical technique but which looked at the 'Cogeneration Efficiency' of a process utility system. The resulting curve is known as the 'fuel utilization curve' and was a plot of cogeneration efficiency versus the process utility system power-to-heat ratio. Although the results of the analysis are obvious (installation of gas turbines and heat recovery steam generators improve fuel efficiency), the graph provided a quantitative measure of the varying efficiency.

The R-Curve was further explored in [169] incorporating the TSA to develop an 'ideal power generation R-Curve'. This used TSA to develop the optimum selection of steam level flows and turbine sizes, and the R-Curve to maximise the fuel efficiency given these flow and shaft work requirements. However, as reviewed in [314], this research did not include basic elements of an industrial process utility system, such as process steam generation or power import/export.

2.3.4 The Chemical Process Simulator

With respect to optimization, most modern process simulators now include the facility to optimize the model which provides an opportunity to be built on for this work. The process simulator has been an industrial technology tool for modelling chemical processes for over fifty years [283, 302, 303]. Benefits include better process insight and scenario modelling for retrofit projects [291], as well as forming the basis for optimal operation via optimization.

The use of chemical process simulators for the modelling of process utility systems allows the use of rigorous thermodynamics to be applied to the unit operation models, reducing the error between actual plant and model results. Several authors have published work on modelling process utility systems within a process simulator [85, 237], however as detailed in our work [71], as well as in [89], utility models do not combine naturally with process simulators. This is due primarily to two reasons: Zero flows can cause the thermodynamic engine to fail to solve and secondly the typical PFD structure of the simulator is too restrictive for the flexibly operated process utility system. A utility system often contains numerous logical operations for the direction of steam flows based on operating conditions (maximum and minimum flows and current demand), which do not have a typical equivalent in a chemical process model, because this is often designed for a single process flow.

Within the process simulators there are two main types: Steady-state and dynamic. Steady-state simulators do not model changes with respect to time, where typically all models are algebraic relationships and represent the final settled value (steady-state value) for a given set of operating conditions. Dynamic simulators in contrast contain ordinary differential equations (ODEs) and differential algebraic equations (DAEs), and can thus model the dynamic response of a plant with respect to time. Dynamic simulators are typically used for the design of process control systems [12], operating training [53] and process start up studies and retrofit investigation [180, 228] which are all considerably more complex.

This research will consider steady-state models due to the focus on optimization of plant economics where the steady-state values represent the most significant cost, versus the 'getting there' cost. This also significantly simplifies the underlying computation and modelling and therefore forms a realistic framework for real-time optimization. A number of steady-state modelling packages exist using different solvers and algorithms and these are detailed in the following subsections.

2.3.4.1 Sequential Modular

One of the earliest approaches to solving process flowsheets was a sequential approach [35, 285], in which the solve order of the unit operations is fixed by the natural flow of the process material, i.e. from the feed streams into the final product out. This strategy was favoured by industry [35] because it was widely accepted to solve the system in the direction of the actual chemical or product flow, with a single flow direction standard in most chemical processes.

The term modular reflects the method in which the equations of the system are solved whereby each unit operation is treated as a standalone module and solved individually in the designated sequence. This reduces the number of initialization variables required for each unit operation, enabling the strategy to be quite robust. The modular strategy also lends itself better to PFD based design.

In order for this system to work one must specify the parameters of the input streams, so that the information can propagate across the flowsheet to the end product. Recycles, or loops, are handled by successively iterating the solving of the unit operations in the loop until the convergence criteria is met. A common term used was to 'tear a break' in a particular stream, so that one part of the feedback loop would be broken and used as the testing / convergence point.

Examples of commercial sequential solver packages include Aspen Plus [18] and COCO [152].

2.3.4.2 Non-Sequential Modular

A later strategy which formed the algorithm of the highly successful HYSYS process simulation was the non-sequential algorithm [33-35]. This has the advantage of being able to propagate information both backwards and forwards, so that specifications in the middle of the system can cause unit operations to solve both before and after the specification.

With respect to process utility systems, the non-sequential algorithm allows boilers, heat exchangers and turbines to be specified based on incoming flows, outgoing flows, duties or temperatures, and still solve. This enables the user to enter the specifications of their system based on their process measurements, without being restricted to preset solve orders such as in the sequential system. This further increases the robustness of the modular solver and enables an interactive process simulation approach, as detailed in [226].

As this solver is still a modular solver, recycles are solved using a stream tear and then iterated until it converges, in a similar manner to the sequential solver.

Examples include HYSYS [19] and VMGSim [318].

2.3.4.3 Equation Based / Global Approach

In direct competition with the sequential methods of the late 1970s and early 1980s, the equation based solvers approached solving the flowsheet as a collection of linear, bilinear and nonlinear equations [293]. Interestingly, this strategy was typically restricted to the academic domain at this time [35], which explains the dominant equation based literature versus the sparse sequential literature.

This strategy, as described later by Barton [28], allows the use of analytical gradients and equation substitution to simplify and speed up the solution of a group of unit operations. Advanced numerical linear algebra algorithms were applied to the sparse gradient and incident matrices in order to solve these systems much faster. These notably also included recycle as part of the system of equations so that it could reduce or even remove the need to iterate the system to converge.

A number of early software packages include Quasilin [135] developed by the University of Cambridge, as well as SPEEDUP [240, 247], developed at the Imperial College. These packages were originally text only and required specialised mainframe computers in order to run, which limited their use. This was in part due to the large memory requirement in order to solve these problems, but also to the architectures for which the source code was written [226].

However, by the 1990s and with the advent of the personal computer, industrial and commercial users now had the processing power and memory to solve these problems [227, 303], and software would be developed that was suitable for a process engineer. Equation based process simulators that resulted from this early work include Chemasim [124].

As a side note, equation based solvers were not limited to chemical processes, and several institutions went on to write general model equation solvers. These advanced packages could also model and solve electrical, mechanical and other physical systems. Three of the main software packages in this area are ASCEND [252] and Modelica [211], as well as the well known Simulink from the Mathworks [208].

2.3.5 Commercial Utility System Tools

Several commercial packages exist for specifically modelling process utility systems (as opposed to chemical/petrochemical processes) which are built on a range of solver algorithms and user interface strategies. In order to identify a research area that has not been covered commercially, the four main utility simulation packages are reviewed in the following subsections.

2.3.5.1 ProSteam

ProSteam [89, 162] is a commercial utility modelling package by KBC, which is effectively an add-in for Microsoft Excel. It utilizes the built in Excel Solver for converging the utility model, which iterates across the sheet using the built in root solver.

Functions are inserted with the assistance of a function wizard, and calculation results connected via cell references. Consequently, the main issue with this arrangement is the amount of manual work required by the user. Not only must the user maintain their own mass and energy balances, but manual recycle points must be identified and inserted. This cell based approach can therefore allow for many errors to go unnoticed within the model.

2.3.5.2 Ariane

Ariane [258, 259] is a utility modelling and optimization package by ProSim, which is a stand alone application. Targeted specifically at power plants which generate steam, electricity and heat, it allows the user to create a graphical model of their system, enter the specifications and constraints, and then subject it to global optimization procedure [29]. From limited information on the product available and from the examples given it does not appear that the system can model closed loop (condensate recovery) process utility systems, which would require a more advanced recycle solver. This can allow for an energy balance to be broken around the deaerator which is in contrast to basic modelling fundamentals, and may result in unrealistic 'optimal' operating points.

2.3.5.3 iCON Utility Optimizer

iCON Utility Optimizer (iUO) was the add on package developed by myself and our research team for PETRONAS [71]. Built on VMGSim [318] a chemical process simulator, it enables a user to create a graphical model of their system, enter the specifications and solve it for a range of operating conditions. The flowsheet is created using a Visio API interface with the unit operations added as Visio shapes. As of 2010 the software did not provide a reasonable optimization framework, but formed the basis of the modelling for this work.

2.3.5.4 Aspen Plus, CHEMCAD, PetroSIM, HYSYS and Other Process Simulators

As with all mature process simulators, process utility systems can be built from fundamental unit operations (mixers, expanders, flash vessels, etc). This is based on our experience constructing iUO for VMGSim, which is specifically a process simulator. There are however a number of problems when building utility models in a process simulator, as listed below and described later in Section 5.2.3:

- Discrete flows can cause convergence issues within the sheet solver.

- Steam header balancing requires external logic to calculate feed and vent mass flows.
- Maximum and minimum flows on equipment are effectively saturation constraints.
- Multi-flow direction of steam is decided by current operating conditions which vary widely.
- Fuel balance and power balance affect gas turbine, boiler and turbo generator operation.

Most importantly however, from our experience in observing process engineers build utility models in process simulators the models are designed for one particular operating point only. This means they are heavily over specified or are solved for known plant operating conditions using numerical solvers. This results in these models not being suitable for optimization because there are not enough free variables to realistically move the operating point of the system. This is discussed further in our paper [71].

2.3.5.5 Commercial Package Summary

Table 2.1 summarizes the features within the packages reviewed above. The features highlighted are as follows:

Live PFD Graphics connect directly to unit operations allowing simple entry of data.

Steam Unit Operations Dedicated unit operations for steam utility modelling.

Recycle Solver Ability to model closed loop steam systems.

Model Constraints Constraints are included as part of the optimization and solver process.

Integer Optimization Ability to optimize equipment on and off.

2.3.6 Design versus Operational Optimization

Optimization of process utility systems is split into two categories, synthesis (or design) and operational. The focus of this research is on operational optimization

Table 2.1: Characteristics of commercially available utility modelling software.

Package	Live PFD	Steam Unit Ops	Recycle Solver	Constraints	Integer
ProSteam		√	√		
Ariane	√	√		√	
iUO	√	√	√		
Process Sim	√		√		

because we believe this is a more industrially significant problem, especially for our industrial clients, and one which is applicable to the majority of process plant operators around the world. However a brief review of synthesis optimization is presented for completion.

2.3.6.1 Unit Operations of Key Equipment

A common theme in all process utility system optimization papers is the discussion of building models for the three main utility operations: Boilers, gas turbines and steam turbines. While conventional models exist for these operations, as described by Aguilar in [4], they are too simple and as a result, calculate performance either as a function of load (fixed unit size) or size (full load only). Several authors have addressed this problem proposing nonlinear models based on regressions of plant measurements, such as in [197, 255, 256, 294, 295, 315]. Although this provides a more accurate estimation of the physical plant equipment performance, modelling of the system becomes a specialist's job in order to generate the regressions, or risk using regressions from other authors' versions of plant equipment.

A problem introduced by using nonlinear models is that the problem now requires a full nonlinear program (versus a much simpler linear program) to solve. This adds complexity to the optimizer resulting in less robust and longer computation time solving. Aguilar [4] addressed this issue in a recent paper proposing linear models for all three unit operations by using an input / output approach, rather than modelling the internal thermodynamics and physical structure of the unit operations. Regressions were performed on various sizes and types of plant equipment in order to generate linear relationships between, for example, fuel duty and electric power output for a gas turbine. As described later in Section 6.3.5, a linear approximation fixes the header enthalpies which breaks the energy balance of the model and results in a poor approximation for part-load modelling.

2.3.6.2 Synthesis Optimization

Synthesis refers to the design stage or 'green fields' whereby the process utility system is not yet built and the optimization refers to the optimal design of a new steam process utility system. This is typically the decision of where to place the steam turbines between pressure levels in order to maximise shaft work, or the inclusion of cogeneration and its effect on the overall efficiency of the system, such as the R-Curve method discussed earlier.

Notable early work by Grossman [117, 242] described the selection and placement of equipment as a mixed integer linear programming (MILP) problem, although this assumed all equipment ran at full load and one operating scenario was evaluated. This work was further refined by Bruno and Grossman in [46] to include nonlinear models and therefore posed the problem as an MINLP problem, based on work with optimization based synthesis in [118] and [119].

While the work by Grossman resulted in substantial economic improvements, it was noted by other researchers that the effect of multiple operating scenarios could influence the resulting optimal configuration of plant equipment, and this formed an important step in the optimization of a process utility system. Two early papers, one using an MILP [134] and the other using simulated annealing [193] explored varying operating conditions and operating demands, respectively, to determine a flexible optimum.

Later work by Marechal [198, 199] introduced a multi-period MILP formulation which allowed the different operating scenarios to be assigned different time lengths, thereby reflecting the actual variation of operation more accurately. Recent work on multi-period optimization of process utility systems is in part two of Aguilar's work [5] and describes a framework for synthesis, retrofit and operational optimization.

2.3.6.3 Operational Optimization

In contrast to synthesis, operational optimization is the optimization of an existing plant whereby pressure levels, flows and temperatures, as well as equipment power output and operating state can be varied in order to obtain the optimum operating parameters. The operational optimization problem is well established with early work by Nath [231] specifically optimizing process utility systems in the mid 1980s. Several authors around this time including Nath developed specific Utility Optimizers such as STEAMPOP [232], UPLAN [241] and others [313] based on linear and mixed integer linear programs. Variations on the standard degrees of freedom included modelling transition costs due to bringing equipment online and shutting it down were investigated in [149]. This required penalty functions associated with the change of a binary variable and discouraged the optimizer switching on a boiler or turbine unless it was actually economically sensible to do so.

Operational optimization was also explored by Varbanov in [315] using a successive MILP approach. A simplified, linear model was iteratively subjected to an MILP optimizer with each solution being tested for accuracy within a rigorous second stage nonlinear simulation. When the solutions of the two simulations converged, the function would exit with the resulting nonlinear optimum.

When considering retrofit optimization, current research requires the user to determine a 'superstructure' for the addition or upgrading of plant equipment. The superstructure is a predefined selection of available investment equipment which could be added to the process utility system, and its size or loads may also be predetermined. The optimizer must now not only decide to add which equipment and where to place it, such as in [17, 123], but if allowed, must also determine the optimum size of this equipment, as in [5]. This adds many more variables to the optimization problem, a number of which are binary. As detailed in [5], a retrofit optimization can take four orders of magnitude longer to solve than an operational optimization, even with a state-of-the-art solver.

2.3.6.4 Scheduling Optimization

A final area of operational optimization is that of planning and scheduling utility operations based on the process, the maintenance and the time requirements. This will not be covered here because it is considered a separate field of research and would increase the scope of the research beyond that achievable within this time frame. For the reader's reference [3, 149, 168, 299, 300, 301] provide a summary of recent work in process utility system scheduling.

2.3.7 Utility System Optimization Summary

The review of the utility system optimization literature has detailed the industrial significance of this optimization problem, given the wide ranging scope for application from the process and manufacturing industry to power generation. As detailed by Fernandez-Polanco et al. in [92], the utility system represents the largest managed cost for a typical petrochemical refinery, yet when operational optimization is applied, requires little to no capital investment and savings of 1-5% are common [90].

Several techniques have been reviewed for optimizing both the operational and design of a utility system, including early graphical techniques such as Pinch Technology and the R-Curve. While Pinch technology is still widely in use today within KBC's supertarget software (as well as within other packages), its application is typically limited to heat-exchanger design and layout, matching hot and cold streams, or when designing distillation columns. For this reason, it is not suitable as a technology for optimizing an entire utility system. Similarly, the R-Curve concept is a simple approximation of the operation of a utility system, and does not take into account sufficient detail to be used.

The application of a chemical process simulator for modelling and optimizing utility systems was investigated, however based on our industrial experience, this was found to be problematic (see Section 5.2.3 for more details). However the design of the simulator, including techniques such as sequential modular, non-sequential modular and equation based provide a basis for the methods to approach modelling a utility system within this work. As will be demonstrated later in Section 6.3.3, both a sequential modular approach and an equation based strategy will be used within this work, where the sequential modular strategy is used to initialize an equation based model, based on findings within this review.

A review of commercially available tools was undertaken to see what technology was available for modelling and optimizing utility systems. From this survey, it was clear that the ability to apply mixed integer optimization was a major gap within commercial tools, thus limiting the benefit of operational optimization.

Within academic optimization studies, operational and synthesis optimization is typically posed as a large-scale mixed integer linear program, which as shown later in Section 6.3.5, can result in unrealistic header enthalpies and possible damage to the utility system, if the solution were to be implemented in practice. Therefore authors had either opted for a successive MILP approach, such as in [315], or the full mixed integer nonlinear model as in [46], but this required implementation on a high-performance server.

What we believe is overlooked, is considerable speed and robustness improvements can be realised from improved equipment models, together with a novel formulation of the complete system. By approximating thermodynamically rigorous unit operation models with linear or bilinear expressions, the models are both faster and more suitable for optimization. Furthermore, the models can be developed to capture the response of the model over the expected operational area, thus increasing the accuracy of the approximation. This approach is made possible because we are focusing on operational optimization which allows us to define offline the typical operating region, based on base-case data. Furthermore, by tailoring the construction of the model to a form that is most compatible with the optimizer, robust solutions should be obtained in just a few seconds, even when optimizing large industrial-scale utility systems. This builds on the approach taken by Bruno et al, further developing the unit operation models and creating a more flexible method of describing and optimizing these systems, which provides more accurate and physically realisable solutions faster.

2.4 A Review of Shortcomings in the Literature

This chapter has provided an overview of industrial optimization, together with a brief summary of major historical events which lead to the formation of the discipline of operations research. Building on this overview, a review of the literature surrounding the two core areas of this research has been undertaken, embedded model predictive control and steam utility system optimization.

Within the field of embedded MPC, the two core areas of research have been identified as offline optimization via explicit MPC, and online optimization via hardware and/or algorithmic enhancements. As stated in the literature, the main issue with explicit MPC is the large memory requirement to store the parametric representation of the controller, thus limiting the algorithm to small problems not of interest within this work. Within hardware enhancements, FPGAs have been identified as the preferred implementation candidate given their inherent parallelism. However as noted further on in Section A.4.1, there are a number of drawbacks to FPGAs, most significantly of which is the extended compilation time required for re-tuning, but also the much longer development time. For these reasons a hybrid DSP/MCU has been chosen as the hardware target within this work, as described later in Section 4.2.1. With regards to algorithm advances, both interior-point and the fast-gradient method are identified as the main variants, with both algorithm being deployed within MCUs, DSPs and FPGAs. As stated in the literature, the fast-gradient method is more efficient than a standard interior-point solver, but suffers an exponential increase in computational load with unstable or poorly conditioned problems. Based on this, the decision to pursue an interior-point solver has been made, with a mixed hand-optimized/auto-coded implementation tailored to solve the problems resulting from an MPC formulation.

The second area considered is the field of steam utility system optimization, specifically looking at operational optimization. As identified within the existing literature, existing research has focused predominately on the synthesis optimization problem, and therefore with much larger and more complex systems, the optimization problem is often approximated using linear relationships. Alternatively, early work considered graphical techniques such as Pinch and the R-Curve, which aimed to optimize sections of a utility system, such as heat exchanger or steam turbine networks. Within this work we will consider the operational optimization problem of an entire utility system, which allows us to define the operating region of interest for our system, and build purpose-built nonlinear models tailored for the optimizer. This overcomes the pitfalls of linear approximations seen in literature, ensuring energy balances around steam headers and preserved, and resulting in a model which solves for a sensible and implementable optimum. Furthermore, by building on industrial modelling experience detailed in Section 5.2, new unit operation models which take into account typical operating parameters will be developed, leveraging regressions within modelling literature to develop part-load expressions.

Within both areas, the literature has shown that predominantly existing industrial optimization problems have been posed and solved in isolation, without the use of a global framework or common methodology. This feature limits the applicability of the research to the academics who posed the problem, and therefore the industrial significance. This work aims to provide both academic and industrially significant results by providing a high level framework for posing real industrial problems, while tailoring the model and optimizer for robust, high speed solutions, using ideas developed within the academic contribution of this work.

Chapter 3

Quadratic Programming for Model Predictive Control

At the core of a standard, linear constrained MPC controller is a Quadratic Programming (QP) solver. The QP solver is tasked with solving the constrained optimization problem that results at each sample, in order to calculate the control move(s) for the next sample. This chapter shows that solving the quadratic program at each sample forms the bulk of the computation required by the control algorithm, and therefore it targets the QP solver as an opportunity to improve existing work, both in terms of sample rate, and in terms of memory usage. By tailoring the QP solver for the class of quadratic programs that result from a linear MPC controller, a suite of heuristics and algorithm modifications are proposed which accelerate convergence, reduce memory requirements and provide an opportunity to embed the solver within an embedded platform. A comprehensive analysis of new and existing algorithm modifications is presented which shows the benefits of the proposed strategy against traditional QP solvers, as well as industry standard QP solvers, for the optimization problems of interest in this work. Results presented show that the `quad_mehrotra` algorithm presented obtains best in class performance against two literature MPC case studies.

This chapter begins with an introduction to the model predictive control algorithm used within this work, namely the finite-horizon algorithm, including the derivation of the resulting quadratic cost function with linear constraints. A survey illustrates issues with existing quadratic programming solvers for both the problems of interest, as well as deployment to an embedded computing platform, before introducing the infeasible interior point method. The algorithm is detailed, including a range of modifications for accelerating convergence and reducing memory consumption, before being benchmarked against two existing literature studies. Finally, the jMPC Toolbox is introduced, which provides a framework for the design, tuning, simulating and testing of linear MPC controllers with the new quadratic programming solvers developed in this chapter.

3.1 Introduction

The standard formulation of the linear MPC control law involves minimizing a quadratic cost function of the form

$$\mathcal{J} = \sum_{j=1}^{N_p} \|\gamma_j \left(\hat{\mathbf{y}}_j - \mathbf{y}_j^\star\right)\|^2 + \sum_{j=1}^{N_c} \|\lambda_j \Delta \mathbf{u}_j\|^2 \tag{3.1}$$

where the decision variables, $\boldsymbol{\Delta}\mathbf{u}$, are the control inputs over an immediate future control horizon, N_c, and are chosen so that the weighted sum of the squared deviations between the predicted output, $\hat{\mathbf{y}}$ and setpoint $\mathbf{y}^\star$ and control moves is minimized over the prediction horizon, N_p. In addition, the system may be constrained so that the control inputs, $\boldsymbol{\Delta}\mathbf{u}$, $\mathbf{u}$, and plant output, $\hat{\mathbf{y}}$, must satisfy some predetermined system limits.

Given the problem definition above, the MPC problem can be constructed and solved as a quadratic program, a superset of Linear Programming (LP) and a subset of Nonlinear Programming (NLP), where the objective function may contain quadratic, bilinear and linear operations only, and which is subject to linear constraints. The term 'linear MPC' refers to the fact that the dynamic model used to predict the system's response is linear, such as a transfer function or state-space model.

3.2 Linear Model Predictive Control

As detailed in Section 2.2, the traditional linear MPC algorithm utilizes a Linear Time Invariant (LTI) system model, such as a transfer-function, zero-pole-gain or state-space description of the plant to be controlled. Using this linear model, coupled with a discrete implementation, allows the linear MPC algorithm to avoid using a generic numerical integration scheme intended for Ordinary Differential Equations (ODEs), and provides a much simpler mechanism for predicting the system response.

3.2.1 Algorithm Overview

A distinguishing feature of MPC is the principle of a receding horizon. At each sampling instant the MPC controller solves a finite horizon control problem for a sequence of optimal control inputs, of which only the first input is applied. This procedure is repeated at every subsequent sample, and the prediction horizon shifted one step further into the future. This constantly receding prediction horizon is where MPC gets the well known name of receding horizon control, and is quite unique from most of the other control strategies (such as traditional PID/LQR). An example of calculation procedure at one sample is shown in Figure 3.1.

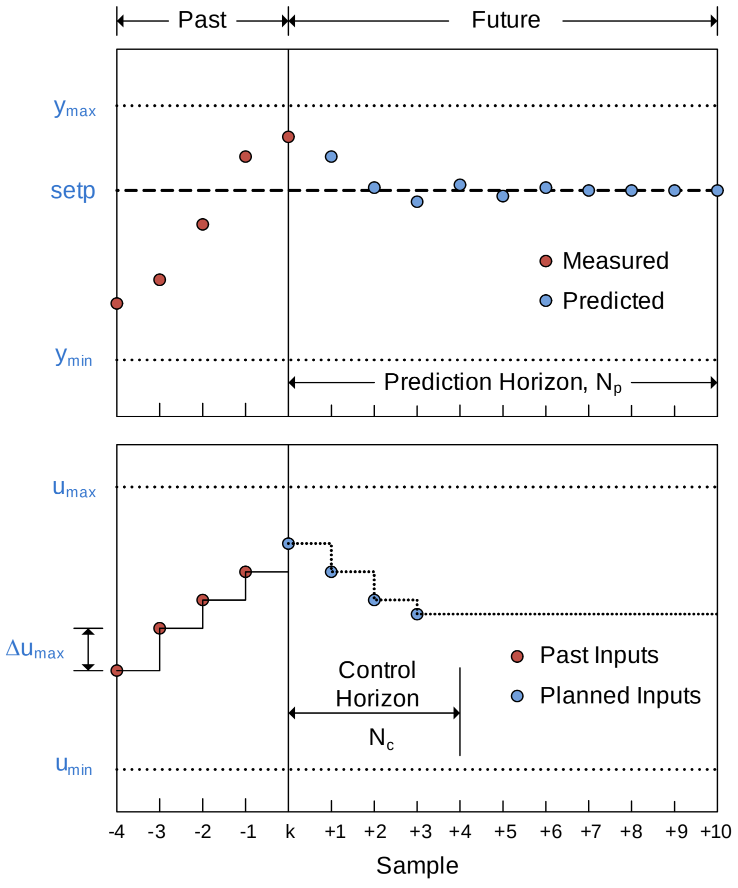


Figure 3.1: A graphical description of the MPC algorithm at k=0.

In order to solve the finite horizon control problem MPC utilizes a system model to predict the system response, based on a sequence of calculated future control moves. This is where two significant advantages of MPC are realised; the system model can be Single Input Single Output (SISO) or Multiple Input Multiple Output (MIMO), or any combination within, meaning multivariable systems can be just as easily controlled as single variable problems. Secondly, by being able to predict the system's response, we can apply constraints to the system outputs, as well as the system inputs, by solving a constrained optimization problem at each time step.

The downside to this powerful control strategy is high computational cost, whereby solving the constrained optimization problem at each time step can consume up to 95% of the computation time at each sample (see Figure 3.3 in Section 3.2.2.6). This therefore limits the applicability of the algorithm to typically large and slow dynamic systems. This work focuses on accelerating the MPC algorithm and resulting constrained optimization problem in order to make the algorithm suitable for implementation on small, low-cost embedded systems controlling fast dynamic systems.

3.2.1.1 Infinite Horizon MPC

At this point it is worth noting that only the finite-horizon MPC algorithm will be considered within this work. This is opposed to the infinite-horizon (or dual-mode) MPC that is now common in recent MPC literature. The reason for this is based on the focus of this work which is taking the optimization algorithm and tailoring it to the control problem, rather than the controller algorithm itself. Moreover, the modification to achieve infinite-horizon MPC for an unconstrained system is simply replacing the final weight (also known as the terminal weight) in the control horizon within the objective function (Equation 3.17) with the solution to the discrete algebraic Riccati equation [214, 287] (via `dare`) in MATLAB, a modification that is considered trivial.

For infinite-horizon MPC of constrained systems (which are the focus of this work), and excepting the computationally unimplementable solution of an infinite horizon (i.e. $N_c = \infty$), the established technique [214, 273, 287] is the addition of terminal constraints, which provide the necessary conditions for an unconstrained infinite-horizon MPC controller to converge to a feasible solution. This follows the unconstrained control law within mode 2 of a dual-mode controller (i.e. from $k = N \to \infty$). The selection of these constraints is independent of control horizon and initial state, however the number of terminal constraints required is based on the controller constraints, model and control law [54]. The issue is more about ensuring the control horizon is 'long enough' so that the terminal state can be reached from any operating point. However, this problem again diverges from the focus on the optimizer, which apart from the addition of a few (typically small in comparison to the prediction horizon) extra linear inequality constraints, is the same quadratic optimization problem.

As stated by Rossiter in [287], "In practice the DMC/GPC algorithm is good enough to handle most industrial problems. Recent advances have given a better understanding of why this is so". This indicates the finite-horizon algorithm of

DMC/GPC works sufficiently well, while the advances within infinite-horizon MPC have aided the theoretical reasoning as to why this is so. With the focus on this work concerning the optimization, only the finite-horizon MPC controller will be considered henceforth. The reader is reminded that a simple implementation of dual-mode MPC only requires minor modifications to the weighting matrix and constraint set, both of which can be handled using optimization techniques developed in this work.

3.2.1.2 System Model

The core requirement of MPC is the availability of a linear system model. This can be obtained by a variety of means, such as first principles modelling, system identification or previous work, but ultimately a linear state space model must be derived. As this work focuses on an implementation of discrete MPC, the model of the system to control takes the following discrete state-space form for a SISO system:

$$\hat{\mathbf{x}}_{k+1} = \hat{\mathbf{A}}\hat{\mathbf{x}}_k + \hat{\mathbf{B}}u_k \tag{3.2}$$

$$\hat{y}_k = \hat{\mathbf{C}}\hat{\mathbf{x}}_k \tag{3.3}$$

and similarly for a MIMO system

$$\hat{\mathbf{x}}_{k+1} = \hat{\mathbf{A}}\hat{\mathbf{x}}_k + \hat{\mathbf{B}}\mathbf{u}_k \tag{3.4}$$

$$\hat{\mathbf{y}}_k = \hat{\mathbf{C}}\hat{\mathbf{x}}_k \tag{3.5}$$

where $\hat{\mathbf{x}} \in \Re^{n\times 1}$, $\hat{\mathbf{A}} \in \Re^{n\times n}$, $\hat{\mathbf{B}} \in \Re^{n\times m}$ and $\hat{\mathbf{C}} \in \Re^{p\times n}$. The hat ($\hat{}$) is used to designate a variable which is part of the estimated system model. In addition, the state-space feedforward $\hat{\mathbf{D}}$ matrix is not used, as is common in predictive control implementations. For the remainder of this chapter, the system model is assumed to represent a SISO system, in order to simplify the algorithm description. For modifications required to control MIMO systems, the reader can refer to Section 1.5 in [326], noting that only minor modifications are required.

3.2.1.3 Augmenting an Integrator

In order to design a tracking (i.e. able to follow a constant reference non-zero setpoint, or setpoint with step changes only) MPC controller using this model, we must augment an integrator to the system model. This can be achieved by either augmenting the past control input(s) to the state vector, or, as done in [326] and illustrated below, augmenting the output(s) to the state vector.

To augment the model output to the state vector, first define two new delta (difference) variables

$$\Delta u_k \overset{\text{def}}{=} u_k - u_{k-1} \tag{3.6}$$

$$\mathbf{\Delta}\hat{\mathbf{x}}_k \overset{\text{def}}{=} \hat{\mathbf{x}}_k - \hat{\mathbf{x}}_{k-1} \tag{3.7}$$

Given these difference variables, the difference of the state-space model state update equation can be rewritten as

$$\mathbf{\Delta}\hat{\mathbf{x}}_{k+1} = \hat{\mathbf{A}}\mathbf{\Delta}\hat{\mathbf{x}}_k + \hat{\mathbf{B}}\Delta u_k \tag{3.8}$$

and the output difference equation as:

$$\hat{y}_{k+1} - \hat{y}_k = \hat{\mathbf{C}}\hat{\mathbf{A}}\mathbf{\Delta}\hat{\mathbf{x}}_k + \hat{\mathbf{C}}\hat{\mathbf{B}}\Delta u_k \tag{3.9}$$

To connect the state variable increment $\mathbf{\Delta}\hat{\mathbf{x}}_k$ with the output $\hat{y}_k$, we define a new state variable vector

$$\mathbf{x}_k \overset{\text{def}}{=} \begin{bmatrix} \mathbf{\Delta}\hat{\mathbf{x}}_k{}^T & \hat{y}_k \end{bmatrix}^T \tag{3.10}$$

and by using equations 3.8 and 3.9, we now form the new augmented model

$$\underbrace{\begin{bmatrix} \mathbf{\Delta}\hat{\mathbf{x}}_{k+1} \\ \hat{y}_{k+1} \end{bmatrix}}_{\mathbf{x}_{k+1}} = \underbrace{\begin{bmatrix} \hat{\mathbf{A}} & \mathbf{0}^T \\ \hat{\mathbf{C}}\hat{\mathbf{A}} & 1 \end{bmatrix}}_{\mathbf{A}} \underbrace{\begin{bmatrix} \mathbf{\Delta}\hat{\mathbf{x}}_k \\ \hat{y}_k \end{bmatrix}}_{\mathbf{x}_k} + \underbrace{\begin{bmatrix} \hat{\mathbf{B}} \\ \hat{\mathbf{C}}\hat{\mathbf{B}} \end{bmatrix}}_{\mathbf{B}} \Delta u_k \tag{3.11}$$

$$\hat{y}_k = \underbrace{\begin{bmatrix} \mathbf{0} & 1 \end{bmatrix}}_{\mathbf{C}} \begin{bmatrix} \mathbf{\Delta}\hat{\mathbf{x}}_k \\ \hat{y}_k \end{bmatrix} \tag{3.12}$$

where $\mathbf{0} \in \Re^{(p\times n)}$. Equations 3.11 and 3.12 now form the basis for the model used for building the remainder of the MPC controller.

3.2.1.4 Generating the Prediction Matrices

The key idea in Model *Predictive* Control is the predictive component, which uses the augmented model defined in equations 3.11 and 3.12 to predict the future outputs of the system ($\hat{y}$), given the control inputs (Δu). However the first question that arises is: how far forward do we predict the plant output? This is known as the Prediction Horizon (N_p), and is defined as the number of samples into the future we will use the model to predict. In addition, we also have the ability to calculate multiple future control moves, where the number we will optimize is defined by the Control Horizon

(N_c). Figure 3.1 shows a graphical definition of both of these horizons, with respect to the current sampling instant.

At the current sampling instant, k, together with a series of present and future control moves

$$\Delta u_k,\ \Delta u_{k+1},\ \cdots,\ \Delta u_{k+N_c-1} \tag{3.13}$$

and the current state vector, $\mathbf{x}_k$, the augmented model can be used to predict the system state variables over the prediction horizon

$$\mathbf{x}_{k+1|k},\ \mathbf{x}_{k+2|k},\ \cdots,\ \mathbf{x}_{k+N_p|k}$$

In order to predict the future states, Equation 3.2 is rolled forward, as shown in Equation 3.14

$$\begin{aligned}
\mathbf{x}_{k+1|k} &= \mathbf{A}\mathbf{x}_k + \mathbf{B}\Delta u_k \\
\mathbf{x}_{k+2|k} &= \mathbf{A}\mathbf{x}_{k+1|k} + \mathbf{B}\Delta u_{k+1} \\
&= \mathbf{A}^2\mathbf{x}_k + \mathbf{A}\mathbf{B}\Delta u_k + \mathbf{B}\Delta u_{k+1} \\
&\vdots \\
\mathbf{x}_{k+N_p|k} &= \mathbf{A}^{N_p}\mathbf{x}_k + \mathbf{A}^{N_p-1}\mathbf{B}\Delta u_k + \mathbf{A}^{N_p-2}\mathbf{B}\Delta u_{k+1} + \cdots + \mathbf{A}^{N_p-N_c}\mathbf{B}\Delta u_{k+N_c-1}
\end{aligned} \tag{3.14}$$

noting that when $N_p = N_c$, the final $\mathbf{A}$ term goes to $\mathbf{I}$ and the final term reduces to $\mathbf{B}\Delta u_{k+N_c-1}$. For controllers where $N_p > N_c$, the controller input is maintained at the last calculated control input, as $\mathbf{\Delta u}$ is set as 0.

To predict the system output, the predicted states are substituted into Equation 3.12

$$\begin{aligned}
\hat{y}_{k+1|k} &= \mathbf{C}\mathbf{A}\mathbf{x}_k + \mathbf{C}\mathbf{B}\Delta u_k \\
\hat{y}_{k+2|k} &= \mathbf{C}\mathbf{A}\mathbf{x}_{k+1|k} + \mathbf{C}\mathbf{B}\Delta u_{k+1} \\
&= \mathbf{C}\mathbf{A}^2\mathbf{x}_k + \mathbf{C}\mathbf{A}\mathbf{B}\Delta u_k + \mathbf{C}\mathbf{B}\Delta u_{k+1} \\
&\vdots \\
\hat{y}_{k+N_p|k} &= \mathbf{C}\mathbf{A}^{N_p}\mathbf{x}_k + \mathbf{C}\mathbf{A}^{N_p-1}\mathbf{B}\Delta u_k + \mathbf{C}\mathbf{A}^{N_p-2}\mathbf{B}\Delta u_{k+1} + \cdots + \mathbf{C}\mathbf{A}^{N_p-N_c}\mathbf{B}\Delta u_{k+N_c-1}
\end{aligned} \tag{3.15}$$

which together with the future calculated control inputs, Δu_k and the current state, $\mathbf{x}_k$, allows a prediction of how the system under control will react.

Equation 3.15 is typically implemented in Matrix-Vector form, by stacking the following two vectors

$$\mathbf{Y} \overset{\text{def}}{=} \begin{bmatrix} \hat{y}_{k+1|k} & \hat{y}_{k+2|k} & \hat{y}_{k+3|k} & \cdots & \hat{y}_{k+N_p|k} \end{bmatrix}^T$$

$$\mathbf{\Delta U} \overset{\text{def}}{=} \begin{bmatrix} \Delta u_k & \Delta u_{k+1} & \Delta u_{k+2} & \cdots & \Delta u_{k+N_c-1} \end{bmatrix}^T$$

and then rewriting as a matrix expression

$$\mathbf{Y} = \mathbf{F}\mathbf{x}_k + \mathbf{\Phi}\mathbf{\Delta U} \tag{3.16}$$

where the two matrices in Equation 3.16 are formed as

$$\mathbf{F} = \begin{bmatrix} \mathbf{CA} \\ \mathbf{CA}^2 \\ \mathbf{CA}^3 \\ \vdots \\ \mathbf{CA}^{N_p} \end{bmatrix}, \mathbf{\Phi} = \begin{bmatrix} \mathbf{CB} & \mathbf{0} & \mathbf{0} & \cdots & \mathbf{0} \\ \mathbf{CAB} & \mathbf{CB} & \mathbf{0} & \cdots & \mathbf{0} \\ \mathbf{CA}^2\mathbf{B} & \mathbf{CAB} & \mathbf{CB} & \cdots & \mathbf{0} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \mathbf{CA}^{N_p-1}\mathbf{B} & \mathbf{CA}^{N_p-2}\mathbf{B} & \mathbf{CA}^{N_p-3}\mathbf{B} & \cdots & \mathbf{CA}^{N_p-N_c}\mathbf{B} \end{bmatrix}.$$

3.2.2 Quadratic Program Formulation

To calculate the future control actions the controller will implement, a cost function is used to define the operational objective of the controller. As described in Section 3.1, the standard MPC cost function is to minimize the squared difference between predicted plant output (using Equation 3.16) and the future setpoint (which provides the reference tracking ability), together with the squared control increments (to minimize aggressive control action). To be able to solve this cost function for the optimal control increments, subject to operational constraints, we must solve a quadratic optimization problem.

3.2.2.1 Cost Function

We define a setpoint expansion matrix which expands the current setpoint, y_k^*, across the entire prediction horizon

$$\mathbf{Y}^* \overset{\text{def}}{=} \underbrace{\begin{bmatrix} 1 & 1 & \cdots & 1 \end{bmatrix}^T}_{\Re^{N_p \times 1}} y_k^*$$

noting that in the above formulation we do not leverage future setpoint information. This can however be done if desired. The setpoint expansion is then used within the following quadratic cost function

$$\mathcal{J} = \left(\mathbf{Y}^{*} - \mathbf{Y}\right)^{T} \mathbf{Q} \left(\mathbf{Y}^{*} - \mathbf{Y}\right) + \mathbf{\Delta U}^{T}\mathbf{R}\mathbf{\Delta U} \tag{3.17}$$

where $\mathbf{Q} \in \Re^{(N_p \times N_p)}$,a symmetric positive-semidefinite tuning matrix for penalizing setpoint deviations, and $\mathbf{R} \in \Re^{(N_c \times N_c)}$, a symmetric positive-semidefinite matrix for penalizing control input movements. For the classic MPC formulation, both $\mathbf{Q}$ and $\mathbf{R}$ are diagonal (containing the tuning weights expanded along the diagonal), however off-diagonal terms can be included for stability reasons or alternative cost functions. Note Equation 3.17 is the matrix-vector form of Equation 3.1, declared at the beginning of this chapter.

With Equation 3.16 substituted into Equation 3.17 and terms collected about $\mathbf{\Delta U}$, we have the following quadratic matrix expression, noting it is a function of $\mathbf{\Delta U}$ only (we assume we can measure or estimate $\mathbf{x}_k$)

$$\mathcal{J} = \mathbf{\Delta U}^{T} \underbrace{\left(\mathbf{\Phi}^{T}\mathbf{Q}\mathbf{\Phi} + \mathbf{R}\right)}_{\mathbf{H}} \mathbf{\Delta U} \underbrace{-2\mathbf{\Phi}^{T}\mathbf{Q}\left(\mathbf{Y}^{*} - \mathbf{F}\mathbf{x}_k\right)}_{2\mathbf{f}} \mathbf{\Delta U} + \underbrace{\left(\mathbf{Y}^{*} - \mathbf{F}\mathbf{x}_k\right)^{T} \mathbf{Q} \left(\mathbf{Y}^{*} - \mathbf{F}\mathbf{x}_k\right)}_{\text{bias}} \tag{3.18}$$

where $\mathbf{H}$ and $\mathbf{f}$ are the standard quadratic program objective matrix and vector, and the bias is simply a constant which can be dropped from the optimizer (it only affects the objective value, $\mathcal{J}$, not the decision variables, $\mathbf{\Delta U}$). The expression for $\mathbf{H}$ has a special property in which it is convex, and therefore this cost function results in a global minimum when solved correctly. This feature will be exploited heavily later in this chapter (see Section 3.4.5.7).

Equation 3.18 is the standard linear MPC cost function which when unconstrained, can be solved using a standard linear equation solver (`-H\f` in MATLAB). However with the addition of operational constraints, as described in following subsections, the problem becomes a constrained optimization problem which will be solved in this work as a quadratic programming problem.

3.2.2.2 Input Rate of Change Constraints

An input rate constraint is used to constrain the maximum rate of change of an input variable. This could be used to implement a constraint on the slew rate of an amplifier, or the change in a pump speed. The input rate constraint is the simplest to implement, and is simply added as bounds on the decision variables, $\mathbf{\Delta U}$

$$\mathbf{\Delta U}_{\min} \leq \mathbf{\Delta U} \leq \mathbf{\Delta U}_{\max} \tag{3.19}$$

To implement the above constraint in a quadratic problem subject only to linear inequality constraints, the following inequalities can be defined

$$\begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \end{bmatrix} \begin{bmatrix} \Delta u_k \\ \Delta u_{k+1} \\ \Delta u_{k+2} \\ \vdots \\ \Delta u_{k+N_c-1} \end{bmatrix} \leq \mathbf{\Delta U}_{\max} \tag{3.20}$$

$$\begin{bmatrix} -1 & 0 & 0 & \cdots & 0 \\ 0 & -1 & 0 & \cdots & 0 \\ 0 & 0 & -1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & -1 \end{bmatrix} \begin{bmatrix} \Delta u_k \\ \Delta u_{k+1} \\ \Delta u_{k+2} \\ \vdots \\ \Delta u_{k+N_c-1} \end{bmatrix} \leq -\mathbf{\Delta U}_{\min} \tag{3.21}$$

or in matrix-vector form

$$\mathbf{I\Delta U} \leq \mathbf{\Delta U}_{\max} \tag{3.22}$$

$$-\mathbf{I\Delta U} \leq -\mathbf{\Delta U}_{\min} \tag{3.23}$$

3.2.2.3 Input Constraints

An input constraint is used to constrain the input actually applied to the plant, rather than the discrete increment, as above. These are used for example to constrain the minimum or maximum input voltage, or minimum or maximum valve position. Input constraints are written as the following two inequalities

$$\mathbf{U}_{\min} \leq \mathbf{U} \leq \mathbf{U}_{\max} \tag{3.24}$$

where the input constraints are applied across the entire control horizon.

Constraints on the control input itself must be formulated as a function of the control increments, meaning we must define an equation which relates the previous control input and planned control increments to the future control inputs

$$\begin{bmatrix} u_k \\ u_{k+1} \\ u_{k+2} \\ \vdots \\ u_{k+N_c-1} \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix} u_{k-1} + \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ 1 & 1 & 0 & \cdots & 0 \\ 1 & 1 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} \Delta u_k \\ \Delta u_{k+1} \\ \Delta u_{k+2} \\ \vdots \\ \Delta u_{k+N_c-1} \end{bmatrix} \tag{3.25}$$

$$= \mathbf{U}_{k-1} + \mathbf{T}\boldsymbol{\Delta}\mathbf{U}$$

where $\mathbf{T}$ is a lower triangular ones matrix in $\Re^{N_c \times N_c}$.

To implement Equation 3.25 as a set of standard linear inequalities, it is substituted into 3.24 which results in the following inequalities

$$\begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ 1 & 1 & 0 & \cdots & 0 \\ 1 & 1 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} \Delta u_k \\ \Delta u_{k+1} \\ \Delta u_{k+2} \\ \vdots \\ \Delta u_{k+N_c-1} \end{bmatrix} \leq \mathbf{U}_{\max} - \begin{bmatrix} 1 \\ 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix} u_{k-1} \tag{3.26}$$

$$\begin{bmatrix} -1 & 0 & 0 & \cdots & 0 \\ -1 & -1 & 0 & \cdots & 0 \\ -1 & -1 & -1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ -1 & -1 & -1 & -1 & -1 \end{bmatrix} \begin{bmatrix} \Delta u_k \\ \Delta u_{k+1} \\ \Delta u_{k+2} \\ \vdots \\ \Delta u_{k+N_c-1} \end{bmatrix} \leq -\mathbf{U}_{\min} + \begin{bmatrix} 1 \\ 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix} u_{k-1}. \tag{3.27}$$

3.2.2.4 Output Constraints

An output constraint is used to constrain the plant output, based on the predicted outputs of the plant. These are used for example to constrain the minimum or maximum angle, or minimum or maximum flow rate. Output constraints are written as the following two inequalities

$$\mathbf{Y}_{\min} \leq \mathbf{Y} \leq \mathbf{Y}_{\max} \tag{3.28}$$

where the output constraints are applied across the entire prediction horizon.

Substituting Equation 3.16 into Equation 3.28 we can write the following inequalities to constrain the plant output

$$\mathbf{\Phi}\mathbf{\Delta U} \leq \mathbf{Y}_{\max} - \mathbf{F}\mathbf{x}_k \tag{3.29}$$

$$-\mathbf{\Phi}\mathbf{\Delta U} \leq -\mathbf{Y}_{\min} + \mathbf{F}\mathbf{x}_k \tag{3.30}$$

Note that with the current formulation state constraints are not implemented explicitly. However by placing a 1 in the corresponding element within the state space **C** matrix, the state can be constrained. Furthermore, as detailed in Section A.2.7, the output can be identified as *uncontrolled*, thereby removing the need for a setpoint and thus removing it from the reference tracking penalty (Equation 3.17) within the cost function, and rendering it purely a state constraint.

3.2.2.5 Hard versus Soft Constraints

All three constraints described so far are *hard* constraints, meaning the solution *must* lie within the interior of these constraints, if a feasible solution exists. This choice of constraint is common for input constraints (both rate-of-change and absolute), as they reflect hard physical limits such as actuator limits. Output constraints, in contrast, are subject to the accuracy of the internal linear model for predicting the future output, as well as any unknown external disturbances, both of which make using hard output constraints a risk to controller feasibility [287]. The solution is to make all output constraints *soft*, meaning they may be temporary violated, but the controller will be penalized via a term in the objective function, which actively discourages violating these constraints.

Within this work, soft constraints are added when the user specifies a penalty weight on an output constraint. Mathematically, soft constraints are added as extra decision variables, one per constraint pair (i.e. upper and lower). The penalty weight is augmented to the quadratic program **H** matrix as

$$\begin{bmatrix} \mathbf{H} & \mathbf{0} \\ \mathbf{0} & \mathrm{diag}(\boldsymbol{\lambda}) \end{bmatrix} \tag{3.31}$$

where $\boldsymbol{\lambda}^T$ is defined as

$$\begin{bmatrix} \lambda_1 & \lambda_2 & \cdots & \lambda_n \end{bmatrix}$$

and λ_i is the associated penalty for constraint pair i. In addition, the output constraint equation is modified as follows

$$\begin{bmatrix} \mathbf{\Phi} & -\mathbf{1} \end{bmatrix} \begin{bmatrix} \mathbf{\Delta U} \\ \mathbf{\Delta U}_{\text{soft}} \end{bmatrix} \leq \mathbf{Y}_{\max} - \mathbf{F}\mathbf{x}_k$$
$$\begin{bmatrix} -\mathbf{\Phi} & -\mathbf{1} \end{bmatrix} \begin{bmatrix} \mathbf{\Delta U} \\ \mathbf{\Delta U}_{\text{soft}} \end{bmatrix} \leq -\mathbf{Y}_{\min} + \mathbf{F}\mathbf{x}_k$$

where $\mathbf{1} \in \Re^{N_p \times 1}$ and $\mathbf{\Delta U}_{\text{soft}}$ are the decision variable(s) associated with the penalty terms (the last element in $\mathbf{\Delta U}$ for a SISO system). For a MIMO system, $\mathbf{1}$ is replaced with the identity matrix, dimensions $q \times q$, repeated for each sample in the prediction horizon.

As stated by Rossiter in [287], violations of soft constraints does not effect nominal stability results, meaning that by using soft output constraints, we can ensure feasibility of the control problem. For the remainder of this work, all constraints will be treated as hard by default, unless specified otherwise. This is done to keep the problem definition simpler.

3.2.2.6 Resulting Quadratic Program

In order to solve the resulting quadratic optimization problem described so far, it must be posed in the standard quadratic programming form. This is done below using standard nomenclature

$$\begin{aligned} \min_{\mathbf{x}} \ & \frac{1}{2}\mathbf{x}^T\mathbf{H}\mathbf{x} + \mathbf{f}^T\mathbf{x} \\ \text{subject to: } & \mathbf{A}\mathbf{x} \leq \mathbf{b} \end{aligned} \tag{3.32}$$

where for this work we assume that the quadratic programming solver only solves problems with inequality constraints. This is possible as the formulation used includes the linear model state space equation within the generation of the prediction matrices, as opposed to a series of linear equality constraints in alternate formulations. This alternate format has been avoided due to additional complexity of handling equality constraints as well as inequality constraints, given this would require additional steps to formulate the problem within the QP solver.

Following the above definition, the optimization problem to solve at each sampling instant is written as follows

$$\begin{aligned} \min_{\mathbf{\Delta u}} \ & \frac{1}{2}\mathbf{\Delta u}^T\mathbf{H}\mathbf{\Delta u} + \mathbf{f}^T\mathbf{\Delta u} \\ \text{subject to: } & \mathbf{M}\mathbf{\Delta u} \leq \mathbf{b} \end{aligned} \tag{3.33}$$

where

$$\mathbf{H} = \mathbf{\Phi}^T \mathbf{Q} \mathbf{\Phi} + \mathbf{R}, \quad \mathbf{f} = -\mathbf{\Phi}^T \mathbf{Q} \left(\mathbf{Y}^* - \mathbf{F} \mathbf{x}_k \right)$$

$$\mathbf{M} = \begin{bmatrix} \mathbf{I} \\ -\mathbf{I} \\ \mathbf{T} \\ -\mathbf{T} \\ \mathbf{\Phi} \\ -\mathbf{\Phi} \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} \mathbf{\Delta U}_{\text{max}} \\ -\mathbf{\Delta U}_{\text{min}} \\ \mathbf{U}_{\text{max}} - \mathbf{U}_{k-1} \\ -\mathbf{U}_{\text{min}} + \mathbf{U}_{k-1} \\ \mathbf{Y}_{\text{max}} - \mathbf{F}\mathbf{x}_k \\ -\mathbf{Y}_{\text{min}} + \mathbf{F}\mathbf{x}_k \end{bmatrix}$$

and assuming all constraints are imposed in the optimization problem, as well as modelling a SISO system. All constraints are assumed present in order to represent the worst case scenario, i.e. the maximum number of constraints. This provides the most challenging optimization problem with respect to both computational demand and memory requirements, and therefore will be used throughout the remainder of this work. In practice however, it is noted not all constraints would be used, but this design approach does allow greater flexibility.

Note extensions to the MIMO case simply expand the optimization problem dimensions, but the same structure and sets of equations remain.

For a fully constrained (i.e. constraints on $\mathbf{\Delta U}, \mathbf{U}$ and $\mathbf{Y}$) SISO system with $N_p = 8, N_c = 3$ the $\mathbf{M}^T$ matrix has the structure shown in Figure 3.2. Note that $\mathbf{M}$

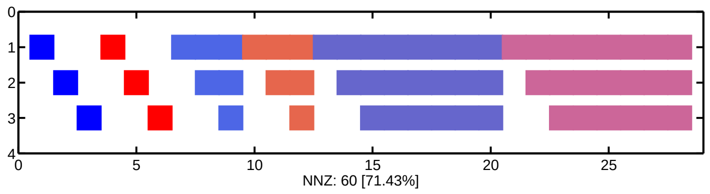


Figure 3.2: Linear inequality constraint $\mathbf{M}^T$ matrix structure.

is particularly dense at greater than 70% non-zero terms, which is typical for these problems. Given this particularly dense structure, sparse routines are unlikely to offer any advantage in computational speed.

Equation 3.33 completes the constrained control law calculation required at each sample step. This forms the bulk of the computation time required at each sample, as shown in Figure 3.3 for a 2 input, 3 output, 6 state system with $N_p = 15, N_c = 10$ on a desktop computer. Note in the jMPC implementation (described later on in Section 3.5.2) the QP only requires solving when the global unconstrained minimum of Equation 3.33 does not satisfy the constraints. When the QP solver is called, it averages 94% of the required computation time, meaning the focus for high-speed MPC should definitely target this step.

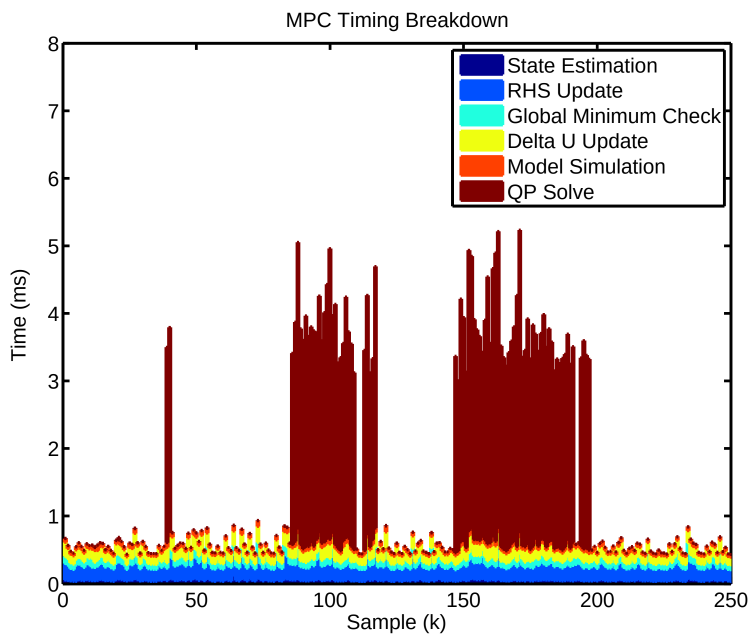


Figure 3.3: Timing breakdown of the MPC computation at each sample.

3.3 Review of Existing Quadratic Programming Solvers

3.3.1 Quadratic Programming Algorithms

Using dedicated custom code to solve quadratic programming problems is an established discipline which aims to solve a linearly constrained nonlinear optimization problem which involves only bilinear and quadratic terms. By leveraging this feature of the objective, the Hessian of the nonlinear problem reduces to a constant matrix. We can thus describe the entire problem with respect to constant matrices and vectors as shown in Equation 3.32. The main advantage of this class of nonlinear problem is based on the problem description requiring only constant matrices (as opposed to black-box functions), and similar to a constrained linear program, a solver can be tailored to exploit all problem information.

The quadratic programs that result from the MPC formulation described so far are always convex (as proved in Section 3.4.2), and thus global optimality can be ensured using a suitable algorithm, provided the problem is of course feasible. The methods for solving a quadratic problem vary, based on factors such as the types of constraints, whether similar QP problems will be solved iteratively, and availability within a modelling framework. The two most common algorithm classes are described below and are based on the overviews described in [235, 336].

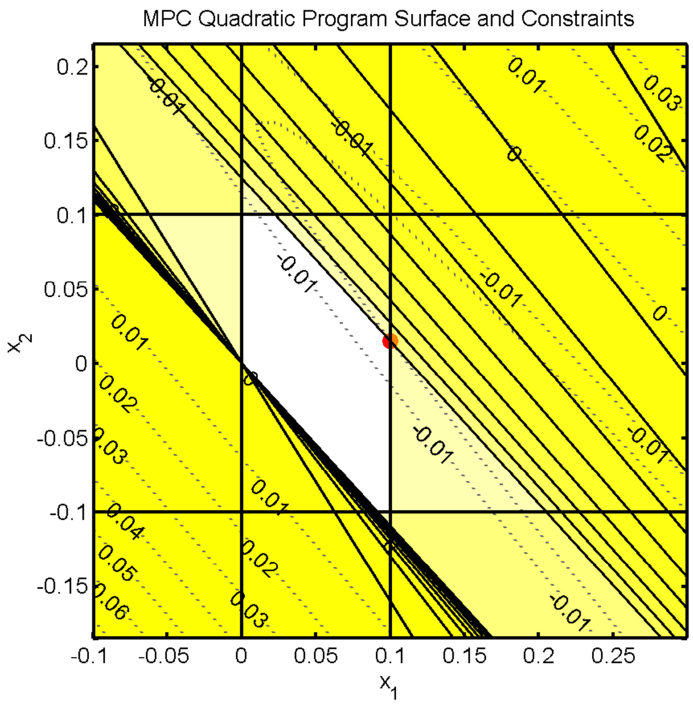


Figure 3.4: Quadratic program optimization surface and constraints resulting from a fully constrained SISO system with $N_p = 10, N_c = 2$. Box constraints result from constraints on $\mathbf{\Delta U}$, while the other linear constraints are the upper and lower output constraints. The red dot indicates the constrained optimum.

3.3.1.1 Active-Set

Active-set methods are best suited to small to medium sized-problems and are one of the oldest techniques for solving QPs, having been widely used since the 1970s. The fundamental idea of the active-set method is to find the set of constraints (bounds, equality and inequality) which is *active* at the solution. With this information it is possible to solve an equality-only constrained sub-problem to determine the optimal value of the objective and decision variables.

The active-set method is typically regarded as a two-phase process, where the first phase finds a feasible starting point (which can be found by solving a linear program, the white central area within Figure 3.4), while the second phase aims to find the optimal solution given this starting point and a guess of the current active constraints (known as the working set) by solving an equality constrained QP sub-problem. The advantage of this approach is that not all constraints are considered and thus solving the QP sub-problem in phase two can be quite small, depending on the size of the working set. As shown in Figure 3.4, it is typical for an MPC problem to only have one or two active constraints at the solution, given for example the output constraints should be at quite different operating points.

The efficiency of the active-set method is realised by clever updates of the factorization of the QP sub-problem at each iteration, which assuming only one constraint is ever added or deleted, only one row is ever changed within the Karush-Kuhn-Tucker (KKT) matrix. By utilizing an initital QR factorization together with a transformation matrix for each iteration, addition of new constraints does not require the entire KKT matrix to be re-factorized. Similarly, when dropping constraints, applying a sequence of plane rotations allows the factors to be updated.

Active-set methods are also well suited to iterative problems where similar QPs will be solved sequentially, such as in the MPC case. By providing the previous final active-set to the solver at the next QP to solve, this can substantially reduce the algorithm searching for the correct working set [27]. The downside of the active-set algorithm, however, is that in the worst case scenario, where it must check every combination of constraints, the algorithm is non-polynomial in time (also called $\mathcal{NP}$-hard), and thus can scale very poorly. Referencing again back to Figure 3.4, this situation can occur when a setpoint changes the output from close to upper output constraints (top of the figure), to now close to the lower output constraints (bottom of the figure). The previous active-set guess will be completely wrong, and the solver must now identify a completely new set of active constraints.

3.3.1.2 Interior-Point

The interior-point method originally gets its name from the characteristic that required all iterates to satisfy all constraints, and thus must always lie in the *interior* of the problem subspace. These methods were typical of formulations used for nonlinear programming, but it was revealed in the 1980s that they could also solve linear (and associated quadratic) programs efficiently, i.e. with polynomial complexity. One of the seminal works on interior-point methods was Karmarkar's projective algorithm [161], which as described by Nocedal and Wright in [235], also showed good practical performance. This was opposed to earlier work such as the ellipsoid method proposed by Khachiyan [167], which although exhibited at worst polynomial complexity, it was in practical terms always at this lower performance limit.

Interior-point methods typically fall into one of three categories. The potential reduction algorithm (similar to Karmarkar's algorithm), the affine scaling algorithm and the path following algorithm. Within the path following category, the primal-dual approach has become the method of choice for large-scale implementations, given that it exhibits excellent practical performance (much better than the worst case polynomial complexity). The term primal-dual refers to the fact the algorithm simultaneously solves the primal and corresponding dual problem, iteratively updating the primal and dual iterates until convergence has been obtained, typically by checking primal and dual feasibility, together with the complementarity (or duality) gap. Moreover by ensuring the KKT conditions are satisfied at each step, together with ensuring the problem is convex, this method will guarantee the global optimum.

Modern (dual) path-following interior-point methods (1990s) no longer require the initial iterates to lie within the feasible region (although the solution must, if the problem is feasible), so that an initial feasibility phase such as required by the active-set algorithm is not required. At each iteration the algorithm solves a system of linear equations that contains all constraints, which can be quite computationally expensive. However the result of this expensive calculation is that the algorithm can make significant progress towards the solution, and thus the number of iterations required is reduced. This is in contrast with the simplex method for LPs (see Chapter 13 in [235]), which requires a large number of iterations, but little work is done at each iteration to compute the simplex pivot.

The most important (modern) contribution to interior-point methods was the predictor-corrector modification proposed by Mehortra [217] which advocated the use of a second-order term to correct for the linearization of the system of nonlinear equations solved by the path-following algorithm. A later modification was proposed by Gondzio [115] which included higher-order correction terms to further reduce this linearization error, and maintain the iterates closer to the central path. One or both modifications are typical in most practical implementations of primal-dual interior-point methods today.

Warm starting path-following interior-point methods has received considerable attention, such as in [345] which proposes several techniques for warm starting based on a perturbed linear program, however it is generally accepted that they do not benefit from warm-starting to the same degree as the active-set method (see Section 16.6 in [235]). This is related to the path-following aspect of the algorithm, which requires the iterates to lie within the central-path in order for large and accurate steps towards the optimum to be made.

3.3.2 Existing Quadratic Programming Solvers

Given the abundance of work within the quadratic programming field there is also a multitude of high-speed QP solver implementations available to use. A small selection of the solvers used for performance comparisons is briefly described in the following sub-sections, and was chosen based on availability for this work. Table 3.1 summarises the availability and functionality of each solver.

Table 3.1: Surveyed quadratic programming solver availability and problem solving functionality.

Solver	Availability	Solver Functionality
CLP	Open source	LP, QP
CPLEX	Commercial	LP, MILP, QP, MIQP, QCQP, MIQCQP
CVXGEN	Open source[1]	LP, QP, QCQP
IPOPT	Open source	QP, NLP
MATLAB - `quadprog`	Commercial	QP
OOQP	Open source	LP, QP
QPC - `qpip,qpas`	Closed source	QP
SCIP	Open source	LP, MILP, QP, MIQP, QCQP, MIQCQP

[1] Code generation source is closed, but the resulting solver is open source.

3.3.2.1 CLP

Written and maintained by John Forrest (IBM, retired), Clp [100] is an open-source solver primarily focused on solving linear programming problems using either a primal or dual simplex algorithm. However it also contains a barrier (interior-point) algorithm for solving quadratic problems, using a Cholesky factorization algorithm supplied with the solver. As noted by John himself, the sparse Cholesky factorization code is quite basic, and thus substantial performance improvements can be found using third party solvers such as MUMPs [13] or the Watson Sparse Matrix Package (WSMP) [121]. For this work the original supplied Cholesky factorization code has been utilized because the interface to MUMPs would intermittently crash, and WSMP was only available under Cygwin.

3.3.2.2 CPLEX

Now owned by IBM, Cplex [144] is a commercial linear and quadratic solver with support for both integer variables, and quadratic constraints. Together with Gurobi, Cplex is one the fastest mixed integer linear programming solvers available using a simplex solver for relaxed linear problems and an advanced branch and cut framework for mixed integer problems. In addition it contains a very efficient parallelized barrier (interior-point) algorithm for solving quadratic problems, as well as a Second Order Cone (SOC) solver for quadratically constrained problems.

3.3.2.3 CVXGEN

CVXGEN [210] is a C code solver generator based on the CVX framework [116]. It generates a fully unrolled implementation of a fixed-size LP or QP problem, i.e. all matrix and linear algebra routines are fully written out as per a symbolic expression. The advantage is that when compiled with full optimization settings enabled, the resulting solver is typically 20 times faster than the original implementation. The downside to this approach is that it requires a large amount of memory in order to store the compiled solver, and can take a considerable amount of time to compile.

3.3.2.4 IPOPT

IPOPT [323] is an open-source large-scale convex nonlinear programming solver which can also efficiently solve quadratic problems using an implementation of Mehrotra's predictor corrector method [217]. IPOPT stands for Interior Point Optimizer, and thus is an interior point solver. It was originally written in Fortran by Andreas Wächter under the supervision of L. Biegler. Carl Laird later re-implemented IPOPT in C++, which is its current form.

3.3.2.5 MATLAB - `quadprog`

MATLAB's Optimization Toolbox [207] provides a number of quadratic programming solver algorithms via the function `quadprog`. As well as an active-set method, an advanced interior-point (convex only) method is supplied which has been used for benchmarking and validation in this work.

3.3.2.6 OOQP

Object Orientated Quadratic Programming (OOQP) [108] is an open-source software package written by Stephen Wright and Michael Gertz. It provides two interior point algorithms based on current state of the art primal-dual techniques: A Gondzio version [115] and a Mehrotra version [217]. It also provides interfaces and linear solver connections for both sparse and dense systems.

3.3.2.7 QPC

Quadratic Programing in C (QPC) [330] is a free, closed-source package written by Adrian Wills. It provides both an active-set (`qpas`) and an interior-point algorithm (`qpip`), with modifications for handling bounded problems only. The interior point algorithm is based on the Mehrotra predictor corrector algorithm [217], using multiple corrections as proposed by Gondzio [115]. The active-set algorithm is based on an algorithm proposed by Goldfarb and Idnani [113], including modifications proposed later by Powell [253]. Both algorithms are suited for dense systems only.

3.3.2.8 SCIP

Reported as the fastest non-commercial mixed integer linear programming solver available [220], SCIP [2] is a constraint integer programming framework which can solve linear, quadratic and nonlinear problems to proven global optimality. It does this using a spatial branch and bound technique, using under and over estimators to reduce nonlinear problems to relaxed linear subproblems, and then solving them to narrow the problem bounds. SCIP uses its own LP solver, SoPlex, for solving relaxed linear problems, and IPOPT for solving relaxed nonlinear (including quadratic) problems.

3.3.2.9 Comparison of Existing Quadratic Programming Solvers for MPC Problems

In order to get a baseline of existing QP solver performance all the above solvers were run over 500 random quadratic programs, each of which were generated from a constrained (constraints on $\mathbf{\Delta U}, \mathbf{U}$ and $\mathbf{Y}$) MPC controller with $N_p = 10, N_c = 8, m = 2, n = 5, p = 2$. This setup results in a QP with 16 decision variables and 104 linear inequality constraints, which is typical of the sizes targeted in this work. Each model is generated using the MATLAB command `drmodel` which generates a random stable discrete model with m inputs, n states and p outputs, as shown in the code listing in Section A.3. Figure 3.5 shows the timing results of each of the solvers, excluding SCIP (due to larger solve times), and run on a 64bit Intel Core i7 laptop at 2.8GHz.

It is worth noting that solvers such as CPLEX, OOQP, IPOPT and SCIP are designed for large-scale and sparse optimization problems and thus this is not a strictly fair comparison. Figure 3.6 shows the relative performance for the solvers which can leverage the problem density.

In addition to comparing solver performance results, Table 3.2 also shows the accuracy and compiled algorithm size of each of the solvers. All solvers were run with a relative tolerance of 10^{-7} and `quadprog` was taken as the reference solution.

What is clear from Table 3.2 is that several high performance QP solvers do exist and are suitable for robustly solving the optimization problems that result from an MPC formulation. Yet as it will be shown in the next subsection, there is still room

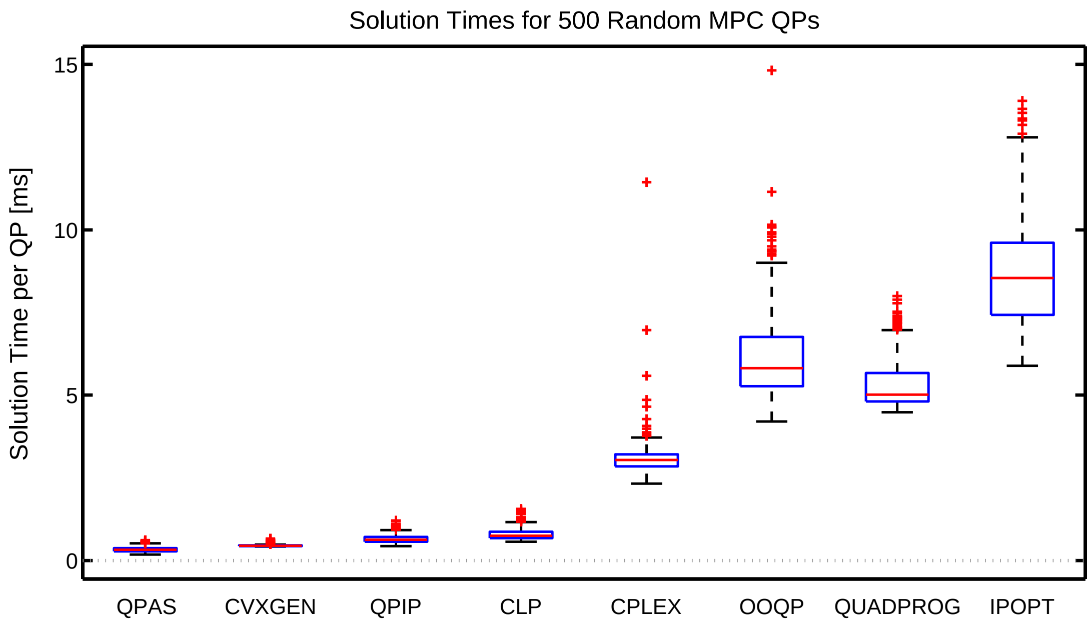


Figure 3.5: Timing comparison of 500 QPs on selected QP solvers (see also Figure 3.6).

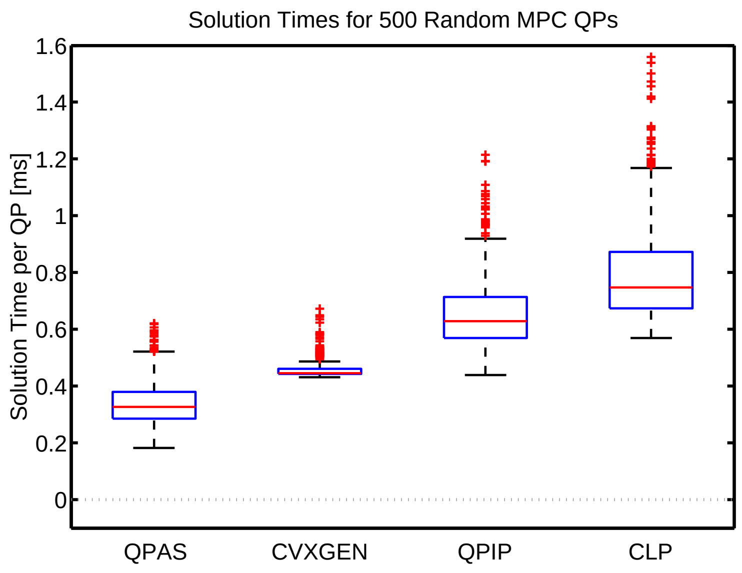


Figure 3.6: Detailed timing comparison of the 4 fastest QP solvers (detail from Figure 3.5).

Table 3.2: Timing and Solver Size comparison for 500 QPs.

Solver	Total Time [s]	Average Time [ms]	Relative Error[1]	File Size [KB]
`qpas`	0.171	0.341	0.000%	95.74
CVXGEN	0.232	0.463	0.000%	1194.5
`qpip`	0.330	0.660	0.005%	114.18
CLP	0.399	0.798	0.000%	2083.84
CPLEX	1.562[2]	3.123[2]	0.000%	16800.96[3]
`quadprog`	2.674	5.348	0.000%	1431.31[4]
OOQP	3.082	6.163	0.000%	26167.81[3]
IPOPT	4.332	8.665	0.000%	11404.8[3]
SCIP	16.087	32.175	0.000%	16063.49[3]

[1] The `quadprog` solution is used as the reference solution.
[2] CPLEX timing includes the getting started parallelization cost.
[3] These solvers are compiled against BLAS, LAPACK and optionally a sparse linear solver (such as PARDISO), increasing the solver size, but also increasing speed.
[4] `quadprog` has been compiled using MATLAB's `mcc` to generate an executable for comparison.

for improvement.

3.4 Development of a New Quadratic Programming Solver

One of the primary motivations to develop a new quadratic programming solver is that the algorithm source code must be available in order to implement it on an embedded system. This is required as target specific libraries, given the multitude of embedded systems and compilers available are virtually impossible to request. Therefore for solvers such as QPIP and QPAS, without purchasing the source (or licensing it in some way), there is no way they can be used for this work.

For solvers where the source is available, such as CVXGEN and CLP, the compiled code size is simply too large to implement on an embedded processor with say, less than 256KB of Flash memory (and/or RAM). This is not to say complex modifications could be made to the existing code in order to get it to fit in memory, but it is also possible that a QP solver specifically tailored to MPC problems could improve on the performance of all solvers surveyed so far, and still require less memory.

For the remainder of this section we will detail the algorithm and development of an Interior-Point (IP) quadratic programming solver that will be tailored for use specifically with MPC problems. While the existing solver survey has shown that an active-set method is actually the fastest, the reasons we have pursued an interior-point method are as follows:

Infeasibility Given the quadratic programs result from an optimal control problem, it is not uncommon that external disturbances may push the system into an infeasible state. This happens when the linear constraints become overly stringent, meaning no control move is possible that will bring the predicted system response back to a feasible operating point. Standard active-set algorithms require that the initial solution guess is feasible before the algorithm can begin to solve for a solution, thus an active-set algorithm will not be able to improve the solution (towards hopefully a less infeasible point) if the problem is infeasible. In contrast, modern interior-point algorithms allow an infeasible initial point, and will work towards the optimal (or less infeasible) point during the normal course of solving. Soft constraints can be used to ensure the system remains feasible, however they increase the problem dimension, add to numerical problems (due to large penalty values) and may not always guarantee a feasible solution in limited precision systems.

Determinism One of the disadvantages of the interior-point algorithm is that all constraints are considered at every iteration. This increases the size of the linear system to be solved at each iteration, meaning the algorithm is quite computationally intensive. An active-set solver, on the other hand, only has to solve for systems that include the active-set of constraints, which for MPC problems, is often only a small percentage of the constraints. The problem is determining which constraints are active, and an active-set solver may have to (theoretically) check every constraint at each iteration before finding a solution, a problem which is $\mathcal{NP}$-hard in the worst case scenario [27] (versus polynomial for modern interior-point methods). The effect is that although slower in general, interior-point methods give a reliable upper-bound on the execution time of the solver, and therefore make them more attractive for real-time control algorithms where worst-case execution time for a specified number of iterations must be known *a-priori*.

Numerical Stability Previous authors such as in [182] have shown that an active-set algorithm may be more numerically sensitive in single precision that an equivalent interior-point algorithm. Their results indicated a 3.65% failure rate for the active-set algorithm due to numerical issues, versus 0.15% for their interior-point implementation. Given the target implementation of this work will be a 32bit microcontroller with a single precision floating point unit, algorithms that have a good numerical stability is an important consideration for the algorithm choice.

3.4.1 Infeasible-Interior-Point Methods

A modern variant of the interior-point algorithm is the Infeasible-Interior-Point (IIP) method. It is based on the Mixed Monotone Linear Complementarity Problem (mLCP), which is a paradigm for describing convex linear and quadratic problems and their optimality conditions. This algorithm and its theoretical underpinning is described by Stephen Wright in [337], and is summarised below in order to detail the algorithm requirements.

An mLCP is defined in terms of a square, positive semidefinite matrix $\mathbf{N} \in \Re^{n\times n}$ and a vector $\mathbf{q} \in \Re^{n}$ where the objective is to find vectors $\mathbf{z}$, $\boldsymbol{\lambda}$ and $\mathbf{t}$ such that

$$\begin{bmatrix}\mathbf{N}_{11} & \mathbf{N}_{12}\\ \mathbf{N}_{21} & \mathbf{N}_{22}\end{bmatrix}\begin{bmatrix}\mathbf{z}\\ \boldsymbol{\lambda}\end{bmatrix} + \begin{bmatrix}\mathbf{q}_1\\ \mathbf{q}_2\end{bmatrix} = \begin{bmatrix}\mathbf{0}\\ \mathbf{t}\end{bmatrix} \tag{3.34}$$

$$\boldsymbol{\lambda} \geq \mathbf{0}, \mathbf{t} \geq \mathbf{0}, \boldsymbol{\lambda}^T\mathbf{t} = \mathbf{0} \tag{3.35}$$

where $\mathbf{N}_{11}$ and $\mathbf{N}_{22}$ are square submatrices of $\mathbf{N}$, $\mathbf{q}$ is partitioned accordingly and $\boldsymbol{\lambda}$ and $\mathbf{t}$ are the lagrange multipliers and slack variables (introduced to convert inequality constraints to equalities) respectively.

In order to maintain algorithm stability while ensuring the iterates stay positive, the complementarity condition $\boldsymbol{\lambda}^T\mathbf{t} = 0$ is replaced with

$$\mathbf{\Lambda}\mathbf{T}\mathbf{e} = \mu\mathbf{e} \tag{3.36}$$

where

$$\mathbf{\Lambda} = \operatorname{diag}(\lambda_1, \lambda_2, ..., \lambda_{m_c}), \quad \mathbf{T} = \operatorname{diag}(t_1, t_2, ..., t_{m_c}), \quad \mathbf{e} = [1, 1, ..., 1]^T.$$

and μ is the complementarity gap, which for iteration k, is defined as

$$\mu^k = \frac{\left(\boldsymbol{\lambda}^k\right)^T \mathbf{t}^k}{m_c} \tag{3.37}$$

Note m_c in Equation 3.37 is the length of $\boldsymbol{\lambda}$ (and thus the length of $\mathbf{t}$), the number of constraints in the problem.

With this modification, given $\boldsymbol{\lambda}, \mathbf{t} \geq 0$, the problem is then to find $\mathbf{z}, \boldsymbol{\lambda}, \mathbf{t}$ such that

$$\mathbf{F}_\mu = \begin{bmatrix}\mathbf{N}_{11}\mathbf{z} + \mathbf{N}_{12}\boldsymbol{\lambda} + \mathbf{q}_1\\ \mathbf{N}_{21}\mathbf{z} + \mathbf{N}_{22}\boldsymbol{\lambda} - \mathbf{t} + \mathbf{q}_2\\ \mathbf{\Lambda}\mathbf{T}\mathbf{e} - \mu\mathbf{e}\end{bmatrix} = 0 \tag{3.38}$$

which is a system of nonlinear equations. In order to solve this system the typical interior-point strategy is to use Newton's method where μ is gradually reduced to 0 (or some pre-defined tolerance) at each iteration. The modified (based on the addition of μ) Newton step is then to find a search direction that satisfies

$$\frac{\partial \mathbf{F}_\mu}{\partial \mathbf{p}} \boldsymbol{\Delta}\mathbf{p} = -\mathbf{F}_\mu \tag{3.39}$$

where

$$\mathbf{p} = \begin{bmatrix} \mathbf{z} \\ \boldsymbol{\lambda} \\ \mathbf{t} \end{bmatrix}, \quad \boldsymbol{\Delta}\mathbf{p} = \begin{bmatrix} \boldsymbol{\Delta}\mathbf{z} \\ \boldsymbol{\Delta}\lambda \\ \boldsymbol{\Delta}\mathbf{t} \end{bmatrix}$$

The infeasible-interior-point method starts with iterates $\mathbf{z}^0, \boldsymbol{\lambda}^0, \mathbf{t}^0$ where $\boldsymbol{\lambda}^0 > 0, \mathbf{t}^0 > 0$, but unlike a traditional interior-point solver, the initial iterates may be infeasible with respect to Equation 3.34. As discussed, this means the algorithm does not need to search for an initial feasible point, and any initial condition can be used. For each subsequent iteration of the algorithm the iterates $\boldsymbol{\lambda}, \mathbf{t}$ remain positive, but the primal and dual infeasibilities, together with the complementarity gap, are gradually reduced to zero. Substituting Equation 3.38 into Equation 3.39 results in the system of linear equations to solve at each iteration

$$\begin{bmatrix} \mathbf{N}_{11} & \mathbf{N}_{12} & \mathbf{0} \\ \mathbf{N}_{21} & \mathbf{N}_{22} & -\mathbf{I} \\ \mathbf{0} & \mathbf{T}^k & \boldsymbol{\Lambda}^k \end{bmatrix} \begin{bmatrix} \boldsymbol{\Delta}\mathbf{z}^k \\ \boldsymbol{\Delta}\lambda^k \\ \boldsymbol{\Delta}\mathbf{t}^k \end{bmatrix} = \begin{bmatrix} -\mathbf{r}_1^k \\ -\mathbf{r}_2^k \\ -\boldsymbol{\Lambda}^k\mathbf{T}^k\mathbf{e} + \sigma^k\mu^k\mathbf{e} \end{bmatrix} \tag{3.40}$$

where

$$\begin{aligned} &\sigma^k \in (0,1] \\ &\mathbf{r}_1^k = \mathbf{N}_{11}\mathbf{z}^k + \mathbf{N}_{12}\boldsymbol{\lambda}^k + \mathbf{q}_1 \\ &\mathbf{r}_2^k = \mathbf{N}_{21}\mathbf{z}^k + \mathbf{N}_{22}\boldsymbol{\lambda}^k - \mathbf{t}^k + \mathbf{q}_2 \end{aligned}$$

To obtain the iterates for the next iteration the increments solved in Equation 3.40 are multiplied by a scalar, then added to the previous iterates

$$\left(\mathbf{z}^{k+1}, \boldsymbol{\lambda}^{k+1}, \mathbf{t}^{k+1}\right) = \left(\mathbf{z}^k, \boldsymbol{\lambda}^k, \mathbf{t}^k\right) + \alpha_k \left(\boldsymbol{\Delta}\mathbf{z}^k, \boldsymbol{\Delta}\lambda^k, \boldsymbol{\Delta}\mathbf{t}^k\right). \tag{3.41}$$

The scaling factor, α, is chosen such that

$$\begin{gathered} \alpha_k \in (0, 1] \\ \left(\boldsymbol{\lambda}^{k+1}, \mathbf{t}^{k+1}\right) > 0. \end{gathered} \tag{3.42}$$

For a proof of global convergence, together with additional conditions required to ensure convergence is obtained, see [337, 339].

3.4.2 Solving Quadratic Programs using the IIP Framework

In order to solve the QP presented in Equation 3.33 using the IIP framework it is easiest to write out the problem again, relating the variables in mLCP problem to the standard QP form

$$\begin{gathered} \min_{\mathbf{z}} \ \frac{1}{2}\mathbf{z}^T\mathbf{H}\mathbf{z} + \mathbf{f}^T\mathbf{z} \\ \text{subject to: } \mathbf{M}\mathbf{z} \le \mathbf{b}. \end{gathered} \tag{3.43}$$

To convert Equation 3.43 into an IIP, we can write the Karush-Kuhn-Tucker (KKT) conditions for the problem. These ensure that for convex problems the feasible solution is also the optimal solution, however they only hold for *convex* problems. Recalling Equation 3.18, $\mathbf{H}$ is defined as

$$\mathbf{H} \overset{\text{def}}{=} \boldsymbol{\Phi}^T\mathbf{Q}\boldsymbol{\Phi} + \mathbf{R}$$

where both $\mathbf{Q}$ and $\mathbf{R}$ are positive-semidefinite matrices, and $\boldsymbol{\Phi}$ is one of the prediction matrices, defined in Equation 3.16. Given $\boldsymbol{\Phi}$ is defined as $\Re^{N_p \times N_c}$ and has a rank of N_c, due to its construction, then following [250]

$$\boldsymbol{\Phi}^T\mathbf{Q}\boldsymbol{\Phi} \succeq \mathbf{0} \tag{3.44}$$

where the expression $\succeq \mathbf{0}$ means positive-semidefinite. The addition of two positive-semidefinite matrices also results in a positive-semidefinite matrix, following the standard definition of a positive-semidefinite expression

$$\begin{aligned} \mathbf{x}^T\left(\boldsymbol{\Phi}^T\mathbf{Q}\boldsymbol{\Phi}\right)\mathbf{x} \ge 0, \forall\mathbf{x} \\ \mathbf{x}^T\mathbf{R}\mathbf{x} \ge 0, \forall\mathbf{x} \\ \therefore \\ \mathbf{x}^T\left(\boldsymbol{\Phi}^T\mathbf{Q}\boldsymbol{\Phi} + \mathbf{R}\right)\mathbf{x} \ge 0, \forall\mathbf{x} \end{aligned} \tag{3.45}$$

Therefore having proved $\mathbf{H}$ is positive-semidefinite, the KKT conditions are as follows

$$\begin{aligned}\mathbf{Hz} + \mathbf{M}^T\boldsymbol{\lambda} + \mathbf{f} &= \mathbf{0}\\ \mathbf{Mz} + \mathbf{t} - \mathbf{b} &= \mathbf{0}\\ \mathbf{t} \geq 0, \quad \boldsymbol{\lambda} \geq 0, \quad \mathbf{t}^T\boldsymbol{\lambda} &= 0\end{aligned} \tag{3.46}$$

Equation 3.46 can be substituted into the mLCP problem, Equation 3.40, by using the following identities

$$\begin{aligned}\mathbf{N}_{11} = \mathbf{H},\, \mathbf{N}_{12} = \mathbf{M}^T,\, \mathbf{N}_{21} = \mathbf{M}\\ \mathbf{q}_1 = \mathbf{f},\, \mathbf{q}_2 = -\mathbf{b}\end{aligned}$$

resulting in the following equation

$$\begin{bmatrix}\mathbf{H} & \mathbf{M}^T & \mathbf{0}\\ \mathbf{M} & \mathbf{0} & \mathbf{I}\\ \mathbf{0} & \mathbf{T}^k & \boldsymbol{\Lambda}^k\end{bmatrix}\begin{bmatrix}\boldsymbol{\Delta}\mathbf{z}^k\\ \boldsymbol{\Delta}\lambda^k\\ \boldsymbol{\Delta}\mathbf{t}^k\end{bmatrix} = -\begin{bmatrix}\mathbf{Hz}^k + \mathbf{M}^T\boldsymbol{\lambda}^k + \mathbf{f}\\ \mathbf{Mz}^k + \mathbf{t}^k - \mathbf{b}\\ \boldsymbol{\Lambda}^k\mathbf{T}^k\mathbf{e} - \sigma^k\mu^k\mathbf{e}\end{bmatrix} \tag{3.47}$$

As the diagonal elements in $\boldsymbol{\Lambda}^k$ are positive, we can simplify the problem to a two step solution

$$\begin{aligned}\begin{bmatrix}\mathbf{H} & \mathbf{M}^T\\ \mathbf{M} & -\left(\mathbf{T}^k\right)^{-1}\boldsymbol{\Lambda}^k\end{bmatrix}\begin{bmatrix}\boldsymbol{\Delta}\mathbf{z}^k\\ \boldsymbol{\Delta}\lambda^k\end{bmatrix} &= -\begin{bmatrix}\mathbf{Hz}^k + \mathbf{M}^T\boldsymbol{\lambda}^k + \mathbf{f}\\ \mathbf{Mz}^k - \mathbf{b} + \sigma^k\mu^k\left(\boldsymbol{\Lambda}^k\right)^{-1}\mathbf{e}\end{bmatrix}\\ \boldsymbol{\Delta}\mathbf{t}_k &= -\mathbf{t}_k + \left(\boldsymbol{\Lambda}^k\right)^{-1}\left(\sigma^k\mu^k\mathbf{e} - \mathbf{T}^k\boldsymbol{\Delta}\lambda_k\right)\end{aligned} \tag{3.48}$$

A further simplification can be made to reduce the linear system size, but increase it to a 3-step solution

$$\begin{aligned}&\left(\mathbf{H} + \mathbf{M}^T\left(\mathbf{T}^k\right)^{-1}\boldsymbol{\Lambda}^k\mathbf{M}\right)\boldsymbol{\Delta}\mathbf{z}^k =\\ &\qquad -\mathbf{Hz}^k - \mathbf{M}^T\boldsymbol{\lambda}^k - \mathbf{f} - \mathbf{M}^T\left(\mathbf{T}^k\right)^{-1}\left(\boldsymbol{\Lambda}^k\left(\mathbf{Mz}^k - b\right) + \sigma^k\mu^k\mathbf{e}\right)\\ &\qquad \boldsymbol{\Delta}\lambda^k = \left(\mathbf{T}^k\right)^{-1}\left(\boldsymbol{\Lambda}^k\left(\mathbf{M}\boldsymbol{\Delta}\mathbf{z} + \mathbf{Mz} - \mathbf{b}\right) + \sigma^k\mu^k\mathbf{e}\right)\\ &\qquad \boldsymbol{\Delta}\mathbf{t}_k = -\mathbf{t}^k + \left(\boldsymbol{\Lambda}^k\right)^{-1}\left(\sigma^k\mu^k\mathbf{e} - \mathbf{T}^k\boldsymbol{\Delta}\lambda^k\right)\end{aligned} \tag{3.49}$$

Given forming and solving Equation 3.49 is the most computationally intensive step, particular focus will be spent on its implementation further on in this chapter.

3.4.3 Exploiting Rectangular Constraints

A common modification of the IIP algorithm is to separately treat decision variable rectangular bounds ($\mathbf{l}_\mathrm{b} \leq \mathbf{x} \leq \mathbf{u}_\mathrm{b}$) from the general linear inequality constraints. The advantage is that the number of inequality constraints is reduced, thus computations involving $\mathbf{M}$ require less operations, and convergence may be improved. Given the MPC formulation presented earlier in this chapter, bounds exist when constraints on the rate of change of the input ($\boldsymbol{\Delta}\mathbf{u}$) are imposed. If bounds are to be considered separately, then the QP problem is now

$$\begin{aligned} \min_{\mathbf{z}} \ & \frac{1}{2}\mathbf{z}^T\mathbf{H}\mathbf{z} + \mathbf{f}^T\mathbf{z} \\ \text{subject to: } & \mathbf{M}\mathbf{z} \leq \mathbf{b} \\ & \mathbf{l}_\mathrm{b} \leq \mathbf{z} \leq \mathbf{u}_\mathrm{b} \end{aligned} \tag{3.50}$$

where $\mathbf{l}_\mathrm{b}$ and $\mathbf{u}_\mathrm{b}$ are the lower and upper bounds ($-\boldsymbol{\Delta}\mathbf{U}_\mathrm{max}$ and $\boldsymbol{\Delta}\mathbf{U}_\mathrm{max}$ in the MPC case) respectively. The KKT conditions for this problem are

$$\begin{aligned} \mathbf{H}\mathbf{z} + \mathbf{M}^T\boldsymbol{\lambda} + \mathbf{f} + \mathbf{u} - \mathbf{l} &= \mathbf{0} \\ \mathbf{M}\mathbf{z} + \mathbf{t} - \mathbf{b} &= \mathbf{0} \\ \mathbf{z} + \mathbf{g} - \mathbf{u}_\mathrm{b} &= 0 \\ \mathbf{z} - \mathbf{s} - \mathbf{l}_\mathrm{b} &= 0 \\ \mathbf{t} \geq 0, \quad \boldsymbol{\lambda} \geq 0, \quad \mathbf{t}^T\boldsymbol{\lambda} &= 0 \\ \mathbf{u} \geq 0, \quad \mathbf{g} \geq 0, \quad \mathbf{u}^T\mathbf{g} &= 0 \\ \mathbf{l} \geq 0, \quad \mathbf{s} \geq 0, \quad \mathbf{l}^T\mathbf{s} &= 0 \end{aligned} \tag{3.51}$$

where we have introduced four new variables, $\mathbf{u}, \mathbf{l}$ are the Lagrange multipliers and $\mathbf{g}, \mathbf{s}$ are the slack variables for upper and lower bounds. The complementarity gap for this system is now defined as

$$\mu^k = \frac{\left(\boldsymbol{\lambda}^k\right)^T \mathbf{t}^k + \left(\mathbf{u}^k\right)^T \mathbf{g}^k + \left(\mathbf{l}^k\right)^T \mathbf{s}^k}{2n + m_c} \tag{3.52}$$

Following the steps in Section 3.4.1 the linear system involving inequalities and bounds to be solved at each iteration is now

$$\begin{bmatrix} \mathbf{H} & \mathbf{M}^T & \mathbf{I} & -\mathbf{I} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{M} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{I} & \mathbf{0} & \mathbf{0} \\ \mathbf{I} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{I} & \mathbf{0} \\ \mathbf{I} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & -\mathbf{I} \\ \mathbf{0} & \mathbf{T}^k & \mathbf{0} & \mathbf{0} & \mathbf{\Lambda}^k & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{G}^k & \mathbf{0} & \mathbf{0} & \mathbf{U}^k & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{S}^k & \mathbf{0} & \mathbf{0} & \mathbf{L}^k \end{bmatrix} \begin{bmatrix} \mathbf{\Delta z}^k \\ \mathbf{\Delta}\lambda^k \\ \mathbf{\Delta u}^k \\ \mathbf{\Delta l}^k \\ \mathbf{\Delta t}^k \\ \mathbf{\Delta g}^k \\ \mathbf{\Delta s}^k \end{bmatrix} = - \begin{bmatrix} \mathbf{Hz}^k + \mathbf{M}^T\boldsymbol{\lambda}^k + \mathbf{f} + \mathbf{u}^k - \mathbf{l}^k \\ \mathbf{Mz}^k + \mathbf{t}^k - \mathbf{b} \\ \mathbf{z}^k + \mathbf{g}^k - \mathbf{u}_\mathrm{b} \\ \mathbf{z}^k - \mathbf{s}^k - \mathbf{l}_\mathrm{b} \\ \mathbf{\Lambda}^k\mathbf{T}^k\mathbf{e} - \sigma^k\mu^k\mathbf{e} \\ \mathbf{U}^k\mathbf{G}^k\mathbf{e} - \sigma^k\mu^k\mathbf{e} \\ \mathbf{L}^k\mathbf{S}^k\mathbf{e} - \sigma^k\mu^k\mathbf{e} \end{bmatrix} \tag{3.53}$$

where

$$\mathbf{U}^k = \mathrm{diag}\left(u_1^k, u_2^k, ..., u_n^k\right), \quad \mathbf{G}^k = \mathrm{diag}\left(g_1^k, g_2^k, ..., g_n^k\right)$$
$$\mathbf{L}^k = \mathrm{diag}\left(l_1^k, l_2^k, ..., l_n^k\right), \quad \mathbf{S}^k = \mathrm{diag}\left(s_1^k, s_2^k, ..., s_n^k\right).$$

The most noticeable difference with Equation 3.53 and Equation 3.47 (inequalities only) is the size; adding bounds has more than doubled the linear system size. By substituting out the lagrange and slack variables as was done in Equation 3.49, we can reduce the system of equations to a seven step process

$$\begin{aligned} \left(\mathbf{H} + \left(\mathbf{G}^k\right)^{-1}\mathbf{U}^k + \left(\mathbf{S}^k\right)^{-1}\mathbf{L}^k + \mathbf{M}^T\left(\mathbf{T}^k\right)^{-1}\mathbf{\Lambda}^k\mathbf{M}\right)\mathbf{\Delta z}^k &= -\mathbf{y}_k \\ \mathbf{\Delta}\lambda^k &= \left(\mathbf{T}^k\right)^{-1}\left(\mathbf{\Lambda}^k\left(\mathbf{M\Delta z} + \mathbf{Mz} - \mathbf{b}\right) + \sigma^k\mu^k\mathbf{e}\right) \\ \mathbf{\Delta u}^k &= \left(\mathbf{G}^k\right)^{-1}\left(\mathbf{U}^k\left(\mathbf{\Delta z} + \mathbf{z} - \mathbf{u}_\mathrm{b}\right) + \sigma^k\mu^k\mathbf{e}\right) \\ \mathbf{\Delta l}^k &= \left(\mathbf{S}^k\right)^{-1}\left(\mathbf{L}^k\left(-\mathbf{\Delta z} - \mathbf{z} + \mathbf{l}_\mathrm{b}\right) + \sigma^k\mu^k\mathbf{e}\right) \\ \mathbf{\Delta t}_k &= -\mathbf{t}_k + \left(\mathbf{\Lambda}^k\right)^{-1}\left(\sigma^k\mu^k\mathbf{e} - \mathbf{T}^k\mathbf{\Delta}\lambda^k\right) \\ \mathbf{\Delta g}_k &= -\mathbf{g}_k + \left(\mathbf{U}^k\right)^{-1}\left(\sigma^k\mu^k\mathbf{e} - \mathbf{G}^k\mathbf{\Delta u}^k\right) \\ \mathbf{\Delta s}_k &= -\mathbf{s}_k + \left(\mathbf{L}^k\right)^{-1}\left(\sigma^k\mu^k\mathbf{e} - \mathbf{S}^k\mathbf{\Delta l}^k\right) \end{aligned} \tag{3.54}$$

where

$$\begin{aligned} \mathbf{y}_k &= \mathbf{Hz}^k + \mathbf{M}^T\boldsymbol{\lambda} + \mathbf{f} + \mathbf{u}^k - \mathbf{l}^k + \mathbf{M}^T\left(\mathbf{T}^k\right)^{-1}\left(\mathbf{\Lambda}^k\left(\mathbf{Mz}^k - b\right) + \sigma^k\mu^k\mathbf{e}\right) + \mathbf{g}_r^k - \mathbf{s}_r^k \\ \mathbf{g}_r^k &= \left(\mathbf{G}^k\right)^{-1}\left(\mathbf{U}^k\left(\mathbf{z}^k - \mathbf{u}_\mathrm{b}\right)\sigma^k\mu^k\mathbf{e}\ \right) \\ \mathbf{s}_r^k &= \left(\mathbf{S}^k\right)^{-1}\left(\mathbf{L}^k\left(-\mathbf{z}^k + \mathbf{l}_\mathrm{b}\right)\sigma^k\mu^k\mathbf{e}\ \right) \end{aligned}$$

Having admittedly made solving the problem larger and more complicated, the question then is: Have we improved the solution speed of the algorithm? The hypothesis is that there will be a payoff point; once the number of decision variables increase sufficiently then a speed improvement will be seen. By increasing the number of decision variables, the effective number of bounds constraints increase in $\mathbf{M}$, and therefore we can conclude the matrix-vector calculations will take longer. By substituting out the bounds and treating them separately, we will reap rewards despite the increase in computation steps.

To test this we have compared the performance of two solvers; `quad_wright`, our implementation of the IIP algorithm, and `qpip`, tested both with and without bounds. The result is four comparative runs across two interior-point solvers, using a set of random QPs generated from random MPC controllers. Each MPC controller was generated using a random stable discrete model with two inputs, two outputs and five states, using the MATLAB command `drmodel`. In addition, each controller was fully constrained ($\mathbf{\Delta u}, \mathbf{u}, \mathbf{y}$ all bounded), and setup with identity tuning weights. The prediction and control horizons are then varied between $N_p = 2, N_c = 1$ in steps of 2 (where $N_p = 2N_c$) to $N_p = 98, N_c = 49$, resulting in a series of 25 test controllers, each run 20 times and the results averaged.

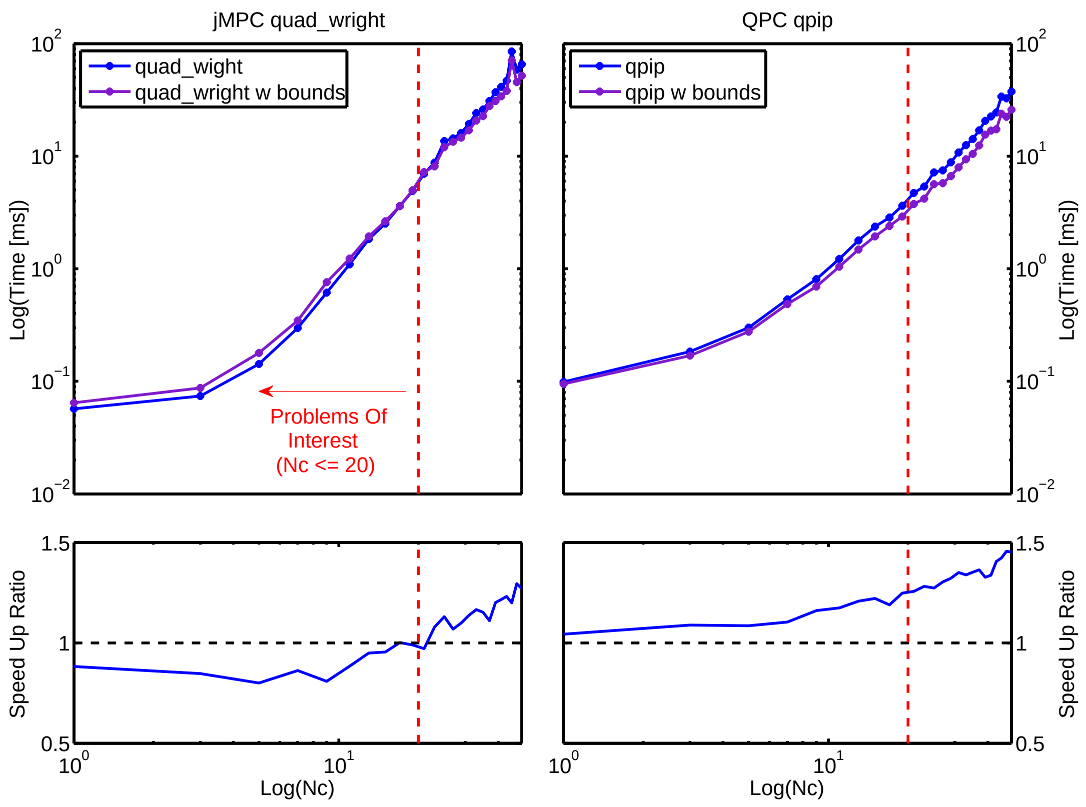


Figure 3.7: Log-Log bound exploitation comparison.

Figure 3.7 shows the results obtained using both `quad_wright` and `qpip`, including the ratio of speedup for exploiting (treating separately) the bounds. As expected, the `quad_wright` algorithm that did not treat bounds separately was faster for small problems (1-40 decision variables). In addition, the `quad_wright` algorithm that treated bounds separately was shown to be faster for larger problems (40+ decision variables). However the `qpip` algorithm did not show the same result, with treating the bounds separately always being faster than lumping them with the inequalities. Closer inspection of the `qpip` algorithm reveals that at each iteration it factorizes and solves a system of linear equations that includes $\mathbf{H}$ and $\mathbf{M}$ (i.e. similar to Equation 3.48) using a symmetric indefinite solver. Given the factorization and linear solver are the most expensive operations, and by removing a number of constraints from this system, thus reducing the size, the results shown for `qpip` in Figure 3.7 are expected.

Because the purpose of this work is to develop a QP solver for implementation on an embedded platform, the size of problems we will attempt to solve is limited. The vertical dashed red lines in Figure 3.7 show the upper bound of problems of interest, i.e. with less than 40 decision variables. Normally problems will be limited to sizes much less than this, say 10-20 decision variables, memory and speed permitting. Therefore it is clear the benefit of treating bounds separately with the `quad_wright` algorithm is not significant for the problems of interest, and thus modifications listed in this subsection will not be incorporated in the final algorithm.

In addition, the practical use of rate-of-change constraints ($\boldsymbol{\Delta}\mathbf{U}_{\max}$) is considered limited for the high-speed systems we are interested in controlling, where typically voltage is the control input and time constants of the implementation circuits far exceed that which is achievable by the proposed control algorithm.

3.4.4 Mehrotra's Predictor-Corrector Modification

One of the most significant contributions to interior-point methods in recent years was the predictor-corrector modification proposed by Sanjay Mehrotra [217]. This method forms the basis for most practical implementations of interior-point solvers, and thus investigation of its applicability to the proposed algorithm has been undertaken.

The core idea of the predictor-corrector algorithm is that the linear system factorization is the most expensive step, and thus if it can be reused, then potential computational savings can be realized. Moreover, by exploiting a higher order derivative of the Newton (affine-scaling) step, the prediction step length can be corrected by performing a second solve operation which utilizes this higher order information. The practical result is that the predictor-corrector algorithm requires more work per iteration, but less iterations are required to solve the problem. Therefore a balance again exists between the payoff of reduced iterations versus extra operations per iteration.

Before exploring this balance, the modifications to the IIP algorithm are presented to illustrate the extra work required. To begin with, Equation 3.47 is modified such that the complementarity gap μ and centering parameter σ are removed. This step is known as the prediction step:

$$\begin{bmatrix} \mathbf{H} & \mathbf{M}^T & \mathbf{0} \\ \mathbf{M} & \mathbf{0} & \mathbf{I} \\ \mathbf{0} & \mathbf{T}^k & \mathbf{\Lambda}^k \end{bmatrix} \begin{bmatrix} \mathbf{\Delta z}^k \\ \mathbf{\Delta}\lambda^k \\ \mathbf{\Delta t}^k \end{bmatrix} = - \begin{bmatrix} \mathbf{Hz}^k + \mathbf{M}^T \boldsymbol{\lambda}^k + \mathbf{f} \\ \mathbf{Mz}^k + \mathbf{t}^k - \mathbf{b} \\ \mathbf{\Lambda}^k \mathbf{T}^k \mathbf{e} \end{bmatrix} \tag{3.55}$$

Next, as with the original algorithm, the step-length α is chosen as the largest value within (0,1] that maintains the Lagrange multipliers and slack variables as positive. The next step uses a heuristic proposed by Sanjay for choosing the centering parameter σ

$$\sigma^k = \left(\frac{\left(\boldsymbol{\lambda}^k + \alpha^k \mathbf{\Delta}\lambda^k\right)^T \left(\mathbf{t}^k + \alpha^k \mathbf{\Delta t}\right)}{\left(\boldsymbol{\lambda}^k\right)^T \mathbf{t}^k} \right)^3 \tag{3.56}$$

which is used to keep the iterates on the central path (i.e. the path-following characteristic). The final modification is the real trick of Sanjay's work, the correction step

$$\begin{bmatrix} \mathbf{H} & \mathbf{M}^T & \mathbf{0} \\ \mathbf{M} & \mathbf{0} & \mathbf{I} \\ \mathbf{0} & \mathbf{T}^k & \mathbf{\Lambda}^k \end{bmatrix} \begin{bmatrix} \mathbf{\Delta z}^k \\ \mathbf{\Delta}\lambda^k \\ \mathbf{\Delta t}^k \end{bmatrix} = - \begin{bmatrix} \mathbf{Hz}^k + \mathbf{M}^T \boldsymbol{\lambda}^k + \mathbf{f} \\ \mathbf{Mz}^k + \mathbf{t}^k - \mathbf{b} \\ \mathbf{\Lambda}^k \mathbf{T}^k \mathbf{e} + \mathbf{\Delta \Lambda}^k \mathbf{\Delta T}^k \mathbf{e} - \mu^k \sigma^k \mathbf{e} \end{bmatrix} \tag{3.57}$$

What is obvious when comparing Equations 3.55 and 3.57 is that the left hand side matrix is identical. This means the Cholesky factorization performed in the prediction step can be reused in the correction step, and only a triangular substitution solver is required to solve the second linear system. Once the second system is solved, α is once again solved for, given the same requirements and Equation 3.41 used to update the iterates. Note modifications to the selection of the final step-length have been proposed (such as in [107]) however for this work it has not been found to noticeably affect algorithm performance.

Returning to the hypothesis of a payoff balance between reducing iterations and extra work per iteration, the two algorithms proposed so far have been compared over the same range of MPC problem sizes as detailed in Section 3.4.3. Figure 3.8 shows a Log-Log comparison of the two algorithms.

What is interesting about Figure 3.8 is the difference in payoff observed between a MATLAB implementation and a C-code implementation of both algorithms. Both `quad_wright` and `quad_mehrotra` have been implemented in both MATLAB and

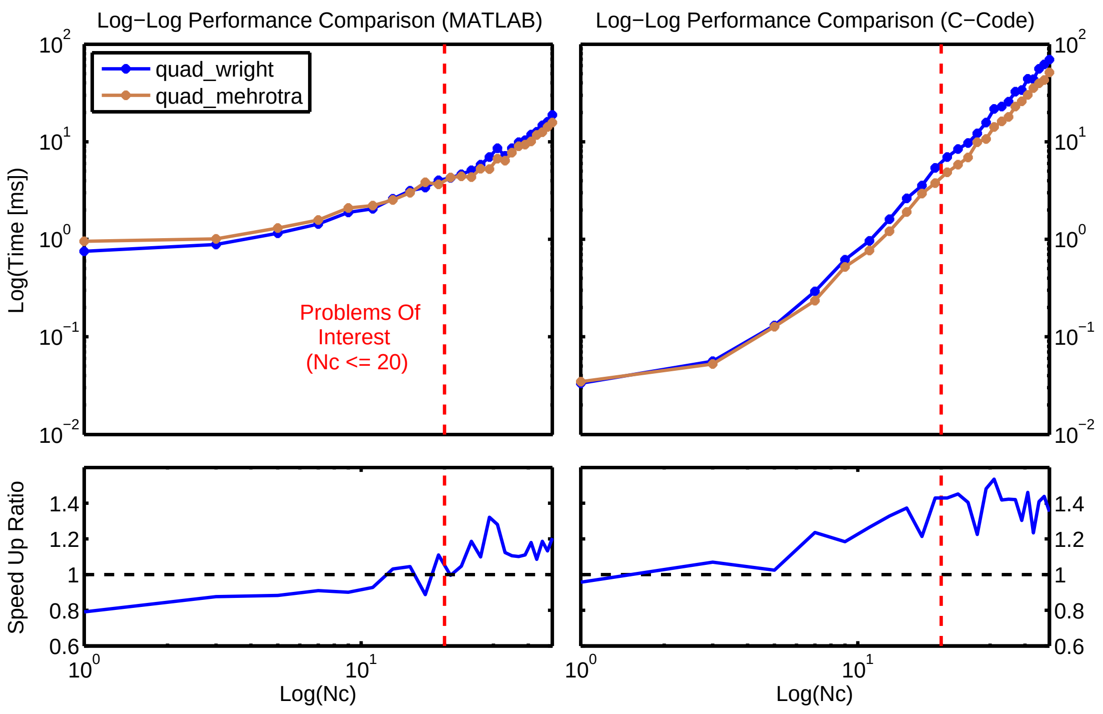


Figure 3.8: Log-Log `quad_wright` vs `quad_mehrotra` comparison.

C, and both obtain virtually identical results (down to `eps`), yet due to the differing linear algebra libraries we see a different crossover point. In MATLAB it's built in linear algebra libraries (BLAS/LAPACK) are utilized, while in C a custom library is used. The result of these two different libraries is that in MATLAB the crossover point exists at around 40 decision variables, while in C it is at 2 decision variables. The gradient differences between the two upper graphs show the effect of advanced linear algebra libraries which exploit parallelism, and the MATLAB code will scale much better with increasing problem size, but has an increased start-up cost in order to create and run the threaded calculations.

Given these results it is clear that Mehrotra's predictor-corrector modification is an advantageous addition to the IIP algorithm, as the average reduction in iterations is approximately 4. However for benchmarking purposes, both `quad_wright` and `quad_mehrotra` algorithms will be described throughout the rest of this work.

One final modification typical of practical implementations of the primal-dual interior-point algorithm is the use of multiple centrality corrections as proposed by Gondzio in [115]. Once again the aim is to reduce the number of iterations by utilizing higher-order corrections to improve the accuracy of each step. Currently this technique has not been investigated, but a small comparative study contrasting QPIP (which implements the IIP algorithm with Gondzio's modification) versus `quad_mehrotra` has been performed and the results shown in Figure 3.9. These indicate the performance gain of the modification is only appreciable at larger problem sizes, outside the range of interest for this work. Note the version of `qpip` used in Figure 3.9 was compiled against sequential BLAS/LAPACK, thus there is no threading overhead and the difference is assumed to be a result of the difference in algorithm (and perhaps to a lesser degree, implementation).

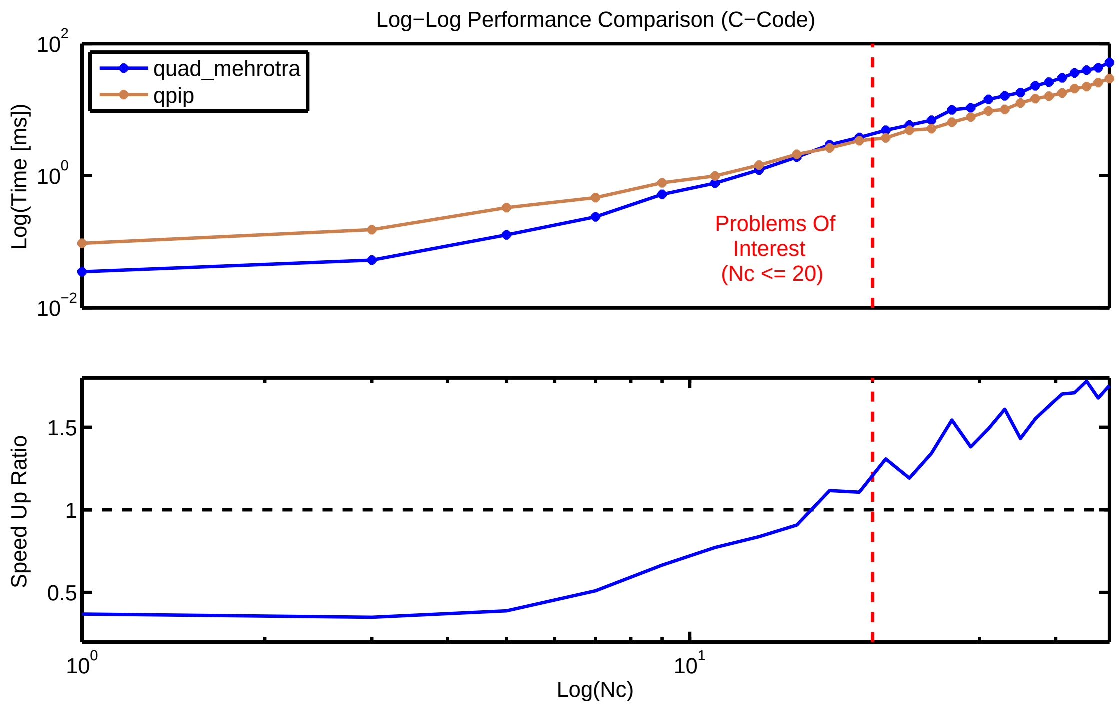


Figure 3.9: Log-Log `quad_mehrotra` vs `qpip` comparison.

3.4.5 Complete IIP QP Algorithm

Given the IIP algorithm and modifications present so far, the two QP algorithms developed in this work are described by the following steps.

3.4.5.1 `quad_wright` Algorithm

Step 1: Choose an initial set of iterates $(\mathbf{z}^0, \boldsymbol{\lambda}^0, \mathbf{t}^0)$ where $(\boldsymbol{\lambda}^0 > 0, \mathbf{t}^0 > 0)$.

Step 2: At the k-th iteration solve for the iterate increments using Equation 3.48/3.49.

Step 3: Solve for the step length α using Equation 3.42 then increment the iterates using Equation 3.41.

Step 4: Update the complementarity gap μ and centering parameter σ.

Step 5: Judge the convergence by comparing the complementarity gap μ and feasibility ϕ to 0. If it is within a specified tolerance, stop, otherwise continue.

Step 6: Check for infeasibility by checking for stalling iterates or if the feasibility ϕ is growing. If deemed infeasible stop, otherwise repeat from Step 2.

3.4.5.2 `quad_mehrotra` Algorithm

Step 1: Choose an initial set of iterates $\left(\mathbf{z}^0, \boldsymbol{\lambda}^0, \mathbf{t}^0\right)$ where $\left(\boldsymbol{\lambda}^0 > 0, \mathbf{t}^0 > 0\right)$.

Step 2: At the k-th iteration solve for the prediction iterate increments using Equation 3.55.

Step 3: Solve for the step length α using Equation 3.42.

Step 4: Update the centering parameter term using Equation 3.56, which is then used to update the right hand side of the linear system to solve.

Step 5: Solve for the correction iterate increments using Equation 3.57.

Step 6: Solve for the step length α using Equation 3.42 then increment the iterates using Equation 3.41.

Step 7: Update the complementarity gap μ.

Step 8: Judge the convergence by comparing the complementarity gap μ and feasibility ϕ to 0. If it is within a specified tolerance, stop, otherwise continue.

Step 9: Check for infeasibility by checking for stalling iterates or if the feasibility ϕ is growing. If deemed infeasible stop, otherwise repeat from Step 2.

3.4.5.3 Algorithm Summary

It is evident the algorithm is remarkably simple, which infers the implementation of the algorithm could also be quite simple. This is advantageous in that the program space required for the algorithm should also be relatively small. However, in order to be fast enough for the sampling rates required by MPC, the algorithm will require a novel implementation with a number of 'sneaky tricks' to exploit the structure of the problem being solved. The following subsections will detail these tricks and heuristics which have been used to accelerate the performance of both algorithms. The complete code listings for both the `quad_wright` and `quad_mehrotra` algorithms are presented in Appendix Section A.1.

For the remainder of this section only the `quad_mehrotra` algorithm will be described, as the implementation heuristics are near identical for the `quad_wright` solver.

3.4.5.4 Choice of Initial Iterates

The choice of initial iterates ($\mathbf{z}, \boldsymbol{\lambda}$ and $\mathbf{t}$) can drastically effect the operation of the solver, and thus care should be taken when selecting these vectors. It is also pointed out in [338] that the IIP algorithm has polynomial complexity when the starting iterates,($\lambda^0, \mathbf{t}^0$), are sufficiently large with respect to the initial residuals (given in Equation 3.40). A common heuristic modified for this work for warm starting these algorithms is to use the maximum absolute value in the problem data as follows

$$p_{\max} = \|H, f, M, b\|_\infty \tag{3.58}$$

where the infinity norm nomenclature is in this case used to represent the maximum absolute value for *both* matrices and vectors. The iterates are then initialized as follows

$$\begin{aligned}
\mathbf{z}^0 &= [0, 0, ..., 0]^T \\
\boldsymbol{\lambda}^0 = \mathbf{t}^0 &= \begin{cases} \left[\sqrt{p_{\max}}, \sqrt{p_{\max}}, ..., \sqrt{p_{\max}}\right]^T, & \text{if } p_{\max} > 1 \\ [0.5, 0.5, ..., 0.5]^T, & \text{otherwise} \end{cases}
\end{aligned}$$

Given the QP to be solved has been scaled (see Section 3.4.5.5) so the maximum element in each problem matrix/vector is 1.0, this results in the following initial iterates

$$\begin{aligned}
\mathbf{z}^0 &= [0, 0, ..., 0]^T \\
\boldsymbol{\lambda}^0 &= [0.5, 0.5, ..., 0.5]^T \\
\mathbf{t}^0 &= [0.5, 0.5, ..., 0.5]^T
\end{aligned}$$

The reasons for the above values are as follows:

- The primal variables ($\mathbf{z}$) are chosen such that the default change in control action ($\boldsymbol{\Delta}\mathbf{u}$) is zero. For a system operating at or close to steady state this will be the standard solution.
- The dual and slack variables are chosen such that they are relative to the size of problem data, with a minimum lower bound enforced. The lower bound is required so that particularly poorly scaled problems are not worsened by poor initial iterates, as described below. For typical problems the square root aims to bring the iterates closer to not active (0 for $\boldsymbol{\lambda}$) and active (a positive number relative in size to $p_{\max}$), providing a robust compromise between fewer iterations and numerical stability.

The above heuristic has been successfully implemented in a range of MPC studies, however this simple method is known as cold starting as it does not take into account our knowledge of the solution from the previous control input. In the case of optimal control problems it is common that the control input solution of each sample is similar (or even identical) to the previous sample's control input. Therefore by utilizing the solution of the previous sample as the initial solution guess for the current sample, it is possible to see a reduction in solver iterations. This is known as warm starting (or in some literature, hot starting) and is the process of providing a solution guess (in our case, $\mathbf{z}^0, \boldsymbol{\lambda}^0, \mathbf{t}^0$) from a previous solution run to the solver. In most circumstances this can reduce the number of iterations required by the solver to converge to the new solution.

As stated in [338], as well as in private communication with John Forrest (author of Clp) [98], warm starting interior-point solvers with the previous *primal* solution does not show a large improvement on convergence time. To illustrate this a series of 2D QPs were solved using different warm starting techniques, shown in Figure 3.10. It is acknowledged a larger system should be used for this comparison, however this would limit the ability to plot the solution path, which is of interest. What is

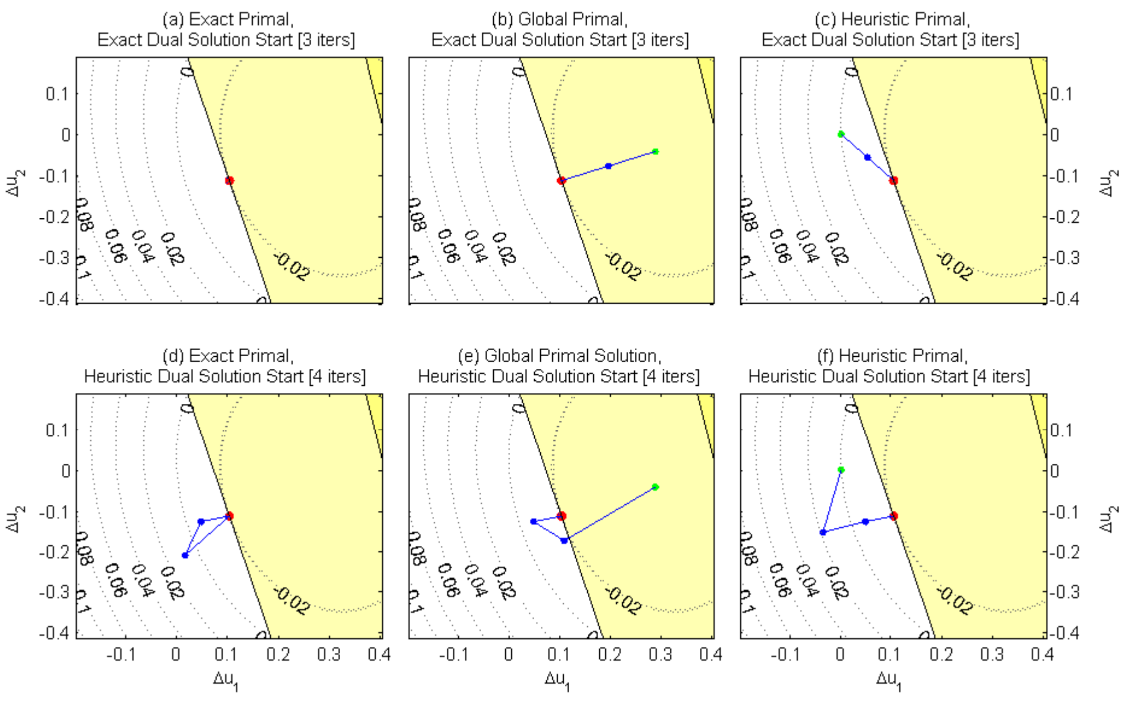


Figure 3.10: Comparison of warm starting methods. The red dot indicates the solution, the green dot the initial primal guess, and blue dots (and connecting lines) the trajectory taken by the primal iterates.

observed is that even when supplying the exact primal and dual solution ($\mathbf{z}$ and $\boldsymbol{\lambda}$), as shown in the plot (a), the solver still requires 3 iterations in order to ascertain optimality. This is attributed the need to solve the slack variables, $\mathbf{t}$, in order to validate the complementarity gap has reduced sufficiently.

The other plots show that for this small system, a range of solution trajectories are used by the solver depending on the initial starting conditions. The solution is found in the best case of 3 iterations, while in the worst case of 4 iterations. This result is similar to that of larger systems, albeit with larger numbers of iterations required for larger systems.

By critically looking at Figure 3.10, a representative plot of typical MPC quadratic programs with output constraints, we can make the following observations:

- Even supplying the exact primal solution does not guarantee the solution will be found in less iterations. Looking at the bottom left plot where we have used a heuristic to choose the dual variables, the exact primal solution does not help the solver isolate the solution in less iterations than a purely heuristic based approach (plot (f)). This result is typical across a range of problems and problem sizes.

- Utilizing the exact dual solution is often the best way to obtain reduced iterations, noting all three warm starting techniques that use the dual solution result in 3 iterations to a solution, regardless of the primal starting point. However it is often very difficult to determine the active constraints (and thus $\boldsymbol{\lambda}, \mathbf{t}$) for a given problem, and simply using the previous solution unmodified can lead to numerical problems (discussed below).

- Utilizing a global primal solution for warm starting does not add any noticeable benefit to the problem. In fact it is preferable that the previous primal solution is used which (it is hoped) is feasible. Additionally if the solver were to stop prematurely (say, due to exceeding maximum iterations), the current iterate should be feasible.

Given these observations, a natural conclusion might be to simply always warm start *both* the primal and dual (or perhaps even the slack as well) variables. Theoretically this may be adequate (with infinite precision), but in practice this can result in substantial numerical errors with a simple implementation. Recalling Equation 3.49 is used to solve for the current iterate increments, the following two operations show why these numerical errors can result

$$\begin{aligned} -\left(\mathbf{T}^k\right)^{-1}\mathbf{\Lambda}^k \\ -\mathbf{t}^k + \left(\mathbf{\Lambda}^k\right)^{-1}\left(\sigma^k\mu^k\mathbf{e} - \mathbf{T}^k\mathbf{\Delta}\lambda^k\right) \end{aligned}$$

As is typical in the optimal solution of an MPC QP, only a small number of the constraints will be active, which infers a large proportion of the elements in $\boldsymbol{\lambda}$ at the solution will be very close to 0 (and conversely the active constraints will have a slack variable of close to 0). As shown above, both the Lagrange and slack variable diagonal matrices are inverted as part of solving for the iterate increments, which can result in very large elements both within the matrix required to be factored, as well as in the right hand side for the triangular substitution solver. Simply supplying these previous dual and slack solution vectors as is (with values close to 0) can lead to substantial numerical errors in finite precision systems, and from our testing is attributed to the main source of numerical error of this algorithm.

A simple solution implemented within this work is to slightly bias the dual and slack variables if warm starting is enabled. After each solution, values within $\boldsymbol{\lambda}$ and $\mathbf{t}$ that are below a predefined tolerance are incremented by a fixed value to ensure numerical stability. For a scaled MPC problem (Section 3.4.5.5), variables that are less than 0.1 in value are incremented by 0.15, chosen based on numerical testing of the algorithms. Figure 3.11 shows the typical result of warm starting the `quad_mehrotra` solver with primal, dual and slack variables using this heuristic on a system with rate and output constraints. Commonly, the total number of iterations drops by around 15-20%, and the maximum iterations reduces by typically 2 or 3. Note however that

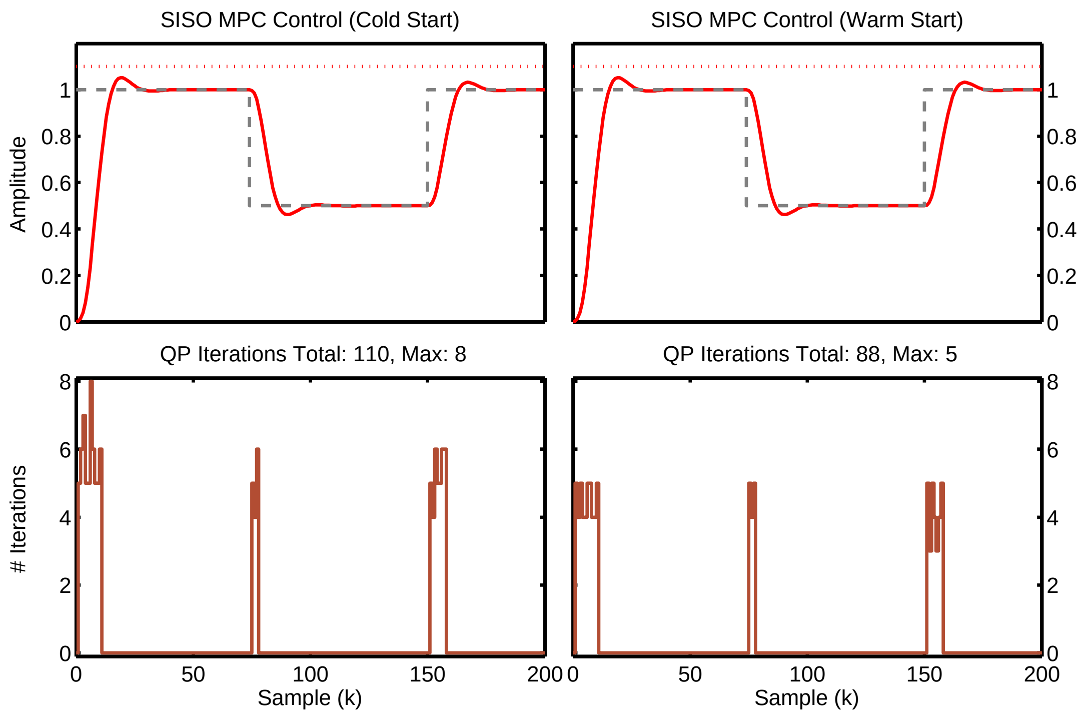


Figure 3.11: Cold start versus warm start MPC control. The setpoint is designated by the dashed grey line, while the upper output constraint is indicated by the dotted red line.

this result is not guaranteed, and depending on the tolerance and bias terms, warm starting can *decrease* performance by increasing the number of iterations required.

A final point on this subject is what to do when an extended number of controller samples have been run when the global, unconstrained minimum has satisfied the constrained minimum. As shown in 3.11 there are a large sections of the controller simulation where the QP is not required (due to example being at steady state), and therefore it is unlikely the previous QP solution is still valid. At the next step change or disturbance, the same constraints from the last QP solution will most likely not be active, and thus warm starting from this solution often degrades the solver solution. In this case the solver is simply cold started once one or more samples where the global solution has been used occurs.

3.4.5.5 System Scaling

The MPC formulation of the prediction matrices, as detailed in Equation 3.16, requires element-wise raising the state-space $\mathbf{A}$ matrix to increasing integer powers. Each time $\mathbf{A}$ is squared, cubed, etc, precision is lost in floating point, as the magnitude of the elements within the matrix increases, such as described in Section 4.2.2 of [326]. This problem is exacerbated in single precision, where there are less bits to represent the numbers, and can result in substantial numerical roundoff errors.

A particularly ill posed problem where this issue is quite obvious is a linearized Cessna model, presented by J. Maciejowski in [192] (this model is described in more detail later in Section 4.5.3.1). The continuous $\mathbf{A}$ matrix is as follows

$$\hat{\mathbf{A}} = \begin{bmatrix} -1.2822 & 0 & 0.98 & 0 \\ 0 & 0 & 1 & 0 \\ -5.4293 & 0 & -1.8366 & 0 \\ -128.2 & 128.2 & 0 & 0 \end{bmatrix}$$

Note the two larger elements in red, which are two orders of magnitude larger than the remainder of the elements. This has the effect of losing the precision of some rows, because the dynamic range required to store this matrix is very large. Using Maciejowski's tuning recommendations of $N_p = 10, N_c = 3$, together with the weighting and constraints indicated in the original reference, the QP $\mathbf{H}$ matrix is calculated as

$$\mathbf{H} = \begin{bmatrix} 29131758 & 22041268 & 16002114 \\ 22041268 & 16762968 & 12239335 \\ 16002114 & 12239335 & 8995198 \end{bmatrix}$$

where the numbers involved are huge relative to the calculated solution (which is in the range of -0.2 to 0.2). Disabling warm-starting in order to generate a side-by-side comparison, Figure 3.12 shows the result of appropriately scaling the problem (left hand side) versus a raw implementation (right hand side) in single precision.

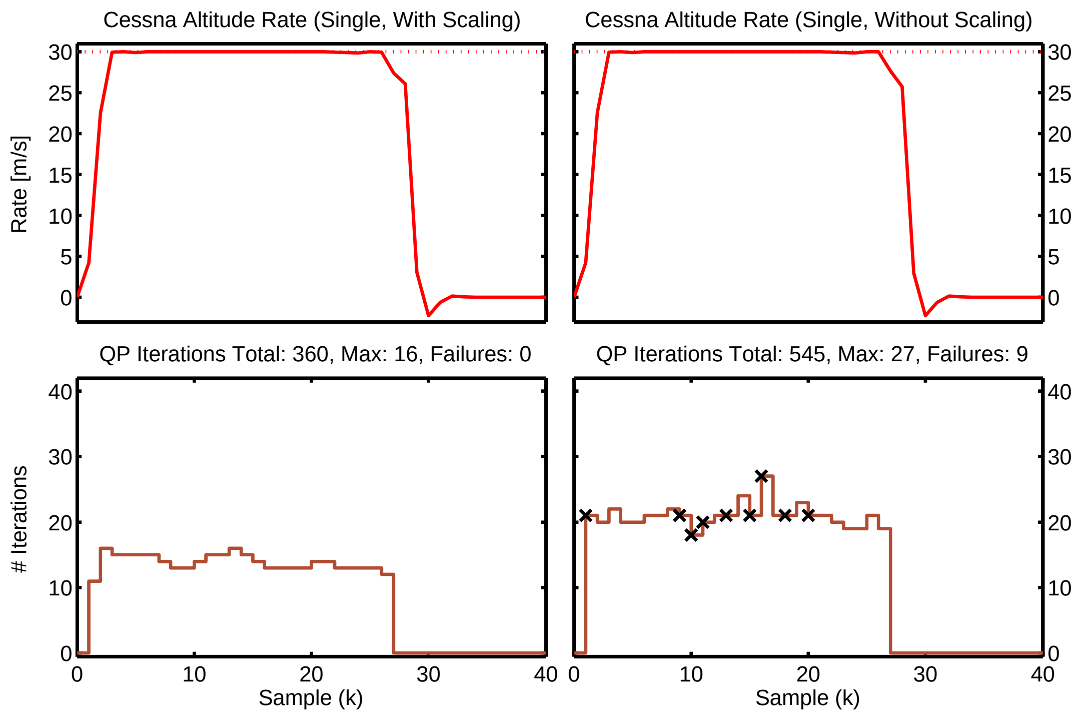


Figure 3.12: Single precision control of the Cessna altitude rate with and without scaling (crosses indicate QP failures).

While the controlled response of the altitude rate is similar between the simulations (due largely to the selection of sensible initial iterates described previously, together with modifications described in the next subsection), the amount of work required by the QP solver varies significantly. For the scaled system (with elements ranging from -10 to 10 in all problem matrices/vectors), the `quad_wright` algorithm has no issue, and solves the problem in a maximum of 16 iterations. This is in contrast to the unscaled simulation, where the solver fails on 9 iterations (23% failure rate, caused by numerical issues such as a negative α) and requires a maximum of 27 iterations. Not only does scaling nearly double the achievable sampling rate, but it has also significantly increased the robustness of the solver. The simple scaling step is implemented as

$$\text{scale}_{\text{H}} = \frac{\text{scale}_{\text{fac}}}{\|\mathbf{H}\|_{\infty}}, \quad \text{scale}_{\text{M}} = \frac{\text{scale}_{\text{fac}}}{\|\mathbf{M}\|_{\infty}} \tag{3.59}$$

where the infinity norm refers (in this work) to the maximum absolute value within the matrix and $\text{scale}_{\text{fac}}$ is a normalizing factor, typically in the range of 1 to 10, and the individual scaling factors are calculated for the objective and constraint matrices. These scaling factors are then element-wise multiplied to each respective problem matrix and vector, including modifications to these vectors as the algorithm progresses.

It is acknowledged a more advanced scaling algorithm would be preferable, such as an algorithm that would examine the conditioning of the original model state-space matrices and correctly scale them prior to formulating the quadratic program, rather than afterwards, such as done here. An example of two strategies include one by Liuping Wang where she describes an exponentially weighted cost function modification in [326], while Rossiter et al proposed an inner-loop state feedback stabilization strategy in [150]. In addition, using the current heuristic, the selection of $\text{scale}_{\text{fac}}$ does impact the robustness of the algorithm and thus should be investigated further. This work is currently being undertaken by a Master's student based on work presented in this chapter, and is discussed within future work in Section 8.3.1.

3.4.5.6 Exploiting Matrix Properties in the QP Algorithm

To calculate intermediate terms such as

$$-\left(\mathbf{\Lambda}^k\right)^{-1}\mathbf{T}^k$$

where we require a matrix inverse multiplied by another matrix can be expensive to evaluate. While there are several factorizations we could use to speed this up, we are much better off analyzing the creation of these matrices. For example, recall both $\mathbf{\Lambda}$ and $\mathbf{T}$ are defined as

$$\mathbf{\Lambda} = \operatorname{diag}\left(\lambda_1, \lambda_2, ..., \lambda_{m_c}\right), \quad \mathbf{T} = \operatorname{diag}\left(t_1, t_2, ..., t_{m_c}\right)$$

so they are both strictly diagonal. This means when calculating the inverse of $\mathbf{\Lambda}$ it is simply the inverse of each element along the diagonal, a substantial computational saving (and improvement in numerical conditioning) over a general full matrix inverse. In addition, multiplying two diagonal matrices together is very inefficient using a standard matrix-multiply routine. Therefore both $\mathbf{\Lambda}$ and $\mathbf{T}$ are treated as vectors for most of the computations, meaning this product is simply an element-wise multiplication (Hadamard product). Particular attention has also been paid to intermediate variables which are saved and then used further on in the code. This avoids unnecessary recalculation of these elements, and a cost of higher memory usage.

Further substantial computational savings can be made when evaluating the term

$$\left(-\left(\mathbf{\Lambda}^k\right)^{-1}\mathbf{T}^k\right)^{-1}\mathbf{M}$$

by again identifying the matrices inside the left hand term as diagonal. This reduces the equation to

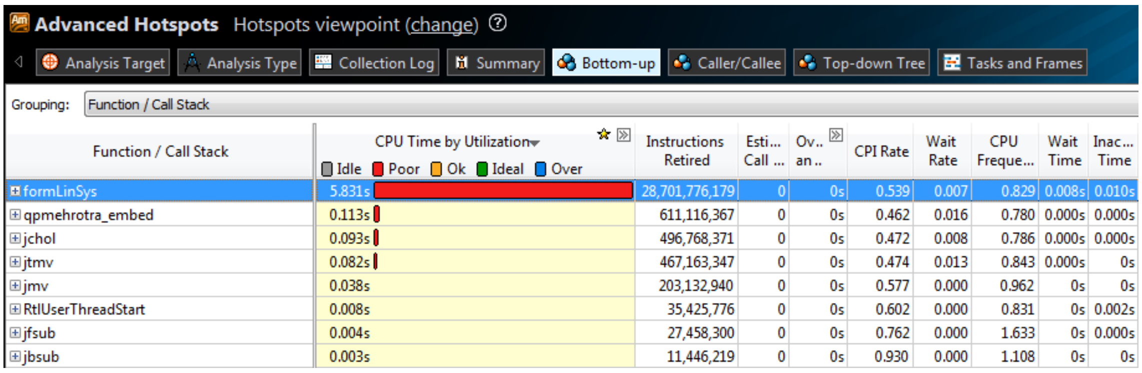
$$-\frac{\boldsymbol{\lambda}}{\mathbf{t}}\mathbf{M}$$

which when implemented using a banded matrix multiplier, again substantially reduces the computational time as typically the matrix $\mathbf{M}$ is very large. This technique is used in the MATLAB implementation of both `quad_wright` and `quad_mehrotra`.

Going one step further, we can examine the complete expression required to generate the linear system left hand side in Equation 3.49

$$\mathbf{H} + \mathbf{M}^T \left(\mathbf{T}^k\right)^{-1} \boldsymbol{\Lambda}^k \mathbf{M} \tag{3.60}$$

Benchmarking has shown that formulating this matrix can be the most expensive step of the algorithm, depending on the size of $\mathbf{M}$ and its implementation. Using the Intel V-Tune Amplifier to profile `quad_mehrotra` solving a large QP revealed just how expensive this step is, as shown in Figure 3.13.


Advanced Hotspots Hotspots viewpoint (change)

Analysis Target | Analysis Type | Collection Log | Summary | Bottom-up | Caller/Callee | Top-down Tree | Tasks and Frames

Grouping: Function / Call Stack

Function / Call Stack	CPU Time by Utilization (Idle, Poor, Ok, Ideal, Over)	Instructions Retired	Esti... Call ...	Ov.. an ..	CPI Rate	Wait Rate	CPU Freque...	Wait Time	Inac... Time
formLinSys	5.831s	28,701,776,179	0	0s	0.539	0.007	0.829	0.008s	0.010s
qpmehrotra_embed	0.113s	611,116,367	0	0s	0.462	0.016	0.780	0.000s	0.000s
jchol	0.093s	496,768,371	0	0s	0.472	0.008	0.786	0.000s	0.000s
jtmv	0.082s	467,163,347	0	0s	0.474	0.013	0.843	0.000s	0s
jmv	0.038s	203,132,940	0	0s	0.577	0.000	0.962	0s	0s
RtlUserThreadStart	0.008s	35,425,776	0	0s	0.602	0.000	0.831	0s	0.002s
jfsub	0.004s	27,458,300	0	0s	0.762	0.000	1.633	0s	0.000s
jbsub	0.003s	11,446,219	0	0s	0.930	0.000	1.108	0s	0s

Figure 3.13: Profile results of the `quad_mehrotra` solver on a large QP. The horizontal bars indicate the amount of time required by each subfunction within the algorithm (see Section A.1 for directions to the code listing), indicating formulating the linear system is the most expensive step.

The equation has a particular structure we can exploit, whereby not only are $\boldsymbol{\Lambda}$ and $\mathbf{T}$ diagonal, but we actually only require the lower triangle of the resulting matrix, given the factorization we will use to solve the system (described in the next subsection). Explicitly writing out this matrix-vector equation, assuming $\mathbf{H} \in \Re^{2\times 2}$ and $\mathbf{M} \in \Re^{3\times 2}$ and

$$\gamma = \frac{\boldsymbol{\lambda}}{\mathbf{t}}$$

results in the following expression

$$\begin{bmatrix} h_{11} + \gamma_1 m_{11}^2 + \gamma_2 m_{21}^2 + \gamma_3 m_{31}^2 & h_{12} + \gamma_1 m_{11} m_{12} + \gamma_2 m_{21} m_{22} + \gamma_3 m_{31} m_{32} \\ h_{21} + \gamma_1 m_{11} m_{12} + \gamma_2 m_{21} m_{22} + \gamma_3 m_{31} m_{32} & h_{22} + \gamma_1 m_{12}^2 + \gamma_2 m_{22}^2 + \gamma_3 m_{32}^2 \end{bmatrix}$$

As expected this matrix is symmetric (noting $\mathbf{H}$ is symmetric as well), therefore only the lower triangular elements need to be calculated and can be copied up if required. However in order to maintain the correct expression order, a symmetric matrix multiplier cannot be explicitly used, only the banded modification as already shown. Therefore a custom linear algebra routine has been written (`formLinSys`) which efficiently calculates this expression, by exploiting not only the symmetric property but also properties of all matrices involved. `formLinSys` is described further in Section A.4.2.1.

3.4.5.7 Linear Equation Solver

Once the linear system has formulated given the techniques described in the last subsection, calculating the solution of this system of linear equations forms the next largest portion of the computation time of each iteration of the algorithm. In order to maximise the efficiency of this step of the algorithm, we need to examine and exploit the properties of the linear system being solved.

By inspecting the linear system matrix in Equation 3.48, it is obvious this is a symmetric indefinite matrix. Even this simple property can result in noticeable improvements in the speed of the algorithm when using a factorization algorithm suited to symmetric indefinite matrices (such as an LU with partial pivoting). However it is possible to further simplify the matrix to restrict it to be positive-semidefinite, meaning we can use a basic Cholesky factorization and a simple triangular solver to solve the system of equations. The simplification converts the problem into a three step process, where not only are the linear system dimensions substantially reduced, but the remaining iterate increments can be solved directly without factorization from the proceeding steps, as detailed in Equation 3.49.

A standard Cholesky factorization is approximately twice as fast as a standard LU factorization, and has the added advantage of factorization stability without the need of pivoting [321]. Note the simplification in Equation 3.49 is made in [186], however the author chose to implement Gaussian elimination in order to solve the linear system, which is less efficient than solving via a Cholesky factorization. The matrix

$$\mathbf{H} + \mathbf{M}^T \left(\mathbf{T}^k\right)^{-1} \mathbf{\Lambda}^k \mathbf{M}$$

will remain positive-definite as $\mathbf{H}$ is restricted to be positive-definite (as discussed in Section 3.4.2), as well $\left(\mathbf{T}^k\right)^{-1} \mathbf{\Lambda}^k$ will also be positive-definite (based on all the elements of $\boldsymbol{\lambda}^k, \mathbf{t}^k$ being $> \mathbf{0}$). As discussed in [250] and shown in Equation 3.45, the resulting matrix will also be positive-definite, meaning a Cholesky decomposition can be used.

One major disadvantage of using a Cholesky decomposition is the requirement that the matrix is positive-definite, which while theoretically correct, in practice implementations in single precision have shown this does not necessarily always hold. While correctly scaling the system (and/or model) can normally correct this issue, on the odd occasion, during one of the many QP solver iterations during a simulation, a non-positive definite matrix can result from Equation 3.60. The problem is then what to do for the solution to the QP and whether precision can be restored.

This issue has been addressed by multiple practical methods, summarized in [340], however the two basic strategies are pivot skipping (ignore negative or zero eigenvalues), or replacing negative pivots with a large number as the algorithm progresses. While more advanced strategies exist (as described in further in [340]), these typically focus on sparse Cholesky algorithms and require reordering which is not as efficient or applicable for dense systems. For this work, both pivot skipping and replacing negative values with a large positive number were tested, however it was found pivot skipping typically resulted in the most robust compromise between increased solver iterations, and finding the correct solution.

To illustrate this point, the following output iteration snippet results from solving one of the unscaled single precision QPs using `quad_wright` from the Cessna problem in Section 3.4.5.5.

```
iter    phi          mu        sigma    alpha    max(r1)     max(r2)
 22   2.1092e-07  0.12248   0.18375  0.9735   1          2.3134e-06
 23   4.9677e-07  0.070755  0.00866  0.42607  11.5       9.177e-07 (C Fail)
 24   0.0057104   0.040334  0.19276  0.52181  1.663e+05  7.975e-07
 25   0.002158    0.019714  0.18524  0.62214  62866      1.8093e-06
 26   0.0015617   0.014838  0.11676  0.27601  45494      1.2946e-06
```

At iteration 23 the QP solver detected the Cholesky Factorization failed (a negative pivot was encountered) and the corresponding pivot was skipped. The C code snippet below shows how the Cholesky Factorization identifies this problem, and how the value is skipped while letting the main algorithm know a problem was encountered.

```c
if(s <= 0.0)
    status = -1; //not positive definite, pivot skipped
else
    A_jn[j] = sqrt(s);
```

Looking back at the iteration print out, the algorithm appears to stumble directly after the failed Cholesky (as indicated by looking at the maximum dual residual, `r1`), however it continues to converge and finds the correct solution by iteration 29. It is suspected the error in this example occurs with a loss of precision due to ϕ and μ not converging at the same rate. For problems where repeated Cholesky failures are encountered, the QP algorithm allows a maximum of 2 failures before exiting with a suitable numerical error flag. In addition, the algorithm reduces the maximum step size (α) once a Cholesky failure has been encountered, to ensure slower, hopefully more robust steps towards the solution. This problem has only been observed in single precision when the problem is close to infeasible, or on deliberately poorly scaled QPs in double precision. It is also much less likely to occur if the problem has been correctly scaled.

It is also worth noting that given this formulation the linear system matrix is also 100% dense, meaning there is no advantage to applying sparse techniques to this step.

3.4.5.8 Checking for Infeasibility

Without soft constraints (and even sometimes with these, depending on precision), the QP resulting from an MPC input calculation may become infeasible. This can be due to events such as excessive disturbances, poor tuning, or even the setpoint moving inadvertently outside the system constraints (see Figure 3.14 for an example). Once the QP problem becomes infeasible it is critical that this is identified by the algorithm so that either corrective action can be taken externally or the system can be shutdown. To correctly identify an infeasible problem, techniques described in [107] have been implemented and modified, specifically if

$$\text{iter} > 6 \text{ and } \frac{\|\mathbf{r}_2{}^k\|_\infty}{p_{\max}} > \epsilon_r \text{ and } \frac{|\mathbf{r}_2{}^k - \mathbf{r}_2{}^{k-1}|}{|\mathbf{r}_2{}^k|} < \epsilon_c \text{ and } \frac{|\mathbf{r}_2{}^{k-1} - \mathbf{r}_2{}^{k-2}|}{|\mathbf{r}_2{}^{k-1}|} < \epsilon_c \quad (3.61)$$

where

$$\mathbf{r}_2{}^k = \mathbf{M}\mathbf{z}^k - \mathbf{b} + \mathbf{t}$$

which is the primal residual, and ϵ_r and ϵ_c are predefined tolerances typically relative to the square root of the precision of the implementation, then the `quad_wright` algorithm determines the problem to be infeasible. Equation 3.61 requires that the algorithm has completed at least 6 iterations in order to avoid falsely identifying early infeasibility, as well as the primal residual is still significantly large with respect to the maximum problem element. The final condition checks to see if the primal residual has stalled, and if so, then infeasibility is declared.

For the `quad_mehrotra` algorithm a further condition was implemented which checked to see if the relative feasibility gap ϕ (also known as the duality gap) began to grow, which was found to be a good indication that the problem was infeasible. The relative feasibility gap is defined as

$$\phi^k = \frac{\|\mathbf{r}_1{}^k, \mathbf{r}_2{}^k\|_\infty + \left(\boldsymbol{\lambda}^k\right)^T \mathbf{t}^k}{\|\mathbf{H}, \mathbf{f}, \mathbf{M}, \mathbf{b}\|_\infty} \tag{3.62}$$

where $\mathbf{r}_1$ is the dual residual defined as

$$\mathbf{r}_1{}^k = -\mathbf{H}\mathbf{z}^k - \mathbf{M}^T\boldsymbol{\lambda}^k - \mathbf{f} \tag{3.63}$$

This is implemented using the same rules as Equation 3.61 but with two modifications: The required iterations is reduced to 4 (due to decreased iterations required by the algorithm) and the following extra conditions added

$$\phi^k > \phi^{k-1} \text{ and } \phi^{k-1} > \phi^{k-2} \tag{3.64}$$

which were found to correctly identify typical infeasible problems. Most practical implementations of an interior point algorithm will also consider the dual residual $\mathbf{r}_1{}^k$, which can indicate if the problem is unbounded below. However as the QP results from an MPC formulation and can be tested under a variety of operating conditions before implementation, it is highly unlikely that the practical real-life problem will be unbounded.

An advantage described earlier in Section 3.4.1 of using an *Infeasible* Interior-Point method is that it can still attempt to solve problems, even if infeasible. This is opposed to active-set methods which typically require an initial feasible solution, or a standard interior-point method. Figure 3.14 shows the controlled response of a 3-DOF Helicopter (simulation only) as described in [7], subjected to a large setpoint change with tight pitch axis constraints. The result is an infeasible QP during the middle of the manoeuvre for approximately 22 samples. In this example no external corrective action is taken, and the controller and solver are left to try to rectify the broken constraint. As shown in the comparison plot, the `quad_wright` solver appears to handle the infeasibility quite well, minimizing the infeasibility and thus the amount of time the constraint is violated. This is in direct contrast to CLP (also an interior-point solver, but using a different formulation), which completely loses control of the system, and alternates between hitting the maximum iterations (1500) and detecting infeasibility.

While this example is rather academic, as the natural motion of the helicopter corrects for the infeasibility and brings the system back under stable control, it

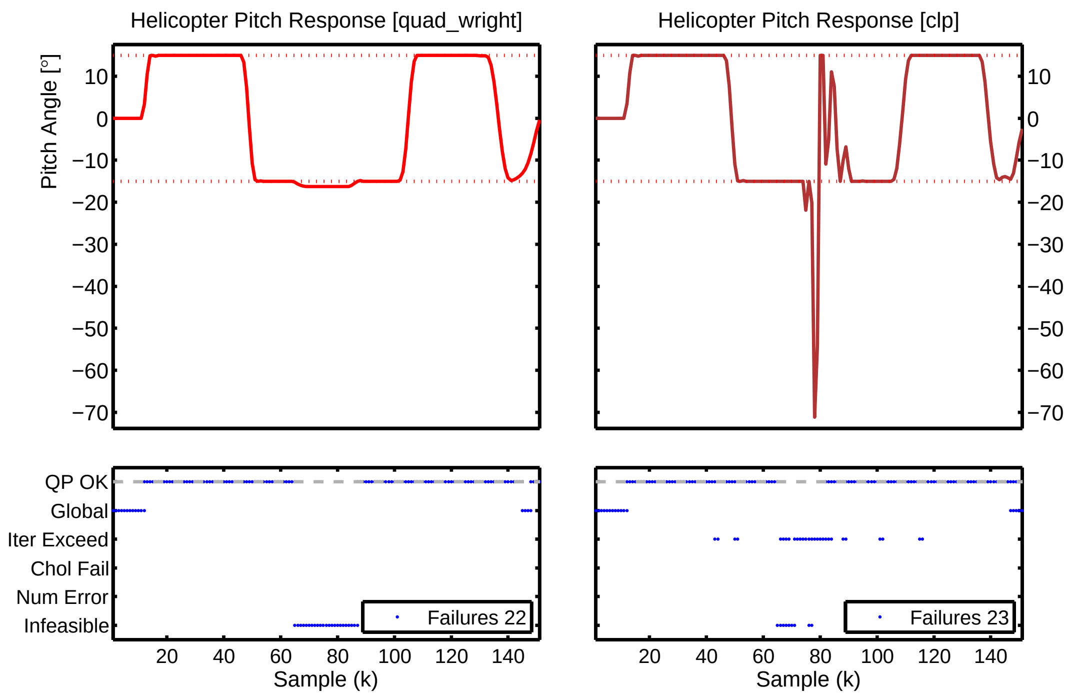


Figure 3.14: Control comparison of an infeasible 3-DOF Helicopter simulation (`quad_wright` vs `clp`), including solver status codes.

does show that under infeasible conditions the infeasible part of this algorithm is advantageous to maintaining partial control. The example also shows the algorithm correctly identifies infeasibility, and exits before numerical errors are encountered. In practice, using single precision, an infeasible problem is more likely to be identified as multiple Cholesky factorization failures, but generally the controller retains reasonable control, as shown in Figure 3.15. Note the single precision solver has over double the failure rate as the double precision solver, but most of these are status code -3, indicating numerical errors were encountered. This is typically due to a loss of precision and the solver cannot find a better solution without a numerical error such as α tending negative, or a NaN encountered. Both of these conditions are caught early on to avoid numerical errors propagating from the solver. In practice the solution found is sufficiently accurate to retain adequate control even when constraints are active.

It is acknowledged a more rigorous treatment of the infeasible constraints should be considered, such as removing the infeasible rows and re-solving the quadratic program in order to retain control, however the point of this example is to demonstrate the behaviour of the proposed algorithm when presented with an infeasible problem. As shown, the performance (in this example) is far superior to alternative solvers, which is attributed to the inherent infeasible component of the IIP algorithm.

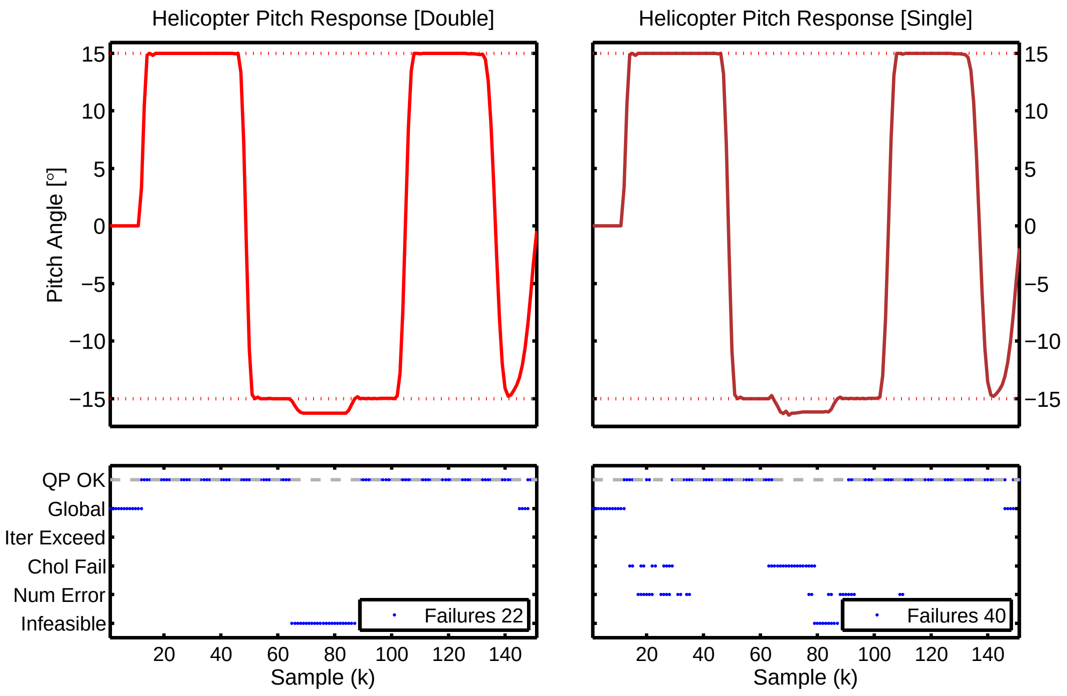


Figure 3.15: Control comparison of an infeasible 3-DOF Helicopter simulation (double versus single using `quad_wright`).

3.4.5.9 Termination Conditions

The successful termination condition of the algorithm relies on comparing two key calculated variables, μ, the complementarity gap (Equation 3.37), and ϕ, the relative feasibility gap (Equation 3.62) to a predefined tolerance, ϵ_r (both variables are compared to the same tolerance). As the algorithm converges to a feasible optimal solution, both μ and ϕ will approach 0 as the primal and dual infeasibilities are also reduced to 0. Therefore a simple comparison of μ and ϕ with a pre-specified tolerance is all that is required to judge the completion of the algorithm. For typical MPC applications this can be set as

$$\mu_{\text{final}} = \phi_{\text{final}} = \epsilon_r = \sqrt{\epsilon} \tag{3.65}$$

where ϵ is the floating point precision (`eps` in MATLAB) of the system on which the solver is implemented. Simply testing μ in isolation does not give a sufficient metric of solution optimality because it is an absolute term, and as the problems are typically scaled, a relative tolerance term is also required. A point to note is that relaxing of the tolerance (i.e. $\mu_{\text{final}} = \phi_{\text{final}} = 10^{-3}$) often has little effect on the overall control performance of the system, given the approximation of the model in the first place and quantization present with typical data acquisition and control systems.

3.4.6 Performance Results

In order to validate the performance of the algorithm a series of tests have been run to benchmark the `quad_wright` and `quad_mehrotra` algorithms against two reference implementations of the interior-point predictor-corrector QP algorithm, the commercial solver `quadprog` [207] and the free, closed source solver `qpip` [330]. For each test, where possible, comparisons will be drawn across the following three areas, all with respect to problem size:

Speed The amount of time taken to solve each problem, run over 25 different problem sizes and averaged over 20 problems at each size. This is the most important attribute of the solver, given the focus on high-speed control. It is also important to see the relationship between average solution time and problem size, so we can predict whether the solver can calculate the required control move(s) within the sampling time available.

Memory Dynamic memory, especially RAM, is very limited within typical small microcontrollers (the target of this algorithm), thus the algorithm must be as memory efficient as possible. It is not possible to easily compare the memory requirements of the other solvers, however the memory requirements for the algorithms developed in this work will be reported. As a reference value, the target microcontroller for this work (described later in Section 4.2.1.3) has only 256KB of RAM.

Accuracy How accurate the solution is, again averaged over 20 problems from 25 sizes. While not required to be highly accurate (such as would be required for parameter estimation or economic calculations), reasonable accuracy (approximately 10^{-3}) is required to ensure the QP solver itself converges and the resulting control moves are still near-optimal. Also note the control moves will be subject to quantization via a Digital to Analog Converter (DAC), so only one or two decimal points of accuracy are realistic.

In addition, rather than use general benchmarking QPs such as those used in the Hans Mittelmann benchmarks [219], we have used a series of randomly generated QPs resulting from the creation of arbitrary MPC controllers, as described in Section 3.4.3. These will stress the `quad_wright` and `quad_mehrotra` algorithms on problems they are designed to solve providing a better indication of their performance.

Shown in Figure 3.16 is a log-log plot of the timing results comparing the four solvers. This figure provides an insight into the expected algorithm performance as the problem size increases. As shown, the timing measurements for all solvers form approximately a straight line on a log-log plot as the problem size increases, indicating they can be expected to have a monomial (or power) relationship between execution time and N_c. Table 3.3 shows the expected execution time power factor with respect to N_c for each solver.

Table 3.3: Power relationship for each IP solver (the gradient of Figure 3.16).

Solver	Power Term
`quad_mehrotra`	2.23
`quad_wright`	2.29
`quadprog`	1.13
`qpip`	1.42

For the solvers developed in this work, `quad_wright` and `quad_mehrotra`, the power term is approximately a quadratic, while for `quadprog` is approximately linear, and `qpip` is in the middle. This indicates the two reference solvers would be expected to scale better as the problem gets larger, which reflects their use of multithreaded linear algebra libraries (BLAS/LAPACK) as well as a more advanced algorithm. However the focus of this work is not on large-scale problems and as noted on the plot, the maximum N_c is expected to be around 20 (corresponding in this example to 40 decision variables and 320 inequality constraints), therefore for most of the region of interest, `quad_mehrotra` is superior in terms of speed.

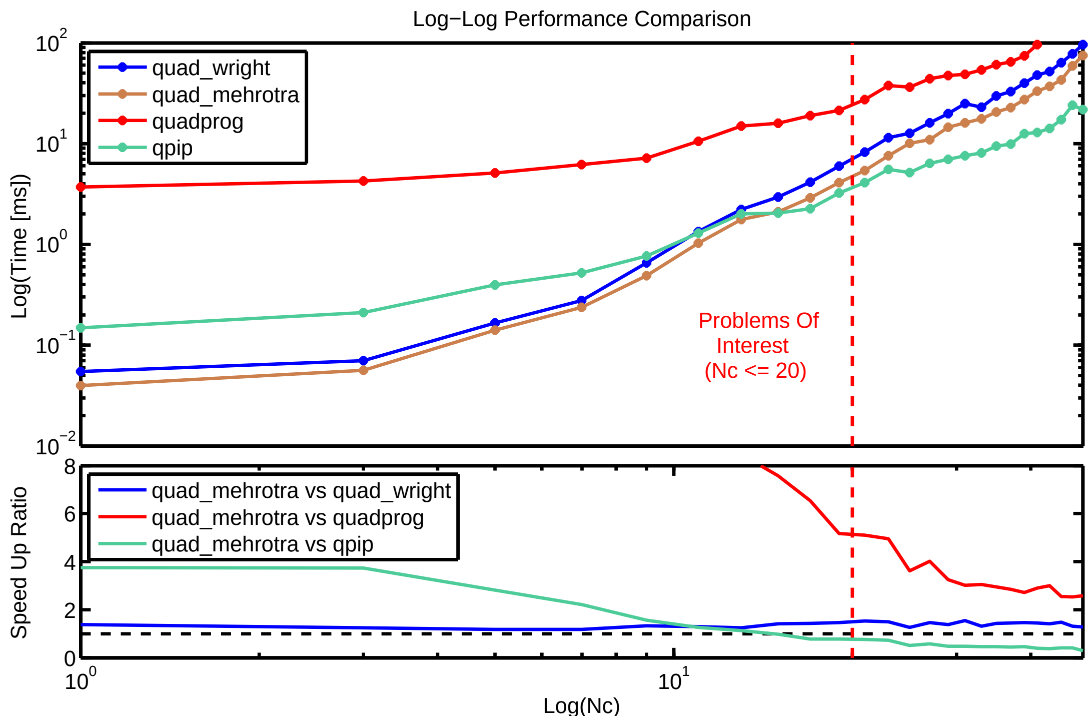


Figure 3.16: Algorithm timing comparison for the surveyed interior point implementations.

With regards to accuracy, Figure 3.17 shows the relative accuracy of the quad_wright algorithm as a function of N_c, taking Cplex as the reference solution (quadprog was found to give remarkably different results from all other solvers, even with the same tolerance of 10^{-7}). Both quad_wright and quad_mehrotra were set up with an

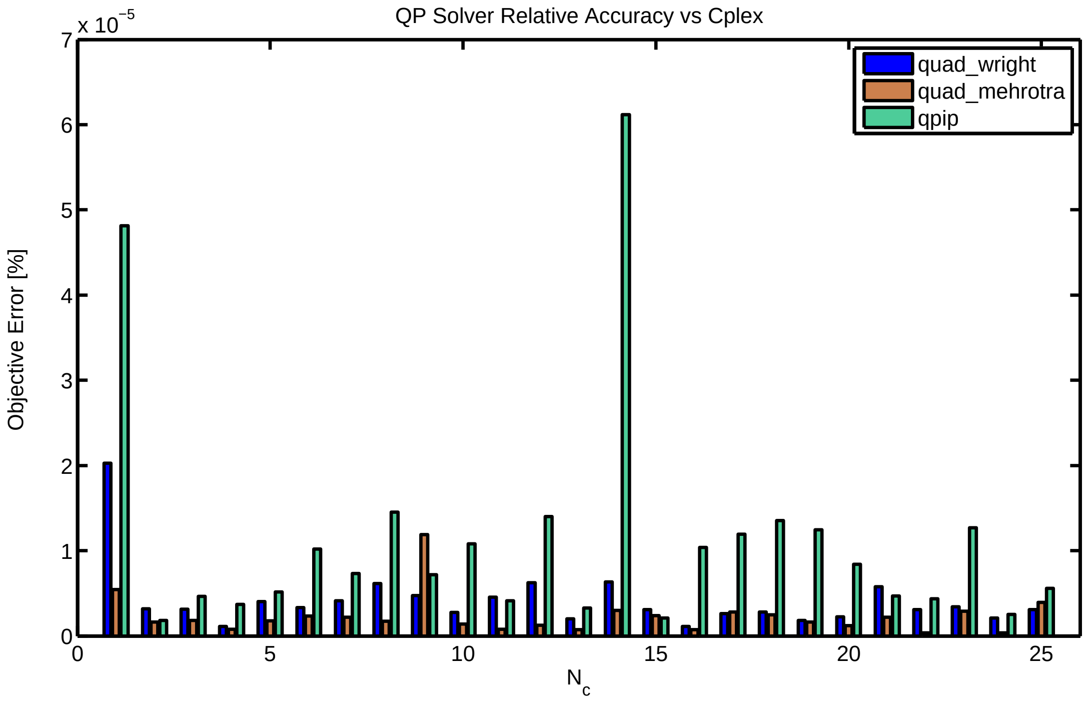


Figure 3.17: Algorithm accuracy comparison.

identical tolerance of 10^{-7}, and as can be seen, achieved a relative error on average of 0.00002%, and a maximum relative error of 0.00007% (average for both solvers). These results shows that both of the algorithms developed in this work are reliably calculating the correct solution across a range of typical MPC problems, and for most of the test problems, more accurately than both quadprog and qpip.

In addition to speed and accuracy, memory usage has also been measured of the quad_wright and quad_mehrotra solvers within C-code implementations. This was not directly possible with any of the other solvers, thus no comparison could be made. However embedded systems typically have very limited on-chip RAM (16KB-256KB for a basic 32bit microcontroller) thus if the algorithm and associated problem data does not fit in memory, control performance must be sacrificed by reducing N_p and/or N_c in order to reduce the algorithm memory footprint. Alternatively off-chip RAM can be used, however typically a time penalty will be incurred, decreasing the achievable sampling rate, as well as complicating the embedded system. Given the current increment in problem size, as described in the timing results, the memory requirement is approximately a quadratic function of N_c for storing the problem data and all required intermediate variables in single precision, as shown in Figure 3.18.

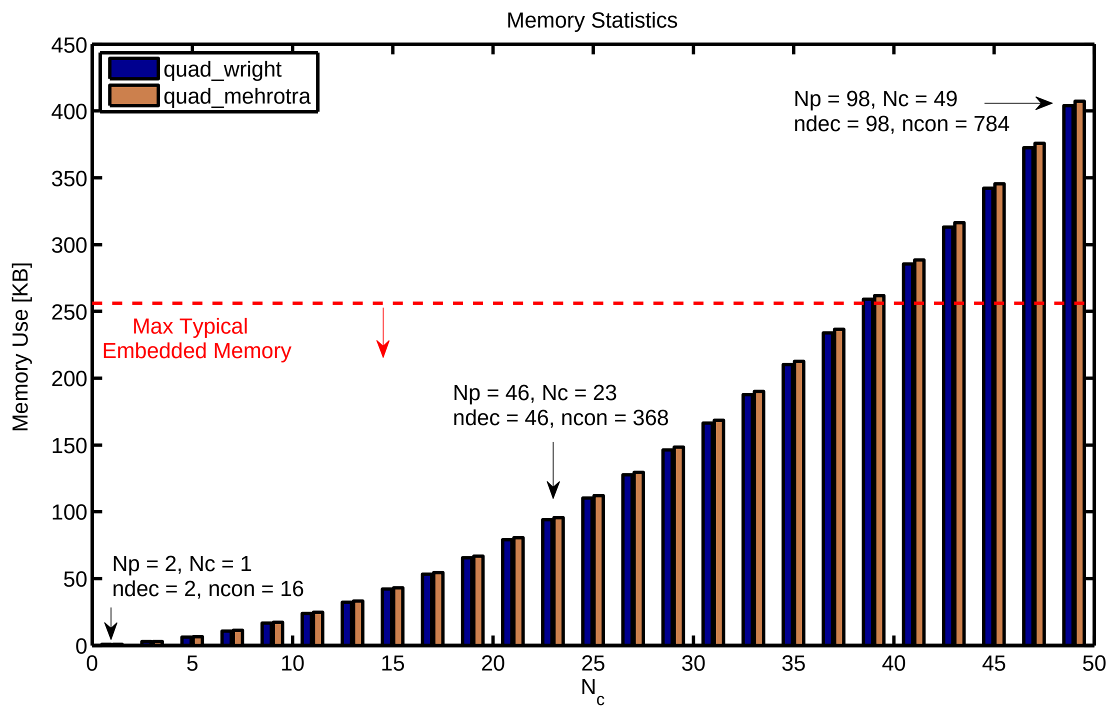


Figure 3.18: `quad_wright` and `quad_mehrotra` algorithm memory requirements.

More information on how the memory requirements of the algorithm are calculated is presented in Section 4.3.

3.4.7 QP Solver Development Summary

To briefly summarize the contribution so far, we have successfully developed two QP solvers based on the Infeasible-Interior-Point framework introduced by Stephen Wright, with one incorporating the predictor-corrector modification proposed by Sanjay Mehrotra. In addition, several implementation heuristics have been proposed which result in a high-speed, low memory-footprint and highly accurate QP solver, suitable for implementation on a range of embedded systems in both single and double floating point.

To briefly demonstrate the performance achievable using the algorithms presented, two MPC case studies are presented from literature.

3.4.7.1 Control of a Linearized Cessna

The complete simulation results of the linear Cessna model presented in Section 3.4.5.5 from [192] are shown in Figure 3.19. The maximum achievable sample rate (including all MPC calculations) on a Windows i7 2.8GHz Laptop is over 30kHz using the ANSI C implementation, when solved with the `quad_mehrotra` solver in double precision and a termination tolerance of $\epsilon_r = 10^{-6}$. The complete controller implementation requires 11.2KB to store all problem data and intermediate variables, with only 3.8KB required for the QP solver. The QP at each iteration consists of 3 decision variables and 42 constraints.

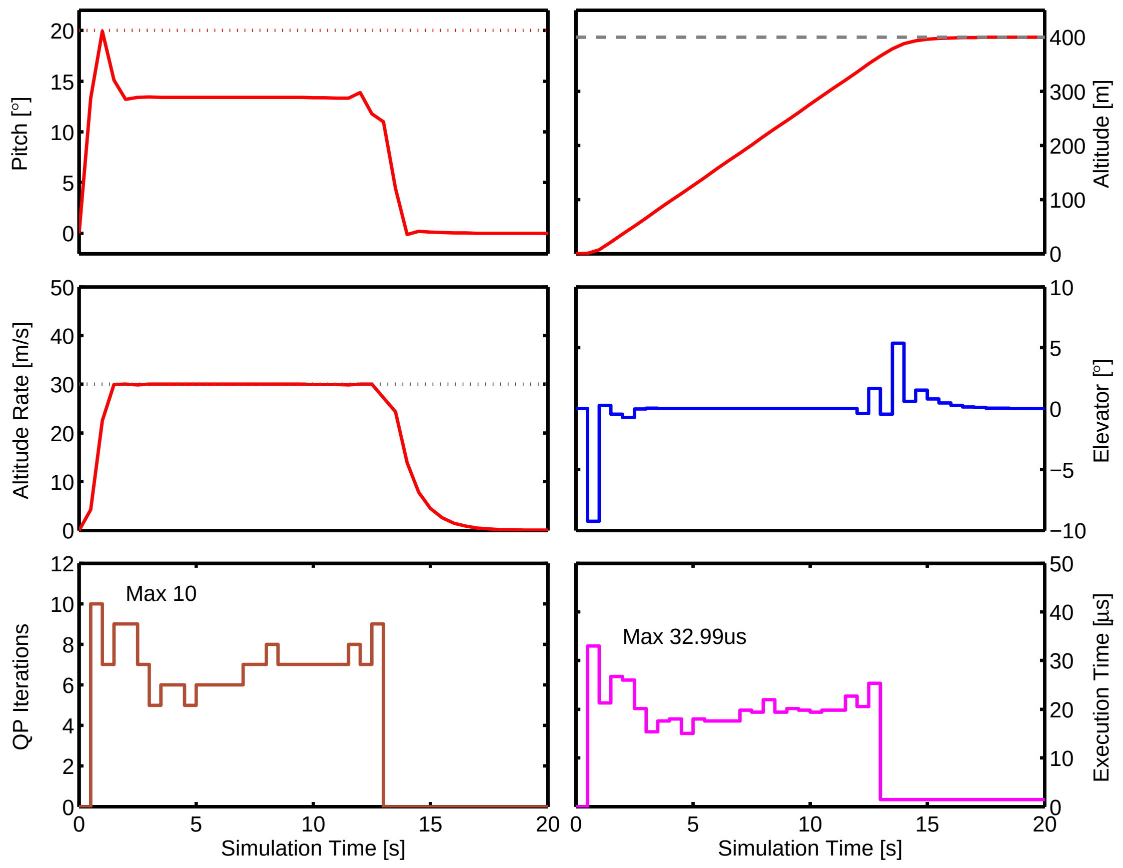


Figure 3.19: Cessna model MPC simulation. The three outputs (in red) are pitch angle of the aircraft, rate of altitude ascent/descent, and actual altitude. The input is the elevator angle (blue), while the number of QP iterations and execution time are shown in brown and pink respectively. The dotted lines indicate output constraints, while the dashed grey line is the setpoint.

When solved via MATLAB as the control engine (the MPC controller is implemented in MATLAB, the QP solver in compiled C, thus allowing QP solvers to be benchmarked), a comparison of the maximum achievable sample rate is shown in Table 3.4. Note that as the simulation is solved via MATLAB and not compiled C code, the times recorded are significantly slower than in Figure 3.19. In order to remove the effect of varying processor speed, each run was repeated 20 times and the results shown below are the average.

Table 3.4: Cessna simulation maximum sample rate comparison in MATLAB.

Solver	Maximum Sample Rate (Hz)
`quad_mehrotra`	956
`quad_wright`	923
`qpip`	797
`quadprog`	130

3.4.7.2 Control of a 3-DOF Helicopter

The complete simulation results of the 3-DOF Helicopter model presented in Section 3.4.5.8 from [7] are shown in Figure 3.20. The maximum achievable rate in C for this example is over 3kHz, when solved using `quad_mehrotra` and the same tolerance as described in the above example. The complete controller implementation requires 46.4KB to store the QP with 21 decision variables and 90 constraints, including all problem data for both the MPC and QP algorithms.

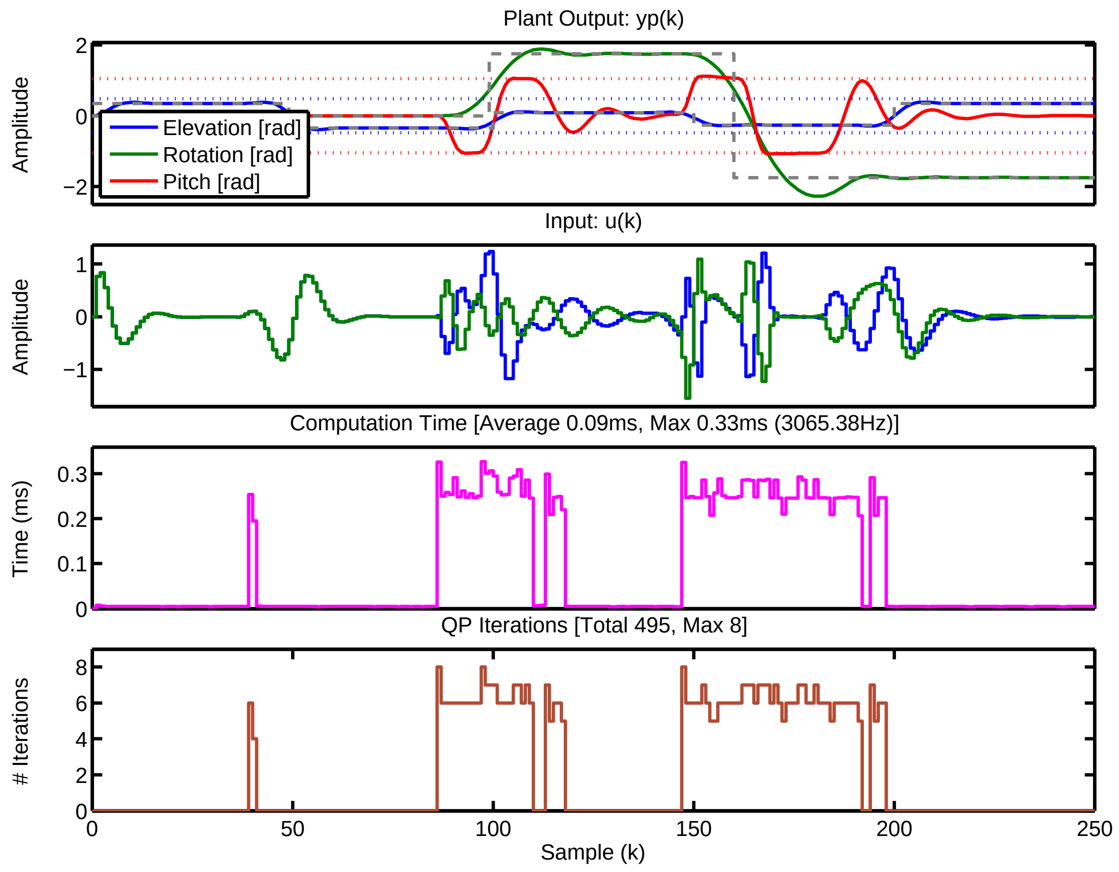


Figure 3.20: 3-DOF helicopter model MPC simulation.

A comparison of the maximum sample rates is shown in Table 3.5, once again showing the two solvers developed in this work are superior in terms of speed. Note `clp` was also compared but failed completely on this problem, as did `qpip` at two difficult control samples (hence the slow sample rate).

Table 3.5: Helicopter simulation maximum sample rate comparison in MATLAB.

Solver	Maximum Sample Rate (Hz)
`quad_mehrotra`	687
`quad_wright`	602
`qpip`[1]	372
`quadprog`	80

[1] Failed on 2 samples

3.5 MATLAB Based MPC Simulation Tools

In order to automate the generation of the MPC and resulting QP problem, together with providing a simulation and testing environment, MATLAB [205] was chosen as the development environment. This is based on powerful dynamic simulation and control functionality built into the software (Simulink, Control Systems Toolbox), as well as its ability to easily generate and manipulate the required matrices of the MPC problem.

Several packages for MPC simulation within MATLAB already exist, however for reasons described next, these were not suitable for the remainder of this work. Each of the packages are briefly reviewed below, before the jMPC Toolbox is introduced in Section 3.5.2, the software tool developed to implement the high-speed MPC algorithm.

3.5.1 Existing MATLAB Linear MPC Tools

MATLAB MPC Toolbox The MATLAB MPC Toolbox [206] is a commercial software tool for building, simulating and deploying linear MPC controllers. It was written primarily by Alberto Bemporad and Manfred Morari (ETH), with contributions from N. Lawrence Ricker who wrote the QP solver. The toolbox is a flexible implementation of MPC, allowing customization and implementation of a range of constraints, tuning weights and advanced parameter such as state estimation. Being a commercial package it was difficult to modify for the research purposes of this work, and thus was only used as a validation tool to ensure our algorithm was correct.

MPCtools One of the best free packages for linear MPC is MPCtools [7], a software package written by Johan Åkesson (Lund). While now quite dated (2006) and never subsequently updated, it supported many of the features required for this work including state estimation, customizable QP solvers and a reference tracking algorithm. Solely written in MATLAB, it did not support code-generation for deployment, but did provide an S-Function for Simulink MPC control. MPCtools was used and referenced frequently when developing the jMPC Toolbox, in order to customize our desired implementation.

YALMIP Written by Johan Löfberg (formerly ETH Zürich, now Linköping), YALMIP (Yet Another Linear Matrix Inequality Program) [189] is an open source modelling language and software tool for the modelling and solution of convex (and more recently, nonconvex) optimization problems. In addition, using the YALMIP modelling language, MPC controllers with arbitrary objective functions and constraints can be built and simulated in MATLAB using a range of competitive open source and commercial QP solvers. The fundamental reason YALMIP was not used was because the modelling language YALMIP supported abstracted the problem too far from an implementable form, given the low-level (C code) goal the project required.

Multi-Parametric Toolbox (MPT) MPT [177, 127] is the result of over 18 people's work, predominately from ETH Zürich. Its focus is on the design and analysis of optimal controllers for linear, nonlinear and hybrid systems. Particular attention is paid to parameterizing systems into polytopes, allowing efficient binary tree searches for calculating the optimal control input (as described in Section 2.2.2). This method of solving control problems includes Explicit MPC, which was not part of this work, primarily because of the high memory requirement as described in the literature review. MPT also uses YALMIP extensively for MPC problems, and thus was not used for the same reason as given above.

fast_mpc A free set of code that accompanies [327] is fast_mpc, which was written by Stephen Boyd (Stanford) and a graduate student, Yang Wang. Its main focus is on accelerating the solution speed of linear MPC, as is the thrust of this work. However there were a number of short-comings associated with the package which made it unsuitable for this work. Most notable was that the control performance was worse than equivalent software packages (e.g. jMPC Toolbox), even with an identical problem setup. In addition, the flexibility of the package was quite limited in terms model/plant mismatch, reference tracking, blocking, state estimation and other features we wanted to investigate. The package was however very fast, solving the package supplied large random system MPC example four times faster than the jMPC implementation (described next). It was worth noting that this example problem had 90 decision variables and 900 constraints, which is much larger than this work is targeting. When reduced to a problem with 20 decision variables and 80 constraints, the jMPC implementation with `quad_mehrotra` is four times faster than fast_mpc.

3.5.2 jMPC Toolbox

As identified in the previous subsection, the existing MPC tools available for MATLAB were unsuitable for use within this work. This was primarily due to their limited flexibility to accommodate the problems of interest, but also to their targeting of different MPC formulations such as explicit MPC and alternative objective functions. To overcome the limitations of the existing tools, a new high-level framework for linear MPC in MATLAB was developed, the jMPC Toolbox.

The jMPC Toolbox provides a high-level framework for describing, building, tuning, simulating and then deploying linear model predictive controllers within MATLAB. One of the primary aims of the toolbox is to tailor the QP solver to the type of problem solved, in order to exploit structure within the problem and speed up convergence. When this approach is used together with a performance orientated MPC implementation, also developed within this work, this leads to high-speed controllers which can achieve over 1MHz sampling rates on a standard desktop PC, allowing rapid simulation and validation. Section A.2 describes the toolbox and its functionality in detail, including a step-by-step example of MPC control of a nonlinear CSTR in Section A.2.8. To see case-studies of MPC applications using jMPC, please refer to the Case Studies Section of the jMPC User's Guide [61], supplied in the jMPC Toolbox/Help directory on the Appendix DVD.

3.6 Summary

This chapter has introduced the finite-horizon MPC algorithm and the formulation of the quadratic programming problem that results from the quadratic cost function, together with the construction of the linear constraints. Based on a survey of existing quadratic programming solvers, the decision was made to develop a new infeasible interior point quadratic solver. This algorithm was chosen based on its applicability to smaller quadratic programming problems, such as those found in finite-horizon MPC problems, as well as benefits including not requiring an initial feasible point, together with an inherent worst-case execution time. This is opposed to an active-set algorithm, which as reviewed, may exhibit worst-case non-polynomial complexity, and therefore provides a severe worst-case execution limit, together with literature examples of poor numerical stability in reduced precision.

Several modifications to the traditional IIP algorithm were proposed, and each was tested with respect to the quadratic programs that result from a linear model predictive controller. A detailed analysis showed that Mehrotra's modification provided a significant converge speed-up, while treating the problem bounds separately had no benefit for the problems of interest. Heuristics were proposed to warm-start the algorithm at each consecutive optimization problem, as well as scaling the optimization problem, and both were written with a reduced numerical precision implementation in mind. In addition, algorithm modifications were proposed to efficiently calculate the system of linear equation at each iteration of the quadratic programming solver, as well as a set of infeasibility and termination tests which were shown to accurately determine when the solver should exit.

The main contribution of this chapter is the development of two new quadratic programming solvers, `quad_wright` and `quad_mehrotra`, both of which incorporate the suite of modifications and heuristics presented. Each solver was benchmarked against a range of standard quadratic programming problems, including model predictive control quadratic problems, for the problem sizes of interest (2-40 decision variables, 16-320 constraints). From this benchmark study it was shown both solvers achieved comparable accuracy to an industry standard solver (CPLEX), as well as outperforming all other surveyed solvers with respect to computational speed. Furthermore, both new solvers were shown to be highly memory efficient, with the maximum problem size of interest easily able to fit into less than 100KB of memory, indicating both solvers were suitable for deployment on an embedded system.

Chapter 4

Embedded Model Predictive Control

Continuing with the focus on model predictive control, this chapter extends both the model predictive control algorithm, and the quadratic programming solvers developed in the last chapter, and tailors them for deployment on an embedded platform. For this work an embedded platform is defined as a small (could fit in the palm of your hand), lightweight (weight of a modern smart-phone) and low-power (less than 2W) platform that contains an embedded processor and required circuitry to power and interface to the processor. These specifications allow model predictive control to be expanded to a range of mobile and lightweight systems, however they also severely constrain the achievable performance of the algorithm, given resource limits that are inherent in an embedded system. This chapter will show that embedded controller performance metrics such as sampling rate, memory requirement and accuracy do not need to be sacrificed due to the resource constraints of an embedded system, and that competitive results can be obtained by tailoring the algorithm for a specific problem dimension and deployment hardware.

The chapter begins with an introduction to the problem with embedded model predictive control, before a survey of suitable hardware targets has been undertaken in order to identify the best compromise of computational power and cost. Using this platform, a new auto-coding framework is described which extends the jMPC Toolbox to generate efficient, verified, embedded model predictive controllers, leveraging the quadratic programming solver developed in the last chapter, together with a new linear algebra library developed specifically for embedded MPC. The chapter concludes with a number of case studies comparing the proposed approach with both results from literature, as well as control of a real, unstable, nonlinear and multivariable, helicopter platform.

4.1 Introduction

It is no surprise that MPC is in fact well suited to the chemical process world, given the relatively slow dynamics (seconds to minutes), which mean the algorithm has plenty of time to converge and deliver the next control input(s). However with faster and smaller processors, together with advancements in numerical algorithms for solving quadratic programs, it is now realistic to achieve high speed MPC on modest hardware, with low power and limited memory requirements. This means MPC can be applied to a large range of smaller, faster and more varied applications.

As discussed in the literature review, MPC has seen a recent surge in research interest by fields other than its traditional application to large, slow chemical processes. This is because MPC has a number of attractive features which are equally applicable to control problems in electrical, mechanical, aerospace and other dynamic systems. These include the native handling of multivariable systems, intuitive tuning and optimal control inputs even with predefined operating limits.

The challenge with bringing these features to a high speed application is the significant computational load solving a quadratic program online adds. Given a modest control problem with horizons of $N_p = 15$ and $N_c = 5$, 3 inputs and outputs, and 6 states, results in a quadratic problem with 15 decision variables and 150 constraints, and a memory footprint of over 20KB (stored in double precision), it starts to become obvious why so much research has been done to reduce the complexity of embedded MPC implementations.

However, as stated in the introduction, we believe what has been overlooked is that an efficient, hand-coded implementation which retains the online optimization step, can be just as effective, if not more practical, for real implementations of MPC on embedded hardware. The remainder of this chapter will detail the development of a high speed embedded MPC implementation, and benchmark it against applications reported in the literature, as well as a real implementation of a challenging nonlinear system. This will validate that a memory and processor efficient implementation of MPC, using techniques developed in this research, whilst retaining online optimization is just as competitive, if not more so, than other methods for our problems of interest.

4.2 Alternative Embedded Hardware Targets

The MPC and QP algorithms described so far have been able to leverage the virtually limitless (with respect to MPC problems) memory and GHz clock speeds common on desktop computers. In addition, high speed linear algebra packages such as the Intel Math Kernel Library [148] can be utilized to provide parallelized matrix routines on multi-core processors with dedicated vectorized double precision instructions [147]. In order to deploy the algorithms on an embedded device, none of the above can be effectively used, meaning problem sizes are severely limited.

The core design of an embedded model predictive controller is related to algorithm modifications and implementation changes, required in order to achieve both a useful sampling rate, as well as fit the controller within memory limits. In addition, due to the large dynamic range required by the quadratic solver, which commonly must deal with numbers ranging from 10^6 to 10^{-6}, floating point representation of the algorithm is essential. This was determined using AccelDSP [343], a high-level synthesis tool for generating FPGA designs from MATLAB, which also includes an automated floating point to fixed point conversion tool. Examining an early (2009) version of `quad_wright` using AccelDSP revealed that for even a simple controller, a fixed-point version of `quad_wright` would not reliably converge, even with up to 54bit word lengths. Therefore the requirement for a floating point unit alone (rather than memory or clock speed) has substantially influenced the selection of suitable target devices.

4.2.1 Candidate Hardware Targets

Traditionally (5-10 years ago) applications that required high-speed arithmetic, especially in fixed or floating point, was the domain of the Digital Signal Processor (DSP). This was based on the architecture of a DSP being optimized for high-speed, high-throughput *continuous* numeric calculations such as in audio, radar and even video processing. Moreover, DSPs often had larger, more advanced multiply, Multiply-Accumulate (MAC), and division hardware, providing increased numerical calculation performance [297]. The downside however was that due to specific architecture optimization (such as memory layout, clever auto-increment registers and looping without branching) these were often quite difficult to program, even with advanced compilers.

In contrast, the Microcontroller Unit (MCU) was traditionally an integer processor with only an 8 or 16bit integer multiplier, and intended for supervisory control with minimal calculations. While floating point was available via vendor specific software libraries, it was often also a vendor specific implementation and therefore did not align with the IEEE 754 [146] standard on floating point. In addition, using only a 8bit integer multiplier to multiply a 32bit floating point number was very expensive, with division and other standard operations even more computationally expensive.

In the last 5 years the separation between a DSP and an MCU has blurred, with many MCUs now including DSP-like functionality. This brings the benefit of simpler programs, interrupt driven processing, and high-speed arithmetic to a single unit. The two MCUs surveyed in this work were based on this hybrid MCU/DSP architecture. A further option emerged with the introduction of smartphones, that is of the high-specification microprocessors such as the ARM which are now cheaper and more powerful than ever, and often include a Floating Point Unit (FPU, a hardware floating point arithmetic logic unit) which makes them a candidate for this work.

The final option considered is a Field-Programmable-Gate-Array (FPGA), which allows a custom hardware architecture to be developed, similar to an Application-Specific-Integrated-Circuit (ASIC), but which also remains reprogrammable. An inherent advantage of the FPGA is that being hardware, parallel algorithms can be developed and thus they see widespread use in video and signal processing. Moreover because the hardware is customizable, floating point units with a user specified precision can be implemented in hardware.

The following subsections summarize the functionality about each of the candidate targets with respect to the important considerations of this work (FPU, RAM, Flash, clock speed and price), with the findings listed in Table 4.1.

4.2.1.1 Atmel AVR UC3 Series [24]

A candidate MCU target from Atmel was based on their 32bit AVR range, the UC3 series of which the 'C' range features a single precision floating point unit (IEEE 754 compatible). The top model within this range, the AT32UC3C0512C features 512KB of on-chip Flash, 64KB on-chip RAM and clock rates up to 66MHz. The FPU features a fused single-cycle MAC unit that conforms to IEEE 754, as well as typical add, subtract and conversion instructions [21].

A further consideration for this work was the availability of a development kit so that the target processor could be evaluated without a purpose built PCB being designed. The UC3 MCU mentioned was available on a suitable development kit, shown in Figure 4.1 for US$299.00 (in 2013).

The UC3 can be programmed with the free Atmel Studio Integrated Development Environment (IDE), which sits on top of Microsoft's Visual Studio. This provides a compiler, debugger and integrated circuit programmer and based on previous

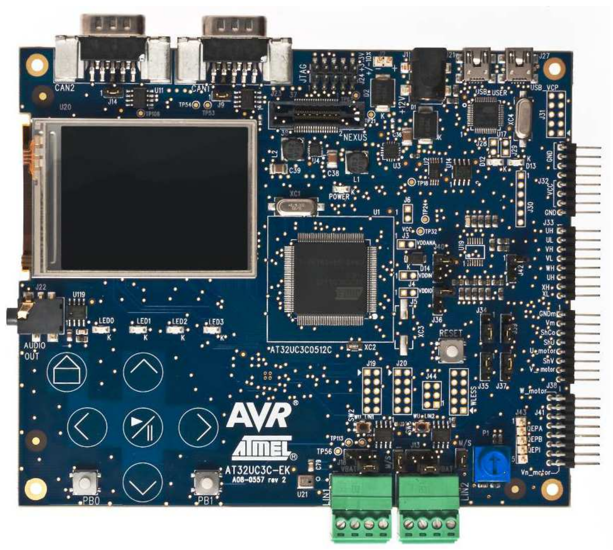


Figure 4.1: Atmel AT32UC3C-EK board [22].

experience, works very well.

4.2.1.2 Atmel AT91SAM

An Atmel IC also used by colleagues within our school was the ARM-based AT91SAM, featuring 256KB of RAM and running at 266MHz. The IC does not feature a floating-point unit, but was the fastest IC available within our department and thus we investigated its applicability within this work. Note this preliminary work was undertaken in late 2009, and this IC is now obsolete.

When developing software to run on the AT91SAM it was immediately discovered this was not a simple task, and required a range of compilation tools, programmers and significant setup. In addition, as our school did not have a commercial compiler for the IC (such as IAR), and there was no funding available, we were using a range of 3rd-party tools which were poorly documented and heavily reliant on the correct version of dependencies. In addition, creating a simple C program to run on the IC required a complex setup function, much of which was written in assembler and thus was a non-trivial task. It was quickly determined this was not going to be a suitable target, based on large investment in time required for most likely a modest to poor result.

Modern ARM-based processors are today almost as cheap as a high-spec MCU, and feature hardware floating point units, significant clock speeds (typically over 500MHz) and high-speed memory interfaces. Together with a colleague we investigated a dual core ARM Cortex A9 on the PandaBoard platform in late 2012 within [65], which is a low-cost (US$174), high performance (1GHz, 1GB off-chip RAM) development kit, as shown in Figure 4.2. Not surprisingly the performance obtained using this platform far surpassed the benchmark Delfino system (described below), however its applicability to real-time control was limited. This is based on the MPC algorithm running as an application of the operating system (Linux), which meant deterministic sampling could not be reliably achieved.

Figure 4.2: PandaBoard with dual core ARM Cortex A9 [344].

For real-time control systems a real-time Linux kernel can be built and downloaded to most modern ARM based development kits. While this was considered, it was determined that this was not worth pursuing given the large amount of auxiliary work required to get even a simple deterministic PID controller running. Alternatively ARM based processors can be programmed in C/C++ without an operating system, however this would be an even larger task than implementing a real-time Linux kernel.

4.2.1.3 Texas Instruments C28x Delfino Series [309]

The TI C28x Delfino series is a high performance hybrid 32bit MCU/DSP based on technology from the C6000 series of TI DSPs. The top-spec model (C28346) features a clock speed of 300MHz with 512KB on-chip RAM and IEEE 754 compatible single precision FPU. The Delfino is equipped with a set of ‘parallel’ floating point instructions for MAC, multiply, addition and subtraction, allowing both a arithmetic operation and memory copy to be done in the same pipeline cycle. TI also supplies free of charge the fastRTS library [306] which provides high-speed, assembler optimized implementations of common floating point functions such as division, square root, and trigonometric functions.

For this work the lower-spec C28343 was chosen as a candidate target based on availability. This IC ran at 200MHz and has 256KB of on-chip RAM, with the rest of the features relevant for this work the same as the C28346. An evaluation kit with the C28343 is available from TI for US$159 and features a small ‘controlCARD’ with a DIM100 connector, allowing it to be placed as-is in a user design, and is shown in Figure 4.3.

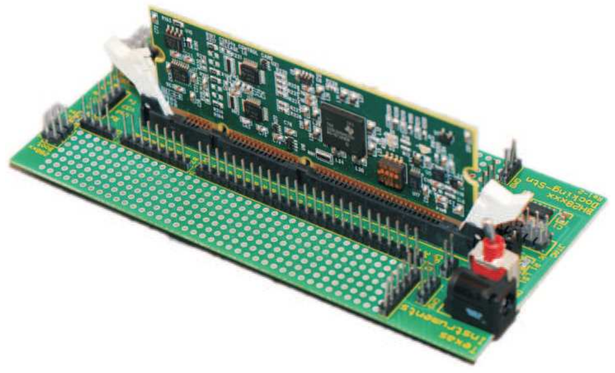


Figure 4.3: TI Delfino C28343 evaluation kit [305].

The C28343 is programmed using Code Composer Studio, a proprietary IDE with optimizing compiler, debugger, and programmer and is based on the Eclipse platform. Licences start from US$450 for a node-locked licence, about the same cost as the required JTAG emulator.

4.2.1.4 Xilinx Spartan-3E FPGA [342]

The Xilinx Spartan-3E XC3S500E is a small (500k gates) FPGA with 20 dedicated 18×18 multipliers and 45KB of on-chip block RAM. Using Xilinx terminology, this equates to 4656 slices or 10476 equivalent logic cells, both of which are alternative methods of describing the available resources of the FPGA. The Spartan-3E was chosen as this was the most advanced Xilinx FPGA accessible within our department, and we had a number of the starter-kits, shown in Figure 4.4.

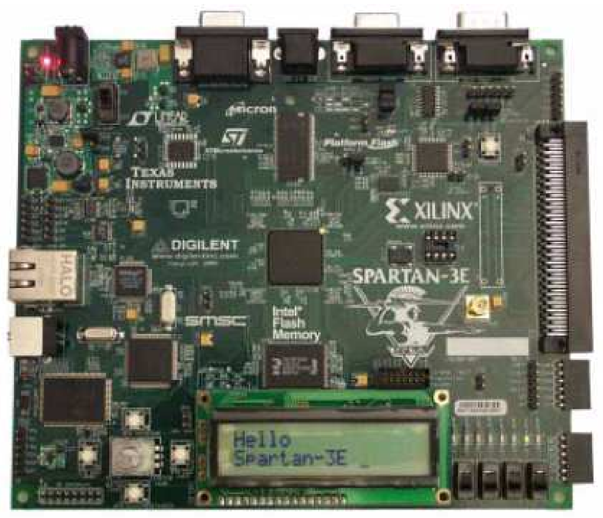


Figure 4.4: Xilinx Spartan 3E starter board [341].

The Spartan-3E is developed using Xilinx's free version of ISE, an IDE for developing Hardware Description Language (HDL) 'programs' using Verilog, VHDL, or graphical connections. Being a FPGA it was possible to implement as many floating point functions as required, with the upper limit set by the available resources (both hardware multipliers and gates) on the FPGA. It is also possible to implement arbitrary precision FPUs, meaning we are not constrained to only 32bit or 64bit units. In reality however this particular FPGA did not support more than 50bit floating point operations.

4.2.1.5 Target Summary

A comparison of the targets surveyed for the embedded MPC implementation is shown in Table 4.1. Note all four of these targets were surveyed in 2009 and thus no longer are a representative view of available hardware.

Table 4.1: Candidate hardware target summary.

Device	Clock Speed	Flash	RAM	Hardware FPU	Eval. Kit Price
Atmel UC3	66MHz	512KB	64KB	Yes	US$299.00
Atmel AT91SAM	266MHz	-	256KB	No	-
TI Delfino	200MHz	0KB	256KB	Yes	US$159.00
Xilinx Spartan	50MHz		45KB[1]	Optional	US$199.00

[1] Available as 18-Kbit dual-port blocks.

Given that the initial focus of this work was to accelerate MPC, and with papers at the time (2009) focusing on the FPGA route (e.g. [186, 187]), it was decided that we would pursue the Xilinx Spartan-3E for our embedded MPC development. However due to issues described later in Section A.4.1, this was later abandoned in favour of the TI Delfino. The TI was chosen over the Atmel UC3 for the following reasons (noted in order of importance):

RAM Given the large memory footprint required by MPC, the Delfino has 4x the RAM available of the UC3.

Clock Speed At 200MHz, the Delfino was over 3x as fast as the UC3.

Parallelized Instructions Leveraging a parallelized pipeline meant the Delfino FPU was much more attractive than the simple UC3 FPU.

fastRTS Library Optimized library functions allow division and square root to be performed in 24 and 28 cycles respectively, greatly accelerating the calculation of for example the Cholesky factorization.

Form Factor The tiny controlCARD form meant the evaluation kit could be re-used for deployment on a real system.

For the remainder of this chapter all embedded MPC results presented were obtained using the TI Delfino C28343 Evaluation Kit running at 200MHz.

4.3 Auto-Coding Framework

One of the contributions of this work is an auto-coding framework specifically developed for generating high-speed embeddable C-code MPC controllers from a MATLAB jMPC object. The framework allows a complete MPC controller, QP solver, and associated linear algebra functions to be generated in ANSI C in as little as 50ms, allowing rapid-prototyping and quick re-tuning of a real MPC controller. In addition, the framework can automatically verify the generated code using auto-generated test benches, thus ensuring the code generated is bit accurate when compared to an equivalent computer implementation. The following subsections will describe the auto-coding framework, noting this forms an integral tool for the remainder of this chapter.

4.3.1 Code Templates

In order to maximize the performance of auto-coder while keeping it as easy to manage as possible, a series of code templates are used for the code generation process. These are generalized hand-coded implementations of the MPC algorithm, QP algorithms, and linear algebra libraries, as discussed further on in Section 4.4.1. The key is they contain a series of 'preprocessor directives', some of which are standard C compiler directives (e.g. `#ifdef`), as well as framework specific directives and data type definitions, to control the generation and later compilation of required functionality. In this way we have a series of easy to read and modify `.c` files which can be easily updated as required, yet which form the basis of the auto-coded algorithm.

This technique avoids a more complex framework which could, for example, attempt to convert the existing MATLAB jMPC and QP algorithms directly into C/C++. There are existing tools that perform this transformation, such as [203], however using these sorts of tools is unlikely to result in scalable, memory efficient and high performance solver that *fully* exploits the benefits of the proposed algorithms. This conclusion is based on these tools typically hard-coding problem dimensions in order to allow for code generation optimizations such as for-loop unrolling. As described in Section 4.4.2 this has been deliberately avoided for this implementation, simply to preserve available data memory for larger controllers, given memory is expected to be an implementation constraint. This can be seen in Table 3.2 when looking in the File Size column for CVXGEN, noting the fully unrolled implementation requires over 1MB to store and does not utilize any threaded libraries (which would artificially further increase the code size for this sort of comparison).

A further workflow investigated used Simulink to auto-code the model/controller into embeddable C-code, as was done in [278]. This required the MPC controller and QP solver to be implemented as Simulink models, which while possible, was found to be quite cumbersome. This was primarily due to the complex indexing required by the MPC controller which was not succinctly expressed as Simulink blocks. Our early experimental QP and MPC Simulink blocks aare retained as part of the jMPC package, available on the Appendix DVD.

Given the motivation to use hand-coded C code versions of the MATLAB algorithms, the framework utilizes up to 7 template files, as described below. The template files can found be on the Appendix DVD, under jMPC Toolbox/Source/Embedded.

4.3.1.1 Common Header File (`jMPCEmbed.h`)

A common header file template is used that contains function prototypes for all available routines, regardless of whether they are included in source or not. This is done as the target compiler will ignore functions not used within the source, and thus enables minimal preprocessing. In reality, most of the routines are used, regardless of whether a Processor-In-the-Loop (PIL) implementation or a real embedded MPC is generated. In addition, four general timing and transmission routines are defined which are required for the user to implement, two for providing a 1μs timer to measure execution time, and two for sending and receiving a single byte via a user nominated serial port. Apart from these four simple routines, the remainder of the code is automatically generated and thus should be very simple to implement on any microcontroller.

4.3.1.2 Linear Algebra Source (`jMathEmbed.c`)

The linear algebra functions, described in detail in Section 4.4.2, form the key mathematical routines used by both the QP solver and the MPC algorithm. All routines are required by all code-generation options, apart from unconstrained controllers (which do not require the QP solver linear algebra functions). This file is typically used verbatim.

4.3.1.3 QP Solver (`QPWrightEmbed.c` or `QPMehrotraEmbed.c`)

Two QP solvers have been developed within this work, and either can be implemented as the QP solver for the MPC controller. Each solver has been hand-coded and hand-optimized in C, and undergone substantial testing as described in Chapter 3. Furthermore, each solver has then undergone minor modifications to allow them to be better implemented within an embedded target. These changes include:

- All memory is allocated by the auto-code generator, so that when the embedded algorithm is compiled, the problem data, intermediate variables and solution data is all allocated and initialized. This avoids the need to dynamically allocate memory for the solution, as well as providing the compiler the best opportunity to optimize the implementation based on the size of the problem generated.
- Solution memory is common between the MPC algorithm and the QP algorithm. This cannot be done with the MATLAB MEX implementation of the QP solvers as it is considered bad practice to modify MATLAB workspace memory within a MEX file, as would happen if common solution memory was used. Therefore the primal, dual and slack vectors do need to be doubled up (reducing memory), and warm starting is automatically available if required, as one set of common memory is used.
- Given both $\mathbf{H}$ and $\mathbf{M}$ are constant they have been removed from the function argument list and are compiled as external constants.
- Following the above point, it is possible to pre-calculate the maximum absolute value in $\mathbf{H}$ and $\mathbf{M}$ during the auto-coding process, reducing online computation. Therefore only $\mathbf{f}$ and $\mathbf{b}$ are analyzed at the beginning of each QP function call.
- Algorithm control parameters such as the maximum iterations and convergence tolerance are hard-coded into the implementation. Note these are read from the MATLAB `jMPC` object, and thus remain user controllable until the code is generated.

As with the control parameters, the QP solver algorithm selection is set up via the MATLAB `jMPC` object. Based on this setting either `quad_wright` or `quad_mehrotra` will be selected by the auto-code generator and become the respective code template used for the final MPC controller.

4.3.1.4 MPC Engine (`jMPCEmbed.c`)

The MPC Engine refers to the main MPC control algorithm, excluding the QP solver, and forms one function. As with the QP solvers this template has been hand-coded and hand-optimized, and thus implements a high-speed embedded algorithm. This main MPC function, `EmbedMPCSolve`, accepts the current plant output(s), setpoint(s), and measured disturbance(s), and returns the calculated control input(s), together with the model output(s) and states(s) and QP solver statistics. Within the function it also completes all calculations required within one sample of the control algorithm, including state estimation, measured disturbance prediction, augmenting unmeasured outputs and control move calculation.

To further increase speed in the embedded algorithm, preprocessor and code-generator directives have been used extensively throughout `EmbedMPCSolve` to enable optional features based on the controller setup. These are summarized below (note all features are disabled by default in order to reduce code size and complexity):

- Measured disturbance prediction calculation for including the disturbance in the control move calculation.
- Unmeasured output augmentation to recover unmeasured (but modelled) outputs via the state estimator.
- Uncontrolled output handling which removes outputs from the QP and augmented state vector the user does not want to control, but still wants to keep within constraints. This reduces the problem size, and can speed up some of the intensive MPC calculations. It also removes the need for a 'dummy' setpoint, which would otherwise be required.
- Constraint updates for $\mathbf{\Delta u}, \mathbf{u}$ and $\mathbf{y}$ are treated individually, allowing calculations to be skipped if the relevant constraint(s) are not present in the controller specification.
- For controllers where there are no constraints, the MPC control law is implemented as $\mathbf{\Delta u} = -\mathbf{H}^{-1}\mathbf{f}$ where $\mathbf{H}$ is pre-factorized using a Cholesky factorization, so that only a triangular substitution solver is required to solve the control input(s).

4.3.1.5 PIL Utilities (`PILEmbed.c`)

One of the most useful features of the auto-code generator, apart from generating real MPC controllers, is the ability to set up a Processor-In-the-Loop (PIL) validation of the generated MPC controller. This technique is described in detail further on in Section 4.5, however the basic idea is that the generated MPC controller runs on the embedded target, and a development computer pretends to be the system plant. The two devices communicate via a two-way communication channel, which in this case is a simple USB-serial link. This allows the generated code and the MPC algorithm to be validated on the target using any system to be simulated and controlled on the PC.

To facilitate this functionality the auto-code generator will by default copy a series of functions to run the controller, 'measure' the plant output from the PC, and apply the 'control inputs' to the PC. In reality measuring the plant output is done by querying the PC for the current plant output, and applying the control inputs is done by sending the values to the PC, therefore we require both transmit and receive functionality. As described in the generated header file, the user is required to implement two functions for serial communications, one for receiving a single byte, and one for sending a single byte. These will be processor and compiler specific, thus cannot be automatically generated (but should be trivial to implement). These are then used by higher level routines which are copied from the code template, and facilitate sending and receiving both single and double precision arrays, as well as larger integer variables. In addition, when receiving floating point arrays, a basic state machine is used in order to receive the correct number of bytes and have them shifted into the correct bit locations correctly.

The main PIL function, `PILSim`, is a basic `while(1)` loop which reads the current plant output(s) and desired setpoint(s), executes and times the MPC controller calculation, then sends back the calculated control move(s), as well as sample and QP statistics. This repeats indefinitely until the processor is restarted or a new controller is downloaded.

4.3.1.6 MEX Test Benches (`TestQP.c` and `TestMPC.c`)

In order to verify the embedded controller and QP solver are functionally correct, two test bench files can be optionally created to verify them on the development computer. Both of these templates provide a MATLAB MEX interface wrapper over the respective generated functions which can be automatically compiled and run during the code-generation process. This provides a simple and automated method for quickly checking the algorithm during development, before moving to the target.

4.3.2 Code Generation and Memory Estimation

In order to generate a complete MPC controller, the auto-code framework requires both the algorithm, supplied via the code templates above, as well as the tuning and specification specific problem data. This data is constructed via the MATLAB `jMPC` object, and is saved within it for use during MATLAB, MEX or Simulink MPC simulations. Using the object-orientated extensions of MATLAB, generating a complete embedded implementation of a `jMPC` controller is as simple as the following line of code:

```
embed(jMPCobj)
```

where `jMPCobj` is a `jMPC` object containing the complete controller specification. However the `jMPC` object is typically used in conjunction with a `jSIM` object, which provides the simulation environment settings such as the system plant model, set-point over the simulation, etc. Supplying both objects to the `embed` method enables both the controller to be generated, as well as a series of code verification tests for validating the controller within the supplied simulation environment. These verification tests are described in detail in the next section.

When `embed` is called several steps occur to generate the required controller. A standard call will consist of the following steps:

0. Basic error checking is performed to check the controller is valid for coding and default options are set up.
1. `jMPC_embed.h` - The embedded header file is copied from the header file template and data types defined. Compiler defines are added to enable options specified in the controller such as unmeasured outputs, uncontrolled outputs, etc.
2. `jMPC_constants.c` - A constants source file is created from scratch which includes constants for both the QP algorithm and MPC algorithm. This is described in more detail below.
3. `jMPC_math.c` - The linear algebra source file is copied from the custom math template.

4. `jMPC_qp.c` - The quadratic programming solver source file is copied from the specified template (based on `quad_wright` or `quad_mehrotra`). Global variable definitions are added based on the problem size and user settings.

5. `jMPC_engine.c` - The MPC algorithm source file is copied from the MPC engine template. Global variable definitions are added based on the problem size, user settings, and initialization requirements.

6. `jMPC_pil.c` - If the simulation options are supplied then a PIL implementation can be run, and the required communication routines are copied from the PIL template file.

An example output from `embed` is shown below for a small MPC controller as described in [320]:

```
jMPC Auto Code Generator for Embedded MPC [v1.5]
Architecture: c2000,  Precision: float [%1.10g]
------------------------------------------------
1) Creating Embedded Header File "jMPC_embed.h" ... Done
2) Creating Embedded Constants File "jMPC_constants.c" ... Done
3) Creating jMATH Source File "jMPC_math.c" ... Done
4) Creating QP Source File "jMPC_qp.c" [Mehrotra] ... Done
5) Creating MPC Source File "jMPC_engine.c" ... Done
6) Creating PIL MPC Source File "jMPC_pil.c" ... Done
------------------------------------------------
Data Memory Summary:
TOTAL:  1.736 KB
FLASH:  0.850 KB    QP:   0.688 KB
RAM:    0.886 KB    MPC:  1.048 KB
```

The entire code generation process takes around 50-100ms, depending on the size of the controller. Large controllers with problem matrices exceeding several thousand variables may take up to 1 second to generate, however the entire process is typically very quick. As shown in the above code output, the controller has been generated in single precision (`float`), and numerical values have been printed with up to ten decimal places, in order to retain repeatable results. Even though a single precision number only has around 7 decimal places of accuracy, it has been observed that up to 10 are required to match the conversions done by the compiler when type casting from double precision to single.

4.3.2.1 Data Type Definitions

Within the code templates the default floating point data type is `realT`. A similar set of names is used for other common data types such as `intT` (integer) `uintT` (unsigned integer), etc. These are non-standard data types and they allow a set of custom type definitions to be created based on the architecture of the processor for which the code is targeted.

A good example of why this might be required is demonstrated by the data type `double` when used on the TI C28343. For a computer user this would be expected to result in a 64bit IEEE 754 double precision implementation, the maximum typical accuracy achievable using native hardware support. On the TI C28343 however, a double is implemented as a 32bit IEEE 754 float, when using the hardware floating point unit. In order to activate a 64bit floating point, the data type must be defined as a `long double`, which then uses a non IEEE 754 software implementation (and does not leverage the floating point unit).

Given these architecture specific configurations, the auto-code generator is set up to define a series of type definitions within the common header file, based on the architecture of the processor of being targeted. The code snippet below shows a few lines for a single precision controller on the TI C28343:

```
typedef float realT;
typedef int intT;
typedef unsigned int uintT;
```

Using type definitions in this way provides the simplest mechanism to provide a global set of types that can be customized to the specific architecture of the target. In addition to data types, single precision variations of functions are automatically used (for example `sqrt` vs `sqrtf`) and the suffix `F` is added onto initialized variables to prevent unnecessary compilation warnings. This is done using a simple regular expression find-and-replace based on a set of common rules. A further consideration is that bit operations are avoided so that endianness (MSB or LSB) of the processor does not affect the algorithm (but are used for PIL routines).

The result of the data type definitions and auto-code rules is a generic MPC controller than can be tailored to most standard architectures.

4.3.2.2 Constants File

The only file that is generated entirely programmatically is the constants file in Step 2. It contains all the constants required by both the QP solver and the MPC algorithm declared in one place for easy inspection. Due to being constant data, it also represents all data that would be stored in Flash. An example output of the QP objective and constraint matrices contained within this file is shown below:

```
// QP CONSTANTS
```

```
const realT H[9] = {
        1,0.5017130181,0.3786669325,
        0.5017130181,0.7558895555,0.3004893826,
        0.3786669325,0.3004893826,0.5900181133};
const realT A[36] = {
        1,0,0,-1,0,0,-1,-1,-1,1,1,1,
        0,1,0,0,-1,0,0,-1,-1,0,1,1,
        0,0,1,0,0,-1,0,0,-1,0,0,1};
```

Note the constraint **M** matrix (called A within the code) has been transposed as memory is stored column major in MATLAB (based on its Fortran origin), and C is a row major language. 2D indexing has also been avoided as this would require increased code changes within the MPC, QP and linear algebra routines based on the C requirement of the leading dimension to be specified.

In addition to problem data, such as the QP matrices above, the constants also include a number of scalars which dictate the number of inputs, outputs, states, unmeasured outputs, etc. Other helper variables for indexing are also generated, which although increasing memory requirements, do keep the MPC code cleaner and easier to maintain.

4.3.2.3 Memory Estimation

Following on from the generation of the constants file is the estimation of data memory required by the controller. This is an important consideration when developing an embedded MPC controller as larger controllers will exceed the memory requirements of most small microcontrollers. Therefore to judge the upper limit on prediction and control horizons (which primarily dictate the problem size for a given model) it has proved very useful to provide this memory estimation.

Given the auto-code generator is responsible for writing every global variable and global constant, estimating the memory use for data is a simple task. For each variable or constant, a routine determines the number of bytes used by the respective data type, then multiplies it by the number of elements in the array. The total is then either summed to the RAM requirement (variables) or Flash requirement (constants). It is possible to simply precalculate these values based on a known controller configuration, however because the code generator has been in a state of flux during development, it was found to be more reliable to calculate these totals during code generation. This also provides a simple mechanism for dealing with the multitude of setup and algorithmic options available via the `jMPC` and `jSIM` objects.

Note the estimation of memory also takes into account initialized global variables, which require both Flash memory for the initialized values, as well as RAM for the variable memory.

4.3.3 Code Verification

Being able to automatically generate an embedded MPC controller is only useful if you can be confident the generated code (and algorithm) is functionally correct. Early implementations of the auto-code generator did not verify the resulting code, and as a result too many hours were spent debugging issues on the target, rather than addressing the errors within the faster, easier to use, development environment. Therefore substantial code verification functionality has been built into the auto-code generator, ensuring the generated controller and QP solver is either bit-accurate, or within numerical precision.

Two modes of verification are available to the user, one on the development computer and one on the target itself. This enables a staged implementation process to ensure that before the final controller is configured, the code has been rigorously verified. Each mode of verification is set up to either test the QP solver, or the complete MPC algorithm.

4.3.3.1 Development PC Verification

As part of the `embed` method two options are available for verification on the development computer:

`verify_qp` Compile a MEX wrapper around the specified QP solver and solve the first sample of a MPC simulation.

`verify_mpc` Compile a MEX wrapper around the MPC engine and run a complete MPC simulation.

Both verification routines use code templates described in Section 4.3.1.6, which are simply copied and modified based on the controller specifications. They are then automatically compiled and run and the verification results printed as part of the code generation process. This enables both the generation and verification to be completed with a single command, and typically takes no longer than 2-3 seconds.

To verify the QP solver three metrics are used; the accuracy of the solution, the number of iterations taken and the exit status. If the 2-norm of the difference between the generated solution and the reference solution meets a specified tolerance, and the status and iterations are identical to the reference results, then the QP solver is deemed correct. For the MPC engine two metrics are used; the accuracy of the plant outputs and the accuracy of the control inputs. As with the QP solver, the 2-norm of the difference between the generated solution and reference solution is used to check for errors.

When verifying the generated code the equivalent precision routine is run to generate the reference results. In addition, the high-speed C-code implementation of the QP solver / MPC algorithm is used, and given these are what the code templates are based on, the result is a bit-accurate verification result. This is only possible due to both algorithms being functionally identical, compiled with the same compiler, using the same precision and not using any non-deterministic functions (such as threaded libraries).

The typical output of a code-verification run on the development computer is shown below:

```
Auto Generated Embedded MPC Verification:
Compiling MEX QP Testbench... Done
Compiling MEX MPC Testbench... Done

MEX QP Verification PASSED
- Successfully Solved
- z_norm: 0

MEX MPC Verification PASSED
- y_norm: 0
- u_norm: 0
```

Note that the results are bit identical. In addition, a comparative figure is generated which compares the plant output, control input, the number of QP iterations taken and the QP status at each sample. Figure 4.5 shows the typical verification comparison of an mildly oscillatory SISO system in single precision, where the top plots show the generated code and reference outputs, and the bottom plots the difference between them. As can be seen the results are bit identical, and thus we can be certain the generated code is correct.

Differences between the generated code and the reference solution can be demonstrated when comparing simulations using different precision. Figure 4.6 shows the verification comparison of a single precision control simulation versus a double precision reference simulation. In this case the generated controller is accurate to around the numerical precision of the generated code (approximately 10^{-6}). Larger, more complex controllers have demonstrated higher inaccuracies between single and double simulations, but this is to be expected given the reduced precision. Larger controllers (20+ decision variables and 150+ constraints) return results typically accurate to 10^{-2} when comparing single and double precision implementations.

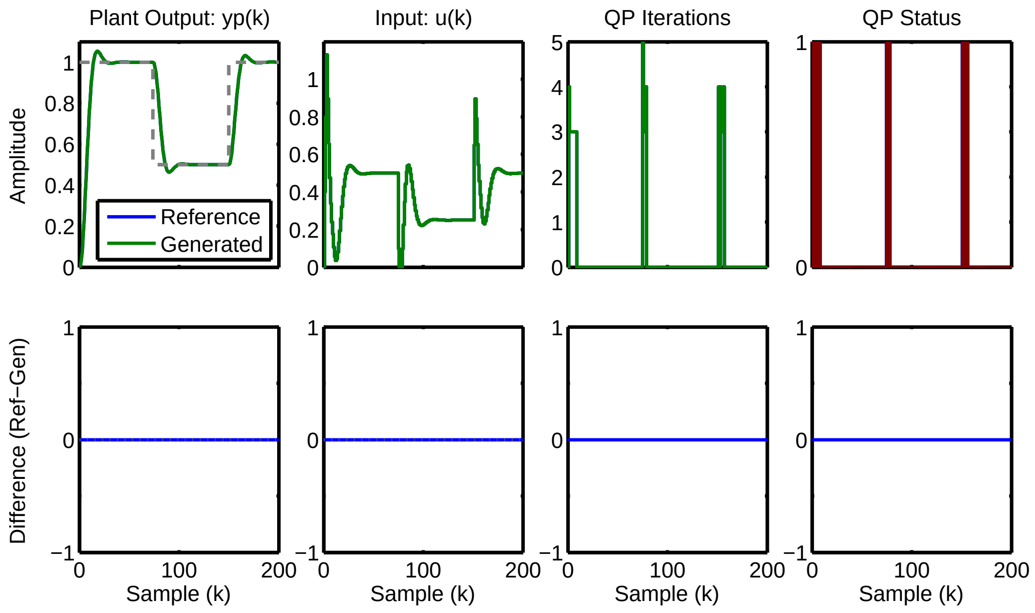


Figure 4.5: Code verification of a SISO MPC controller on the development computer (both in single precision).

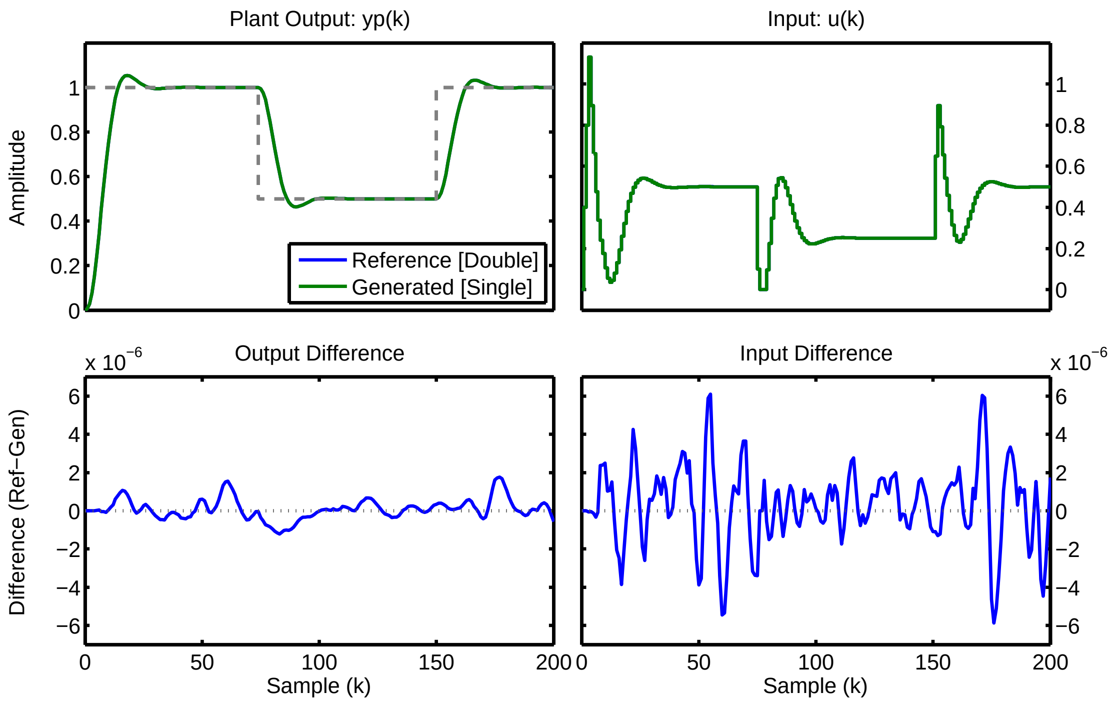


Figure 4.6: Code verification of a SISO MPC controller on the development computer (single precision code versus double precision reference).

Another possibility for differences is when the data types of the generated code do not correspond to equivalent types on the development computer. This particular issue has not been encountered during this work, but could exist, for example, due to the way the data type `double` is handled on the TI C23843 (as noted in Section 4.3.2.1).

4.3.3.2 Target Verification

The next important step once the code has been verified on the development computer is to verify the code on the target microcontroller/microprocessor. Even though we know from the development computer verification the code is functionally correct, it is very unlikely to produce the same result as the reference simulation. As described later in Section 4.4.4.2, this is based on different compilers performing different optimizations, so that the operation order of the floating point calculations will likely vary. Therefore the target is expected to give different results from the development computer, but most likely only down to numerical precision.

In order to verify the algorithms on the target, two Processor-In-the-Loop routines are automatically generated by the auto-code generator, one for verifying the QP solver and one for the MPC engine. As described in Section 4.3.1.5, these routines use the development computer as the system plant (or for the QP solver, to supply the initial iterates), and leverage a simple USB serial link to communicate between the target and development computer.

Verifying the QP solver on the target is quite possibly the most important part of the code verification process, given it not only forms the bulk of the MPC controller, but also is the most sensitive to numerical errors. Therefore to effectively verify the QP solver it must be rigorously tested across a range of MPC QPs. A simple method of doing this is to substitute the QP solver on the development computer with the target QP solver, thus utilizing the target to solve the constrained optimization problem at each sample. This can be compared with the result of a MPC simulation run purely on the development PC, and errors identified at specific samples. Figure 4.7 shows the PIL setup used to verify the QP solver on the TI C28343 within a development computer based MPC simulation.

Using the same SISO MPC control example from the previous section, Figure 4.8 shows the result of solving the QP resulting from each sample of the MPC simulation on the TI C28343, compared with the result on the development computer. As expected there are small differences between the two implementations, but the errors are consistent with the numerical precision of both systems. In addition, within this example the number of QP iterations recorded matches between both

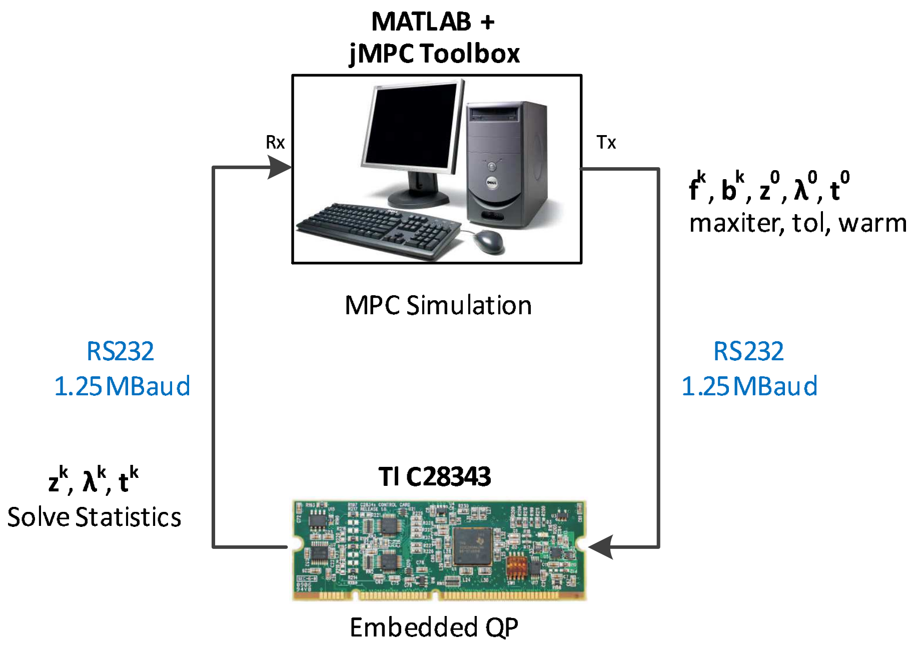


Figure 4.7: TI C28343 embedded QP verification using PIL.

implementations, however this is not always the case. It has been observed in larger systems that due to implementation differences between the target and development computer, the QP solver may require a different number of iterations in order to converge. This is typically within one or two iterations and the control performance is not obviously degraded.

The MPC PIL verification is described in detail later in Section 4.5, however for comparison, the entire MPC controller from the QP verification above is shown verified on the TI C28343 in single precision in Figure 4.9. Once again there are small differences between the two implementations, but the errors are consistent with the numerical precision of both systems.

4.3.4 Framework Summary

The auto-code framework presented so far is summarized below in terms of its design advantages.

4.3.4.1 Design Advantages:

High Level Control Design Using the jMPC Toolbox to design and simulate MPC controllers in MATLAB means a high level language can be used to succinctly describe the controller specifications and simulation environment. This allows rapid prototyping and validation of the design before implementation in a lower level language, such as C.

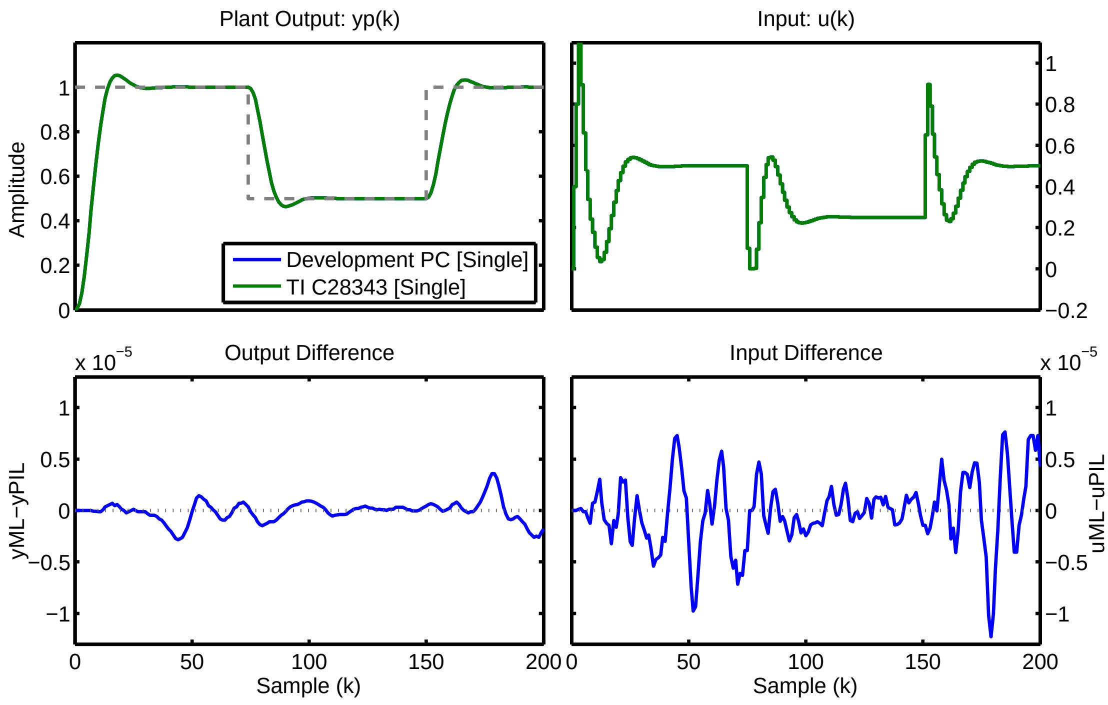


Figure 4.8: Code verification of `quad_mehrotra` on the TI C28343, called from a MPC simulation on the development computer (both in single precision).

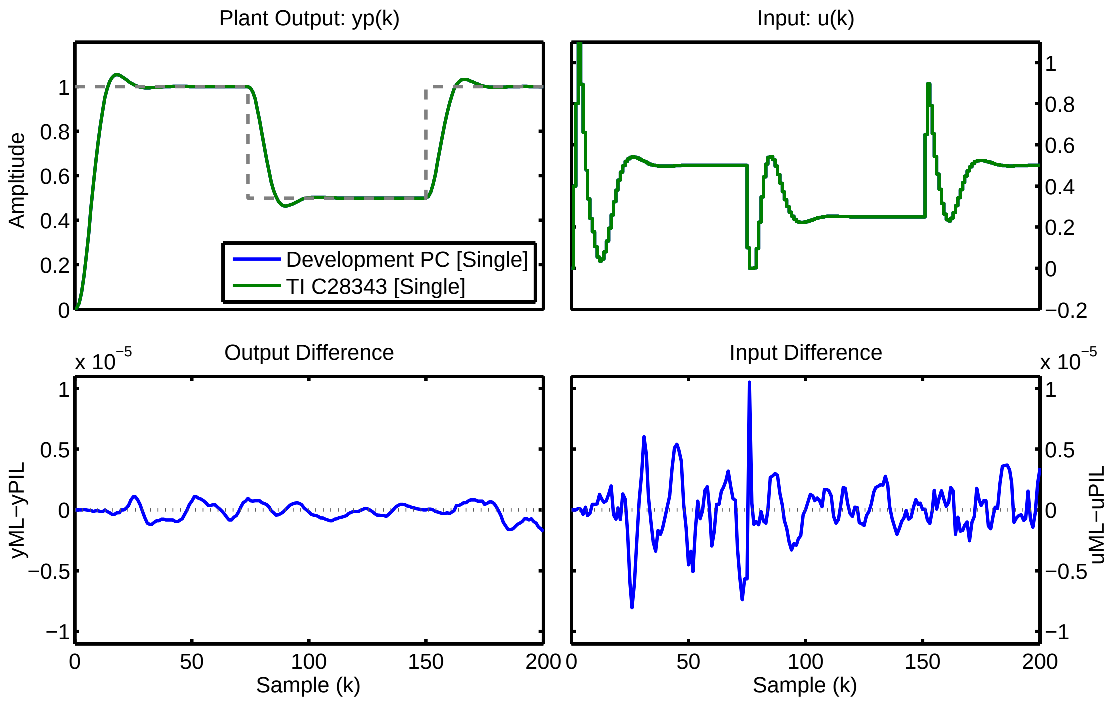


Figure 4.9: Code verification of a complete SISO MPC controller on the TI C28343 (both in single precision).

Algorithm Speed By utilizing a combination of auto-generated code and hand-coded templates, the resulting MPC controller leverages both the advantage of optimized hand-coded algorithms, together with the tuning dependent problem data being automatically inserted. This not only greatly aids code maintainability (based on being able to visually inspect the code templates), but allows for finer control of the generated code.

Memory Footprint As demonstrated, the memory footprint of the controller data memory is estimated as the code is generated, thus allowing a user to judge whether the algorithm will fit before undertaking compilation. The memory required for the QP and MPC algorithm is compiler dependent, but typically requires around 8KB including program code.

Code Generation Speed Generating a small MPC controller and QP solver can be performed in as little as 50ms, meaning re-tuning a controller or developing a new controller is not an onerous task. When developing using Code Composer Studio and the TI C28343, a controller can be tuned, generated, compiled and deployed in as little as 10 seconds, allowing rapid development of an optimal control implementation.

No 3rd Party Dependencies The framework does not rely on external tools such as the MATLAB or Simulink Compiler in order to generate C-Code. Apart from MATLAB itself, the jMPC toolbox and code-generation functionality is completely stand alone. The exception is the Control Systems Toolbox, which can be used to increase functionality, if required.

4.4 Quadratic Program Solver Implementation

The core requirement of a successful embedded MPC algorithm is an efficient quadratic programming solver, given that it is responsible for up to 95% of the computational effort at each sample and is responsible for the calculation of control moves. Chapter 3 described the highly efficient QP solvers we have developed in this work, together with performance results for a MATLAB implementation. This section describes the implementation of the `quad_wright` and `quad_mehrotra` algorithms in C-code to maximize efficiency and reduce code-size, and demonstrates the implementation within the custom auto-code generation environment.

4.4.1 ANSI C `quad_wright` and `quad_mehrotra`

The `quad_wright` and `quad_mehrotra` algorithms have been implemented in ANSI C-code which makes it suitable for a range of embedded hardware targets. Moreover, implementing the algorithm in a lower level language has enabled increased code optimization to be performed, using both advanced compiler optimizations and hand-optimized routines. In addition, full control of the memory allocation and release is available in languages such as a C or C++, meaning the code can leverage pointer-arithmetic and avoid dynamic memory allocation where possible.

It may be argued that a true hand-coded implementation should be done in assembler, such as in [331] (which achieves a very high sampling rate), in order to fully utilize the system resources. This however limits the implementation to a particular instruction set and as this work has considered multiple hardware targets, becomes a limitation and thus C will be the lowest level language used.

Complementing the efficient algorithm implementation is a custom library of linear algebra routines and a simple memory management scheme, which are described in the following subsections.

4.4.2 Linear Algebra Library

As with typical optimization solvers `quad_wright` and `quad_mehrotra` are described using a series of matrix, vector and linear algebra operations and functions. On a typical PC these functions can be implemented via BLAS[183]/LAPACK[14] routines such as `dgemm` (double precision, general matrix$\times$matrix) and `dpotrf` (double precision Cholesky factorization). Implementations of BLAS/LAPACK include those available via Netlib, as well as tuned implementations such as ATLAS [329] and Intel's MKL [148]. The tuned implementations leverage multithreaded computations to parallelize the repetitive calculations involved and thus scale well with problem size. In addition they often also take advantage of architecture specific instructions, such as the Intel AVX/AVX2 [147] instructions for Single Instruction, Multiple Data (SIMD) for further vectorization, as well as implementations tuned for specific processor cache sizes.

In contrast with the multi-core PC is the embedded microcontroller which is often severely limited in terms of memory, clock speed, cache-size (if any) and restricted to a single-core. Therefore dedicated PC linear algebra and matrix routines will not transfer to an embedded platform without substantial re-writing, and often may be impossible to transfer without the original source for compilation to the target architecture. Libraries such as BLAS are available as Fortran source, but even after being run through `f2c` (Fortran to C converter) so that the target compiler may compile them, the library will be difficult to read and may not function correctly under high optimization settings.

To alleviate the issues surrounding what is effectively PC code for a microcontroller, a dedicated set of linear algebra and matrix/vector routines has been written in C which is used both for the QP solver and the MPC algorithm. These routines are listed in Table 4.2 together with their mathematical implementation. All routines are once again hand-optimized in order to maximize code efficiency, as well as reduce the overall code size. An example is presented in Section A.4.2 of the implementation of `jtmv` together with details of the techniques used to optimize it.

Table 4.2: Custom linear algebra routines.

Routine	Equation	Description
`jdot`	$y = \mathbf{a}^T\mathbf{b}$	Dot Product
`jmv`	$\mathbf{y} = \alpha\mathbf{A}\mathbf{b} + \beta\mathbf{y}$	Scaled Matrix × Vector + Vector
`jtmv`	$\mathbf{y} = \alpha\mathbf{A}^T\mathbf{b} + \beta\mathbf{y}$	Scaled Matrix^T × Vector + Vector
`jfsub`	$\mathbf{y} = \left(\mathbf{A}^T\right)^{-1}\mathbf{b}$	Forward Substitution Solver
`jbsub`	$\mathbf{y} = \left(\mathbf{A}\right)^{-1}\mathbf{b}$	Backward Substitution Solver
`jtris`	$\mathbf{y} = \mathbf{A}^{-1}\left(\left(\mathbf{A}^T\right)^{-1}\mathbf{b}\right)$	Triangular Solver
`jchol`	$\mathbf{A} = \mathbf{R}^T\mathbf{R}$	Cholesky Factorization
`formLinSys`	$\mathbf{H} = \mathbf{A}^T\text{diag}\left(\frac{\boldsymbol{\lambda}}{\mathbf{t}}\right)\mathbf{A}$	Form Linear System LHS
`vecmax`	$\gamma = \lVert\mathbf{t}\rVert_\infty$	Max Absolute Value in a Vector
`vecmmax`	$\gamma = \lVert\boldsymbol{\lambda} - \mathbf{t}\rVert_\infty$	Max Absolute Value in (Vector-Vector)

4.4.3 Memory Management

Dynamically allocating and releasing memory on a PC using `malloc` and `free` (or `new` and `delete`) is often done without much thought of performance penalties. For example, the original (pre 2013a) MATLAB MPC Toolbox [206] C-code engine block would dynamically allocate and release QP memory at every sample, rather than maintaining a set of data memory between calls. This approach to memory management on a embedded platform is actively discouraged for multiple reasons. Primarily this is due to limited dynamic memory (RAM) available, and as memory is allocated and released, fragmentation can occur. While this can also occur on a PC, the processor speed and sheer size of available memory means performance penalties are not normally noticed when memory compacting (defragmentation) is run. This is not the case on an embedded processor, and can lead to non-deterministic runtime performance. Given this algorithm is to result in a real-time controller, dynamic memory allocation is not an option.

Given a defined controller specification it is however possible to completely precalculate the required Flash (constants) and RAM (variables) memory when the controller is created using the jMPC Toolbox. This means all memory requirements are known before implementation, and variables can be preallocated, such as below.

```
realT RQP[25] = {
        0,0,0,0,0,
        0,0,0,0,0,
        0,0,0,0,0,
        0,0,0,0,0,
        0,0,0,0,0};
```

Using this preallocation strategy has two advantages; first, all variables are initialized to a known value (non-standard for a C compiler) and secondly, when the program is compiled the compiler is able to make better code optimization choices knowing the size and dimensions of all variables. The downside is that generating these preallocated constants and variables can be extremely tedious, which is where the auto-coding framework described earlier allows true rapid-prototyping.

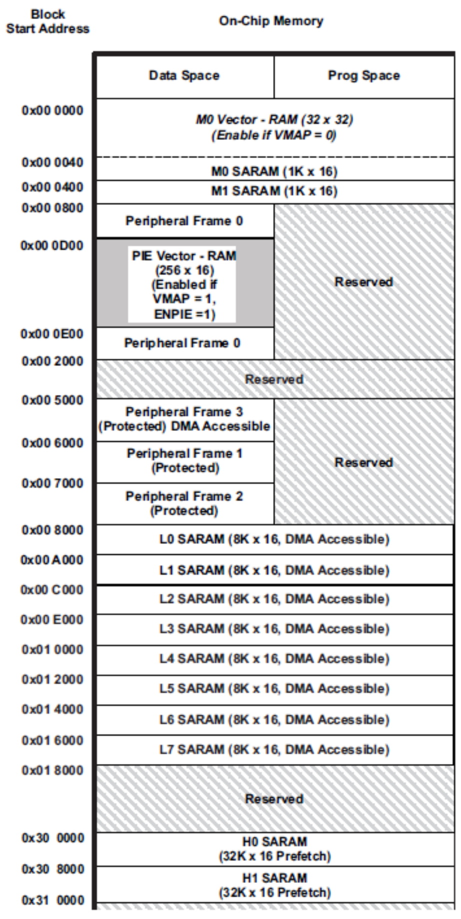
4.4.3.1 Constants vs Variables

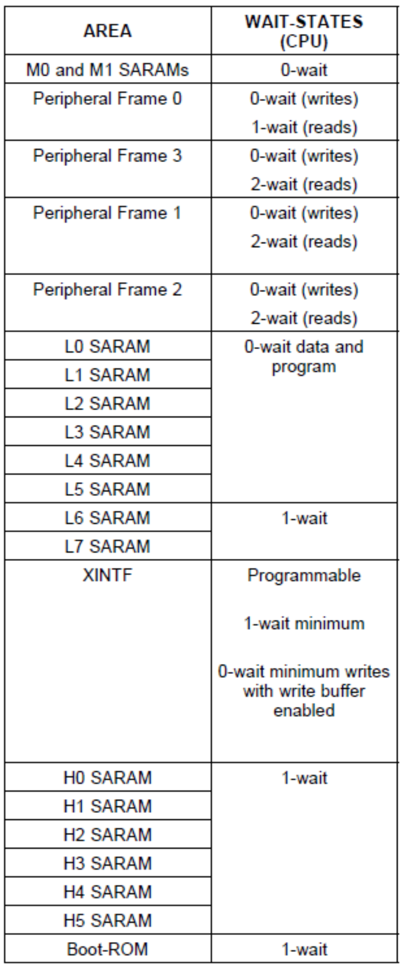
A point worth mentioning in this section is the difference between declaring problem data as `const` and normal (modifiable) variables. Depending on the architecture of the processor being used, together with compilation settings, variables declared as `const` are typically stored in Flash rather than RAM. This is normally advantageous as a processor will have much more Flash memory than RAM. For example, take a common 8-bit microcontroller used in our undergraduate classes, the Atmel ATmega128 [23] which has 128KB of Flash and 4KB of RAM. This reflects the normal distribution of memory of small microcontrollers, where the program and constants are significantly larger than the required dynamic variables. For the 3-DOF Helicopter example in this work, constants alone take up 80% of the memory requirement of the MPC controller, without taking into account the program itself.

An issue arises when examining the access speed of the two memory types. While *reading* Flash memory is fast, it is not (typically) as fast as accessing RAM [307]. Again looking at the Atmel ATmega128 data sheet, instructions which load data from Flash (`LPM, ELPM`) take 3 clock cycles, versus typical load instructions (from CPU registers or RAM, e.g. `LDI, LD`) requiring 1-2 clock cycles. When up to 80% of the memory requires an extra one or two clock cycles to access *every time* it is accessed, a heavy performance penalty can be experienced. Therefore if the processor has available RAM resources (such as the TI Delfino), it is advantageous *not* to declare constants as `const`, in order to avoid this time penalty. Alternatively, as described in the next subsection, the linker can be setup to place constants within RAM, overriding the default allocation to Flash.

4.4.3.2 Memory Block Size, Location and Wait-States

As described in Section 4.2 the TI C28343 microcontroller is equipped with 260KB of on-chip RAM. It might be expected then that apart from program space (noting this particular IC has no onboard Flash) and memory dedicated to the peripherals, that most of this RAM is available for use, and is basically identical regardless which address data is stored at. This does not turn out to be the case, and as shown in Figure 4.10 only approximately 100KB is 0 wait-state memory, meaning the remainder of the memory requires 1 or more wasted clock cycles in order to read or write.


Block Start Address
On-Chip Memory
Data Space
Prog Space
0x00 0000
M0 Vector - RAM (32 x 32) (Enable if VMAP = 0)
0x00 0040
M0 SARAM (1K x 16)
0x00 0400
M1 SARAM (1K x 16)
0x00 0800
Peripheral Frame 0
0x00 0D00
PIE Vector - RAM (256 x 16) (Enabled if VMAP = 1, ENPIE =1)
Reserved
0x00 0E00
Peripheral Frame 0
0x00 2000
Reserved
0x00 5000
Peripheral Frame 3 (Protected) DMA Accessible
0x00 6000
Peripheral Frame 1 (Protected)
Reserved
0x00 7000
Peripheral Frame 2 (Protected)
0x00 8000
L0 SARAM (8K x 16, DMA Accessible)
0x00 A000
L1 SARAM (8K x 16, DMA Accessible)
0x00 C000
L2 SARAM (8K x 16, DMA Accessible)
0x00 E000
L3 SARAM (8K x 16, DMA Accessible)
0x01 0000
L4 SARAM (8K x 16, DMA Accessible)
0x01 2000
L5 SARAM (8K x 16, DMA Accessible)
0x01 4000
L6 SARAM (8K x 16, DMA Accessible)
0x01 6000
L7 SARAM (8K x 16, DMA Accessible)
0x01 8000
Reserved
0x30 0000
H0 SARAM (32K x 16 Prefetch)
0x30 8000
H1 SARAM (32K x 16 Prefetch)
0x31 0000



AREA	WAIT-STATES (CPU)
M0 and M1 SARAMs	0-wait
Peripheral Frame 0	0-wait (writes) 1-wait (reads)
Peripheral Frame 3	0-wait (writes) 2-wait (reads)
Peripheral Frame 1	0-wait (writes) 2-wait (reads)
Peripheral Frame 2	0-wait (writes) 2-wait (reads)
L0 SARAM	0-wait data and program
L1 SARAM	
L2 SARAM	
L3 SARAM	
L4 SARAM	
L5 SARAM	
L6 SARAM	1-wait
L7 SARAM	
XINTF	Programmable 1-wait minimum 0-wait minimum writes with write buffer enabled
H0 SARAM	1-wait
H1 SARAM	
H2 SARAM	
H3 SARAM	
H4 SARAM	
H5 SARAM	
Boot-ROM	1-wait

Figure 4.10: TI C28343 on-chip memory characteristics (Figure 3-3 and Table 3-1 in [309]).

In addition, following the linking information on the TI Wiki [307], we see further complications of variable block sizes, and constraints such as program and data must be located in separate blocks (typical in embedded systems, but as the TI uses a 'unified' memory architecture, overlaps can occur). Therefore the optimal allocation of the number of blocks, the size of each block, and which data maps to which block all become an interesting problem, the solution to which is also dependent on the tuning characteristics of the MPC controller. For this work a compromise was found which allowed large tuning horizons to be implemented, while retaining efficient use of 0 wait-state SARAM blocks. The linker command file for this work is listed in Section A.4.3, and summarized in Table 4.3.

Table 4.3: TI C28343 custom linker command file summary.

Name	Length	Used[1]	Page	Wait-States	Function
RAML0	1KB	1%	Program	0	Switch Case Tables, `ramfuncs`
RAML1	16KB	60%	Program	0	Program code, IQmath code
RAML2	24KB	92%	Data	0	Global and Static Variables
RAML3	90KB	99%	Data	0/1[2]	Constant Data
RAMM1	2KB	0%	Data	0	Stack
RAMH0	130KB	16%	Data	1	Initialized Global Variables

[1]3-DOF Helicopter MPC Controller with 35 decision variables and 308 constraints

[2]1 wait-state for addresses 0x14000 and above

By examining the generated `.map` file after an MPC controller has been compiled it is possible to determine how much memory is placed into each section. Also shown in the third column of Table 4.3 is the percentage used by a very large MPC controller implemented with long horizons ($N_p = 60, N_c = 17$) and constraints on all inputs, and two of three outputs. The resulting controller has an estimated memory footprint of 110KB, which matches the actual memory requirements quite closely. Differences occur due to the Code Composer Compiler allocating global arrays of lengths that may vary from that specified in code, in order to optimize page boundary access (see the discussion on the TI Forum [310]).

A further observation of Table 4.3 is the use of `RAMH0` and its allocation of initialized global variables. In a normal embedded system with Flash memory, these would be the initialized values of the global variable arrays, and thus are written into Flash, *in addition to* the memory being allocated in RAM. Part of the boot process would be then to copy these values into RAM, thus enabling their use. However it appears this particular IC still separates the values from the variables, even when the program is written directly into RAM. Therefore we effectively 'double up' the required memory for initialized global variables using my configuration. In addition, `RAMH0` has not been used for any other data, specifically because of the 1 wait-state penalty when using it. Given the example presented, the maximum sampling rate is already limited to around 6Hz, thus larger controllers are not expected to be implemented using this platform and `RAMH0`. There is therefore around 100KB free memory on this platform if required for other purposes.

4.4.4 Performance Testing on the TI C28343

Using the auto-coding framework described in Section 4.3, both the `quad_wright` and `quad_mehrotra` solvers have been implemented on the TI C28343 microcontroller and their performance, accuracy and memory consumption benchmarked across a range of QPs resulting from MPC problems. In order to benchmark the solvers, the C28343 microcontroller has been interfaced to MATLAB using a Processor In The Loop (PIL) implementation as shown in Figure 4.11.

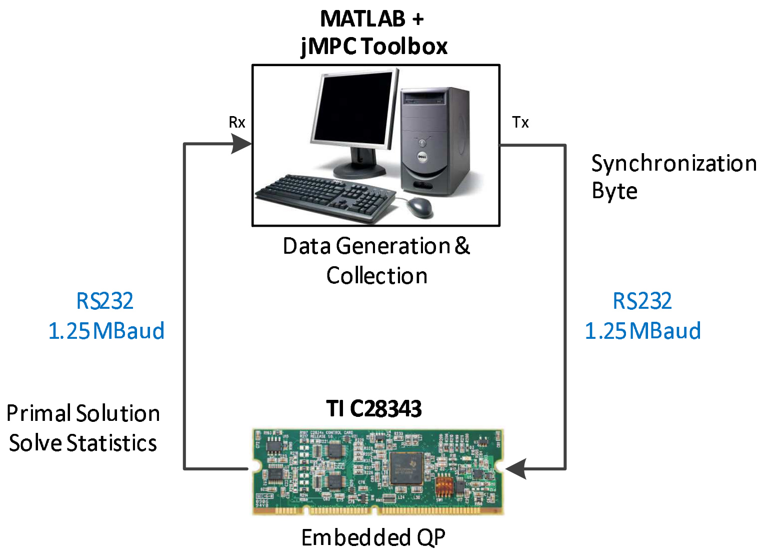


Figure 4.11: TI C28343 embedded QP PIL setup.

The QP solver is compiled using maximum optimization settings in Code Composer Studio then downloaded to the target microcontroller via JTAG, which then waits for a synchronization byte from MATLAB. Once initiated, the QP solver on the C28343 solves the compiled problem and transmits back the primal solution, the solver status, the number of iterations taken and the length of time taken, measured using a 1μs timer interrupt.

The problem chosen to benchmark the QP solvers is based on solving the QP that results from the first sample of a tightly constrained SISO system. The system model is as follows

$$\mathrm{G}(s) = \frac{2}{0.7s^2 + 0.2s + 1}$$

and is discretized at a sampling rate of 0.1s.

The controller is designed using $\mathbf{Q} = (5^2)\,\mathbf{I}_{N_p}$ and $\mathbf{R} = (0.5^2)\,\mathbf{I}_{N_c}$, and is constrained within the following regions

$$\begin{aligned} -2.0 &\leq u \leq 4.5 \\ 0.49 &\leq y \leq 1.01 \end{aligned}$$

In order to test the QP performance, the initial systems states are set as $(2.5, 2.5)^T$ and the setpoint is set as 1. Given the large weight on output deviation, together with the tight constraint (setpoint of 1, upper output constraint of 1.01), a large proportion of the upper output constraints are expected to be active at the solution. To vary the size of the QP being solved, N_p and N_c will be varied, thus testing the performance of the solver over a range of decision variable and constraint sizes.

For the reader's reference, a sample solution run of the MPC controller with $N_p = 20, N_c = 8$ can be viewed in Figure 4.12. The QP being solved for this benchmarking study is the QP that results at sample 1, using single precision and a tolerance of 5×10^{-4}. Given the tuning constants shown, 17/20 upper output constraints are active at the solution, resulting in a challenging QP to be solved. The relationship between prediction horizons and problem dimensions for this problem is as follows

$$\begin{aligned} \#\text{Variables} &= N_c \\ \#\text{Constraints} &= 2N_p + 2N_c \end{aligned}$$

The following subsections detail each of the metrics used to validate the QP solver performance.

4.4.4.1 Speed

To demonstrate the achievable speed of both algorithms, a range of QPs with varying sizes have been compiled and tested on the TI C28343. Two distinct ranges have been tested, small problems with 2-8 decision variables and 12-48 constraints, and medium to large problems (scale relative to typical embedded implementations) with 10-40 decision variables and 60-320 constraints. Gaps within the respective ranges are due to the formulation constraint that $N_p \geq N_c$ (lower bound in each Figure), and the range of horizons chosen (upper bound in each Figure).

Figure 4.13 shows the solution times of each algorithm with respect to increasing N_p and N_c, while Tables 4.5 and 4.4 detail the timing results. An interesting result

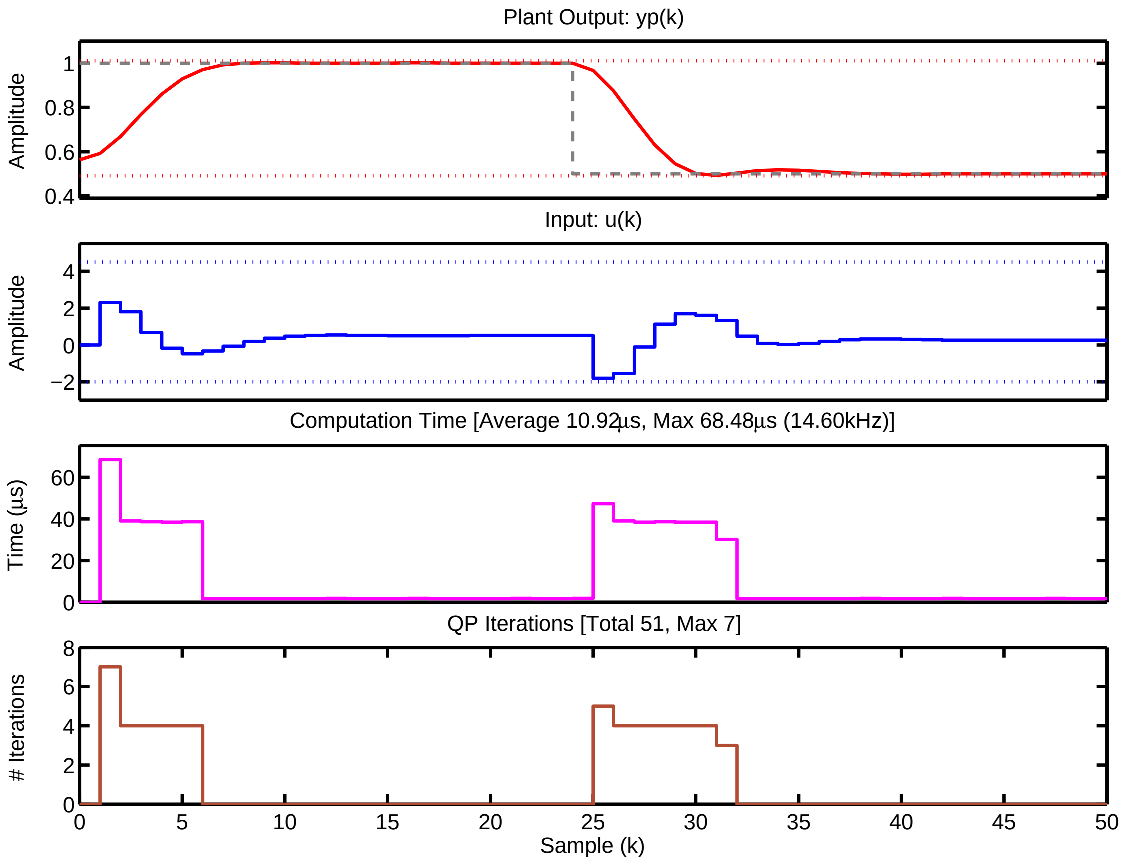


Figure 4.12: Reference MPC solution used for QP performance testing.

is observed in Table 4.5 where for a few particular tuning configurations (e.g. $N_p = 20, N_c = 8$), a larger problem is solved in less time than an adjacent smaller problem. This is a result of the solver requiring less iterations to solve this particular problem than the equivalent smaller problem. For all results presented, both algorithms required the same number of iterations to solve all problems when compared to the same algorithm on the PC.

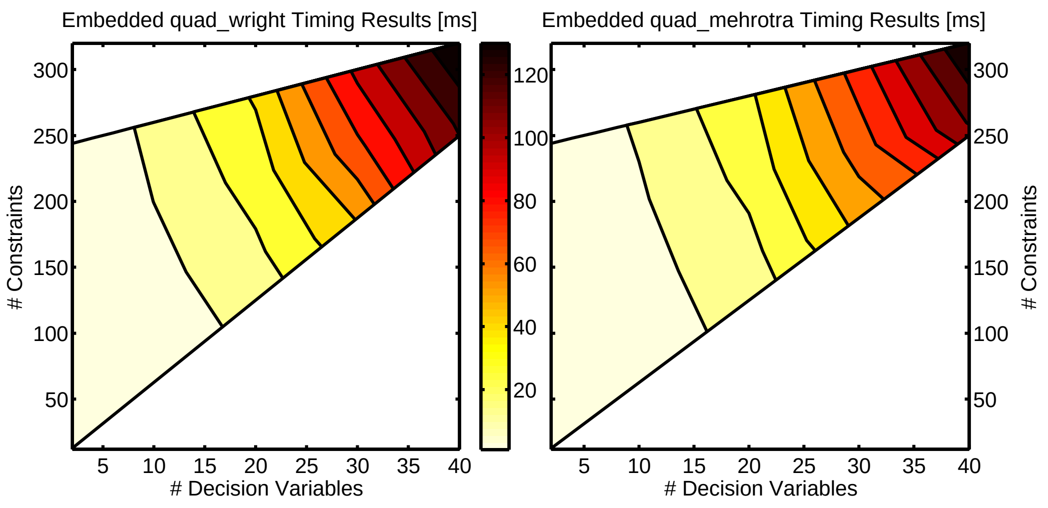


Figure 4.13: Embedded QP solver timing results (times reported in ms).

Table 4.4: Embedded `quad_wright` timing summary (times reported in ms).

N_c / N_p	2	4	6	8	10	20	30	40
4	0.163	0.387						
8	0.425	0.840	1.347	2.025				
12	0.572	0.944	1.659	2.440	3.443			
16	0.720	1.150	1.973	2.850	3.959			
20	0.747	1.355	2.285	3.253	4.475	15.149		
30	1.076	2.136	3.079	4.292	5.788	16.051	36.646	
60	2.022	3.455	5.513	7.433	8.574	24.335	52.036	107.637
90	4.541	5.762	7.901	10.531	13.673	32.592	76.916	123.515
120	7.275	10.258	10.291	13.388	17.296	42.032	97.664	147.638

4.4.4.2 Accuracy

To determine the accuracy of the embedded versions of the `quad_wright` and `quad_mehrotra` algorithms, the embedded results can be compared directly to the results obtained via the development computer (also in single precision). This is possible as the the accuracy of both the `quad_wright` and `quad_mehrotra` algorithms has been verified in Section 3.4.6, meaning we can be confident one set of results is correct. Unlike the auto-coded verification step (discussed in Section 4.3) which returns bit-accurate solutions, the embedded version is expected to return solutions accurate to

Table 4.5: Embedded `quad_mehrotra` timing summary (times reported in ms).

N_c / N_p	2	4	6	8	10	20	30	40
4	0.238	0.435						
8	0.447	0.753	1.606	2.987				
12	0.604	1.147	1.992	2.041	2.801			
16	0.762	1.398	2.034	3.768	3.848			
20	0.742	1.378	2.350	3.255	4.350	14.018		
30	1.312	1.897	2.653	3.596	4.721	16.961	37.199	
60	2.470	4.225	4.747	6.224	7.958	21.625	53.027	78.399
90	4.437	6.099	6.798	8.821	11.165	29.040	68.888	114.766
120	6.815	10.594	8.867	11.481	14.523	35.480	82.121	138.936

a fraction of percent. The reason for a difference is due to the non-associative and non-distributive properties of floating point (exaggerated in single precision), which will be caused by the different compiler optimizations present, and thus operation order, between TI Code Composer and Microsoft Visual C++.

As with the speed measurements, accuracy has been evaluated over two ranges: Small problems and medium to large problems. Figure 4.14 shows the absolute error of the embedded algorithm against the respective algorithm on the PC, when comparing the calculated objective values. The comparison is done in double precision to provide a better estimate of error. For complete accuracy details see Tables 4.6 and 4.7.

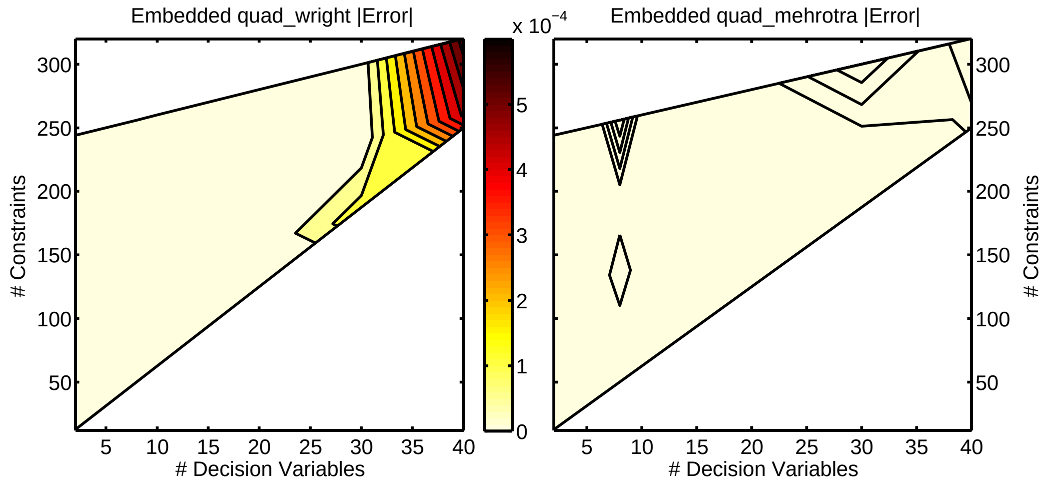


Figure 4.14: Embedded QP solver timing results [%]. Note the right hand figure exhibits no contour colours as compared to the left hand plot (sharing the same scale), very little error exists.

Table 4.6: Embedded `quad_wright` accuracy summary (absolute error).

N_c / N_p	2	4	6	8	10	20	30	40
4	$< 10^{-6}$	$< 10^{-7}$						
8	$< 10^{-6}$	$< 10^{-6}$	$< 10^{-7}$	$< 10^{-6}$				
12	$< 10^{-6}$	$< 10^{-6}$	$< 10^{-6}$	$< 10^{-7}$	$< 10^{-6}$			
16	$< 10^{-6}$	$< 10^{-6}$	$< 10^{-6}$	$< 10^{-6}$	$< 10^{-7}$			
20	$< 10^{-6}$	$< 10^{-6}$	$< 10^{-6}$	$< 10^{-7}$	$< 10^{-7}$	$< 10^{-7}$		
30	$< 10^{-7}$	$< 10^{-6}$	$< 10^{-6}$	$< 10^{-6}$	$< 10^{-6}$	$< 10^{-7}$	$< 10^{-6}$	
60	$< 10^{-7}$	$< 10^{-6}$	$< 10^{-5}$	$< 10^{-6}$	$< 10^{-5}$	$< 10^{-5}$	0.0001	$< 10^{-5}$
90	$< 10^{-7}$	$< 10^{-5}$	$< 10^{-7}$	$< 10^{-6}$	$< 10^{-7}$	$< 10^{-6}$	$< 10^{-6}$	0.0005
120	$< 10^{-5}$	$< 10^{-5}$	$< 10^{-6}$	$< 10^{-7}$	$< 10^{-6}$	$< 10^{-5}$	$< 10^{-4}$	0.0006

Table 4.7: Embedded `quad_mehrotra` accuracy summary (absolute error).

N_c / N_p	2	4	6	8	10	20	30	40
4	$< 10^{-6}$	$< 10^{-7}$						
8	$< 10^{-7}$	$< 10^{-7}$	$< 10^{-7}$	$< 10^{-6}$				
12	$< 10^{-7}$	$< 10^{-6}$	$< 10^{-7}$	$< 10^{-7}$	$< 10^{-7}$			
16	$< 10^{-6}$	$< 10^{-7}$	$< 10^{-6}$	$< 10^{-5}$	$< 10^{-6}$			
20	$< 10^{-7}$	$< 10^{-6}$	$< 10^{-7}$	$< 10^{-6}$	$< 10^{-7}$	$< 10^{-7}$		
30	$< 10^{-7}$	$< 10^{-6}$	$< 10^{-6}$	$< 10^{-6}$	$< 10^{-6}$	$< 10^{-8}$	$< 10^{-7}$	
60	$< 10^{-8}$	$< 10^{-6}$	$< 10^{-5}$	$< 10^{-4}$	$< 10^{-5}$	$< 10^{-7}$	$< 10^{-7}$	$< 10^{-5}$
90	$< 10^{-7}$	$< 10^{-6}$	$< 10^{-5}$	$< 10^{-5}$	$< 10^{-5}$	$< 10^{-6}$	$< 10^{-5}$	$< 10^{-4}$
120	$< 10^{-7}$	$< 10^{-5}$	$< 10^{-6}$	$< 10^{-4}$	$< 10^{-6}$	$< 10^{-6}$	$< 10^{-4}$	$< 10^{-5}$

4.4.4.3 Memory Consumption

As described in Section 4.4.3.2 the Code Composer Studio compiler may allocate arrays with a larger number of elements than specified in order to optimize page boundary access. Therefore in order to accurately benchmark the memory consumption of the solver, the compiler generated `.map` file is parsed to determine the number of bytes required for the switch-case tables, the compiled algorithm, variables, constants, and initialized variables (RAM blocks L0, L1, L2, L3 and H0 respectively). Note that as no dynamic memory is allocated, this calculated figure should give an accurate estimation of memory required. The compiled algorithm requires 6.5KB, with the remainder of memory required by the problem data.

As with both speed and accuracy, memory has been evaluated over two ranges: Small problems and medium to large problems. Figure 4.15 illustrates the memory requirements for both the `quad_wright` and `quad_mehrotra` algorithms, noting that the `quad_mehrotra` algorithm only requires two extra vectors, one of the number of decision variables long, and one of the number of constraints long. For memory requirement details consult Tables 4.8 and 4.9.

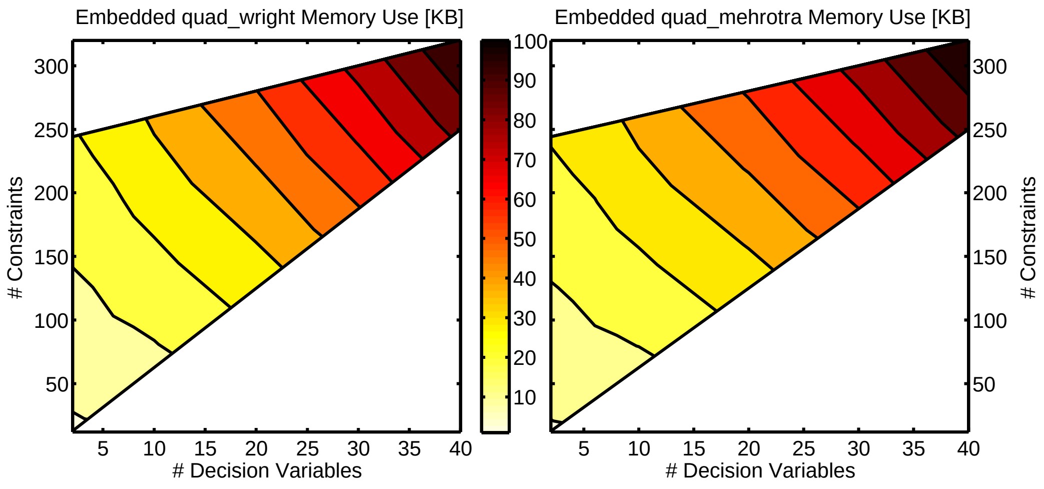


Figure 4.15: Embedded QP solver memory use [KB].

4.4.4.4 Performance Summary

The benchmark results so far have demonstrated the three key requirements of a successful embedded quadratic programming solver, namely fast solution times, repeatable results between compilers and processors, and lastly a modest memory footprint, even for large problems. Tables 4.10 and 4.11 summarize the performance metrics and results for small and medium-large problems respectively.

A key observation that can be drawn from these results is that for small problems, `quad_wright` is faster than `quad_mehrotra` . This was hypothesized and demon-

Table 4.8: Embedded `quad_wright` memory summary (memory reported in KB).

N_c / N_p	2	4	6	8	10	20	30	40
4	7.90	8.76						
8	8.76	9.67	10.62	11.45				
12	9.32	10.23	12.15	13.05	14.20			
16	10.80	11.64	12.67	13.63	14.84			
20	11.18	12.09	13.20	14.22	15.48	25.36		
30	12.21	14.25	15.49	16.61	18.12	28.78	42.19	
60	17.14	18.76	21.45	23.05	25.04	37.01	53.84	75.09
90	22.16	24.17	26.38	29.42	31.88	46.26	65.49	89.19
120	27.02	29.60	32.27	34.85	38.79	55.56	77.19	102.22

Table 4.9: Embedded `quad_mehrotra` memory summary (memory reported in KB).

N_c / N_p	2	4	6	8	10	20	30	40
4	8.59	9.41						
8	9.49	10.42	11.39	12.25				
12	10.08	11.01	13.08	14.01	15.18			
16	11.71	12.58	13.63	14.62	15.86			
20	12.13	13.06	14.19	15.24	16.53	26.78		
30	13.24	15.43	16.70	17.84	19.37	30.41	43.94	
60	18.67	20.32	23.15	24.78	26.78	39.01	56.09	77.71
90	24.18	26.22	28.45	31.64	34.13	48.75	68.23	92.32
120	29.54	32.14	34.83	37.43	41.54	58.56	80.44	105.71

strated in Section 3.4.4 in MATLAB, but was not reproduced in C. However for the TI C28343, this is quite obvious, and hence why *both* algorithms have been developed and tested within this framework. The intention is then that for small MPC controllers, `quad_wright` will be used, while for larger MPC controllers, `quad_mehrotra` will be used, thus allowing an adaptive solution. The reason `quad_wright` is faster on the TI C28343 will be partly based on the earlier hypothesis (it does less work per iteration), but will also be due to a different compiler optimized implementation than the Microsoft VC++ compiler.

In addition, it is evident that on the TI C28343 the accuracy of the `quad_wright` degrades as performance increases. By comparing the infeasibility and value of the objective, it was determined the computer solution was 'more' optimal, and thus the TI processor solution was deviating from the 'correct' solution. This result was observed within the `quad_mehrotra` algorithm as well, but not to the same degree of error as with the `quad_wright` algorithm.

With regards to memory consumption, as stated there is only a difference of two vectors between the two solvers, and thus their memory requirements are very similar. However it is worth pointing out that both algorithms are very memory efficient, requiring on average 11KB and 50KB for small and medium-large problems respectively. This puts these algorithms within reach of a number of small microcontrollers, where the limiting factor will now be the clock speed for achievable sampling rates. A further observation is that seeing the average memory footprint for a small MPC QP is so small, it is conceivable the algorithm could fit on a simple 8-bit microcontroller. This is based on the RAM requirement being approximately 2KB and that most microcontrollers will have greater than 16KB of Flash available for the code and constants.

In summary, it has been shown that both `quad_wright` and `quad_mehrotra` are competitive solvers for solving the QPs that result from a range of MPC tuning parameters on the TI C28343 floating point microcontroller. Sampling rates up to and exceeding 6000Hz are realistic for small control problems, with the average sampling rate expected to be between 200-2000Hz depending on the system model and tuning parameters used.

4.4.5 Benchmarking Against FORCES

A recent significant contribution to the field of embedded MPC is the Fast Optimization for Real-time Control on Embedded Systems (FORCES) framework [84], which is a MATLAB package for generating embedded model predictive controllers, as well as general linear and quadratic (including second order cone) optimization

Table 4.10: Embedded QP solver performance results - small problems (2-8 decision variables, 12-48 constraints).

Metric	Statistic	`quad_wright`	`quad_mehrotra`
Solution Time	mean	1.25ms (800.23Hz)	1.44ms (692.66Hz)
	max	2.85ms (350.88Hz)	3.77ms (265.39Hz)
	min	0.16ms (6134.97Hz)	0.24ms (4201.68Hz)
|Objective Error|	mean	$< 10^{-6}$	$< 10^{-6}$
	max	$< 10^{-6}$	$< 10^{-5}$
	min	$< 10^{-7}$	$< 10^{-7}$
Memory	mean	10.8KB	11.6KB
	max	13.6KB	14.6KB
	min	7.9KB	8.59KB

Table 4.11: Embedded QP solver performance results - medium to large problems (10-40 decision variables, 60-320 constraints).

Metric	Statistic	`quad_wright`	`quad_mehrotra`
Solution Time	mean	48.35ms (20.68Hz)	43.13ms (23.19Hz)
	max	147.64ms (6.77Hz)	138.94ms (7.2Hz)
	min	4.47ms (223.46Hz)	4.35ms (229.46Hz)
|Objective Error|	mean	$< 10^{-4}$	$< 10^{-5}$
	max	0.00577	$< 10^{-4}$
	min	$< 10^{-7}$	$< 10^{-7}$
Memory	mean	48.7KB	51.0KB
	max	102.0KB	106.0KB
	min	15.5KB	16.5KB

solvers. The package is written by Alexander Domahidi of ETH Zürich, and includes a web-based automatic code generator [82], very similar in concept to that proposed by this work. For this reason a comparison with the framework of this work is presented, together with a performance comparison.

One of the main advantages of the FORCES package is the ability to solve Quadratically Constrained Quadratic Programs (QCQP), also known as Second Order Cone Problems (SOCP). It is acknowledged that this is not possible within the jMPC framework, however it was also not the design intention, as the problems of interest do not contain quadratic constraints. Otherwise the two packages are quite similar in philosophy and design: they both utilize interior-point methods for solving the online optimization problems (reinforcing our argument that interior-point methods are suited for MPC problems), they are both MATLAB based, and they both generate ANSI C code targeted at embedded controllers. However the high-level language is vastly different between the two packages, with FORCES designed for time varying MPC formulations, requiring manual entry of the objective, constraints and dimensions at each stage (i.e. time within the horizon), while jMPC is designed for a constant formulation, based on its simplicity.

The code-generator included in the FORCES package utilizes a web-server to generate the embedded code, which is then transmitted back to your PC. The code-generation process is remarkably quick, taking only a few seconds for a small problem. This is opposed to CVXGEN [210], which can take several minutes to generate the C code implementation. Furthermore, FORCES adopts a similar strategy to this work, whereby the solver is not completely unrolled (as in CVXGEN), so as to limit the resulting code-size. The code is however generated for a particular problem size, and thus would require regeneration to re-tune.

To compare the performance of FORCES with the proposed approach, the FORCES simple MPC example (`http://forces.ethz.ch/doku.php?id=examples:simplempc`) is used as benchmark. This example applies MPC to control of a unstable double integrator, which results in a quadratic program with 10 variables and 60 constraints, thus falls within the problem size range of this work. The FORCES package formulates the MPC problem in a slightly different fashion (no reference tracking and adds additional terminal weights and constraints for stability), thus the controlled responses are slightly different. However as shown in Figure 4.16, the controlled responses are very similar. Table 4.12 compares the execution times of the samples for which the jMPC controlled solved a QP (at the beginning of the simulation), and the same samples within the FORCES simulation, to provide a fair comparison. In addition, the file memory (not RAM requirement, just the file size) is reported, together with the amount of time required to generate *and* compile the solver.

Table 4.12: Performance comparison between jMPC and FORCES - timing and memory results. Note the jMPC results were obtained using the `quad_mehrotra` solver.

Metric	Statistic	jMPC	FORCES
Solution Time	mean	0.069ms	0.166ms
	max	0.1ms	0.203ms
	min	0.047ms	0.143ms
Code Generation Time		1.72s	68s
File Memory		37KB	67KB

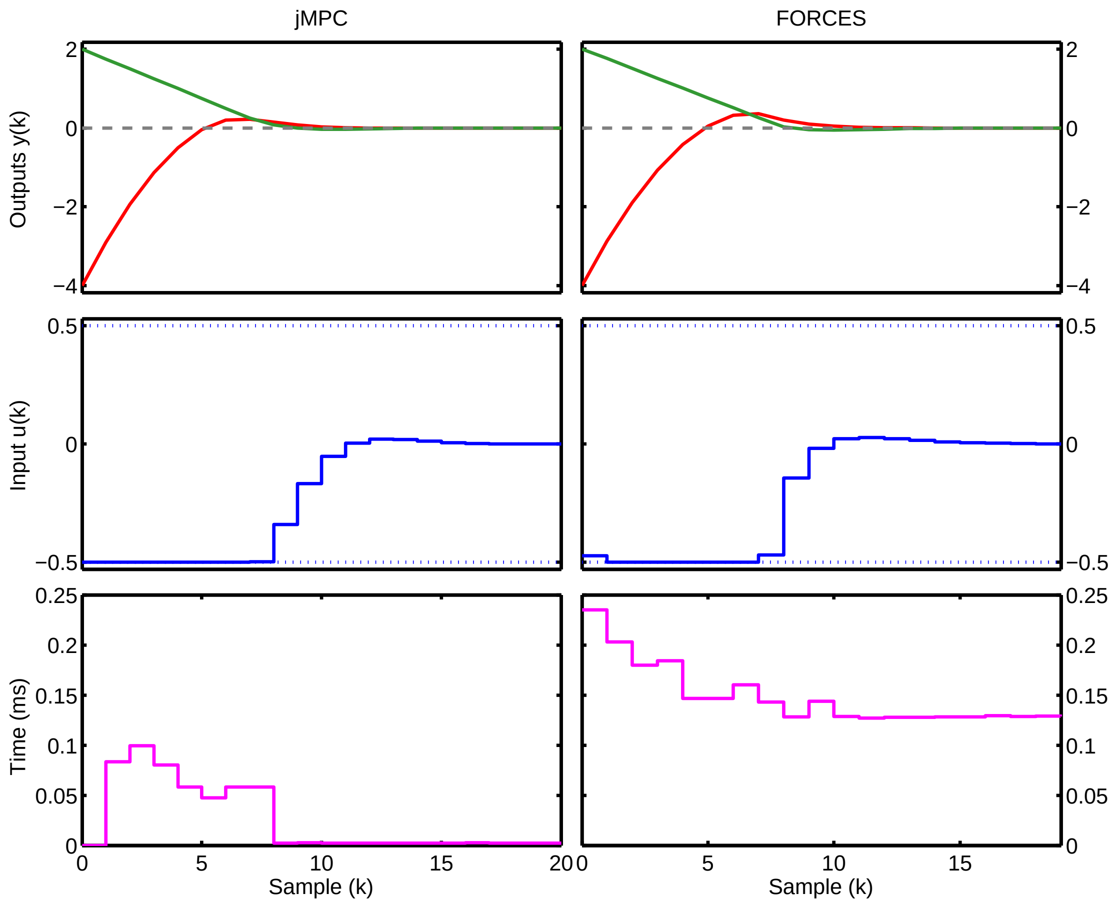


Figure 4.16: Performance comparison between jMPC and FORCES.

As shown in Table 4.12, the jMPC implementation outperforms FORCES on all metrics. It is not only on average 2.4× faster, but requires 45% less memory (in terms of file size), and was 40× faster to generate. While the FORCES solver may be able to solve more complex (and larger) MPC problems, for the problems of interest within this work, the jMPC algorithm together with the `quad_mehrotra` solver is the preferred implementation. Note the solution times are recorded on a 2.8GHz i7 laptop using the operating system's high performance timer, which allows timing approximately accurate to one microsecond.

4.5 Processor-In-The-Loop Embedded MPC

With two efficient QP solvers now developed and the MPC algorithm proven against the FORCES framework, the next task required is to add the complete model predictive control algorithm to the embedded target implementation. With efficient preprocessing and problem generation within MATLAB, together with the auto-code framework described in Section 4.3, a complete MPC controller with QP algorithm can be deployed in as little as 10 seconds. This section will detail verifying the embedded MPC implementation on the target microcontroller via an automated Processor-In-the-Loop (PIL) testing method.

4.5.1 A Processor-In-the-Loop (PIL) Implementation

A PIL implementation places the target processor within the testing loop so that the algorithm under test is run on the final implementation hardware. The key difference though between a final implementation and a PIL implementation is that the development computer retains some control over the target hardware. This allows the algorithm to be verified on the final production hardware, whilst retaining an easy to use, monitor, and control testing environment.

For this work a PIL implementation has been extensively used for verifying the functionality and accuracy of the QP solver (described in Section 4.4.4), however the core focus of the development of the PIL system was for validating the MPC algorithm. Figure 4.17 shows the intended verification system. Within the PIL system used the development computer acts in a dual role, both as the system plant and a supervisory controller. In its role as the system plant, its job is to simulate the dynamic system that is being controlled, whether this be linear or nonlinear. In its role as a supervisory controller, it supplies the current setpoint, providing the reference trajectory for the controller to follow. The embedded MPC controller is run on the target hardware, and communicates with the development computer via a two-way communication channel. For this work, a simple USB serial link is run at 1.25MBaud, with a custom packet system written for communication between the two devices.

The initialization and subsequent operation of a typical PIL verification run is as follows:

1. (PC) An embedded MPC controller is generated from a specified `jMPC` object within MATLAB.
2. (PC) The generated controller is compiled and downloaded to the target.

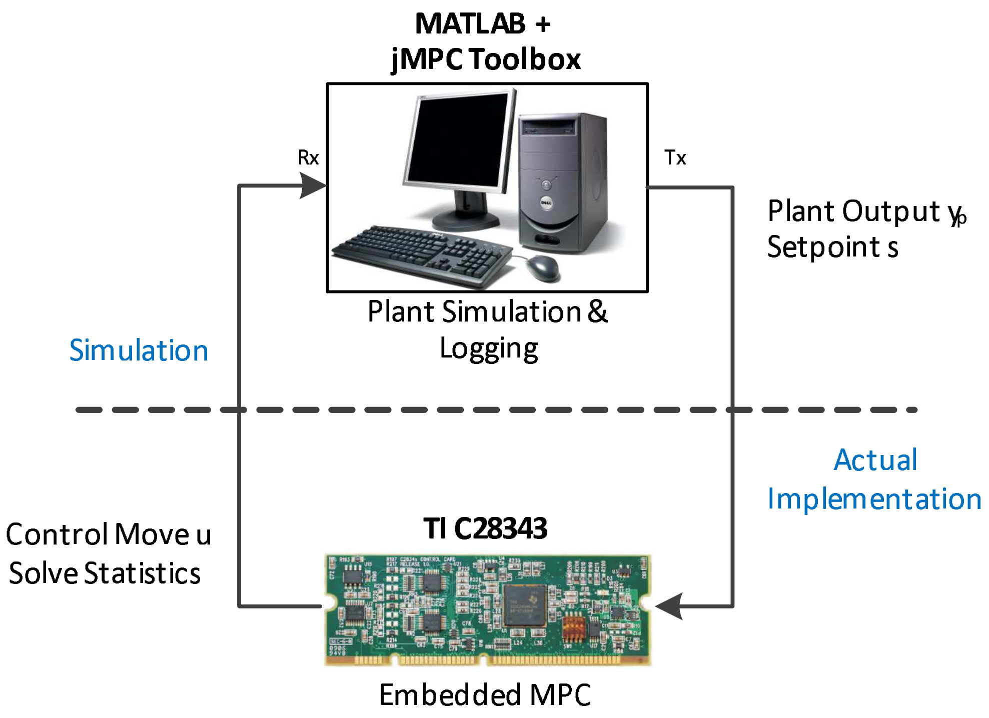


Figure 4.17: Embedded MPC verification using PIL and jMPC with the TI C28343.

3. (TARGET) The controller, now running on the target, completes its setup routines and enters the PIL simulation mode. It now waits for the first set of measurements from the development computer.

4. (PC) A PIL simulation is invoked on the development computer, using the same `jMPC` syntax for normal (PC based) simulations, except with `PIL` specified as the simulation environment.

— —

5. (PC) The current plant output(s) and desired setpoint(s) are transmitted from the development computer to the target via the serial link.

6. (TARGET) Using the transmitted measurements and setpoint the target completes a full MPC control move calculation.

7. (TARGET) The calculated control move, together with solution statistics and model values are transmitted back to the development computer.

8. (PC) The development computer reads and stores the transmitted data, applies the control move(s) to the simulated plant, and the time step is incremented one sample. The PIL verification run repeats from step 5 and continues until no further data is received from the development computer.

As described in the above list, steps 1-4 are the setup and initialization of the PIL loop, while steps 5-8 are the iterative loop. A detailed example of how a

PIL implementation is run under the jMPC Toolbox is described in Section A.4.4, including timing and accuracy results.

4.5.2 Nonlinear Verification using a PIL Embedded MPC

A further advantage of using the jMPC Toolbox and a PIL simulation is the ability to verify the controller on the target within a nonlinear simulation environment. By utilizing the development PC as the system plant, we are free to use an ODE integrator (such as `ode45`) to simulate the nonlinear system dynamics, and thus provide a more robust verification of our linear controller's performance.

To illustrate this process we are going to control a nonlinear *model* of an inverted pendulum on a motorized cart, based on the physical system sold by Quanser [267]. This is a traditional nonlinear control problem that exhibits fast dynamics and thus requires high-speed control. Figure 4.18 shows the physical system.

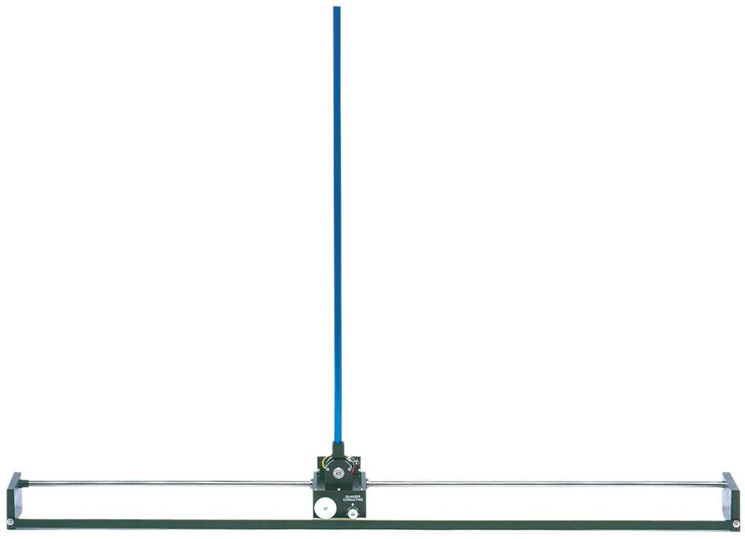


Figure 4.18: Quanser IP02 inverted pendulum control challenge [267].

As this is a traditional system the nonlinear Equations Of Motion (EOM) are well known. These are repeated below from [264]

$$\begin{aligned}\ddot{y} &= \frac{u + ml\,(\sin\theta)\,\dot{\theta}^2 - mg\cos\theta\sin\theta}{M + m - m\cos^2\theta} \\ \ddot{\theta} &= \frac{u\cos\theta - (M+m)\,g\sin\theta + ml\,(\cos\theta\sin\theta)\,\dot{\theta}^2}{ml\cos^2\theta - (M+m)\,l}\end{aligned} \tag{4.1}$$

and are used for both the system plant, as well as generating a linearized control model. The parameters for this system are listed in Table 4.13 for the example Quanser system.

Table 4.13: Quanser IP02 inputs, outputs and system parameters.

u	N	Force applied to the cart
θ	rad	Angle of pendulum from vertical
y	m	Position of cart
M	0.455kg	Mass of the cart
m	0.21kg	Mass of the pendulum
l	0.305m	Distance to the centre of mass
g	9.81m/s^2	Gravitational constant

The system differential equations are entered into a MATLAB function in Cauchy form, and saved as the function `nl_pend`. To generate a linear approximation of the pendulum the `jNL` class is used together with the `nl_pend` function as follows:

```
% Nonlinear Plant Model
Plant = jNL(@nl_pend,C,{M,m,l,g});

% Linearize about u = 0N
Model = linearize(Plant,0)
```

This results in the following linear state-space system:

$$\begin{bmatrix} \dot{y} \\ \ddot{y} \\ \dot{\theta} \\ \ddot{\theta} \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & -4.528 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 47.01 & 0 \end{bmatrix} \begin{bmatrix} y \\ \dot{y} \\ \theta \\ \dot{\theta} \end{bmatrix} + \begin{bmatrix} 0 \\ 2.198 \\ 0 \\ -7.206 \end{bmatrix} \begin{bmatrix} u \end{bmatrix}$$

Note the derived linear system is identical to that described in the original reference [264]. In addition, the `linearize` method has automatically solved for the steady state of the system, found to be [0,0,0,0] (centred, vertical, not moving). The output matrix, C, simply picks off y and θ as the system outputs.

At this point an MPC controller can be designed using the linearized model. The tuning weights are set as `ywt` = [1.5, 0] (do not control pendulum angle, only constrain it) and `uwt` = 2, and predictions of $N_p = 30$ and $N_c = 5$. In addition the following constraints are used

$$-10\text{N} \leq u \leq 10\text{N}$$
$$-2\text{m} \leq y \leq 2\text{m}$$
$$-0.785\text{rad} \leq \theta \leq 0.785\text{rad}$$

The model is discretized at a sampling rate of 20Hz and an embedded MPC controller is generated.

When the PIL implementation is run, rather than using a simple discrete state-space update of the plant state, MATLAB's `ode45` is used to integrate the system ODEs across each sample on the development computer. The plant model can now predict the system behaviour much more accurately, including the nonlinear effects as the pendulum moves further from the linearization point (in this case, vertically upright). Figure 4.19 shows the result of the PIL implementation, utilizing `quad_mehrotra` as the QP solver and implemented in single precision. As shown

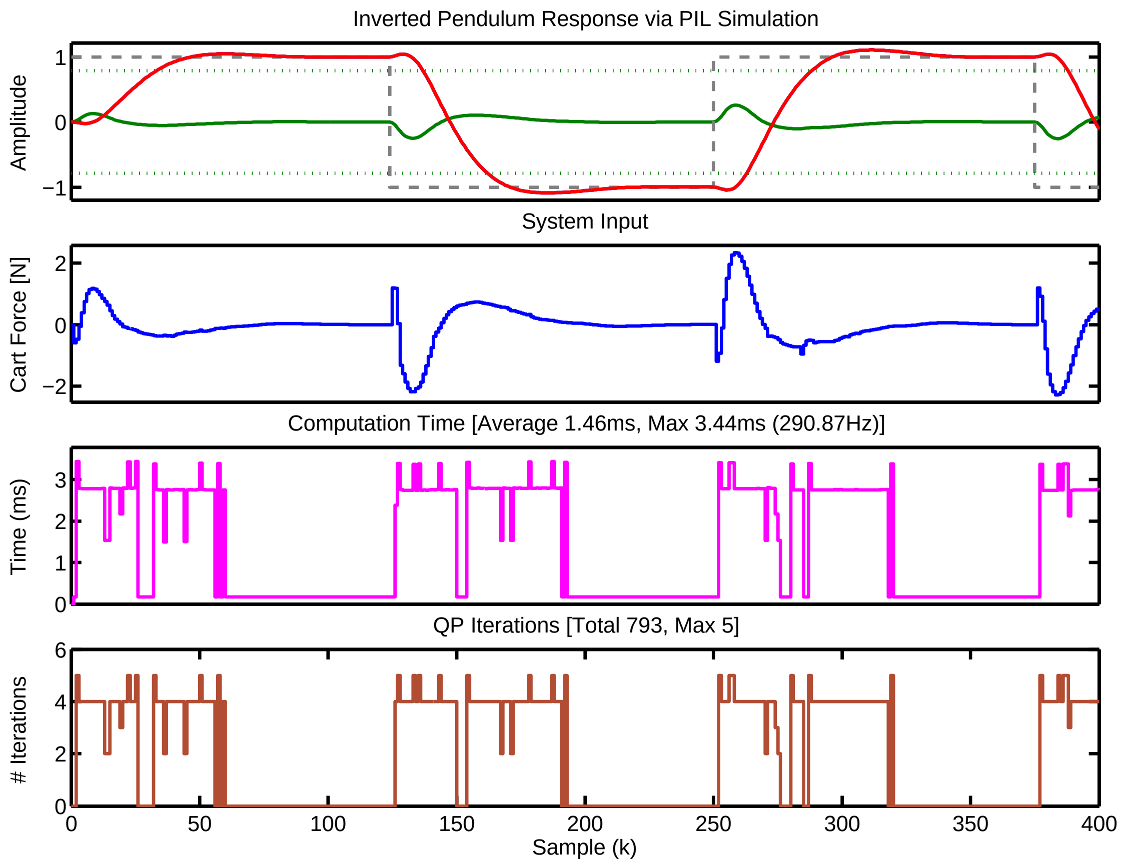


Figure 4.19: Embedded linear MPC control of a nonlinear inverted pendulum model. The pendulum model is simulated on the development computer, and control actions obtained via a PIL implementation of the TI C28343. The top red trace is the cart position in metres (controlled) and green trace is the pendulum angle from vertical in radians (uncontrolled).

the embedded MPC controller achieves a maximum sampling rate of around 290Hz, indicating the controller could be pushed much faster. However the control performance of the system is satisfactory given the desired tuning and thus there is no further reason to increase the sampling rate. A more aggressive controller on the other hand may require a faster sampling rate. Inspecting the compiler `.map` file the controller required 28.3KB to store all problem data and the algorithm, which includes data for solving the QP at each sample with 5 decision variables and 130 constraints.

To verify the control performance is as expected, `compare` is used to validate the PIL results against a simulation run within MATLAB. Note increased differences are to be expected due to the MATLAB MPC and QP algorithms varying slightly from the embedded MPC algorithm, based on leveraging different linear algebra libraries and a substantially different operation order (due to different compilers). The comparison results are shown in Figure 4.20.

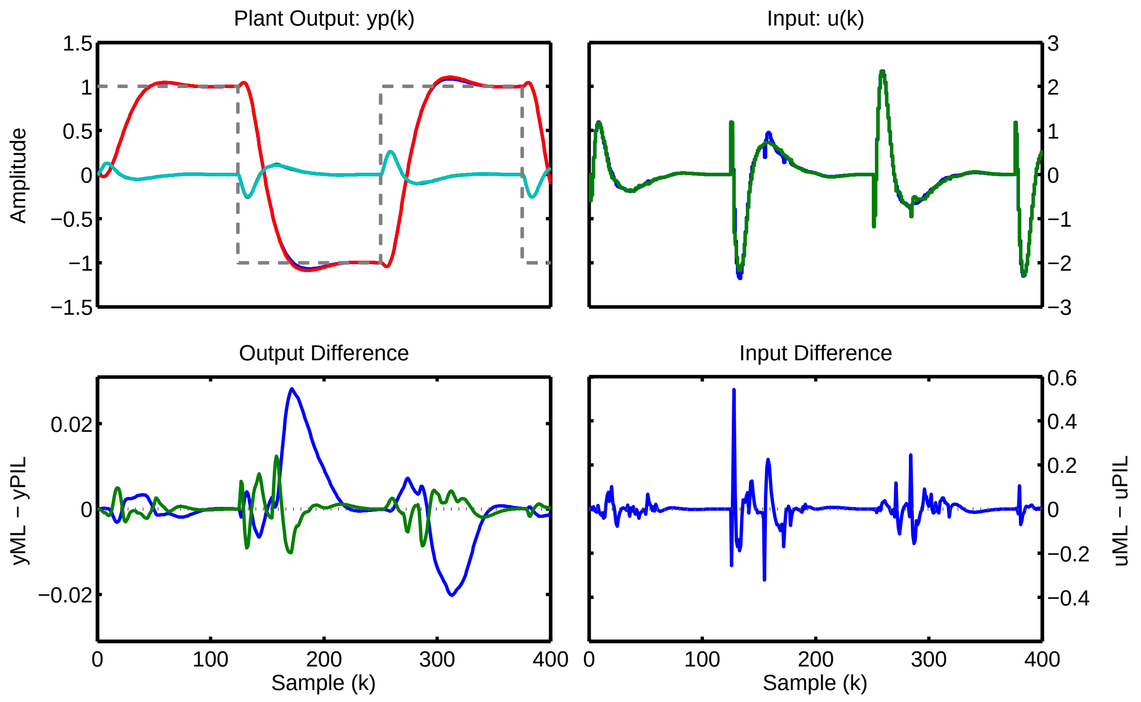


Figure 4.20: Embedded linear MPC control of a nonlinear inverted pendulum model comparison: MATLAB vs TI C28343.

This example has demonstrated one of the powerful code verification and tuning validation features of this work: The ability to run the generated MPC controller on the target hardware while controlling a detailed nonlinear plant via a PIL simulation. In addition, the linear model is automatically derived about a user specified operating point, allowing the user to concentrate on tuning and verification. The result is that effort and re-tuning required when transferring the embedded MPC controller from a PIL implementation to a real implementation on the physical plant should be substantially reduced, provided the nonlinear model is correctly identified.

4.5.3 Comparison with Literature

To compare the performance of the proposed algorithm with that reported in the literature, two PIL case-studies are presented.

4.5.3.1 Cessna Citation 500 Model [186]

One of the de-facto models used for validating MPC performance is the Cessna Citation 500 model used by J. Maciejowski in his book, "Predictive Control with Constraints" [192]. The linearized, continuous time model is shown in Equation 4.2 and models the aeroplane pitch angle (y_1, rad), altitude (y_2, m) and altitude rate (y_3, m/s) as a function of the control input, elevator angle (u, rad) and current state.

$$\begin{aligned}
\hat{\mathbf{A}} &= \begin{bmatrix} -1.2822 & 0 & 0.98 & 0 \\ 0 & 0 & 1 & 0 \\ -5.4293 & 0 & -1.8366 & 0 \\ -128.2 & 128.2 & 0 & 0 \end{bmatrix}, \quad &\hat{\mathbf{B}} &= \begin{bmatrix} -0.3 \\ 0 \\ -17 \\ 0 \end{bmatrix} \\
\hat{\mathbf{C}} &= \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ -128.2 & 128.2 & 0 & 0 \end{bmatrix}, \quad &\hat{\mathbf{D}} &= \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}
\end{aligned} \tag{4.2}$$

The performance comparison will be made against an embedded MPC FPGA implementation of this system as described in [186]. The model is discretized with a sampling rate of 0.5 seconds and an MPC controller is designed with $N_p = 10$, $N_c = 3$ and tuning weights of 1 for the input and all outputs.

Two case studies are presented in the paper which vary only by the constraints imposed on the system. The first case study applies the following constant constraints

$$\begin{gathered}
-0.524 \leq \Delta u \leq 0.524 \\
-0.262 \text{ rad} \leq u \leq 0.262 \text{ rad} \\
-0.349 \text{ rad} \leq y_1 \leq 0.349 \text{ rad}
\end{gathered}$$

where y_1 is the pitch angle of the Cessna. From this description a `jMPC` controller is created in MATLAB, a single precision controller is automatically generated in C and a PIL simulation run which mimics the control specification within the paper. Figure 4.21 shows the PIL implementation result and Table 4.14 compares the results from the paper.

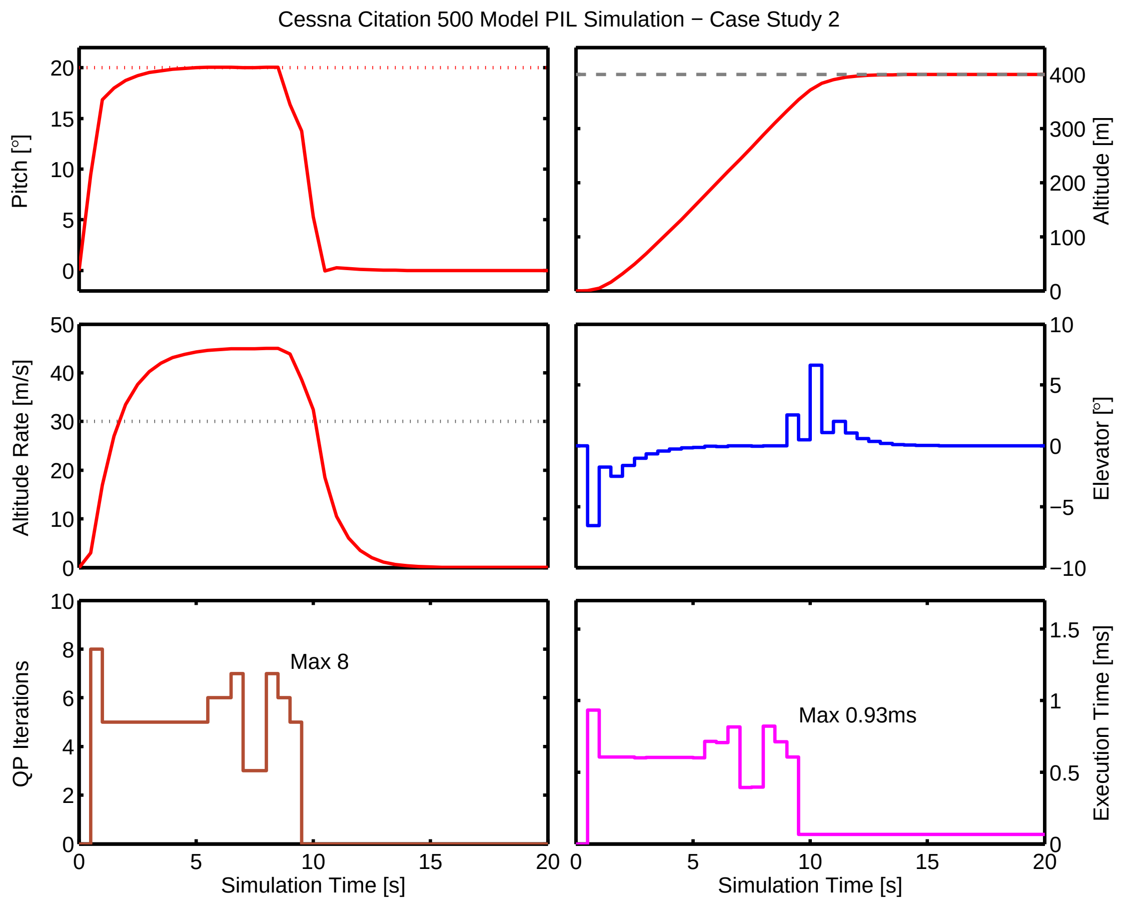


Figure 4.21: Embedded MPC control on the TI C28343 of a Cessna Citation 500 model, case study 2 in [186]. Note the 30m/s altitude rate constraint is not active.

Table 4.14: Cessna Citation 500 case study 2 results comparison.

Implementation	Scheme	Average MPC Sampling Interval	Average Number of QP Iterations
Ling et al. [186]	Sequential (25MHz)	5.2ms	12.3
Ling et al. [186]	Parallel (25MHz)	2.9ms	12.3
`jMPC`	Sequential (200MHz)	0.64ms[1]	5.4[1]
`jMPC`	Sequential (200MHz)	0.32ms[2]	2.3[2]

[1] Calculated over the active QP samples only (for direct comparison).
[2] Calculated over all samples.

One important metric missing from the original reference was the maximum sampling interval, which directly corresponds to the maximum achievable sampling rate. However by inspecting Figure 5(b) within the reference it is possible to estimate the maximum number of QP iterations is 17. This can then be used to estimate the maximum sampling rate using Table 3 (within the reference) as 7.19ms (139Hz) for the sequential algorithm, 4ms (250Hz) for the parallel algorithm, versus 0.93ms (1075Hz) for the `jMPC` sequential implementation. In addition, the memory consumption of the reference algorithm is not specified (e.g. block RAM usage), thus no direct comparison can be made. For the reader's reference, the `jMPC` controller required 18.7KB to store the entire controller and data memory.

The final case study presented by Ling et al. includes an extra constraint on the altitude rate of change

$$-30m/s \leq y_3 \leq 30m/s$$

which replaces the pitch constraint as the active constraint for the majority of the simulation. Figure 4.22 shows the result for this case on the TI C28343 and Table 4.15 compares the results to the paper.

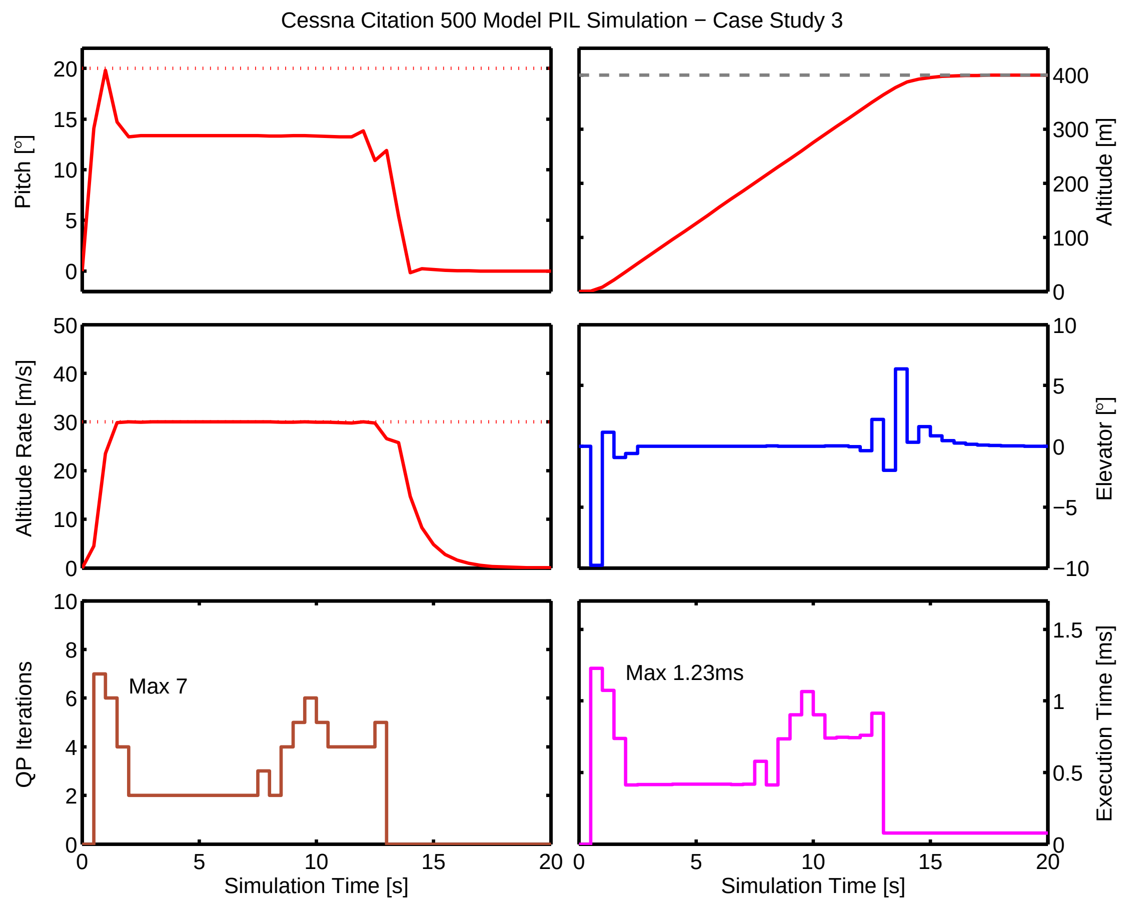


Figure 4.22: Embedded MPC control on the TI C28343 of a Cessna Citation 500 model, case study 3 in [186].

Table 4.15: Cessna Citation 500 case study 3 results comparison.

Implementation	Scheme	Average MPC Sampling Interval	Average Number of QP Iterations
Ling et al. [186]	Sequential (25MHz)	9.1ms	14.8
Ling et al. [186]	Parallel (25MHz)	4.9ms	14.7
`jMPC`	Sequential (200MHz)	0.62ms[1]	3.25[1]
`jMPC`	Sequential (200MHz)	0.42ms[2]	2.1[2]

[1] Calculated over the active QP samples only (for direct comparison).
[2] Calculated over all samples.

Comparing Tables 4.14 and 4.15 it is evident why simply analyzing the average MPC sample interval can give misleading results. The larger QP of Case 3 averages at 0.62ms (active QP samples), while the smaller QP of Case 2 averages at 0.64ms. This is a result of both QP warm starting, as well as the effect of the different constraints on the control problem. Comparing the maximum sample interval, Case 3 requires 1.23ms, versus 0.93ms for the smaller Case 2, the expected result of a faster time for a smaller problem.

One conclusion that could be drawn from this comparison is that it is unfair in terms of raw clock speed, given the `jMPC` TI C28343 is running at 8x the speed of the Ling et al. Xilinx FPGA. However there are a number of points that need to be considered when comparing these two embedded implementations:

FPGAs vs Microcontrollers As described in Section A.4.1, it is very hard to compare the same algorithm implemented in an FPGA, to that implemented in a microcontroller. This is due to the inherent parallelized data paths available within an FPGA as well as the substantially different architecture (customizable vs fixed). The typical clock speed of a small and inexpensive modern FPGA is still around 50MHz, which although can be multiplied using a Phase-Locked-Loop (PLL), is still typically limited by memory (internal Block RAM) access speeds to around 200MHz. Obviously higher performance FPGAs are available, but so too are higher performance microcontrollers (such as the common ARM architecture at circa 1GHz). It is not the focus of this work to target high performance ICs, but rather low-cost, low-power, easy to use processors.

Maximum Clock Speed Given the aim of both this work, and Ling et al. was to develop the fastest embedded MPC implementation, there would be no reason why a lower clock rate than the maximum achievable were to be used. For the TI C28343 the maximum (safe) clock rate is 200MHz, so this was used. It is assumed that the maximum clock rate available for the Xilinx FPGA that still allows the MPC algorithm to be implemented was 25MHz. Therefore using this Xilinx FPGA, simply increasing the clock rate is most likely not possible, and a different (more powerful) embedded target would be required.

Development Tools Handel-C was used by Ling et al. to generate the Hardware Description Language (HDL) from a C-code representation of the algorithm, which is required to generate the FPGA design. This was part of Celoxica DK2, a software package that cost over US$7000.00 (based on information that can be found online - the package is now obsolete). In addition, MATLAB and Simulink were used as part of the PIL testing environment, leading to an expensive proposition to develop low-cost controllers. For the `jMPC` auto-coding framework, only MATLAB and TI Code Composer Studio (CCS) are required, and CCS can be obtained free for academics, or from only US$450.00 for a single user license.

Compilation Time FPGA compilation is notoriously time consuming as the design tool must actually synthesize a real hardware layout, then place it within the available FPGA resources. It is not uncommon for this process to take minutes to hours in time, even for modest designs. In contrast, compiling a C program is a routine exercise, and only takes a few seconds using CCS. Depending on how the FPGA design is implemented, recompilation might be required for every tuning adjustment (for example if N_p is increased, the problem dimensions all also increase), thus limiting the ability to re-tune 'on the fly'.

Hardware Cost The complete TI C28343 development kit is available from US$159.00 [308], whereas the (now obsolete) Ceroxica RC203 sold for around US$1500.00. While it is true for modern FPGA systems that development kit prices have also rapidly decreased, the Ceroxica (now Mentor Graphics) boards are still comparatively more expensive than the equivalent TI development kit.

Based on the above points a direct comparison of clock speed using a constant factor should not be performed, but rather the entire design system and the final timing results viewed as published. However a quick calculation shows the `jMPC` implementation on the TI C28343 is still faster, achieving a sampling rate of 814Hz for Case 3 at 200MHz, and (hypothetically) the RC203 achieving an estimated sampling rate of 730Hz for the parallel implementation if run at 200MHz.

With respect to accuracy, the results presented in the original paper appear to correlate well with Figures 4.21 and 4.22, indicating the control algorithm is performing as expected.

4.5.3.2 Rotating Antenna Assembly [320]

The next case study which will be investigated is the control of a small SISO rotating antenna model used by M.Kothare in [174] for investigating robust MPC, which was also used in [38] for an embedded MPC implementation on a Motorola MPC555 processor. For this case-study we will be looking at recent work in [320] where the authors implemented an ADCUS EISC processor together with a patented (by the authors) matrix coprocessor in a Xilinx Virtex-IV FPGA. This setup is a complex, non-standard implementation requiring specialist knowledge, compilers, and hardware and thus provides an effective benchmark.

It is worth pointing out that the MPC cost function used in [320] is nonlinear, and is solved using a modified Newton's method. However the case-study presented utilizes a linear model, and thus the resulting cost-function reduces to a QP. For the purposes of a fair(er) comparison, the time penalty associated with approximating the gradient and Hessian online has been removed, with only the algorithm and solver time being compared.

The linearized, discrete time model of the rotating antenna is shown in Equation 4.3 and models the dynamics of an electric motor driven antenna. The control problem is to rotate the antenna so that it is always facing the desired target. As with the original reference, it is assumed both the antenna angular position (rad) and angular velocity (rad/sec) are measurable.

$$\begin{aligned} \hat{\mathbf{A}} &= \begin{bmatrix} 1 & 0.1 \\ 0 & 0.9 \end{bmatrix}, & \hat{\mathbf{B}} &= \begin{bmatrix} 0 \\ 0.0787 \end{bmatrix} \\ \hat{\mathbf{C}} &= \begin{bmatrix} 1 & 0 \end{bmatrix}, & \hat{\mathbf{D}} &= \begin{bmatrix} 0 \end{bmatrix} \end{aligned} \tag{4.3}$$

The model has been discretized with a sampling rate of 0.1 seconds and an MPC controller designed with $N_p = 10$ and N_c ranging from 3 to 10. The tuning weights are specified as `uwt` $= 1$ and `ywt` $= 3$. In addition the voltage supplied to drive the antenna is constrained to

$$-2\text{V} \leq u \leq 2\text{V}$$

As with the previous case-study, a `jMPC` controller is created in MATLAB, a single precision controller is automatically generated in C and a PIL simulation is run which mimics the control specification within the paper. Figure 4.21 shows the PIL implementation result and Table 4.16 compares the results from the paper.

Viewable in the results comparison table we see that the TI implementation

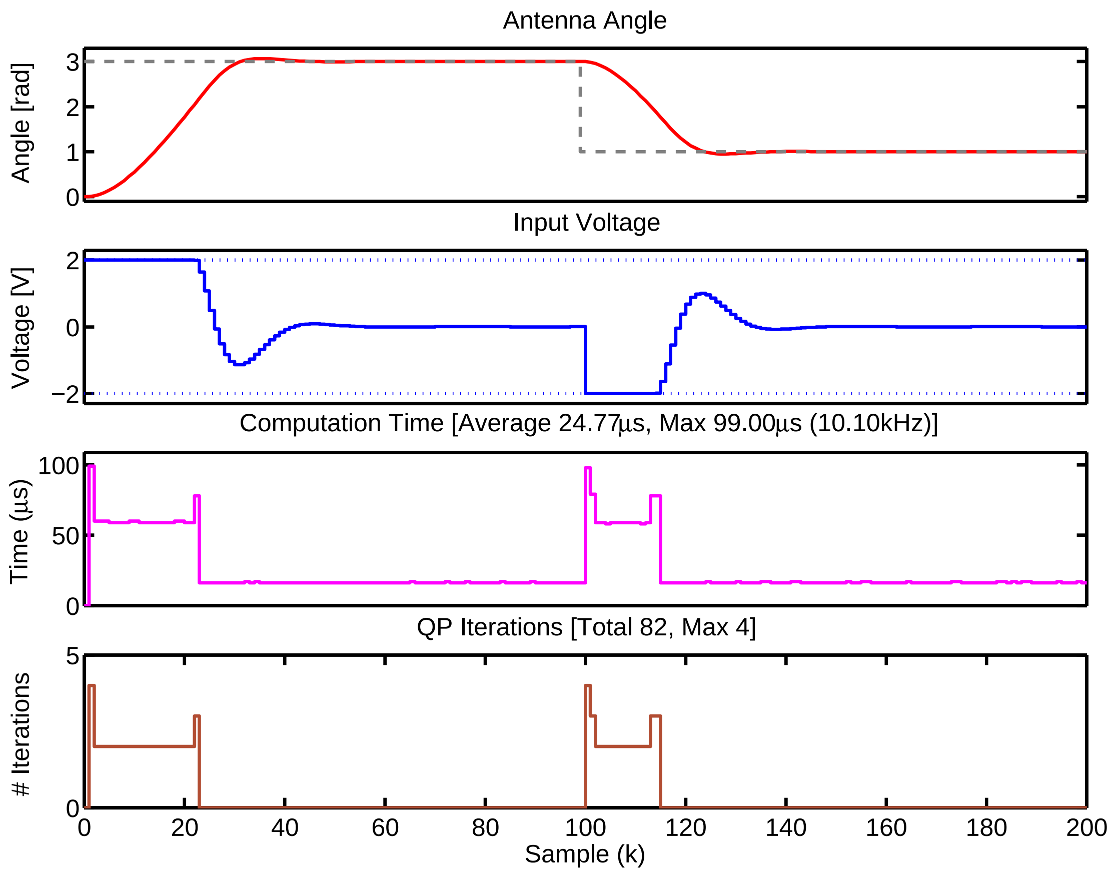


Figure 4.23: Embedded MPC control on the TI C28343 of a rotating antenna assembly model, case study 1 in [320] with $N_c = 2$.

achieves sampling rates exceeding 10kHz with a modest memory footprint for the entire controller (as measured from the compiler `.map` file). Comparing the performance to the reference results, which should be viewed with sensible judgement given the TI is solving a QP, and the Vouzis controller solving an unconstrained nonlinear program, we see a substantial performance improvement of up to 50 times. Even when clock rates are hypothetically aligned at 200MHz (the reference work is at 50MHz), the TI with the jMPC controller and `quad_mehrotra` solver is still substantially faster.

Table 4.16: Rotating antenna assembly results comparison.

Metric	TI Max	TI Total	TI C28343 Max	Vouzis [320] Max
N_c	QP Iter	Memory [KB]	Time [ms]	Time[1] [ms]
2	4	9.14	0.098	5.69
3	4	9.86	0.153	8.4
4	4	10.1	0.212	11.9
5	4	10.7	0.288	16
6	5	11.3	0.456	20.9
7	5	12.1	0.580	26.6
8	5	12.5	0.714	33.3
9	5	13.9	0.882	40.6
10	6	14.5	1.25	48.7

[1] This data is not reported in the original reference and thus has been estimated from the data in Table 3 and from the single piece of timing information where it is stated 10952 clock cycles required 0.45ms to execute (reported in Section A of [320]). In addition, a maximum of 30 iterations has been estimated as required by the Newton solver, based on the same algorithm in Figure 7 in [38] (a very conservative number). Finally, as mentioned, only the Newton solver and GJ inversion cycles have been used to provide a comparative estimate between the nonlinear and quadratic cost functions (a rough approximation).

A few final thoughts on this comparison is that while the Vouzis controller is solving an unconstrained nonlinear program at each sample, there is no need for this type of formulation given the system is linear (as reported in the original paper) and thus a QP formulation will suffice. In addition, the time unit reported in Figure 5 in [320] is in minutes, which indicates the real system is very slow. If this is the case (and not a typographic error), then this system does not provide a representative example of where high speed control would be required. To therefore test the `jMPC` controller on a high-speed system, we now move from an PIL implementation to a real MPC implementation controlling a real nonlinear system with fast dynamics.

4.6 Embedded MPC Case-Study

One of the themes of this work is a strong practical focus and therefore this chapter would not be complete without an actual implementation of MPC on a real system. A challenging piece of experimental equipment we had access to was a 3 Degree-Of-Freedom (3DOF) Helicopter from Quanser [268], pictured in Figure 4.24. The

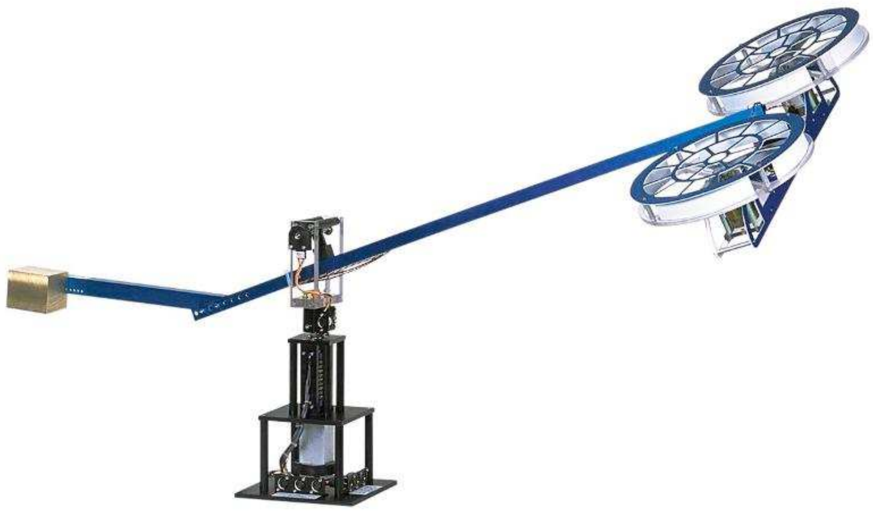


Figure 4.24: Quanser 3DOF Helicopter plant [268].

system is equipped with 3 quadrature encoders mounted on each of the three rotation axes, as well as two powerful DC motors which provide thrust via each of the propellers. Electrical signals are transmitted via slip rings mounted on the central shaft, allowing full 360 degree rotation.

The system is supplied with a USB based data acquisition card (Q8-USB, to read the encoder values), linear voltage amplifier (VoltPAQ-X2 to drive the motors) and modelling software (QuaRC) to control the system via Simulink and MATLAB. Typically the system is controlled via a Quanser-supplied LQR controller that runs at 1kHz on a development computer, which achieves reasonably good control performance. However, for this work only the linear voltage amplifier will be retained (in order to supply the required motor current), and custom hardware and software will replace the LQR controller within QuaRC and Simulink.

4.6.1 3DOF Helicopter Model

A dynamic model for the system was developed by Quanser and is illustrated in Figure 4.25. Each of the three axes is modelled in terms of the thrust acting upon it due to the voltage supplied to the motors, the mass of each component as well as system inertia and gravity. The result is the set of second-order nonlinear differential equations shown in Equation 4.4 with parameters given in Table 4.17.

$$
\begin{aligned}
\ddot{\epsilon} &= \left(\frac{K_f l_a}{J_e}\right)(V_f + V_b)\cos(p) - \frac{T_g}{J_e} \\
\ddot{p} &= \left(\frac{K_f l_h}{J_p}\right)(V_f - V_b) \\
\ddot{\lambda} &= -\left(\frac{F_g l_a}{J_t}\right)\sin(p)
\end{aligned}
\tag{4.4}
$$

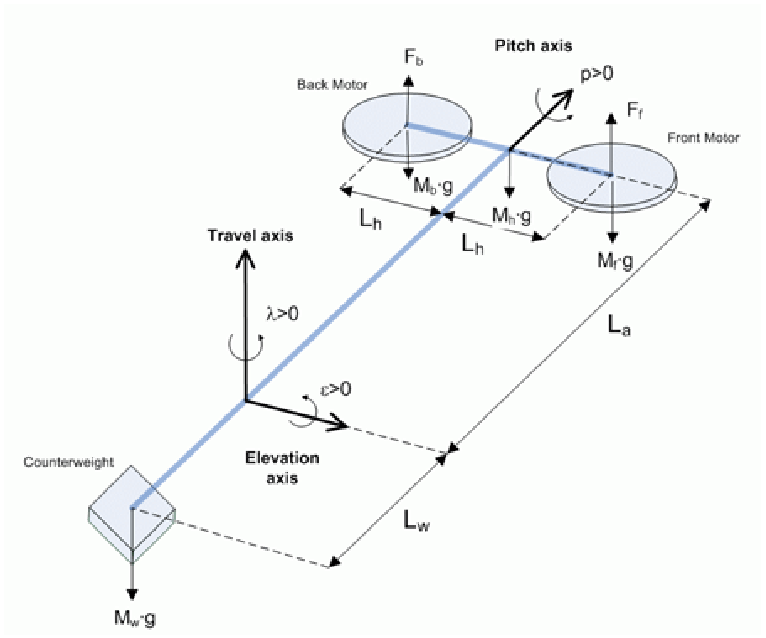


Figure 4.25: 3DOF Helicopter free-body diagram [265].

Table 4.17: Quanser 3DOF helicopter inputs, outputs and system parameters.

$V_f = u_1$	V	Front motor Voltage
$V_b = u_2$	V	Back motor Voltage
$\epsilon = y_1$	rad	Elevation angle
$p = y_2$	rad	Pitch angle
$\lambda = y_3$	rad	Travel (rotation) angle
K_f	0.1188 N/V	Propeller force-thrust constant
m_h	1.15 kg	Mass of the helicopter body
m_w	1.87 kg	Mass of the counter-weight
l_a	0.6604 m	Distance from elevation pivot to helicopter body
l_h	0.1778 m	Distance from elevation pivot to counter-weight
J_e	0.91 kg·m^2	Moment of inertia about elevation axis
J_p	0.0364 kg·m^2	Moment of inertia about pitch axis
J_t	0.9508 kg·m^2	Moment of inertia about travel(rotation) axis
F_g	1.7715 N	Mass differential about elevation axis
T_g	1.1699 Nm	Effective differential gravitational torque due to F_g
g	9.81m/s^2	Gravitational constant

As with the inverted pendulum example, the differential equations are entered into a MATLAB function in Cauchy form, but this time saved as `nl_heli`. The resulting function contains six coupled first-order ordinary differential equations suitable for use with the toolbox. A linear approximation is found about the following input operating point

$$V_{op} = \frac{g\left(l_w m_w - l_a m_h\right)}{2 l_a K_f} \tag{4.5}$$

which is the voltage required to both the front and back motors in order for the helicopter to be level (elevation and pitch axis = 0 rad, travel assumed = 0 rad as well). Using the `jMPC linearize` method results in the following linear approximation about this point

$$\begin{bmatrix} \dot{\epsilon} \\ \dot{p} \\ \dot{\lambda} \\ \ddot{\epsilon} \\ \ddot{p} \\ \ddot{\lambda} \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1.23 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} \epsilon \\ p \\ \lambda \\ \dot{\epsilon} \\ \dot{p} \\ \dot{\lambda} \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0.0862 & 0.0862 \\ 0.5803 & -0.5803 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} V_f \\ V_b \end{bmatrix}$$

which closely matches the symbolic solution obtained in [265]. Small variations are expected as this model is obtained via finite difference, while the original model was developed symbolically using Maple.

4.6.2 Nonlinear PIL Validation

Using the PIL framework together with the nonlinear model we can design and test an MPC controller on the target and verify its performance before implementation on the real system. Given the real system is quite nonlinear, very unstable, and rather expensive, this is an effective means of developing baseline controller tuning parameters without damaging the real system.

The outputs of the system we will control are the elevation and rotation angles, while the pitch angle is left implicitly controlled via the rotation setpoint. The system is constrained as follows

$$\begin{gathered} -20\text{V} \leq V_f \leq 20\text{V} \\ -20\text{V} \leq V_b \leq 20\text{V} \\ -30^\circ \leq p^* \leq 30^\circ \end{gathered}$$

where the input voltage limits are set by the motor and amplifier specifications, and the pitch constraint is implemented to reduce unmodelled nonlinear effects about the pitch axis (as well as providing a visible indication of correct constraint handling). The pitch constraint is implemented as a soft constraint (dictated by the asterisk) with a penalty weight of 350 so that the QP remains feasible at each sample. Tuning weights of `uwt` = [1,1] and `ywt` = [12,0,8] are used to design the controller so that it will attempt to rapidly minimize setpoint deviations.

The remaining tuning parameters are the sampling rate and prediction and control horizons. These cannot be determined independently as the prediction and control horizons are specified in *samples*, and thus the faster the sampling rate, the shorter the horizons in real time. This conundrum means a balance needs to be found between fast sampling rates (for faster disturbance rejection, better dynamic handling) and the required prediction and control horizon lengths. Typically, increasing the sampling rate requires the horizons to be also increased (in terms of samples), which then requires a larger QP to be solved, leading to longer computation times. As the problem size grows and computation times increase, eventually the controller will not be able to keep up with the requested sample rate, or a memory limit will be hit. As described in later in Section 4.6.4, a sampling rate of 33.3Hz was found to provide adequate control performance, and N_p and N_c are set as 80 and [5,5,70] (blocking moves) respectively.

The sampling rate was chosen based on that achievable by the current implementation, which as shown in Figure 4.26, was limited to approximately 100Hz. To provide a buffer if further iterations were required, this sample rate was reduced to one third of the maximum achievable. The horizons were chosen with consideration of the required number of decision variables (hence the use of blocking moves), as well as the desire for a long prediction to reduce rotation axis overshoot, which would increase the number of constraints. All three metrics were tuned subsequently when implemented on the real system, described in Section 4.6.4. Note a faster sampling rate was found to make little difference to the controller performance, provided it was greater than 10Hz.

Figure 4.26 illustrates the control performance of the MPC controller implemented on the TI C28343, and connected via a PIL implementation to the nonlinear model. The system is subjected to a simple elevation setpoint change (15 degrees for 'lift off') and two rotation axis setpoint changes, one of 90 degrees and one of 150 degrees, providing a challenging control problem. These large rotation setpoint changes will stress the pitch constraints, as well as test whether the prediction horizon is long enough to avoid significant overshoot. Given the current tuning, the controller averages around a maximum 100Hz sampling rate and requires 56KB to store the algorithm and problem data, which includes the large QP with 7 decision variables (2 inputs × 3 blocking moves + 1 variable for the soft constrain penalty term) and 172 constraints.

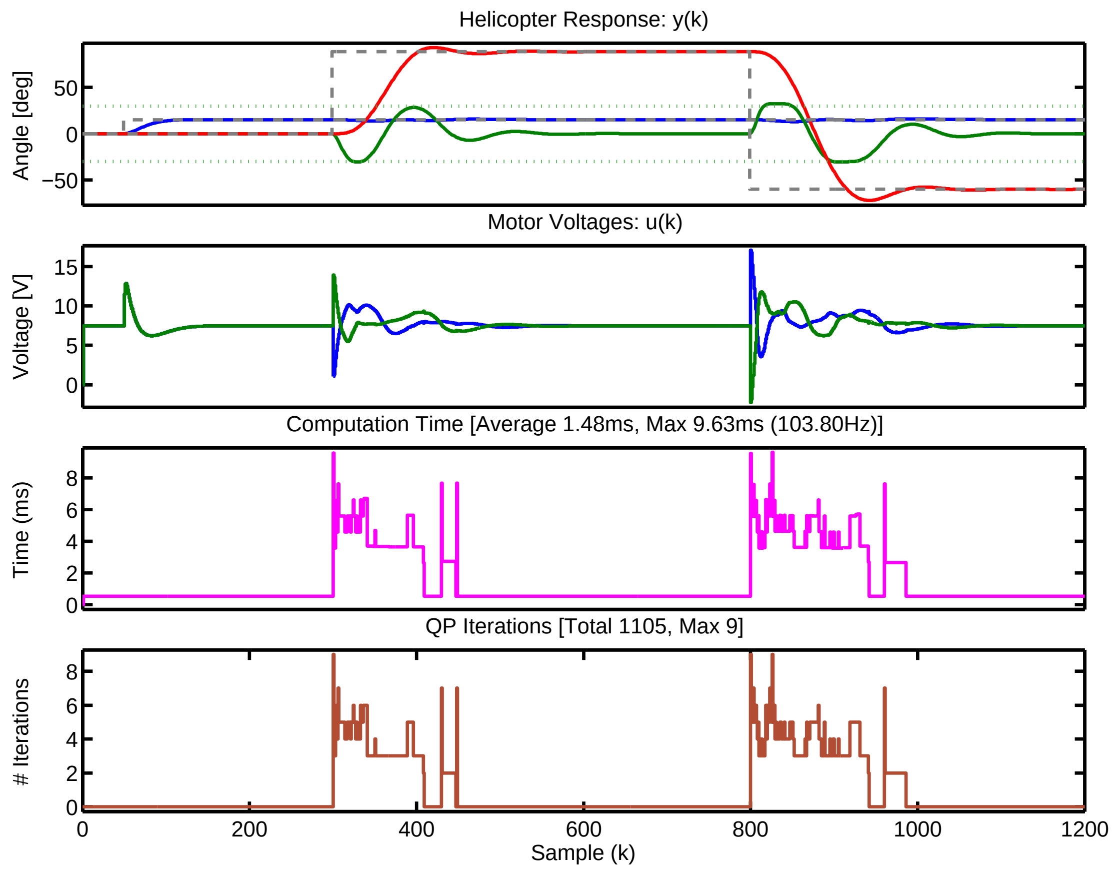


Figure 4.26: Embedded MPC control of a nonlinear 3DOF Helicopter model using a PIL implementation. In the top plot the blue trace is the elevation angle, the green the pitch angle and the red the rotation (travel) angle. For the control inputs, the blue is the front motor and the green the back motor. Note the nonlinear helicopter model is run as a simulation on the development computer.

The validation of the generated controller is shown in Figure 4.27 and shows the TI C28343 implementation matches the MATLAB results within an acceptable tolerance. Based on this result we can be confident the controller will adequately control the real system, provided the nonlinear model has been correctly identified.

4.6.3 Custom DAQ Development

In order to interface the TI C28343 to the Quanser 3DOF Helicopter, a custom hardware data acquisition board had to be developed. The specifications of this board were as follows:

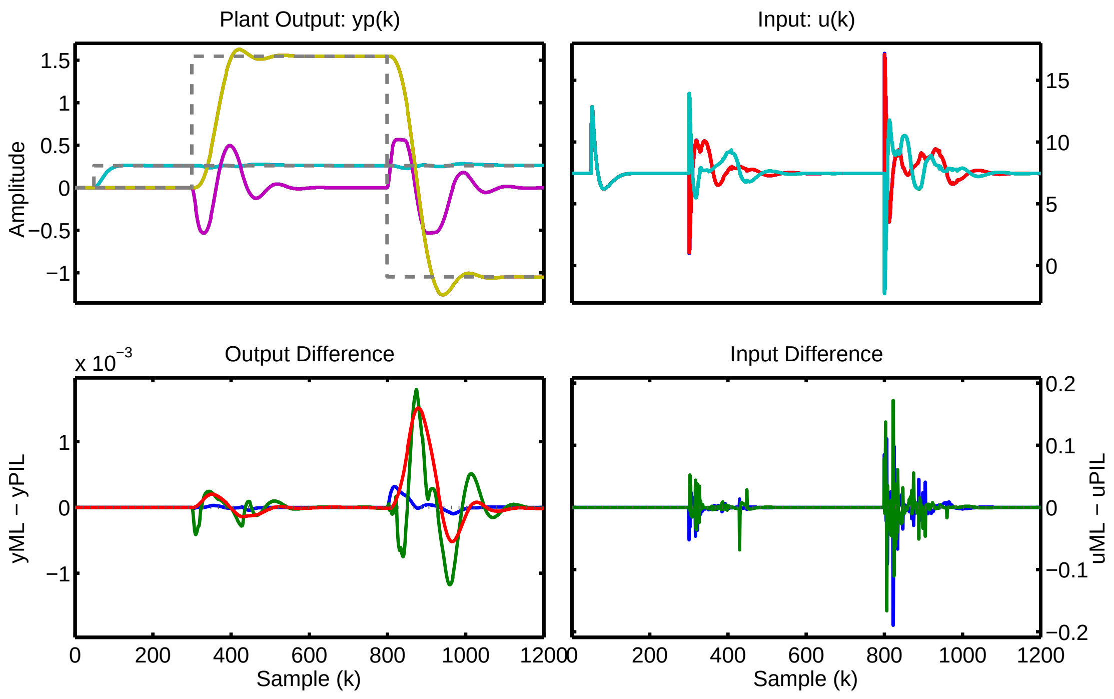


Figure 4.27: Embedded linear MPC control of a nonlinear 3DOF Helicopter model comparison: MATLAB versus TI C28343. Small differences are expected due to the different compilers used for each MPC implementation.

- Be able to read three 512 count quadrature encoders at high-speed simultaneously (for reading each axis encoder)
- Have at least two output Digital to Analog Convertor (DAC) channels at ±10V (for supplying control signals to the linear amplifier, which then drive the motors)
- Have at least two input Analog to Digital Convertor (ADC) channels at 0-10V (future-proofing)
- High-speed serial port of at least 500k baud (for communication with the TI C28343)

While some of this functionality was available via the C28343 Control Card (e.g. input capture, ADC), it was decided to develop a custom hardware solution which contained all required functionality. Being most familiar with the Atmel series of microcontrollers, the decision was made to use a 8/16bit XMEGA 128A3AU which had the required peripheral functionality built-in and ran at a respectable 32MHz. Together with two operational amplifiers and a handful of passive components, the first prototype was designed and built, and is shown in Figure 4.28.

As with most initial prototypes there were some 'teething problems' (the switch mode power supply proved to be too noisy for the quadrature encoders, as well

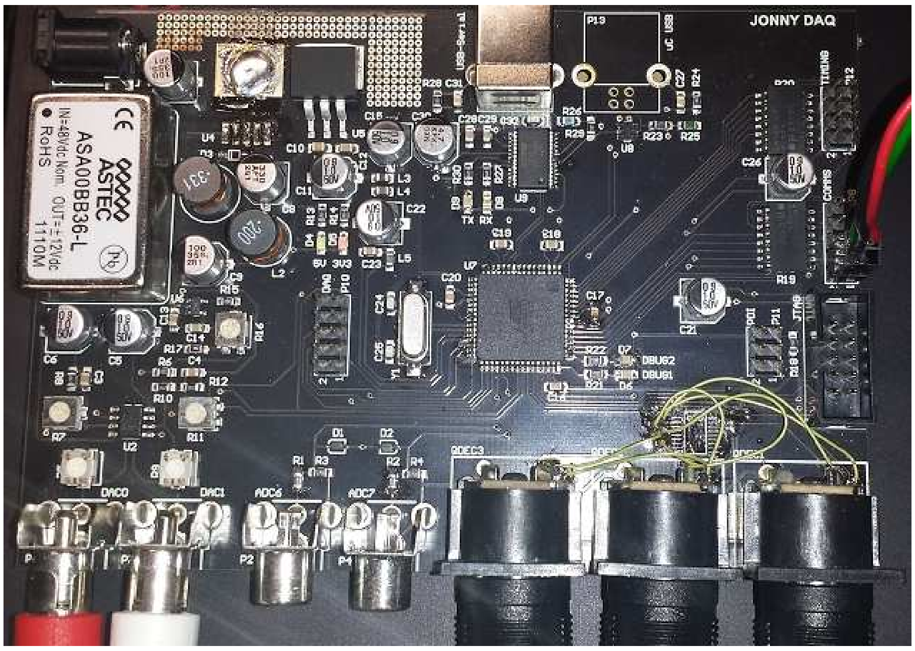


Figure 4.28: Custom DAQ developed to interface to the Quanser 3DOF Helicopter. Each quadrature encoder connects to the bottom right DIN sockets, motor voltages are supplied via the bottom left RCA connectors, and communication with the TI C28343 is via a serial port on the right hand side pin header.

as a bi-directional level shifter would get ‘hung up’ on the quadrature channels), however with some minor fixes (as noted in the figure) the board is functional. A revised board was designed but has never been built because the initial prototype has proved reliable, albeit requiring an external power supply.

A data acquisition program was developed in C using CodeVision and programmed to the XMEGA 128A3AU’s Flash memory. Substantial programmatic calibration of the ADC and DAC channels was required in order for accurate measurements/control outputs and thus we would not recommend this IC for future DAQ development. The hardware based event system handled the quadrature decoding without processor involvement, and this worked remarkably well.

Communication between the XMEGA and TI C28343 (or any other serial capable device) is provided via a hardware serial port operating at 625k baud. The XMEGA operates as a slave and responds to requests for measurements or applies control updates as it receives them. This functionality is implemented via a simple state-machine based decoding system. A small level of error handling is implemented, but it was not required because the communication channel proved to be quite robust.

The complete 3DOF Helicopter hardware setup is shown in Figure 4.29. The TI C28343 is mounted on its breakout board at the bottom right, and communicates to the custom DAQ via the green, red, and black twisted cable. Both boards are mounted on top of the Quanser VoltPAQ-X2 linear voltage amplifier, which connects to the base of the helicopter. The Quanser Q8-USB DAQ can be seen in the background, and is only required to provide the safety override to enable the amplifier output. During normal operation the TI C28343 and custom DAQ (including encoders) required approximately 450mA at 5V to operate, with the remainder of the current supplied by the amplifier.

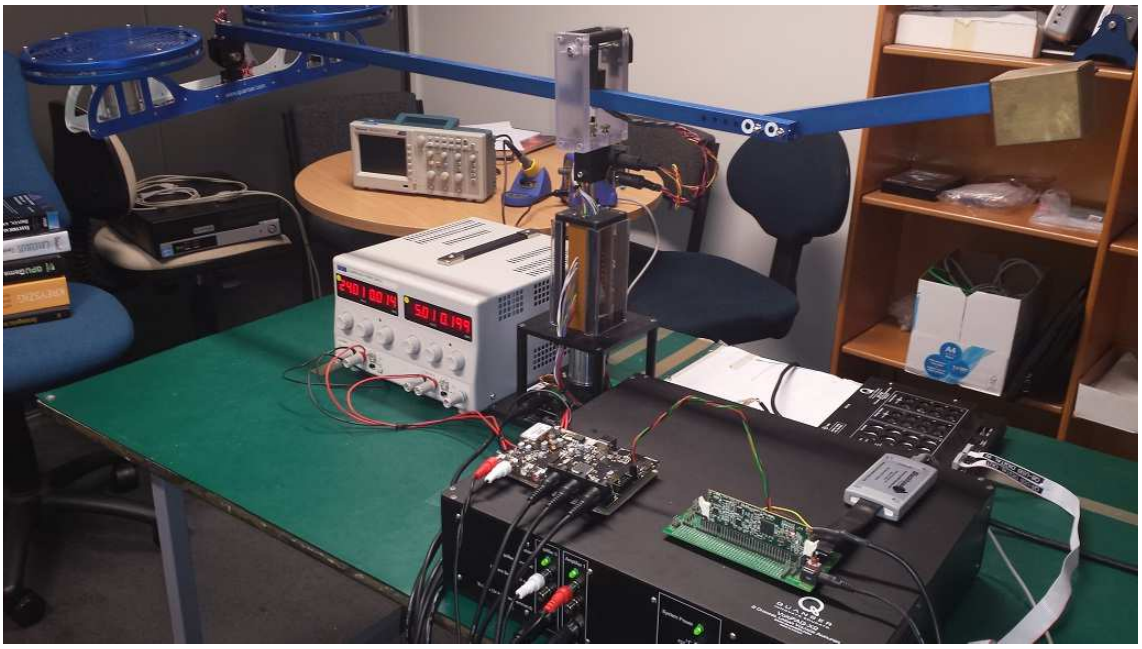


Figure 4.29: 3DOF Helicopter complete hardware setup including the TI C28343, custom DAQ, Quanser amplifier, benchtop powersupply and 3DOF helicopter.

4.6.4 Embedded MPC Implementation

With the acquisition hardware built and calibrated, and a `jMPC` controller designed and validated on the TI C28343, the last step is to implement it and compare the actual performance to that expected. To begin with, an initialization routine was written within the controller to perform the following steps:

1. Initialize the DAQ via the serial port. This included resetting the quadrature encoder timers to 0 within the XMEGA (to set the initial states) and output motor voltages to 0 (off).
2. Initialize derivative filters and discrete integrators, if used (typically disabled).

Once initialization is complete, the controller waits for a start byte from a connected serial device (typically the development computer) to synchronize the collection of run-time data. Once the start byte is received, a hardware interrupt is used to initiate the following steps:

1. The position of the 3DOF Helicopter is requested from the DAQ via the serial port. The angles of each of the axes are returned as integers and then converted to radians.

2. The MPC control law is calculated using the same function as used within the PIL implementations.

3. The calculated control inputs (motor Voltages) are sent to the DAQ, again via the serial port. The conversion from floating point to integers is performed on the TI C28343, including scaling and calibration, because the DAQ microcontroller was not fast enough for floating point calculations.

4. Runtime information and statistics are sent back to the development computer for logging.

5. Wait for the remainder of the sample time, then repeat from step (1) at the next timer interrupt.

The above steps repeat each time the hardware interrupt fires, which is set at the desired sampling rate of the controller. Currently the setpoint is hardcoded into the program, but it would be simple to request this from the development computer.

The reader is reminded that the control and acquisition board plus interface software above is a standard requirement of any control system, and thus could be expected to normally be supplied. Furthermore, the work flow designed within this section is independent of this acquisition hardware, and therefore remains sufficiently general for application to other plants and hardware. The only requirement is there must be some form of serial communication (whether RS232, SPI, I^2C, etc) between the acquisition hardware and the proposed TI control board.

The next complication on the real controller is that we do not have full state feedback, i.e. we only have measurements of $\epsilon, p,$ and λ via the encoders and not their derivatives. With the supplied Quanser system, derivative filters are implemented to obtain estimates of the velocities about each axis; however using the jMPC Toolbox it is simple to add a Kalman filter to estimate the derivative states. Using the Control Systems Toolbox `dlqe` function to design a discrete linear observer, the gain matrix is automatically used by the `jMPC` controller to estimate the missing states. For this case-study the following tuning matrices were used when designing the observer

$$\mathbf{Q} = \begin{bmatrix} 2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}, \quad \mathbf{R} = \begin{bmatrix} 0.1 & 0 & 0 \\ 0 & 0.1 & 0 \\ 0 & 0 & 0.1 \end{bmatrix}$$

where **R** is chosen to contain small elements relative to **Q** as the measurements within the Quanser system are quite accurate (the encoder resolution is high and noise is minimal). Conversely the model uncertainty is assumed to be large, hence the larger elements in **Q**.

Using the tuning constants specified in the nonlinear PIL section, together with the same setpoints, Figure 4.30 shows the result of a real `jMPC` implementation on the Quanser 3 DOF Helicopter. It is obvious the result differs from the PIL implementation, which is to be expected as the nonlinear model used is very simple, and the `jMPC` controller uses only a linear approximation. In addition, the current implementation does not factor in computational delay (due to the controller typically being much faster than the required sample rate) but this is possible to add via the Control Systems Toolbox `delay2z` function.

Figure 4.31 provides a side-by-side comparison of the responses achieved via the real implementation and the PIL implementation. One obvious difference is that the pitch angle violates the upper soft constraint by a much larger degree in the real system. This is attributed to model/plant mismatch whereby the motor dynamics and system parameters most likely need (re)identification.

In order to address the large constraint violations on the pitch axis the constraint penalty is increased from 350 to 1000, and the rotation penalty weight decreased from 8 to 5. Figure 4.32 illustrates the result of this simple re-tuning whereby the pitch angle is now much more tightly constrained, with minimal control performance degradation around the rotation axis setpoint.

As this is only a case-study on the MPC implementation, optimal tuning of this system for a particular application is not the focus, but rather to prove the controller is robust and flexible. Based on this goal, additional operational scenarios have been set up to examine the control performance under various conditions. Figure 4.33 compares the control system response based on varying the pitch constraint from 30° to 50°, while Figure 4.34 introduces a challenging semi-hard (penalty of 50000) lower constraint on the rotation angle. In each case the controller exploits the available

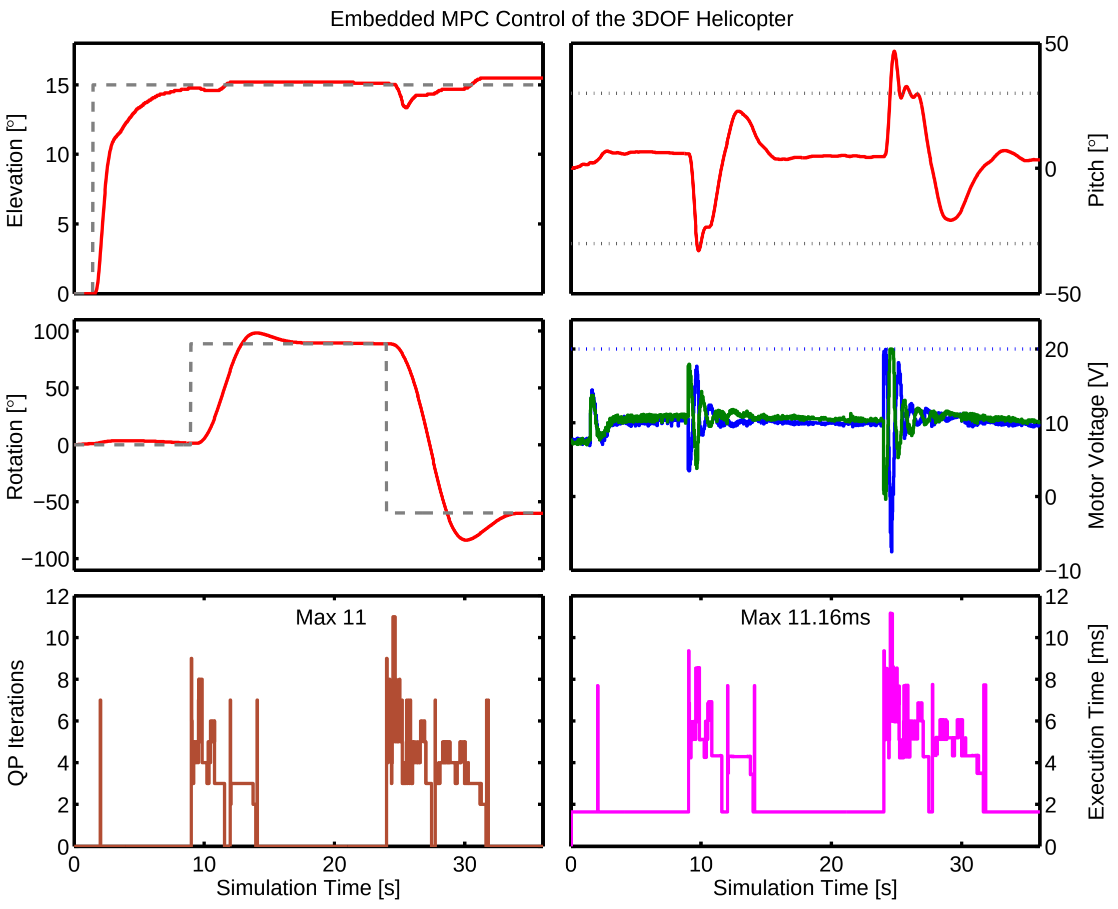


Figure 4.30: Embedded MPC control of the real Quanser 3DOF Helicopter using the TI C28343. Note the pitch constraints are soft and thus drawn lightly.

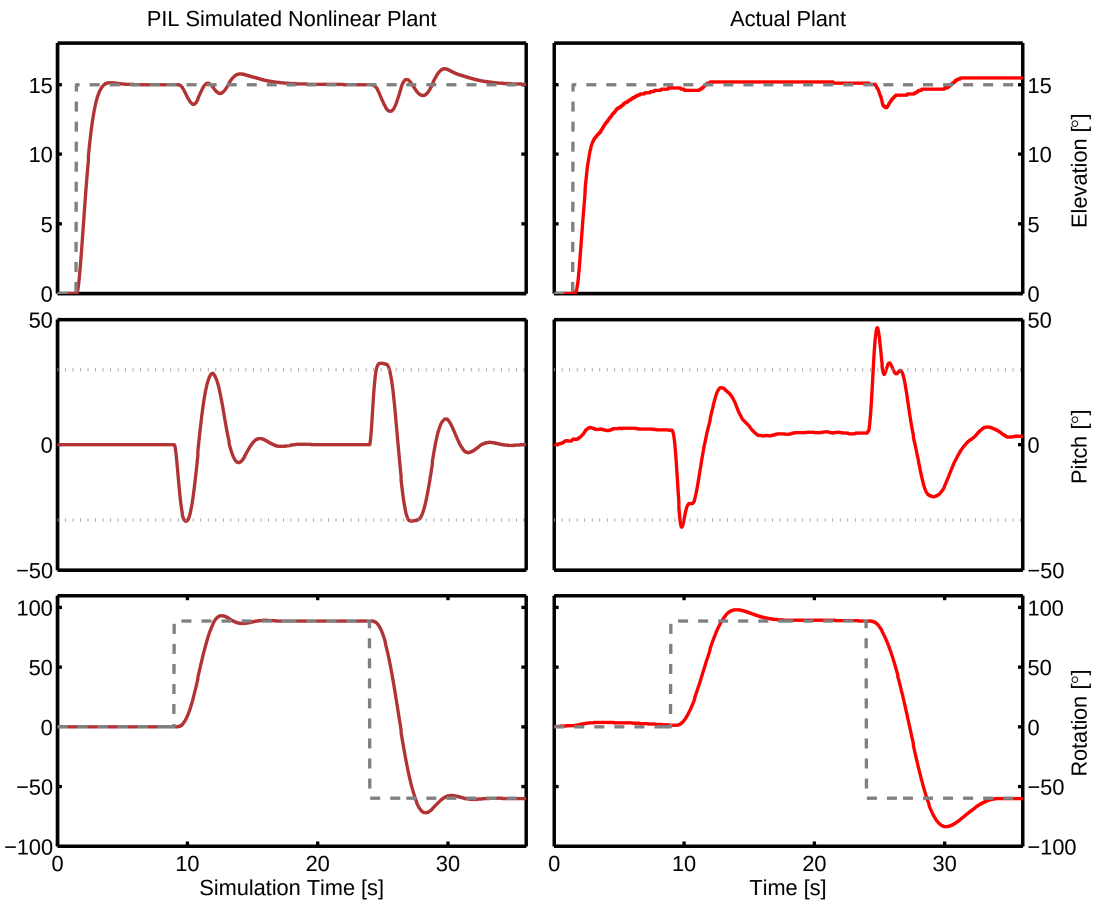


Figure 4.31: A comparison between the PIL with nonlinear simulation results (left hand column) and the real Embedded MPC results (right hand column).

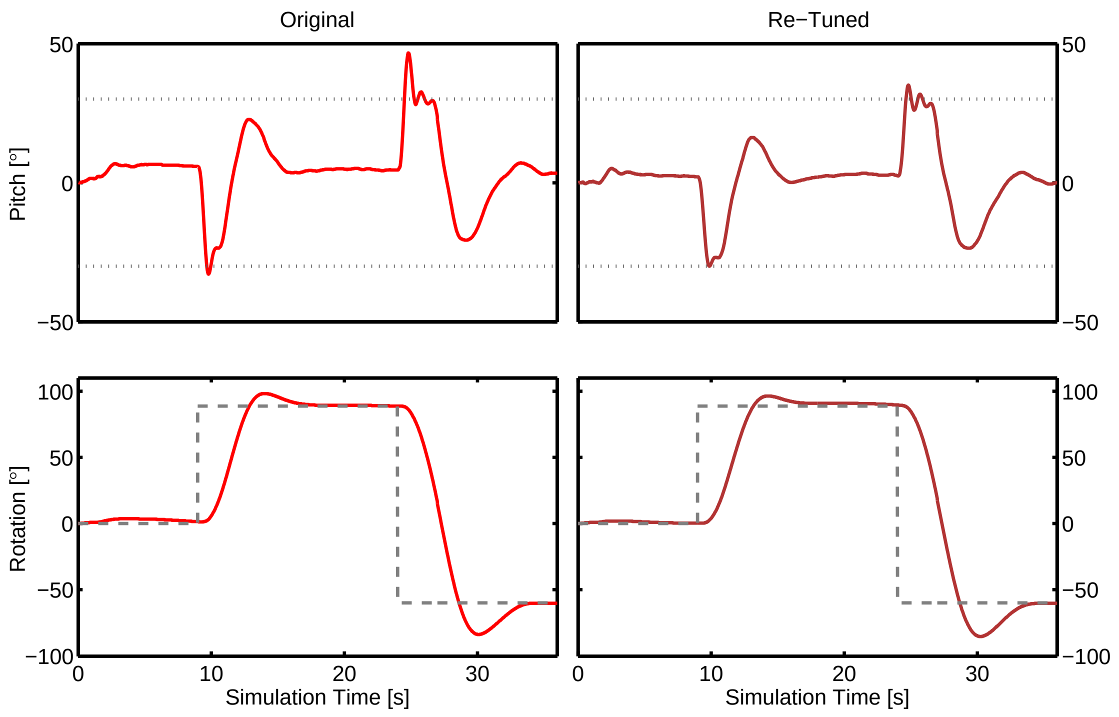


Figure 4.32: A comparison of the original tuning and new tuning to tighten the pitch control on the real 3DOF Helicopter.

control space, such as violating the soft pitch constraint in order to maintain the higher penalty rotation constraint, as demonstrated in right-hand column plots in Figure 4.34.

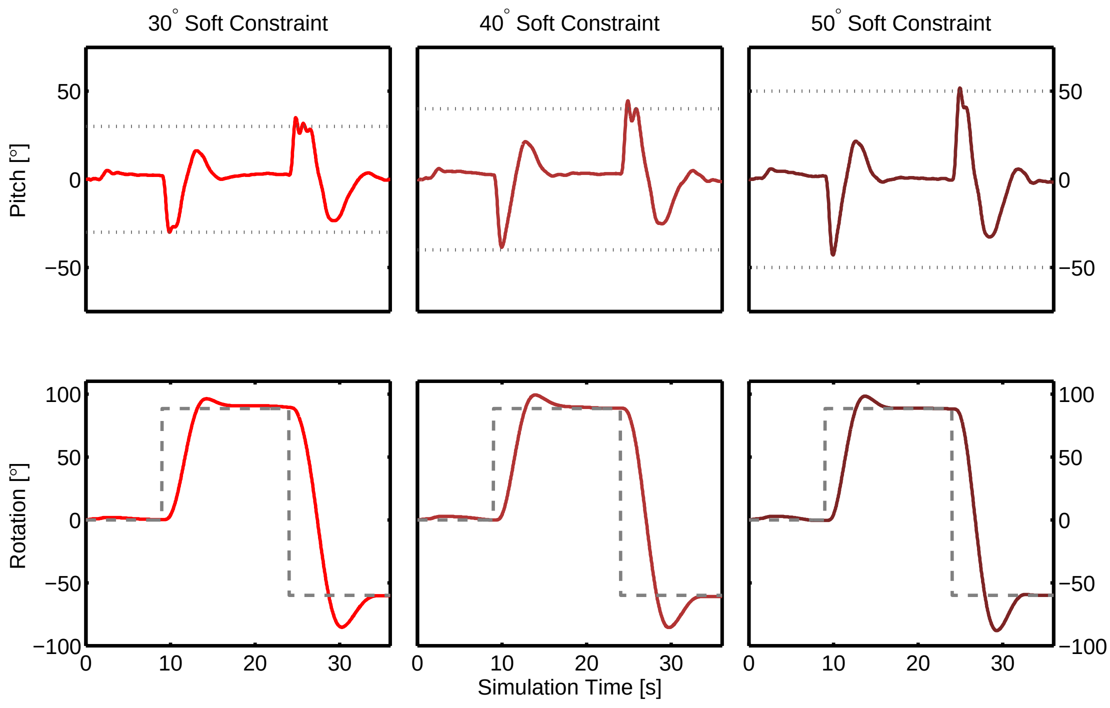


Figure 4.33: A comparison of the system response based on relaxing the pitch constraint from 30° to 50°.

A video of the performance of the embedded `jMPC` controller with the 3DOF Helicopter is available on YouTube at `http://www.youtube.com/watch?v=IReEJy0p3Oc`.

4.6.5 Summary

In this chapter we have introduced the embedded model predictive control problem where the control algorithm must now be solved within limited precision, limited memory and with limited computational power. As identified in Section 1.2.1, new application opportunities are opening up as MPC can be applied to smaller, mobile systems, but these systems inherently need an embedded platform (due to power/weight/size), and thus severely limit the available computing power when contrasted with a standard desktop computer.

This work surveyed the current range of embedded processor technology available in 2009, based on the requirement of a floating point unit, as well as fitting within budget constraints of this research. A Texas Instruments C2000 hybrid MCU/DSP was chosen based on its (comparatively) high clock rate (200MHz), ample memory (256KB) and single precision floating point processor. This is 6% of the clock rate, 0.006% of the memory, and half the floating point precision compared to a typical

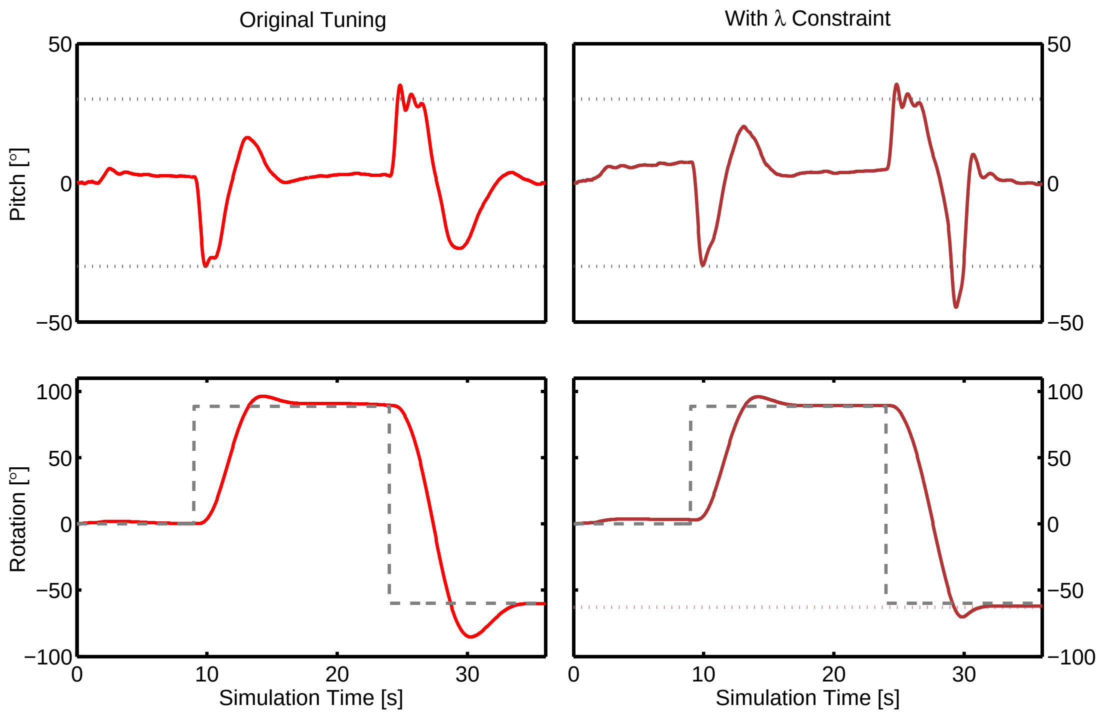


Figure 4.34: A comparison of the system response when introducing a tight lower constraint on the rotation angle. Notice the pitch constraint is temporarily violated in order to minimize the rotation angle constraint violation, given it has a much higher penalty term.

desktop computer. These limits have focused the research to better tailor the MPC algorithm and associated QP solver so as to best exploit these limited resources.

One of the methods used in this chapter to tailor the problem was a new auto-coding tool developed within this work. It provided the ability to automatically generate a full linear MPC controller while retaining benefits of hand-optimized code, using a series of code-templates. Furthermore, by utilizing both compiler directives and functionality of the code generator, the controller could be tailored for the resulting hardware and controller implementation. Moreover, built into the code generator was the ability to automatically validate the generated code within the development environment, as well as within a processor-in-the-loop setup. Combined with the jMPC Toolbox, the two packages allow an embedded MPC controller to be designed, simulated, tuned, deployed and validated all within the one framework, allowing true beginning to end design, and we believe a significant contribution to this field of research.

To validate this claim, two case-studies illustrated the effectiveness of both the framework and its implementation. A processor-in-the-loop implementation of the control of a simulated nonlinear inverted pendulum showed that robustness and speed of the proposed approach, while a real implementation with a nonlinear, multivariate, unstable helicopter proved the approach was also both practically significant and viable.

This chapter concludes our research into the optimization algorithm within an embedded model predictive controller. The following chapter will introduce the next optimization case study, namely the operational optimization of industrial steam utility systems.

Chapter 5

Utility System Model Development

This chapter begins the investigation into the optimization of large industrial utility systems, representing the opposite end of the industrial optimization spectrum from the high-speed controllers presented in the proceeding chapters. On the surface, the difference between these systems is obvious: Embedded MPC looked at small, linear, dynamic models, while now the focus is on large steady-state, nonlinear, non-convex and discrete models. However a common modelling methodology will tie these two contrasting problems together, and the significance of the framework approach will show that common design and construction techniques can be used for optimizing both systems.

The chapter begins with a description of the industrial project which initiated this research, namely iCON Utility Optimizer, conducted in partnership with PETRONAS, and was concerned with the modelling of a steam utility system. From this industrial experience, the decision to pursue our own utility system modelling environment is detailed, which leads into the development of our own library of thermodynamic functions and steady-state, full-load equipment models. The chapter concludes with an overview of the existing literature in utility equipment models, which when combined with the knowledge from our library models, and together with our industrial experience, presents four new detailed models to predict the off-design (part-load) performance of common utility equipment. The significance of these models will be realised in the next chapter, where they be exploited within a new optimization framework.

5.1 Introduction

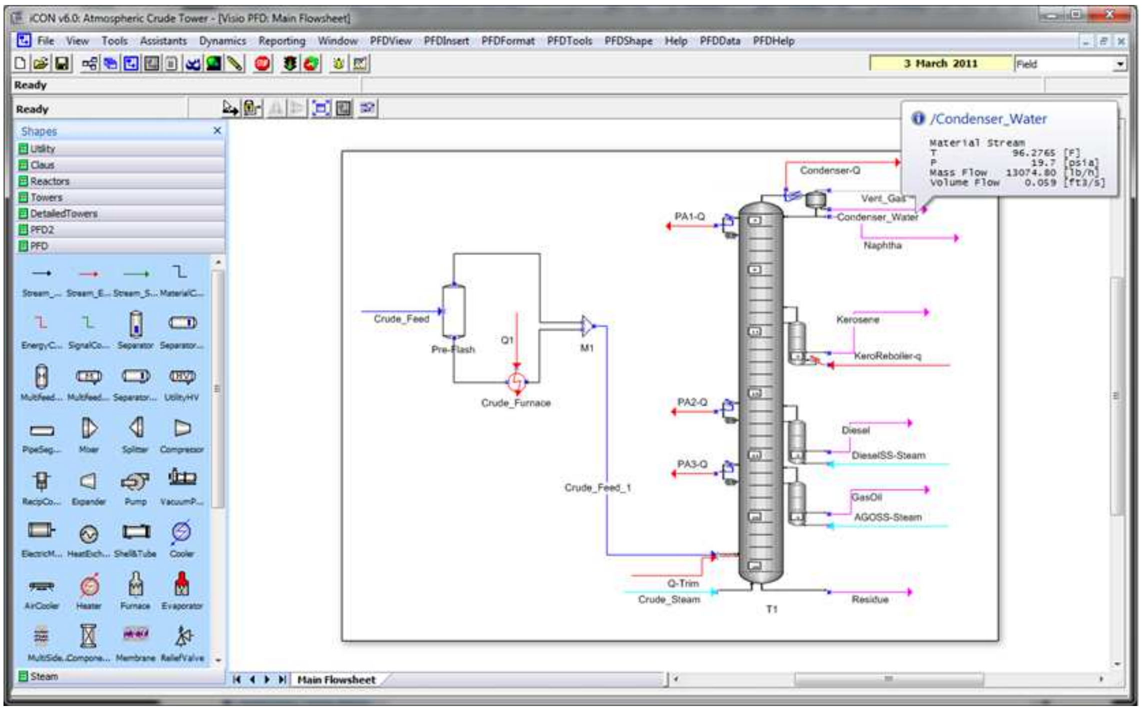
A steam utility system is a large industrial plant which supplies the heating, mechanical and electrical power to an associated process plant, such as chemical plant or petrochemical facility. For the systems of interest for this work, the working fluid is steam, which is generated by gas fired boilers, recovered from process waste heat via waste heat boilers, or generated by heat recovery steam generators. The steam is used by process drivers such as steam turbines (mechanical demands), process users (heating demands) or steam turbo generators (electrical demands), before being collected, condensed and then pumped back to begin the cycle again.

In order to accurately predict the operation (the first step in any optimization study) of these large, complex, nonlinear systems, three major components are required: First, a thermodynamic library which can accurately predict the state properties of the working fluid, Second, a set of ideal (i.e. design) relationships that relate the thermodynamic properties to the mechanical utility equipment, and Third, regressions that provide an estimate of the off-design performance of this utility system equipment. This chapter will address all three requirements, beginning with the background to this research.

5.2 iCON Utility Optimizer (iUO)

At the beginning of 2010 an opportunity arose to shift to Kuala Lumpur to work on a consulting project with PETRONAS, a large multi-national oil and gas company. Together with a post-doctoral fellow from the University of Auckland, Dr. Nick Depree, we were to develop a steam utility modelling framework on top of PETRONAS's existing process simulator, iCON. The key idea was to be able to model both the utility and process systems within the one simulator, allowing for changes on each side to be automatically propagated and accounted.

The platform we were to build on, iCON, is a mature chemical process simulator developed by Virtual Materials Group (VMG), based almost entirely on their flagship product, VMGSim [318]. As shown in Figure 5.1, VMGSim/iCON is a Visio based modelling platform that includes a suite of common chemical and process unit operations, which are dragged and dropped onto the Process Flow Diagram (PFD) and connected like normal Visio elements. The difference is underneath: a Python based simulator engine based on Sim42 [60] is monitoring connections of each unit and forming a detailed simulation model. In addition, VMGThermo [319], a C++ based thermodynamic engine is used for the calculation of thermodynamic properties, which when combined with the unit operation models, a non-sequential flowsheet solver and optional dynamics engine, forms a complete simulation package.



```
Material Stream
T              96.2765 [F]
P                 19.7 [psia]
Mass Flow     13074.80 [lb/h]
Volume Flow      0.059 [ft3/s]
```

Figure 5.1: Screenshot of iCON Process Simulator.

PETRONAS bought the source for VMGSim in 2003 and requested that it be customized to suit their specific needs, which formed iCON. While virtually identical in functionality, iCON contains several PETRONAS only customizations which include new unit operations and thermodynamics tuned for their use.

5.2.1 Project Overview

At the time the project began, PETRONAS was using ProSteam [162], an Excel based modelling package developed by KBC. While adequate for their needs, PETRONAS strategy dictated that they should rely less on external software vendors, and instead develop their own modelling packages where applicable. Therefore our project was to build a complete steady-state utility modelling system within iCON, that would not only match (or exceed) the existing vendor's functionality, but also accurately model PETRONAS's utility systems by validating these against plant data.

An assumption made when the project was proposed was that adding new unit operations to iCON (or VMGSim) would be relatively simple, and thus once a suitable analytical model had been derived, implementation within the simulator would be easy. This did not turn out to be the case, and is described further on in Section 5.2.3. However, aside from this technical issue, Dr. Depree and I developed a suite of common utility system models, as shown in utility modelling Visio palette in Figure 5.2.

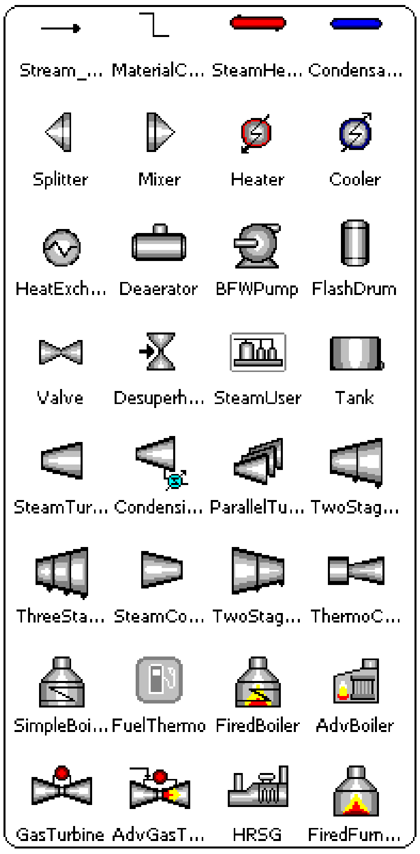


Figure 5.2: iUO function palette.

Each unit operation model was initially derived from a relatively simple mass and energy balance, and then customized based on a known configuration of the real unit. For example, for the steam side of a boiler we assume that the blowdown was at saturated conditions and specified as a continuous fraction of the steam production, thus we could complete the required degrees of freedom. Every unit operation model underwent extensive validation by PETRONAS staff, including comparisons against their own models, ProSteam models, and plant data, where available. The result was by the time we were modelling complete (and very large) utility systems, such as the LNG model in Figure 5.3, we obtained an average accuracy of around 2-3% across the entire system, even on systems producing over 2,500 metric tonnes per hour of steam and containing over 80 individual units.

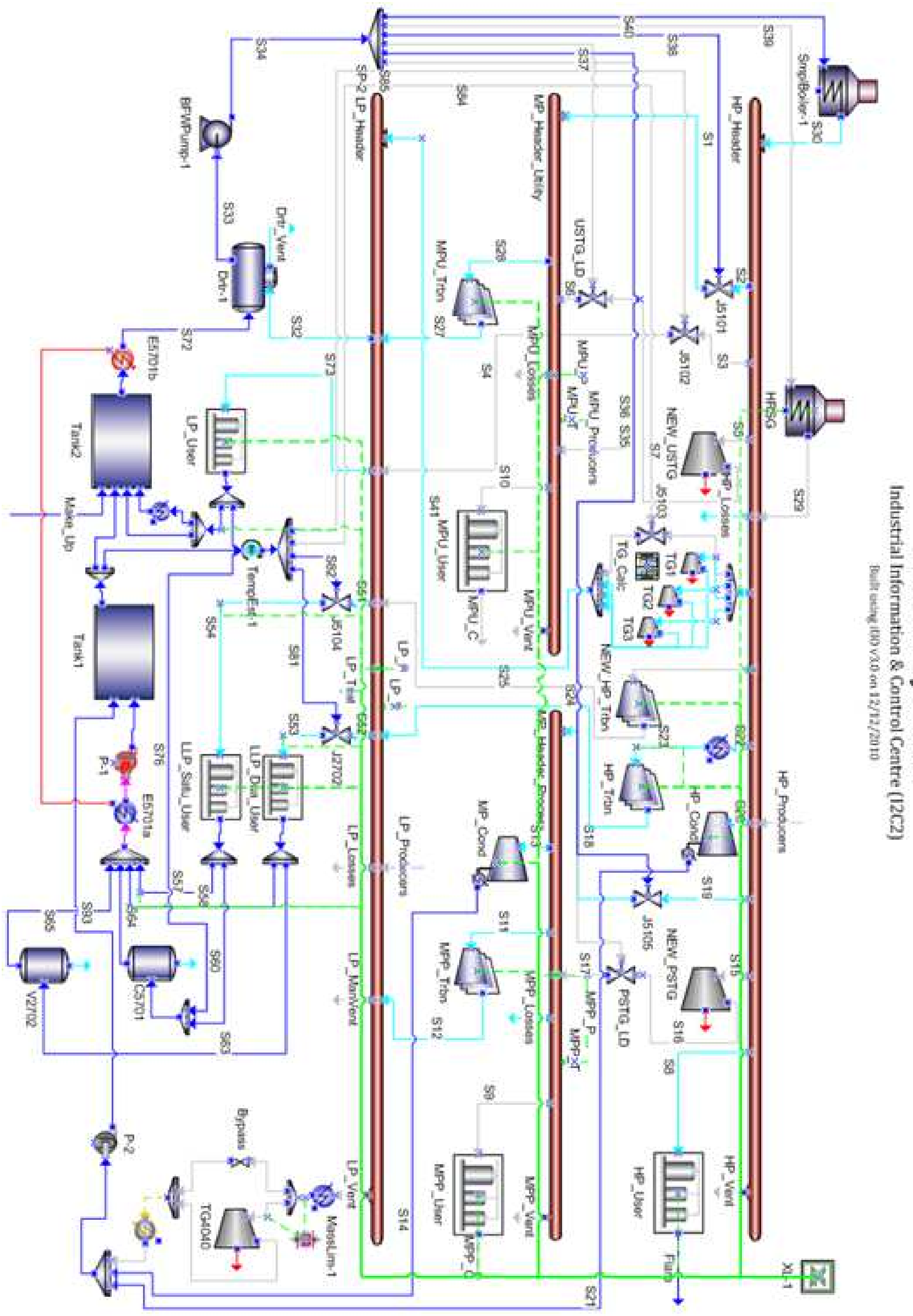


Figure 5.3: Industrial LNG utility system modelled with iUO [71].

5.2.2 Project Summary

The utility modelling package for iCON (officially known as iCON Utility Optimizer (iUO), although no official optimization was performed) was officially completed in July of 2010, having undergone extensive internal review by PETRONAS and validation against existing utility system models. The project met or exceeded virtually all requirements and was generally well received by the process simulation and optimization team.

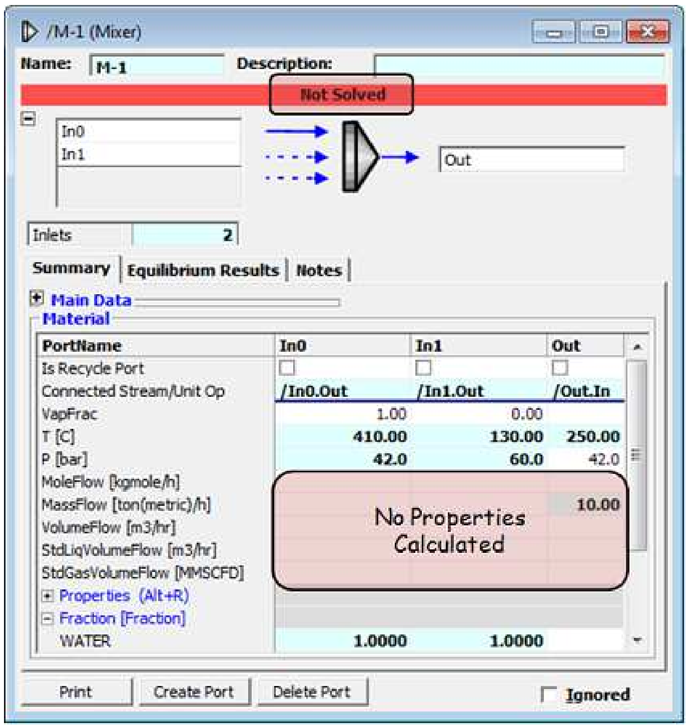
The only major criticism from PETRONAS users of the iUO framework was the simulation speed; it was simply too slow to converge large utility system models. This was primarily caused by the thermodynamic package being used, Steam95, which implemented the scientific formulation from IAPWS [142]. This required a complex formulation of Hemholtz energy for each state property calculation, and as model sizes increased, became quite a bottleneck. A new version of the thermodynamic engine, Steam97, was released later in 2010 and was built into iUO as the default property package. Steam 97 utilized the industrial formulation from IAPWS, which as described further on in Section 5.3.1, is optimized for speed rather than accuracy [140].

5.2.3 Limitations of a Petrochemical Process Simulator for Utility Modelling

The initial thinking when the PETRONAS project was proposed was that utility system modelling within a chemical process simulator should be relatively simple, given there is only a single component (H_2O) and its thermodynamics are well known; as well the unit operations are all relatively simple, especially when compared to the thermodynamics in catalytic crackers and multi-distillation columns. This did not prove to be the case, as described in a joint paper with PETRONAS [71] and detailed below.

Typical problems encountered included issues with properties propagating through models, such as the mixer shown in Figure 5.4. Being a single pure component system, and having both inputs as water, one would assume water would also exit the mixer. However, as shown in the figure, this assumption was not made by the thermodynamic engine inside iCON and had to be automatically added by the framework in order to converge these models.

Another issue was the very common occurrence within utility systems of a zero-flow, i.e. a pipe or unit with no steam in it, which is rare in a chemical process


/M-1 (Mixer)

Name: M-1 Description:

Not Solved

In0
In1

Out

Inlets 2

Summary | Equilibrium Results | Notes

Main Data
Material

PortName	In0	In1	Out
Is Recycle Port	☐	☐	☐
Connected Stream/Unit Op	/In0.Out	/In1.Out	/Out.In
VapFrac	1.00	0.00	
T [C]	410.00	130.00	250.00
P [bar]	42.0	60.0	42.0
MoleFlow [kgmole/h]			
MassFlow [ton(metric)/h]			10.00
VolumeFlow [m3/hr]			
StdLiqVolumeFlow [m3/hr]			
StdGasVolumeFlow [MMSCFD]			
Properties (Alt+R)			
Fraction [Fraction]			
WATER	1.0000	1.0000	

No Properties Calculated

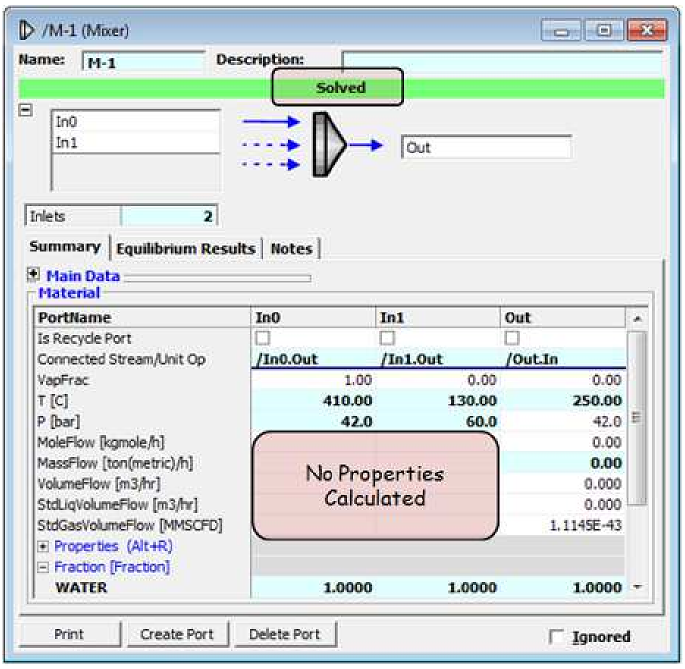
Figure 5.4: Single component convergence issue in iCON. The annotated box indicates missing properties that would normally be calculated.

model. Utility systems commonly switch steam flow between units as process and generation demands change, and thus the framework must be able to cope with no mass (or mole) flow within a connection. This problem alone proved one of the most difficult to overcome, as shown by Figure 5.5 where a mixer would solve correctly even when calculated properties were missing. This was due to the way iCON would calculate an energy balance, such as shown in Equation 5.1 where the denominator is the outlet mass flow.

$$H_{\text{out}} = \frac{H_{\text{In1}} M_{\text{In1}} + H_{\text{In2}} M_{\text{In2}}}{M_{\text{out}}} \tag{5.1}$$

When the outlet approaches zero, the output enthalpy approaches infinity, and this causes a numerical exception and the unit fails to display the results. A rudimentary solution to this problem was to bias the flows so that at least 10^{-10} tonne/hr of steam would always flow, small enough for iCON to think it was zero (and change stream connector colours to show so), but large enough for the calculations to work as required. This solution has not proved problematic for the 3 years the software has been deployed, but it is acknowledged a more suitable solution should be implemented.

Together with a number of other issues as described in [71], it was decided that if we were to continue research within the utility modelling and optimization field, a process simulator was not the correct base platform. These reasons are summarized below:



Name: M-1

Description:

In0
In1

Out

Inlets 2

PortName	In0	In1	Out
Is Recycle Port	☐	☐	☐
Connected Stream/Unit Op	/In0.Out	/In1.Out	/Out.In
VapFrac	1.00	0.00	0.00
T [C]	410.00	130.00	250.00
P [bar]	42.0	60.0	42.0
MoleFlow [kgmole/h]			0.00
MassFlow [ton(metric)/h]			0.00
VolumeFlow [m3/hr]			0.000
StdLiqVolumeFlow [m3/hr]			0.000
StdGasVolumeFlow [MMSCFD]			1.1145E-43
WATER	1.0000	1.0000	1.0000

Figure 5.5: Zero flow convergence issue in iCON.

1. VMGSim had no supported method for building custom unit operations, and this process is actively discouraged by the developers.

2. Rigorous first principle modelling which calculates many more parameters than strictly required, and takes significantly longer to solve.

3. The thermodynamic engine for calculating water and steam properties was too slow for large systems.

4. The flowsheet solver was unable to handle zero-flows without bias flows being added.

5. Optimization had only recently been introduced into VMGSim, and there was little opportunity to customize it further.

Therefore, to continue research into utility system optimization we would need to develop our own utility modelling software, using the knowledge gained during the PETRONAS project. The remaining sections within this chapter detail the development of our own thermodynamic engine, together with a new suite of utility system models.

5.3 Development of the JSteam Utility Modelling Platform

Given the limitations of the process simulator, especially with respect to optimization, it was decided to develop our own utility system modelling platform based on what we had learned from developing iUO. While this would basically require reinventing a number of standard components, ultimately it would provide the flexibility and performance required to create a platform for optimization of these systems. This decision is also based on our early literature review (Section 2.3.5) which indicated commercial tools would not be suitable for this work.

This section will introduce the reader to the thermodynamic engine developed as part of this work, before leading into a description of the steady-state models used. These models are presented to illustrate the underlying equations and relations within this problem field, which will then be exploited both at the end of this chapter, with the presentation of new, detailed utility equipment models, and also in Chapter 6, where these equations form the basis of the steam utility optimization model. Note that the remainder of this Chapter was undertaken independently of the iUO project.

5.3.1 Water and Steam Thermodynamic Engine

The engine of any process modelling package is the thermodynamic package and for utility modelling is the steam and water thermodynamic package. There are three basic methods of obtaining the thermodynamic properties (enthalpy, entropy, quality, etc) of water/steam:

Steam Tables Steams tables are arguably one of the simplest thermodynamic tools, and require a simple grid search and possible (but unlikely) interpolation to find the required property given the known state properties. However, given the modelling package is destined to be a computer package, it is worthwhile to exploit the regressions used to build these tables, rather than enter the tables as they are.

Equation Of State General Equations of State (EOS) (such as described later in the next subsection) are well used for predicting the properties of hydrocarbons, inerts and other components, as well as mixtures. Standard EOS include the Peng-Robinson [245], the earlier Redlich-Kwong [274], and others including modifications for increased accuracies of certain components and mixtures.

However, by design they are deliberately targeted at approximating the properties of a range of components, and thus cause inaccuracies when observing a single component system such as steam. For steam and water this can be exacerbated closer to the saturation line, and because many unit operations will be operating close to this line, it was decided to pursue a dedicated regression for steam.

Dedicated Regression To obtain the most accurate estimate of thermodynamic properties, a dedicated set of regressions can be used. There are two main sets of regressions used in commercial applications: the International Association for the Properties of Water and Steam (IAPWS) 1995 standard targeted at general and scientific use [325, 142], and the IAPWS-IF97 standard targeted at the steam power industry [324, 140]. The industrial formulation is slightly less accurate than the scientific formulation, but it is designed for computational speed.

For this work the IAPWS-IF97 standard has been adopted, including supplementary and revised releases [136, 137, 138, 139, 141, 143]. This choice was based on the focus on optimization of these systems, and thus computational speed features as the predominant selection factor [140]. For a complete list of functionality included in the JSteam package, which includes symbols and functions used throughout the remainder of this Chapter, see Section B.1.2.

5.3.1.1 Implementation

Following the theme of this work, the development environment originally chosen for the thermodynamic engine was MATLAB, given its rapid-prototyping ability. However given that MATLAB is an interpreted language, and the required regressed calculations were quite basic, in order to maximize speed the engine was written in native C++. This avoided the compilation step that MATLAB would be required to perform at each invocation, as well as scalar algorithms being typically faster in compiled code (from our experience) than MATLAB. An example of the calculation of specific enthalpy of steam from pressure and temperature in Region 1 is shown in Equation 5.2

$$\begin{aligned} \frac{h\left(\pi,\tau\right)}{RT} &= \tau\gamma_\tau \\ &= \frac{1386}{T}\sum_{i=1}^{34} n_i \left(7.1 - \frac{p}{16.53}\right)^{I_i} J_i \left(\frac{1386}{T} - 1.222\right)^{J_i - 1} \end{aligned} \tag{5.2}$$

where h is the specific enthalpy in kJ/kg, T is the temperature in K, p is the pressure in MPa and the regressed coefficients are contained in n, I and J. This equation is coded in C++ and detailed in Section B.1.1.

In order to benchmark the performance of the JSteam implementation, the package was compared against three other software implementations of IAPWS-IF97. These are briefly covered below:

freeSteam [260] Released under the GNU General Public License (GPL), freeSteam is an open-source C (based) implementation of Regions 1-4 (specified pressure and temperature ranges) of the IAPWS-IF97 standard. It includes routines for most common property pairings, and utilizes the GNU Scientific Library (GSL) for nonlinear root solving problems. The package appears to be in active development, although an official release has not been announced since January 2010. Note also the underlying code does not conform to modern C/C++ standards and thus recompiling from source has proved impossible using Windows based compilers.

X Steam [129] Released under the BSD license, X Steam is an open-source MATLAB implementation of the complete IAPWS-IF97 standard, including additional supplementary standards. It forms part of the author's 'xeng' suite, which includes steam tables for Excel, Open-Office and a general DLL. The package has not been updated since August 2007, and the author's homepage appears to have been removed. Requests for bug-fixes have also gone unanswered, thus the project appears to have been disbanded.

IAPWS_IF97 [200] Also released under the BSD license, IAPWS_IF97 is another open-source MATLAB implementation of the IAPWS-IF97 standard, however with a particular focus on vectorization to allow for parallel calculations. The implementation is quite limited in terms of property pairings, and misses key property calculations such as entropy and quality. The last package update was in March 2012, with the Github package over 2 years old.

To benchmark the packages the calculation of specific enthalpy (from this point referred to as just enthalpy) over Regions 1-3 (see Figure 5.6) is run by utilizing test points within $T \in (1, 800°C), p \in (1, 1000\text{bar})$. Note that Region 4 (along the saturation line) is not covered within this benchmark because when specifying pressure and temperature, it is not possible to define the quality fraction. In addition, Region 3 is specified as a function of density (ρ) and temperature and thus the volume must first be calculated and then used to calculate enthalpy. Each of the three packages surveyed uses a different method for determining the volume/density within Region 3, thus small calculation differences are expected.

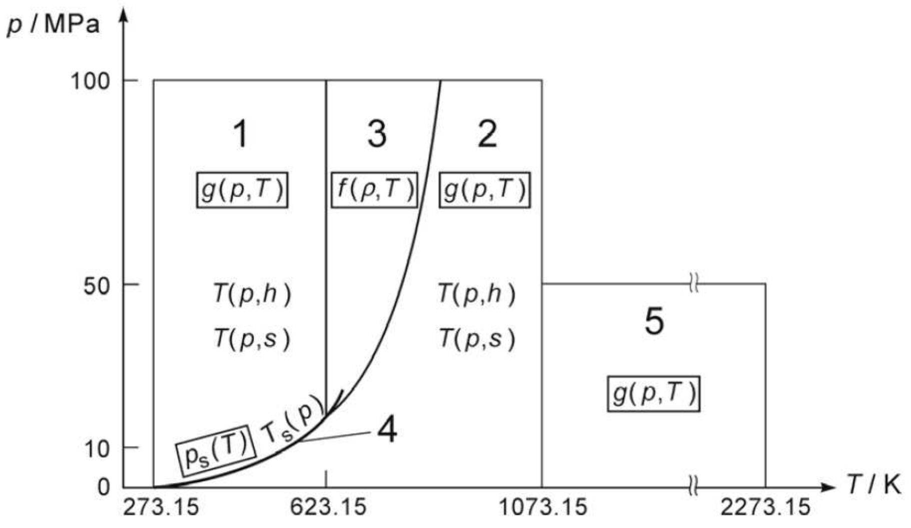


Figure 5.6: IAPWS-IF97 calculation regions (Figure 1 in [140]).

Figure 5.7 shows the sequential performance results of each of the solvers when run on a 2.8GHz i7 Laptop in 32bit. As stated earlier the MATLAB code is not competitive when used for these small scalar problems, and this is justified comparing the two MATLAB implementations versus the two binary implementations. As expected, the Log-Log plot shows that all four implementations scale the same with the number of enthalpy calculations (the gradients are very similar), however the fixed costs (vertical distance) show the performance tuning advantage of JSteam. Comparing JSteam with freeSteam, JSteam is on average 2.1x faster for this example.

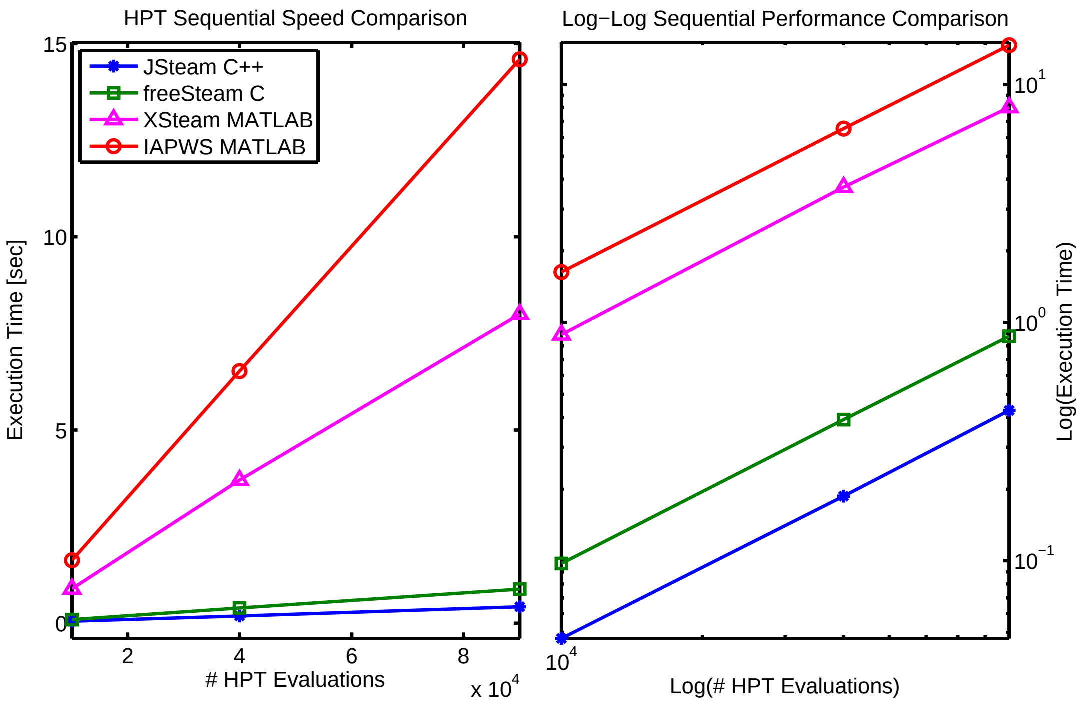


Figure 5.7: Speed comparison of sequential IF97 implementations.

The two parallelized implementations are compared in Figure 5.8 where once again JSteam is much faster, achieving 90,000 enthalpy calculations in less than 25ms. With respect to the sequential results, JSteam averaged a speedup of 16x on a dual-core laptop (with hyperthreading thus 4 logical cores), with the extra speedup (i.e. above 4x) related to the nested calculation for-loops not run within MATLAB (a typical slow point), as well as fixed costs associated with calling the JSteam MEX function.

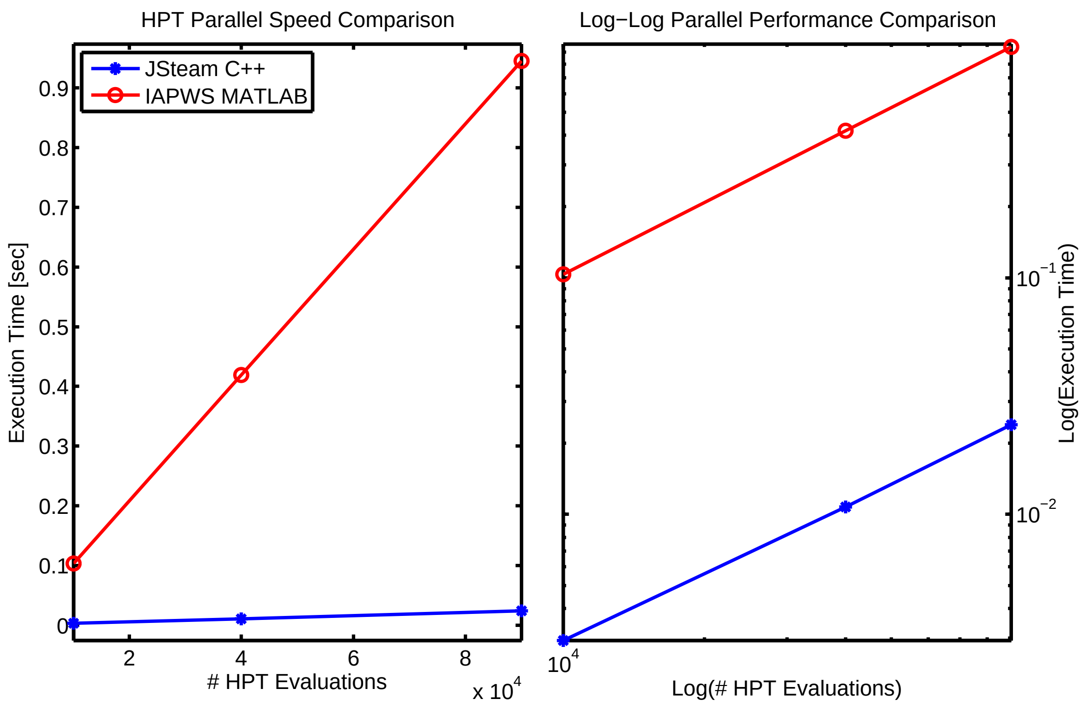


Figure 5.8: Speed comparison of parallelized IF97 implementations.

The parallelization of JSteam was achieved using the Microsoft Parallel Patterns Library (PPL) [218] which executes the outer calculation for-loop in parallel, automatically using the number of cores available. The JSteam library was deliberately written with thread-safety in mind, meaning all routines can be called safely in parallel. Parallelization of the core regression equations (such as Equation 5.2) was initially investigated however these were found to evaluate too fast to overcome the start-up penalty of generating threads for parallelization.

5.3.1.2 Verification

To ensure the implementation of IAPWS regressions are accurate requires a careful choice of verification points across all properties and regression regions. For the example presented in the last subsection, we can easily compare the error between JSteam and any of the other packages at all tested points. Figure 5.9 shows the absolute error between JSteam and the MATLAB implementation IAPWS_IF97 for the enthalpy routine HPT. Observed in the surface plot is the accuracy for all 90,000

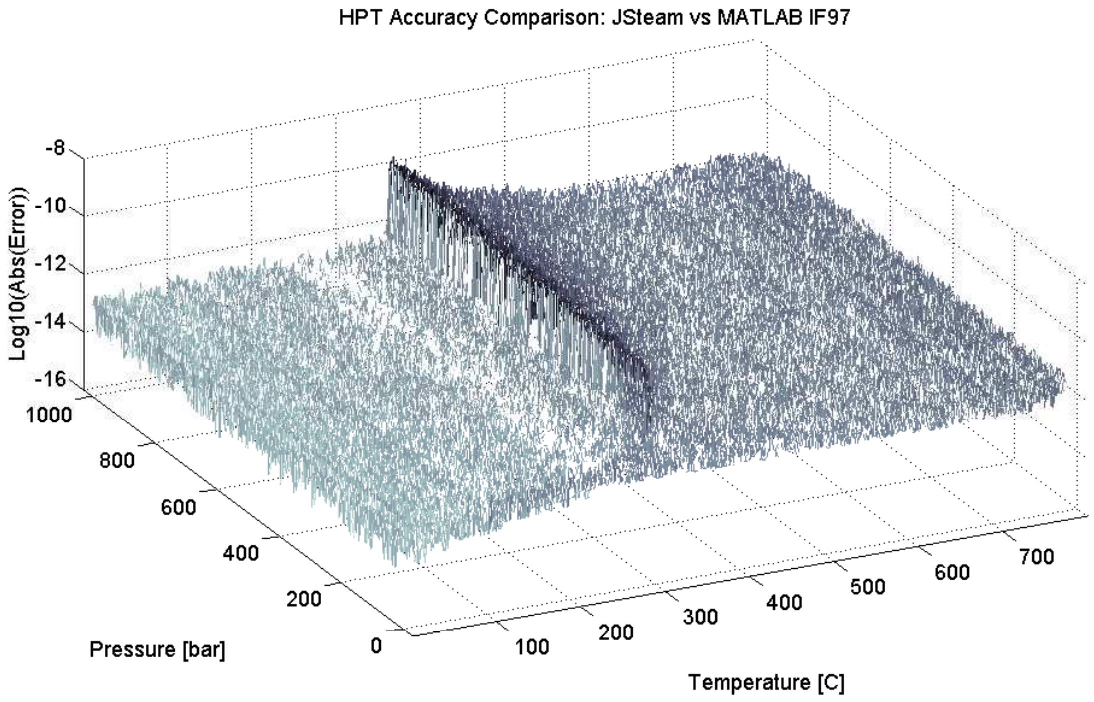


Figure 5.9: Accuracy comparison of HPT: JSteam vs IAPWS-IF97 [200].

test points across the three IF-97 Regions, with most results within the expected numerical precision (double), and the slightly larger errors observed within Region 3. This is due to this region requiring density to be calculated before enthalpy, thus larger numerical differences can be expected. In addition, Region 3 is divided into 26 sub-regions which further increases the differences between the implementations.

Comparing the results to freeSteam, we see a much larger variance within Region 3, however the solution found still has an absolute error averaging less than 10^{-4} within this Region. The larger difference is attributed to freeSteam utilizing a nonlinear root solver to find the Density in Region 3, which is then used to find the enthalpy. In contrast JSteam (and the MATLAB IAPWS_IF97) utilize the supplementary IAPWS release on Volume in Region 3 [139] which avoids the need to iterate. This difference highlights that not all implementations of IAPWS-IF97 are the same, with most not implementing the full set of revised and supplementary releases as available from IAPWS. In addition, many do not include all functionality, such as the MATLAB IAPWS_IF97 missing the fairly important entropy calculations. This reinforces the decision to build our own steam and water thermodynamic package in order to fully utilize the available standards and iteration-free backward calculations.

For verification of the remaining functions, such as entropy, volume, heat capacity, etc, the IAPWS-IF97 specification lists a series of test points and the calculated solutions, such as Table 5 in [140] for Region 1. Each of these test points (around

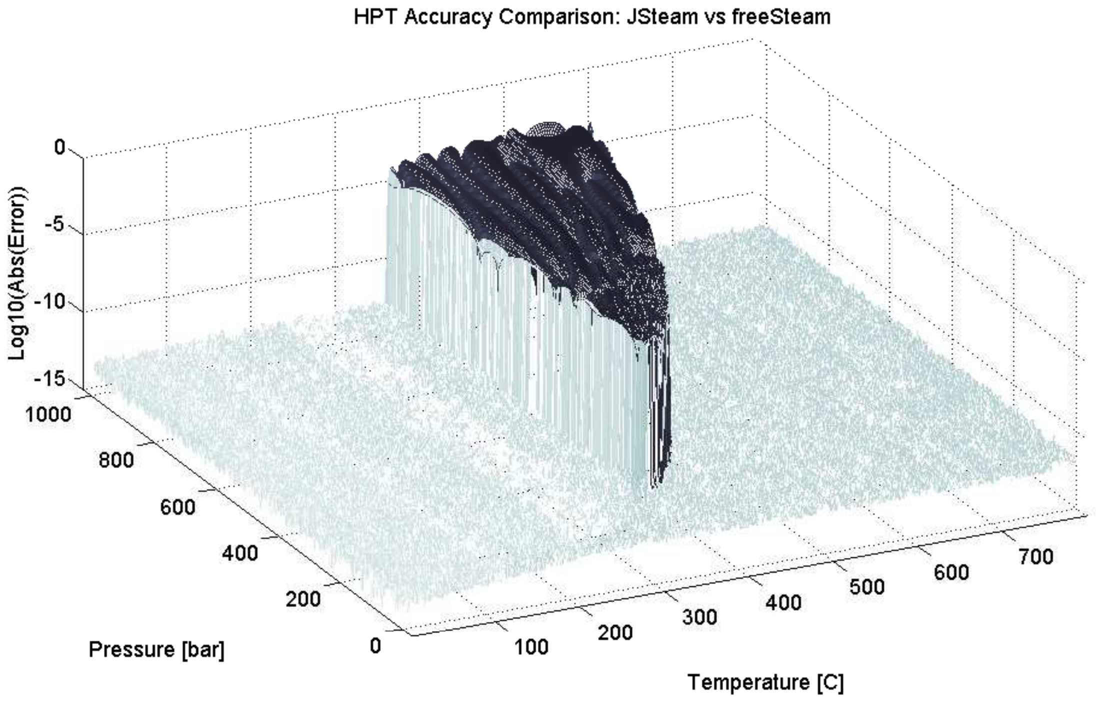


Figure 5.10: Accuracy comparison of HPT: JSteam versus freeSteam.

180 in all) is hard-coded into the JSteam package for automatic self-verification of the package, which includes robustly testing the region identification routines as a function of various property pairings. In addition, a further 320 points are tested against the results from ProSteam over a combination of functions and regions, including terms which must be iterated to solve. Together, these 500 test points are used to automatically verify the accuracy of the JSteam water and steam package using a relative tolerance of 0.05%, thus rebuilds and code changes can be quickly validated.

5.3.2 Steady-Flow Modelling Assumption

Before detailing the unit operations developed in this work, the core modelling assumption must be defined. This work assumes that all equipment within the utility system is operating at steady-state, i.e. the transients due to the start-up period or change in operating condition have completed, and thus can be represented as a *steady-flow* process. This is defined from a thermodynamic point of view as a process where fluid flows through a control volume steadily, and therefore does not change with time. This assumption greatly simplifies the modelling of common unit operations within a utility system, as the standard energy balance equation

$$Q - W = M\left[H_2 - H_1 + \frac{V_2^2 - V_1^2}{2} + g\left(z_2 - z_1\right)\right] \tag{5.3}$$

reduces to

$$Q - W = M\left[H_2 - H_1\right] \tag{5.4}$$

where Q is the rate of heat transfer, W is mechanical power, M is the mass flow rate, H is specific enthalpy, V is velocity, g is the gravitational constant and z is elevation. This approximation is possible because for common utility system unit operations, the velocity change between inlet ($_1$) and outlet ($_2$) is negligible, as is the change in elevation across the unit operation. Furthermore, turbines, compressors, throttling valves, heat exchangers and mixing chambers are all typically regarded as steady-flow devices, which are all common equipment found in a utility system [57]. For these reasons, both the kinetic and potential energy terms will be neglected for the remainder of this work.

5.3.3 Steam Unit Operations

Utilizing the JSteam water and steam thermodynamic engine allowed the implementation of a suite of common water and steam unit operations. The underlying equations are developed both from industrial experience building the iCON Utility Optimizer, together with simple mass and energy balance equations. These are written in C++ and form part of the JSteam modelling package.

Each unit operation is summarized briefly in the following subsections, with detailed models developed later in this chapter. Variables are identified by colour as described below in Table 5.1. Note for many unit operations the choice of inputs

Table 5.1: Unit operation equation colour codes.

Red	Calculated Output
Blue	User Specified Input
Black	Intermediate Variable

and outputs is arbitrary, as long as the degrees of freedom are met to solve the mass and energy balance equations, most inputs can be outputs, and vice-versa. For this work inputs are chosen based on what is commonly measured within a utility system, or is known based on a modelling approach. For modelling the reverse problem, the JSteam package also provides a suite of reverse models, i.e. calculate mass flow through a turbine for a given shaft work requirement, as well calculate the shaft work for a given mass flow. For brevity, only the standard forward models are described in the remainder of this work.

5.3.3.1 Steam Boiler

The Steam Boiler unit operation models the steam side of a typical steam boiler. The input Boiler Feed Water (BFW) is heated to saturated water at the specified output pressure by a single exchanger (the economiser), and a fraction removed as the BlowDown (BD), noting this model assumes a constant blowdown ratio with respect to the amount of steam generated. The saturated water is then heated to superheated steam via another single exchanger (representing both the evaporator and superheater) and this results in the output steam.

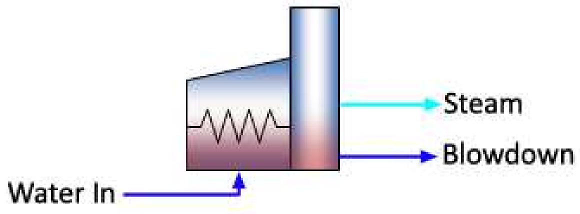


Figure 5.11: Steam boiler unit operation.

It is acknowledged that in a real boiler there will be multiple exchangers with multiple drums. However from a thermodynamic point of view, the model described in Equation 5.5 is effectively equivalent as a first approximation, negating losses. The power required by both exchangers divided by the boiler efficiency (typically taken as a constant, but can be a user specified function, or varied as a function of load such as in Section 5.5.1) determines the duty of the boiler. This will be used later to determine the required mass flow of fuel gas.

The unit is typically specified in terms of input and output pressures and temperatures, and the JSteam `HPT` function is used to calculate the respective enthalpies. In addition, `HPX` is used to calculate the enthalpy of the saturated blowdown, noting this cannot be specified using pressure and temperature alone.

$$\begin{aligned}
Q_{\text{Boiler}} &= \frac{Q_{\text{Economiser}} + Q_{\text{SuperHeater}}}{\eta_{\text{Fir}}} \\
Q_{\text{Economiser}} &= M_{\text{BFW}} \left(H_{\text{BD}} - H_{\text{BFW}} \right) \\
Q_{\text{SuperHeater}} &= M_{\text{Steam}} \left(H_{\text{Steam}} - H_{\text{BD}} \right) \\
M_{\text{BFW}} &= M_{\text{Steam}} + M_{\text{BD}} \\
M_{\text{BD}} &= M_{\text{Steam}} F_{\text{BD}}
\end{aligned} \tag{5.5}$$

where

Table 5.2: Steam boiler parameters.

Q_{Boiler}	kW	Boiler Duty
η_{Fir}	fraction	Firing Efficiency
M_{Steam}	kg/s	Mass Flow of Steam to Generate
M_{BD}	kg/s	Mass Flow of Blow Down
M_{BFW}	kg/s	Mass Flow of Boiler Feed Water
H_{Steam}	kJ/kg	Steam Enthalpy
H_{BD}	kJ/kg	Blow Down Enthalpy (Saturated Water)
H_{BFW}	kJ/kg	Boiler Feed Water Enthalpy
F_{BD}	fraction	Blow Down Fraction

5.3.3.2 Steam Compressor

The steam compressor is very similar to the steam turbine as described in Section 5.3.3.3 in which it models a single inlet, single extraction compressor. The unit is specified in terms of an isentropic efficiency and this is used to calculate the required input power to compress a given mass flow of steam to a specified pressure.

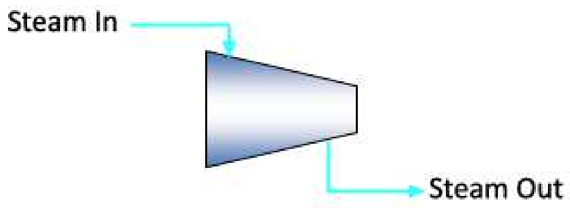


Figure 5.12: Steam compressor unit operation.

For multi-stage compressors the derivation is again very similar to the multi-stage turbine, and is described in more detail in the next subsection.

$$\begin{aligned} M_{\text{out}} &= \frac{W_{\text{Compressor}}}{H_{\text{out}} - H_{\text{in}}} \\ H_{\text{out}} &= \frac{H_{\text{sout}} - H_{\text{in}}}{\eta_s} + H_{\text{in}} \end{aligned} \tag{5.6}$$

where

Table 5.3: Steam compressor parameters.

$W_{\text{Compressor}}$	kW	Required Compressor Power
η_s	fraction	Isentropic Efficiency
M_{out}	kg/s	Mass Flow of Steam
H_{in}	kJ/kg	Input Enthalpy
H_{out}	kJ/kg	Output Enthalpy
H_{sout}	kJ/kg	Isentropic Output Enthalpy

5.3.3.3 Steam Turbine

Representing the most commonly used utility model, the steam turbine (or expander) unit operation models a single-stage, non-condensing, back-pressure turbine with a single inlet and a single extraction. The unit is specified in terms of an isentropic efficiency, which specifies the efficiency of the turbine with respect to pure isentropic operation (i.e. vertical on the T-S diagram). By using this efficiency, together with the inlet and outlet conditions and mass flow, Equation 5.7 describes the amount of power generated by the unit. Alternatively, if the mass flow is specified, the equation can be rearranged to solve for the output power.

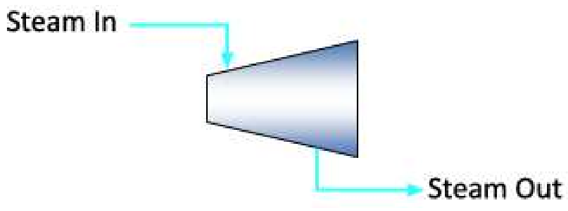


Figure 5.13: Steam turbine unit operation.

$$\begin{aligned} M_{\text{out}} &= \frac{W_{\text{Turbine}}}{H_{\text{in}} - H_{\text{out}}} \\ H_{\text{out}} &= \left(H_{\text{sout}} - H_{\text{in}}\right)\eta_s + H_{\text{in}} \end{aligned} \tag{5.7}$$

where

Table 5.4: Steam turbine parameters.

W_{Turbine}	kW	Generated Turbine Power
η_s	fraction	Isentropic Efficiency
M_{out}	kg/s	Mass Flow of Steam
H_{in}	kJ/kg	Input Enthalpy
H_{out}	kJ/kg	Output Enthalpy
H_{sout}	kJ/kg	Isentropic Output Enthalpy

For multi-stage turbines, such as dual and triple stage versions with a single inlet and multiple extractions, Equation 5.7 is lumped to represent multiple turbines in series, such as shown in Equation 5.8 for a dual stage set

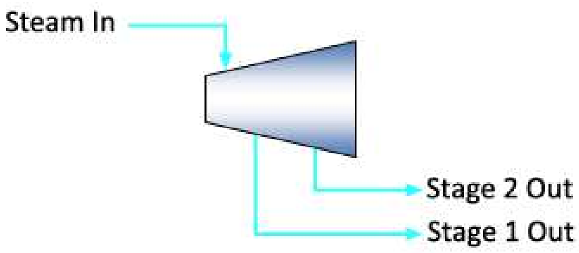


Figure 5.14: Dual stage steam turbine unit operation.

$$\begin{aligned} M_{\mathrm{Out}_2} &= \frac{W_{\mathrm{Turbine}} + M_{\mathrm{Out}_1}\left(H_{\mathrm{Out}_1} - H_{\mathrm{in}}\right)}{H_{\mathrm{in}} - H_{\mathrm{Out}_2}} \\ H_{\mathrm{Out}_2} &= \left(H_{\mathrm{IOut}_2} - H_{\mathrm{Out}_1}\right)\eta_{\mathrm{Isen}_2} + H_{\mathrm{Out}_1} \\ H_{\mathrm{Out}_1} &= \left(H_{\mathrm{IOut}_1} - H_{\mathrm{in}}\right)\eta_{\mathrm{Isen}_1} + H_{\mathrm{in}} \end{aligned} \tag{5.8}$$

where the subscripts 1 and 2 refer to the first stage and second (final) stage respectively.

5.3.3.4 Deaerator

Within a typical utility system, a deaerator is used after the Low Pressure (LP) header and before the Boiler Feed Water (BFW) pump to remove oxygen and other dissolved gases to prevent corrosion and scaling. The deaerator model developed takes a simple thermodynamic approach to this unit, modelling it using a simple mass and energy balance around the two inputs and two outputs. The model assumes the input LP steam and return condensate (collected from e.g. the liquid side of flash drums, condensing turbines, etc) is mixed and dropped to the operating pressure of the deaerator. This allows a small fraction of the input to be flashed off as the 'steam' vent, representing a continuous vent ratio of non-condensables and other unwanted gases (noting thermodynamically these are treated as steam only). The remaining liquid then exits the deaerator to be pumped up to the desired pressure by the BFW pump.

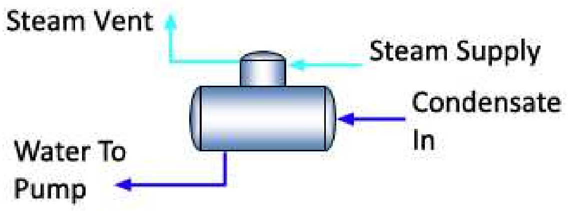


Figure 5.15: Deaerator unit operation.

The model's core job is to calculate the required input LP steam in order to balance the mass and energy across the unit, given the condensate is fixed in terms of mass flow and enthalpy, and the mass flow to the pump is fixed. Equation 5.9 describes the energy balance (top equation) and mass balance (other two equations) of this simplified model.

$$\begin{aligned} M_{\text{Steam}} &= \frac{M_{\text{Vent}} H_{\text{Vent}} + M_{\text{Pump}} H_{\text{Pump}} - H_{\text{Cond}} \left(M_{\text{Pump}} + M_{\text{Vent}} \right)}{H_{\text{Steam}} - H_{\text{Cond}}} \\ M_{\text{Cond}} &= M_{\text{Pump}} + M_{\text{Vent}} - M_{\text{Steam}} \\ M_{\text{Vent}} &= M_{\text{Pump}} F_{\text{Vent}} \end{aligned} \tag{5.9}$$

where

Table 5.5: Deaerator parameters.

M_{Steam}	kg/s	Mass Flow of Input Steam
M_{Cond}	kg/s	Mass Flow of Condensate
M_{Pump}	kg/s	Mass Flow of Output Pump Water
M_{Vent}	kg/s	Mass Flow of Vented Steam
H_{Steam}	kJ/kg	Input Steam Enthalpy
H_{Cond}	kJ/kg	Condensate Return Enthalpy
H_{Pump}	kJ/kg	Output Pump Water Enthalpy (Saturated Water)
H_{Vent}	kJ/kg	Steam Vent Enthalpy (Saturated Steam)
F_{Vent}	fraction	Vent Fraction

5.3.3.5 Desuperheater

While its use is typically avoided in practical situations (due to wasted higher pressure steam), the desuperheater forms a integral component of multi-header utility systems. It provides a simple means to let-down steam between headers, when, for example, parallel connecting equipment is shutdown (such as steam turbines), or a larger mass flow of lower-pressure steam is required at a header below. The model is a simple energy and mass balance, where the amount of cooling boiler feedwater and input steam is calculated in order to satisfy the exit conditions. Alternatively, the input steam mass flow can be calculated, and the equation rearranged to solve for the output mass flow.

$$\begin{aligned} M_{\text{SteamIn}} &= \frac{M_{\text{SteamOut}} H_{\text{SteamOut}} - M_{\text{SteamOut}} H_{\text{Water}}}{H_{\text{SteamIn}} - H_{\text{Water}}} \\ M_{\text{Water}} &= M_{\text{SteamOut}} - M_{\text{SteamIn}} \end{aligned} \tag{5.10}$$

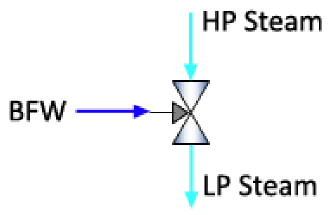


Figure 5.16: Desuperheater unit operation.

where

Table 5.6: Desuperheater parameters.

M_{SteamIn}	kg/s	Mass Flow of Input Steam
M_{SteamOut}	kg/s	Mass Flow of Output Steam
M_{Water}	kg/s	Mass Flow of Cooling Water
H_{SteamIn}	kJ/kg	Input Steam Enthalpy
H_{SteamOut}	kJ/kg	Output Steam Enthalpy
H_{Water}	kJ/kg	Cooling Water Enthalpy

5.3.3.6 Flash Drum

The flash drum is a standard flash operation, but given that it only contains a single component, it is a simple ratio calculation. The model uses the JSteam `XPH` function to calculate the inlet quality at the specified drum pressure, then splits the vapour and liquid fractions into the two outputs.

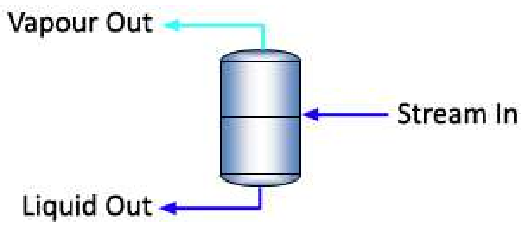


Figure 5.17: Flash drum unit operation.

$$\begin{aligned} M_{\text{Vapour}} &= X_{\text{Drum}} M_{\text{In}} \\ M_{\text{Liquid}} &= \left(1 - X_{\text{Drum}}\right) M_{\text{In}} \end{aligned} \tag{5.11}$$

where

Table 5.7: Flash drum parameters.

M_{Vapour}	kg/s	Mass Flow of Output Vapour
M_{Liquid}	kg/s	Mass Flow of Output Liquid
M_{In}	kg/s	Mass Flow of Input Stream
X_{Drum}	fraction	Quality of Input Stream at Specified Drum Pressure

5.3.3.7 Simplified Heat Exchanger

The heat exchanger unit operation has not been extensively modelled within the JSteam package simply because it has not been extensively used within the iCON modelling studies. Instead, the heat exchanger is typically modelled as a cooler and heater with the duties required to be equal, thus representing a basic thermodynamic energy balance of the system. This is opposed to more industry standard models which use physical properties of the exchangers such as heat transfer coefficients and surface area, or other specifications such as the Log Mean Temperature Difference (LMTD) and heuristics based on the configuration (i.e. concurrent or counter-current flow). Implementation of these standard industry heat exchanger equations has been undertaken by a final year project student and is described briefly later in Section 5.3.7.

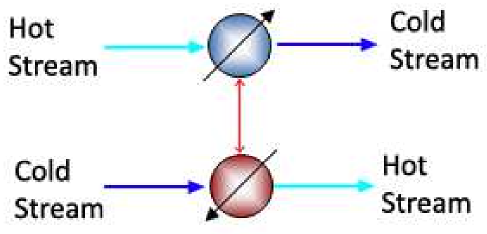


Figure 5.18: Heat exchanger via cooler & heater unit operations.

As can be seen in Figure 5.18 the streams are treated independently and are only connected by the duty of each heater/cooler. Equation 5.12 is the basic set of equations for this system and can be rearranged as required to solve for the required variables.

$$\begin{aligned} Q_{\text{Exchanger}} &= M_{\text{Out}_H}\left(H_{\text{Out}_H} - H_{\text{In}_H}\right) \\ &= M_{\text{Out}_C}\left(H_{\text{In}_C} - H_{\text{Out}_C}\right) \end{aligned} \tag{5.12}$$

where

Table 5.8: Heat exchanger parameters.

$Q_{\text{Exchanger}}$	kW	Heat Exchanger Duty
M_{Out_H}	kg/s	Mass Flow (Heater)
M_{Out_C}	kg/s	Mass Flow (Cooler)
H_{In_H}	kJ/kg	Heater Inlet Enthalpy
H_{Out_H}	kJ/kg	Heater Outlet Enthalpy
H_{In_C}	kJ/kg	Cooler Inlet Enthalpy
H_{Out_C}	kJ/kg	Cooler Outlet Enthalpy

5.3.3.8 Water Pump

The Boiler Feed Water (BFW) pump is modelled as a standard pump with specified isentropic efficiency. Together with the outlet pressure and required mass flow, the model calculates the required power to pump the input water (typically from the deaerator) to the specified boiler pressure level (or slightly above depending on the boiler pressure drop), as well as calculating the slight increase in output temperature due to non-adiabatic operation.

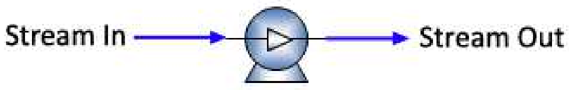


Figure 5.19: Pump unit operation.

$$\begin{aligned} H_{\text{out}} &= H_{\text{in}} - \frac{H_{\text{in}}}{\eta_s} + \frac{H_{\text{sout}}}{\eta_s} \\ W_{\text{Pump}} &= M_{\text{out}} \left(H_{\text{out}} - H_{\text{in}} \right) \end{aligned} \tag{5.13}$$

where

Table 5.9: Water pump parameters.

W_{Pump}	kW	Required Pump Power
η_s	fraction	Isentropic Efficiency
M_{out}	kg/s	Mass Flow of Water Through Pump
H_{in}	kJ/kg	Input Enthalpy
H_{out}	kJ/kg	Output Enthalpy
H_{sout}	kJ/kg	Isentropic Output Enthalpy

5.3.3.9 Valve

The simplest unit operation included in the JSteam suite is the throttling valve, which models a typical isenthalpic (constant enthalpy) valve. The JSteam `TPH` function is used to calculate the output temperature, given the pressure drop across the valve and input conditions.

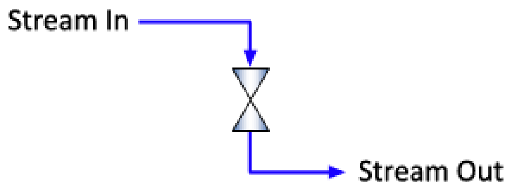


Figure 5.20: Valve unit operation.

5.3.4 Fuel Gas Thermodynamic Engine

To be able to calculate the thermodynamic properties of common components found with both the fuel gas and combustion products, a new thermodynamic engine was developed. The thermodynamics of these components is required to be able to calculate common thermodynamic properties such as enthalpy and entropy for energy balances, as well as to estimate the heating duty available by combusting a fuel and air mixture. The complete list of components available within the JSteam engine are listed in Table 5.10.

Table 5.10: JSteam supported components list.

CH_4	Methane	CO	Carbon Monoxide
C_2H_4	Ethylene	CO_2	Carbon Dioxide
C_2H_6	Ethane	N_2	Nitrogen
C_3H_6	Propylene	O_2	Oxygen
C_3H_8	Propane	H_2O	Water/Steam
iC_4H_8	IsoButene	H_2	Hydrogen
nC_4H_{10}	n-Butane	H_2S	Hydrogen Sulfide
iC_4H_{10}	IsoButane	SO_2	Sulfur Dioxide
nC_5H_{12}	n-Pentane		
C_5H_{12}	IsoPentane		
nC_6H_{14}	n-Hexane		

An important assumption was made during the development of the fuel gas thermodynamic engine and this was that all components were assumed to be gas. This meant that all calculations would be much simpler because a flash calculation (a vapour-liquid equilibrium solver) would not be required, and thus no iteration was necessary for basic properties such as enthalpy and entropy. This assumption is valid because the utility systems of interest are typically run on natural gas, predominantly composed of methane and other short-chain hydrocarbons which are in their vapour form at atmospheric pressure.

Two sets of equations are implemented for calculating state properties of the fuel gas and combustion product streams, based on pressure of the stream. Each of these are described in the following two subsections.

5.3.4.1 NASA Glenn Thermodynamic Database 9-Term Polynomials

For calculations of state properties at (or close to) atmospheric pressure (defined as 1 atm or 1.01325 bar), the NASA Glenn Thermodynamic Database supplies a set of simple-to-implement 9-term polynomials split over two temperature ranges: 200-1000K and 1000-6000K, together with coefficients for a huge number of common components. The database is available via an online application, ThermoBuild [229], which allows a user to select the species they are interested in, including compounds, and then presents a set of coefficients in a computer readable form.

The 3 main thermodynamic properties used in this work can be calculated from the equations as described in [48]

$$\frac{C_p^\circ}{R} = a_1T^{-2} + a_2T^{-1} + a_3 + a_4T + a_5T^2 + a_6T^3 + a_7T^4 \tag{5.14}$$

$$\frac{H^\circ}{RT} = -a_1T^{-2} + \frac{a_2 \ln T}{T} + a_3 + \frac{a_4T}{2} + \frac{a_5T^2}{3} + \frac{a_6T^3}{4} + \frac{a_7T^4}{5} + \frac{a_8}{T} \tag{5.15}$$

$$\frac{S^\circ}{R} = -\frac{a_1T^{-2}}{2} - a_2T^{-1} + a_3 \ln T + a_4T + \frac{a_5T^2}{2} + \frac{a_6T^3}{3} + \frac{a_7T^4}{4} + a_9 \tag{5.16}$$

where the respective coefficients are substituted in from the Glenn Database. Note the enthalpy returned from these correlations is the "engineering enthalpy" which is defined as

$$H^\circ = \Delta_f H^\circ_{298.15} + \int_{298.15}^{T} C_p^\circ \, dT \tag{5.17}$$

which includes the enthalpy of formation at a reference point of 298.15K. In addition, T in each of the equations above is specified in K, C_p in kJ/(kmol K), S in kJ/(kmol K), H in kJ/kmol and R is the gas constant 8.314472.

To verify the implementation, the ThermoBuild application was used to generate 15 test points across each component and each calculated state property, resulting in 855 validation points. Using the JSteam implementation the same test points were used, and the accuracy compared across all 855 points. The JSteam packaged achieved a maximum absolute error of less than 0.005 kJ/kmol (or kJ/kmol K) across all components and properties, indicating a successful implementation. Moreover, the JSteam implementation is able to calculate 90,000 enthalpy calculations of Methane in less than 9ms, noting this is around 70x faster than the Equation of State engine described in the next subsection. For this reason, the NASA 9-term polynomials are used for all thermodynamic calculations involving fuel and combustion products, unless a compressed gas is specified.

5.3.4.2 Peng-Robinson Equation of State

For calculations of compressed fuel gas, such as within a gas turbine, the thermodynamic engine requires functions which are specified in terms of both pressure and temperature. While a simple implementation would use the ideal gas law, simply

$$PV = nRT \tag{5.18}$$

it is well known this law does not hold well at higher pressures (or lower temperatures, however we are not concerned with this region within this work) nor with strongly polar gases. It was therefore decided to implement an Equation of State (EOS), the Peng-Robinson [245] cubic equation of state

$$Z^3 + \alpha Z^2 + \beta Z + \gamma = 0 \tag{5.19}$$

which is the standard compressibility form (Z is the compressibility factor) where the constants α, β and γ depend on the pressure, temperature and physical characteristics of the component.

In order to solve the required state properties such as enthalpy and entropy using this EOS, the cubic expression must be solved based on the known properties, in this work typically pressure and temperature. In JSteam the cubic expression is solved analytically and, if more than one real root exists, the maximum value root is used (this represents the vapour phase compressibility). From the compressibility, a departure function (such as in [179]) is used to determine the difference of the property from the ideal gas value.

For this work the Peng-Robinson values for each component (such as critical temperature (T_c) and pressure (P_c), acentric factor (ω) and ideal gas correlations) have been obtained from a variety of sources including Perry's Chemical Engineering Handbook (Table 2-164, Chapter 2 in [248]), Carl Yaw's Thermophysical Properties of Chemicals and Hydrocarbons [55] and Transport Properties of Chemicals and Hydrocarbons [56], as well as reference data from VMGThermo [319], part of the VMGSim process simulation software [318].

A freely available Windows and MATLAB-compatible implementation of the Peng-Robinson EOS that also included all required calculations was not found to date thus a speed comparison was not made. However for the reader's reference, the JSteam implementation is able to sequentially calculate 90,000 enthalpy calculations of Methane from across 1-30bar, 1-800°C in less than 650ms, which is approximately half the speed of the JSteam steam thermodynamic engine. For a parallelized implementation on the same PC as used in the steam results, the same 90,000 calculations takes around 80ms. Note, because a flash calculation is not required and the result is always assumed to be vapour, this calculation is relatively simple and thus can be executed very quickly.

With respect to accuracy, the EOS is verified against results obtained via VMGSim using a comparative Peng-Robinson thermodynamic package. A total of 950 points is used to test both enthalpy and entropy calculations across each individual component, as well as across varying pressures and temperatures. The accuracy achieved obtains a maximum absolute error of less than 0.05 kJ/kmol (or kJ/kmol K for entropy) for all calculations.

5.3.4.3 Composition Calculations

As is standard in industry, the fuel gas supplied to boilers, gas turbines, furnaces, and other utility equipment is a mixture of hydrocarbon and inert gases, with typically methane, carbon dioxide and nitrogen being the main components. In order to calculate the physical and thermodynamic properties of a mixture of gases (from this point on referred to as a composition) a C++ class (`Comp`) was created to handle the required intermediate calculations. The `Comp` class allows the user to create a composition based on any of the components listed in Table 5.10, and specified either as molar or mass fractions. With the object created, Table B.2 in Section B.1.3 lists the thermodynamic methods available for use with this object. In addition, methods exist for mixing composition objects, shortcuts for setting as air, as well as methods for returning common composition fractions such as Oxygen and Carbon Dioxide.

To calculate the thermodynamic properties of mixtures the current implementation ignores the binary interaction parameters (K_{ij} all assumed to be zero), and returns the mole weighted sum of the thermodynamic property calculated across each component. The only exception to this rule is the calculation of transport properties, for which a mass weighted sum is used (this was found to better match reference data from VMGThermo). Further development of the mixture calculations has been carried out, including a full multi-component flash calculation using the analytical partial derivatives of the Peng-Robinson EOS. This has however yet to be fine tuned and implemented within the JSteam package.

5.3.4.4 Combustion Calculations

In order to complete the fuel gas calculations, a set of combustion calculations has been implemented. These do not attempt to model the complex reaction kinetics present in combustion of natural gas, but rather to present a simple stoichiometric approach which assumes complete combustion at atmospheric pressure using standard stoichiometric coefficients for each component. When working with the engineering team at PETRONAS to develop iCON, this was also a similar approach taken when modelling combustion, and we have therefore taken this to be industry acceptable.

Table B.3 in Section B.1.3 lists the methods available within the JSteam `Comp` object for combusting and calculating combustion properties of a specified fuel gas and air stream. Note the final three solve methods are used to solve for a required flow of air or fuel, in order to match common combustion limits such as minimum inlet excess O_2, or minimum stack O_2. An example of the methods developed and how they are applied within the next section is detailed in Section B.1.4.

5.3.5 Fuel Gas Unit Operations

The fuel requirements are where the real economic considerations come in when optimizing the utility system, and thus modelling the consumption of fuel for various unit operations is a major part of this chapter. The four units below are based on similar models created for PETRONAS in the iCON project, however with multiple functionality differences. In addition, given that these models are derived from an industry project in which we were modelling against real equipment, they have all been validated within a specified tolerance against real PETRONAS equipment.

Note all fuel gas unit operations are specified using a molar basis, as opposed to a mass basis for the steam unit operations. This convention was adopted based on the underlying thermodynamic engine units. In addition, equations for the fuel gas unit operations will include functions from Table B.3. While this mixes mathematics with code, we believe this allows us to describe the model operation in a clearer manner than using pseudo-code, or detailing the tedious internal calculations. As standard in this thesis, variables written in `teletype` are code objects or function calls.

5.3.5.1 Furnace

The Furnace is used to model the combustion of a fuel gas mixed with air and to calculate the resulting combustion products, together with the output temperature (adiabatic) or output duty (non-adiabatic). The model assumes that both fuel and air enter at atmospheric pressure as well as that there is always sufficient oxygen to ensure complete stoichiometric combustion.

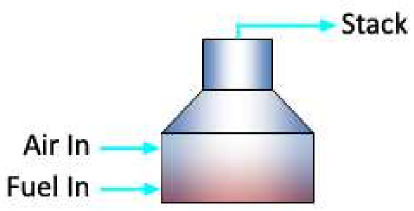


Figure 5.21: Furnace unit operation.

The furnace can be specified as solving for a required fuel flow to generate a specified heat output, or vice-versa. For the example given below, as per Equation 5.20, we are solving for the required fuel flow. Note that in the current implementation the mole flow of fuel must be iteratively solved in order to converge the furnace's output duty to that specified. In practice the system is approximately linear with respect to duty and fuel flow, and thus can be solved using two test points to establish the linear relationship, given the current specifications

$$\begin{aligned}
\texttt{Stack} &= \texttt{FuelAir->Combust}(H_{\text{In}}, T_{\text{Stack}}, M_{\text{In}}, Q_{\text{Furnace}}) \\
\texttt{FuelAir} &= \texttt{Fuel->MixMole}(\texttt{Air}, M_{\text{Fuel}}, M_{\text{Air}}) \\
M_{\text{Air}} &= \texttt{Fuel->SolveAirMoleF}(\texttt{Air}, F_{\text{O}_2}, M_{\text{Fuel}}) \\
\eta_{\text{Furnace}} &= \frac{Q_{\text{Furnace}}}{Q_{\text{Fuel}} M_{\text{Fuel}}} \qquad (5.20) \\
H_{\text{In}} &= \frac{M_{\text{Fuel}} H_{\text{Fuel}} + M_{\text{Air}} H_{\text{Air}}}{M_{\text{In}}} \\
M_{\text{In}} &= M_{\text{Fuel}} + M_{\text{Air}}
\end{aligned}$$

and where the parameters are listed in Table 5.11.

Section B.1.5.1 provides an example of the use of the Furnace model.

5.3.5.2 Fired Boiler

The Fired Boiler extends the Steam Boiler unit operation from Section 5.3.3.1 by adding a Furnace model (from above) to model the combustion side. The duty

Table 5.11: Furnace parameters and units.

Q_{Furnace}	kW	Furnace Duty
Q_{Fuel}	kJ/kmol	Fuel NHV
η_{Furnace}	fraction	Furnace Efficiency
M_{Fuel}	kmol/s	Mole Flow of Input Fuel
M_{Air}	kmol/s	Mole Flow of Input Air
M_{In}	kmol/s	Input Mole Flow
H_{Fuel}	kJ/kmol	Input Fuel Enthalpy
H_{Air}	kJ/kmol	Input Air Enthalpy
T_{Stack}	C	Stack Temperature
H_{In}	kJ/kmol	Input Enthalpy
F_{O_2}	fraction	Inlet Excess O_2 or Stack O_2
`Fuel`		Fuel Gas Composition Object
`Air`		Air Composition Object
`Stack`		Combustion Products Composition Object
`FuelAir`		Fuel + Air Combustion Mixture

required by the two exchangers to heat the boiler feed water to superheated steam is used as a duty target for the furnace, which then solves for the required fuel and air flow in order to balance the energy across the unit. For the purpose of simplicity, the steam side is assumed 100% efficient (i.e. the firing efficiency used previously is 1.0, this is typical as most boilers are > 85% efficient), with the unit efficiency calculated as per the furnace efficiency.

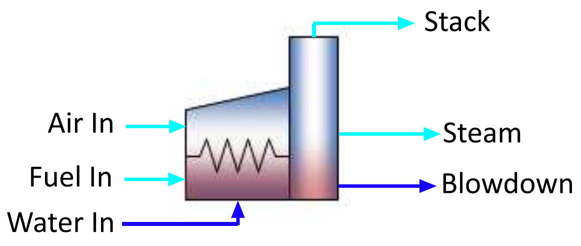


Figure 5.22: Fired boiler unit operation.

The equations for the fired boiler are the Steam Boiler and Furnace equations, run sequentially in this order. Section B.1.5.2 provides an example of the use of the Fired Boiler model.

5.3.5.3 Gas Turbine

The gas turbine model is a mass and energy balance of a typical open cycle gas turbine with optional steam injection. While the model is quite rudimentary, it performs well in practice once the manufacturer's efficiency relationships have been entered from published or operational data. The model incorporates a standard compressor model to compress the incoming air, then mixes it with fuel and the option of steam. The mixture is then combusted and expanded through a standard turbine which generates the turbines output power.

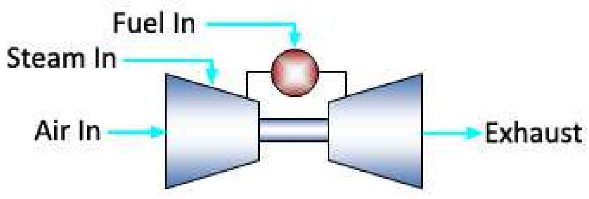


Figure 5.23: Gas turbine unit operation.

As with the furnace, in order to solve the required air flow the system must be iterated. However the system is approximately linear with respect to output power versus air flow, and as per the furnace, a linear approximation can be made across two test points to solve for the required air flow. Equation 5.21 illustrates the calculation required for a GTG with no steam injection, noting the first equation is the main energy balance across the unit. In addition, a fixed isentropic efficiency across the compressor is assumed between 75% and 85%, a value which agrees with validation done in the iCON project and in standard gas turbine analysis [230].

$$\begin{aligned}
W_{\text{GTG}} &= M_{\text{Exhaust}}\left(H_{\text{ExpIn}} - H_{\text{Exhaust}}\right) - M_{\text{Air}}\left(H_{\text{CmpOut}} - H_{\text{Air}}\right) \\
[\texttt{Exhaust}, H_{\text{ExpIn}}] &= \texttt{FuelAir->AdCombust}(H_{\text{CmpIn}}, M_{\text{Fuel}} + M_{\text{Air}}) \\
\texttt{FuelAir} &= \texttt{Fuel->MixMole}(\texttt{Air}, M_{\text{Fuel}}, M_{\text{Air}}) \\
H_{\text{CmpIn}} &= \frac{M_{\text{Fuel}} H_{\text{Fuel}} + M_{\text{Air}} H_{\text{Air}}}{M_{\text{Fuel}} + M_{\text{Air}}} \\
M_{\text{Fuel}} &= \frac{W_{\text{GTG}}}{Q_{\text{Fuel}} \eta_{\text{GTG}}} \\
H_{\text{CmpOut}} &= \frac{H_{\text{ICmpOut}} - H_{\text{Air}}}{\eta_{\text{comp}}} + H_{\text{Air}}
\end{aligned} \tag{5.21}$$

The parameters are listed in Table 5.12.

Section B.1.5.3 provides an example of the use of the Gas Turbine model.

5.3.5.4 Heat Recovery Steam Generator (HRSG)

The HRSG unit models the heat recovery and steam generation of a base-load or supplementary fired HRSG generating a single temperature and pressure level steam output. Two heat exchangers are assumed to be within the boiler, an economiser

Table 5.12: Gas turbine parameters.

W_{GTG}	kW	Gas Turbine Output Power
Q_{Fuel}	kJ/kmol	Fuel NHV
η_{GTG}	fraction	GTG Efficiency
η_{comp}	fraction	Compressor Isentropic Efficiency
M_{Fuel}	kmol/s	Mole Flow of Input Fuel
M_{Air}	kmol/s	Mole Flow of Input Air
M_{Exhaust}	kmol/s	Mole Flow of Exhaust Gas
H_{Fuel}	kJ/kmol	Input Fuel Enthalpy
H_{Air}	kJ/kmol	Input Air Enthalpy
H_{Exhaust}	kJ/kmol	Exhaust Enthalpy
H_{CmpIn}	kJ/kmol	Compressor Input Enthalpy
H_{CmpOut}	kJ/kmol	Compressor Output Enthalpy
H_{ICmpOut}	kJ/kmol	Isentropic Compressor Output Enthalpy
H_{ExpIn}	kJ/kmol	Expander Input Enthalpy
`Fuel`		Fuel Gas Composition Object
`Air`		Air Composition Object
`Exhaust`		Exhaust Gas Composition Object
`FuelAir`		Fuel + Air Combustion Mixture

and a lumped evaporator and superheater. In addition, the model has the facility to calculate steam production as a function of 3 operating conditions:

Base Load No secondary firing is used and only the heat available in the incoming exhaust gas is used to generate steam. The steam production is limited by the available heat in this exhaust gas, as well as the minimum stack temperature and approach temperatures of the HRSG exchangers.

Secondary Fired Secondary (supplementary) firing is used to meet the specified steam demand. Steam production is limited by the available O_2 and stack temperature.

Maximum Firing Typically used for exploratory studies only, the unit is fired to produce the maximum amount of steam, limited by the available O_2 in the incoming exhaust gas and stack temperature.

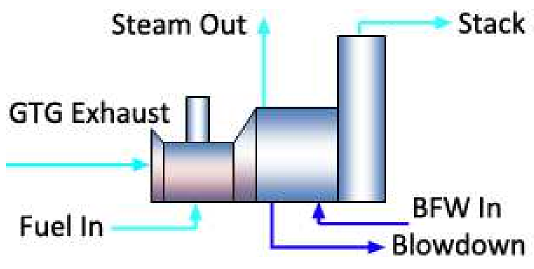


Figure 5.24: HRSG unit operation.

For the base load operating condition, Equation 5.22 describes the operation of the model

$$\begin{aligned}
Q_{\text{Stack}} &= Q_{\text{IntStack}} - M_{\text{Steam}}\left(1 + F_{\text{BD}}\right)\left(H_{\text{BD}} - H_{\text{BFW}}\right) \\
Q_{\text{IntStack}} &= Q_{\text{Exhaust}} - M_{\text{Steam}}\left(H_{\text{Steam}} - H_{\text{BD}}\right) \\
M_{\text{Steam}} &= \min\left(M_{\text{SteamST}}, M_{\text{SteamEC}}\right) \\
M_{\text{SteamST}} &= \frac{Q_{\text{MinStackT}} - Q_{\text{Exhaust}}}{H_{\text{BFW}} - H_{\text{Steam}} + F_{\text{BD}}\left(H_{\text{BFW}} - H_{\text{BD}}\right)} \\
M_{\text{SteamEC}} &= \frac{Q_{\text{MinEconDT}} - Q_{\text{Exhaust}}}{H_{\text{BD}} - H_{\text{Steam}}}
\end{aligned} \tag{5.22}$$

where the parameters are listed in Table 5.13.

Table 5.13: Base load HRSG parameters.

Q_{Exhaust}	kW	Gas Turbine Exhaust Duty (Input to HRSG)
Q_{Stack}	kW	HRSG Stack Duty
Q_{IntStack}	kW	Stack Duty Between Economizer and Superheater
$Q_{\text{MinStackT}}$	kW	Minimum Stack Duty at Minimum Stack Temperature
$Q_{\text{MinEconDT}}$	kW	Minimum Stack Duty Between Economizer and Superheater at Minimum Economiser ΔT
M_{Steam}	kmol/s	Mole Flow of Steam Generated
M_{SteamST}	kmol/s	Max Steam Flow Based on Minimum Stack Temperature
M_{SteamEC}	kmol/s	Max Steam Flow Based on Minimum Economiser ΔT
H_{Steam}	kJ/kmol	Steam Enthalpy
H_{BD}	kJ/kmol	Blow Down Enthalpy
H_{BFW}	kJ/kmol	Boiler Feed Water Enthalpy
F_{BD}	fraction	Blow Down Fraction

The base-load model automatically solves for the two maximum steam flows possible, based on the two limiting operational constraints: the economiser approach temperature (ΔT, typically 20-30°C) and the minimum stack temperature (typically 150-250°C). The approach temperature constraint is used to prevent temperature crossover within the heat exchangers, while the minimum stack temperature is used to prevent condensation forming within the stack which causes corrosion. From these two maximum steam flows, the minimum is chosen as this will always satisfy both constraints (provided the incoming exhaust is hot enough). Section B.1.5.4 provides an example of the use of the base load HRSG model.

The maximum steam production is typically limited by the minimum O_2 fraction in the stack and thus this can be used to determine both the maximum steam flow, as well as the required secondary fuel flow, as shown in Equation 5.23.

$$\begin{aligned}
M_{\text{MaxSteam}} &= \frac{Q_{\text{MinStackT}} - Q_{\text{HRSGIn}}}{H_{\text{BFW}} - H_{\text{Steam}} + F_{\text{BD}}\left(H_{\text{BFW}} - H_{\text{BD}}\right)} \\
Q_{\text{HRSGIn}} &= H_{\text{In}} M_{\text{In}} \\
[\texttt{Stack}, H_{\text{In}}, M_{\text{In}}] &= \texttt{FuelAir->AdCombust}(H_{\text{In}}, M_{\text{Exhaust}} + M_{\text{SecFuel}}) \\
\texttt{FuelAir} &= \texttt{SecFuel->MixMole}(\texttt{GTExhaust}, M_{\text{SecFuel}}, M_{\text{Exhaust}}) \\
M_{\text{SecFuel}} &= \texttt{SecFuel->SolveFuelMoleF}(\texttt{GTExhaust}, F_{\text{O}_2}, M_{\text{Exhaust}}) \\
H_{\text{In}} &= \frac{M_{\text{SecFuel}} H_{\text{SecFuel}} + M_{\text{Exhaust}} H_{\text{Exhaust}}}{M_{\text{Exhaust}} + M_{\text{SecFuel}}}
\end{aligned} \tag{5.23}$$

The parameters for the secondary fired HRSG are listed in Table 5.14 (as well as the earlier Table 5.13).

Table 5.14: Secondary fired HRSG parameters.

Q_{HRSGIn}	kW	Duty of Hot Gas (Combusted GT Exhaust + Fuel) Entering HRSG
M_{Exhaust}	kmol/s	Mole Flow of GT Exhaust
M_{In}	kmol/s	Mole Flow of Hot Gas Entering HRSG
M_{MaxSteam}	kmol/s	Maximum Mole Flow of Steam Generated
M_{SecFuel}	kmol/s	Mole Flow of Secondary Fuel
H_{Exhaust}	kJ/kmol	Enthalpy of GT Exhaust
H_{In}	kJ/kmol	Enthalpy of Hot Gas Entering HRSG
F_{O_2}	fraction	Stack O_2
`SecFuel`		Secondary Fuel Gas Composition Object
`GTExhaust`		GTG Exhaust Composition Object
`Stack`		Stack Composition Object
`FuelAir`		Fuel + Air Combustion Mixture

The final operating mode is to solve for the required secondary fuel flow to generate a specified amount of steam. As with the furnace and gas turbine, a linear approximation can be found which relates the steam mass flow to the secondary fuel mass flow. A set of similar equations to those in Equation 5.23 is then used to solve for the required fuel flow. This method however does not constrain the model to obey the approach temperatures across the exchangers, and if it results in a broken approach constraint, the model is solved using bisection (it is now nonlinear) to find a valid operating condition.

In addition to normal operation, there are a number of invalid operating points that the model will automatically try to identify and report on. These include situations when the incoming exhaust is too cold to generate the required temperature steam, when the unit is constrained so that the required amount of steam cannot be generated (such as limited by approach temperatures or minimum stack O_2), or when there are convergence errors due to internal simplifications. In general, when correctly specified with a real operating point, the unit solves a realistic estimate of the actual equipment operating values.

5.3.6 Excel Modelling Environment

To facilitate the building and tuning of utility system models an Excel front-end was developed over the JSteam Engine in early 2010. The package, “JSteam Excel Add-In” [62] was subsequently released as a commercial product in early 2012 after a commercialization stage in 2011. The package has been sold to a number of international users and is available from our spin-out commercialization company that I, my supervisor and another Ph.D. student (Arrian Prince-Pike) formed in October 2012, Inverse Problem Limited (`www.inverseproblem.co.nz`). Note that neither my supervisor nor Arrian contributed in any way to the development of the Excel Add-In, and it is a direct result of this work alone.

The JSteam Excel Add-In is not intended to provide a significant research contribution, however it does provide a simple to use mechanism to visualize and validate solutions obtained by the optimizer, as described in the next chapter. As detailed in the remaining sections of this chapter, the models described so far will be used to generate approximations suitable for use within an optimizer, and therefore JSteam Excel will be used for visualization and validation of these approximated solutions. Details of the JSteam Excel package are provided in Section B.1.6.

5.3.7 Case Study: Contact Energy Otahuhu B Power Station

In addition to the validation undertaken during the initial modelling of the utility unit operations, a final year electrical engineering student, Hanon Lim [185], used the JSteam Excel Add-In to model a 400MW combined-cycle gas turbine power station in Otahuhu, Auckland, New Zealand. The plant is owned by Contact Energy, one of New Zealand’s largest energy generators and was commissioned in January 2000 [59]. Figure 5.25 shows the plant in October 2013.

This case-study is presented to illustrate that the thermodynamic functions, steam utility unit operations and resulting Excel modelling environment presented so far can accurately model real industrial problems. While the modelling work was carried out by Hanon, his work was under direct supervision of this project as part of a final year undergraduate project.

Figure 5.25: Contact Otahuhu B power station.

The plant consists of a single Siemens SGT5-4000F gas turbine with a rated output of 275MW and is connected to an unfired (base-load only) triple-pressure HRSG with reheat. The boiler generates steam at 3 pressure levels; 113 bar (HP), 28 bar (MP) and 3.5 bar (LP), which are then supplied to a triple stage steam turbine with a rated output of 125MW. The gas turbine is supplied with dry natural gas from a national pipeline which is composed of predominately Methane (80%), Ethane (7.5%), Carbon Dioxide (6%) and Propane (3.5%), as well as small amounts of other components. The gas supply, as calculated by JSteam, has a Net Heating Value of 42471 kJ/kg, which matches quite accurately to data published from the gas supplier.

The PFD of the system is shown in Figure 5.26 and shows the model specified to match a maximum load condition of the plant. This is achieved by entering relevant model inputs from actual plant data (mass flows, temperatures, pressures), supplied by Contact, as well as by using the JSteam functionality to estimate and solve the remaining unknowns. In this early model version where there is no measurements, several assumptions have been made, especially within the HRSG. These were required to predict the temperatures between each of the coils and required assumptions such as assuming saturated conditions after each economiser, as well no pressure drops between the coils.

Using JSteam Excel Add-In with the functionality described so far in this chapter, Hanon Lim was able to quite accurately model the entire system with several operating points modelled and shown in Table 5.15. Considering this was performed with simple models, minimal hand-tuning and little manufacturer's data, this was an impressive result. To correct for the assumptions within the HRSG, Hanon Lim has taken this project further and built in several heat exchanger models based on the physical configuration of the HRSG, and he is currently looking to complete this work as part of a Master's Thesis. The latest version of the model, including Hanon

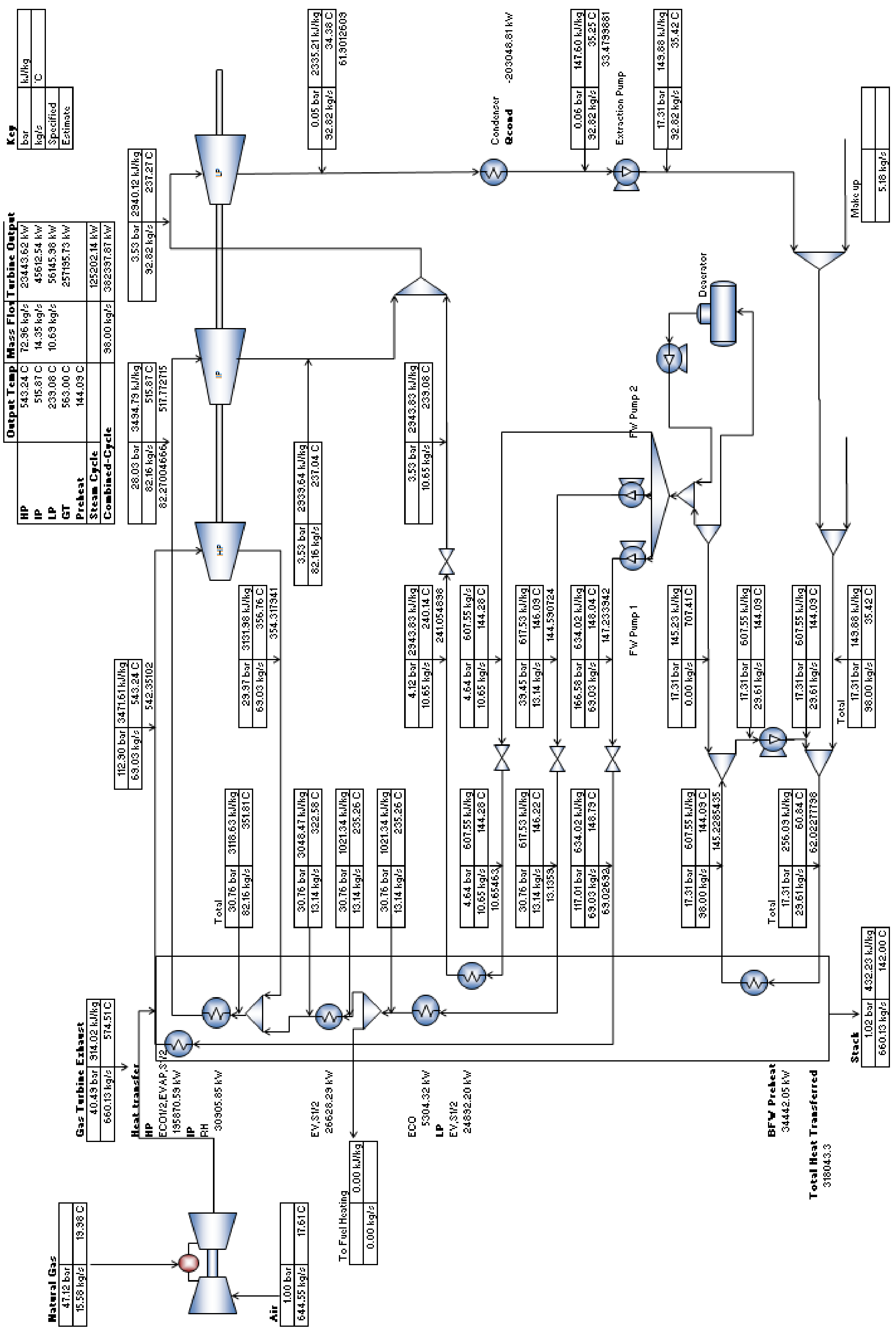


Figure 5.26: JSteam Excel Otahuhu B model at maximum load [185].

Lim's heat exchanger models, eliminates the need for assumptions and allows the system to converge from plant data alone.

Table 5.15: Otahuhu B model validation results.

Load	Measurement	Plant Data	Model Output	Error
Maximum	Steam Cycle Power Output	124.58 MW	125.2 MW	0.5%
	HP Turbine Inlet Temperature	542.4°C	543.2°C	0.15%
	MP Turbine Inlet Temperature	517.8°C	515.9°C	0.37%
	LP Turbine Inlet Temperature	241.1°C	240.1°C	0.41%
	HP Turbine Outlet Temperature	354.3°C	356.8°C	0.71%
Moderate	Steam Cycle Power Output	103.38 MW	98.25 MW	4.9%
	HP Turbine Inlet Temperature	543.6°C	551.5°C	1.45%
	MP Turbine Inlet Temperature	521.0°C	522.8°C	0.35%
	LP Turbine Inlet Temperature	229.2°C	228.8°C	0.17%
	HP Turbine Outlet Temperature	354.0°C	363.8°C	2.77%
Minimum	Steam Cycle Power Output	94.36 MW	89.15 MW	5.5%
	HP Turbine Inlet Temperature	547.3°C	555.5°C	1.5%
	MP Turbine Inlet Temperature	525.8°C	525.0°C	0.15%
	LP Turbine Inlet Temperature	224.6°C	224.4°C	0.01%
	HP Turbine Outlet Temperature	354.1°C	366.1°C	3.39%

The following sections build on the simple models presented so far by developing detailed part-load models. These will provide models of key pieces of equipment suitable for operational optimization, as will be detailed in the next Chapter. Each model described builds on both the work presented so far, as well as models and regressions from literature and industrial data.

5.4 Detailed Steam Turbine Model

As stated in Section 5.3.3.3, the steam turbine is the most common piece of equipment in steam utility systems, and thus considerable attention has been paid to developing a rigorous model. The model shown in Equation 5.7 represents a simple mass and energy balance around the unit, using standard textbook equations for an isentropic turbine. However, for optimization of a utility system a detailed model which includes an efficiency regression is required, especially when investigating part-load performance. This section implements a more detailed literature-based turbine model, using industrial data to develop a set of regression coefficients to predict turbine output characteristics.

5.4.1 Maximum Efficiency Regression

The maximum isentropic efficiency of a back-pressure steam turbine is an important design parameter when modelling real steam turbines, and even more so for steam turbo-generators which have a large impact on the economic calculations.

A common reference used by modern work (such as [4, 212]) is the early experimental work by Peterson and Mann [249], which characterizes the relationship between maximum isentropic efficiency and maximum turbine output power, as a function of inlet pressure. Using this data, it is then possible to regress a series of functions that represent each pressure level, as shown in Figure 5.27. Note a surface fit was attempted, however a simple function could not be found to describe the three problem dimensions as accurately as individual fits with a second stage interpolation (described below).

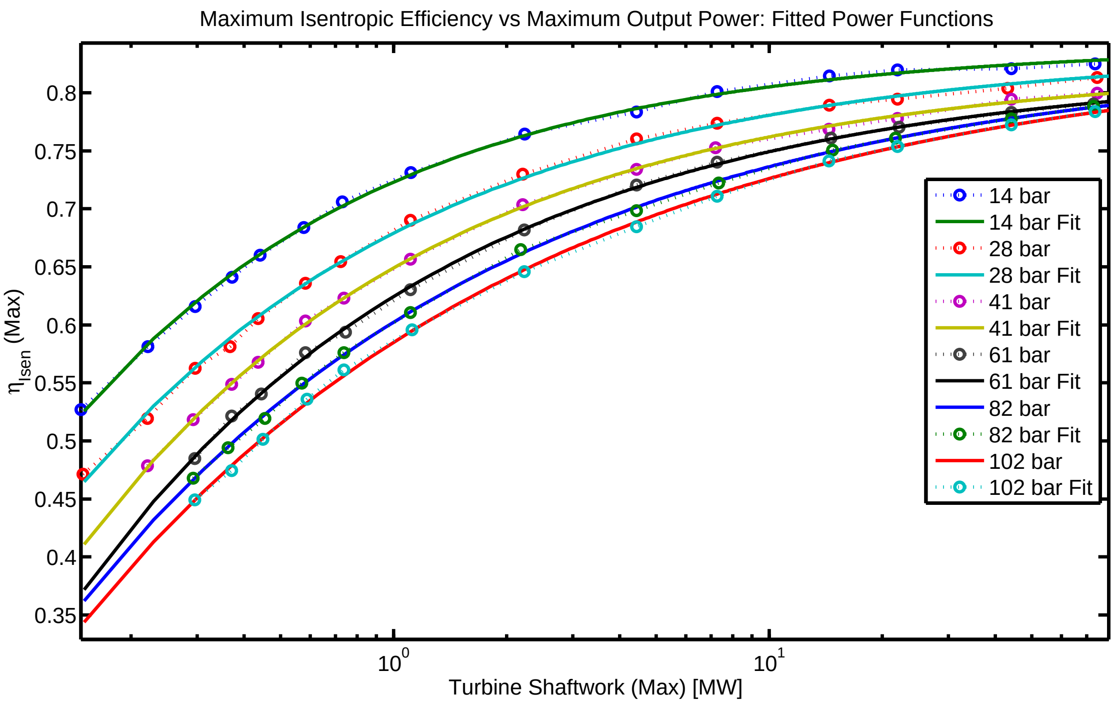


Figure 5.27: Power function fits to the Peterson and Mann turbine data, digitized from [212].

The function fitted to the efficiency curves was a power function

$$\eta = \alpha W^{\beta} + \gamma \tag{5.24}$$

where a function was fitted to each pressure level. The coefficients found are shown in Table 5.16.

To solve the maximum efficiency for a turbine not included in the experimental pressure levels, a shape-preserving piecewise cubic interpolation is used across all

Table 5.16: Regressed power function coefficients from Equation 5.24 for the turbine efficiency data.

Pressure	α	β	γ
14 bar	-0.1175	-0.5198	0.8406
28 bar	-0.1569	-0.4538	0.8359
41 bar	-0.1743	-0.454	0.8233
61 bar	-0.1975	-0.4344	0.8219
82 bar	-0.2314	-0.3755	0.8338
102 bar	-0.2542	-0.3519	0.8392

fitted functions at the specified power output. In MATLAB, this is performed using the `interp1` function with `'pchip'` selected as the interpolation method. The result of this procedure is the maximum isentropic efficiency for an arbitrary sized steam turbine, based on a specified inlet pressure level. This information can then be used as the upper limit for the part-load modelling, described in the next subsection.

5.4.2 Part-Load Modelling

To be able to optimize the operation of utility systems one must be able to move the operating point, which typically involves either switching steam flows between equipment, or manipulating the amount of steam through individual pieces of equipment. In the first case, which is typical of a back-pressure turbine, the turbine is connected to a fixed mechanical load (i.e. a pump, fan or compressor), and therefore it is assumed that the mass flow changes only a small amount with varying operating points: it is either on (steam supplied) or off (an electric motor is used instead to drive the load). In this case, a part-load model is not particularly useful.

However, steam turbines are also used as turbo-generators, in which the turbine is connected to an electrical generator. The amount of electricity to generate can vary depending on site power requirements, generation potential, and electricity prices and thus the connected load can vary widely. In this case, a part-load model must be used to predict the amount of power generated as a function of inlet conditions, mass flow through the turbine and the rated power output. Typically this information can be extracted from a manufacturer as a plot of isentropic efficiency versus output power, or mass flow versus output power (a Willans Line [58]). However, for the purposes of this work where this information may not be available, the following heuristics are proposed.

Equation 5.7 from Section 5.3.3.3 is the basic equation for a steam turbine is

$$W_{\text{shaftwork}} = M\Delta H_{\text{isen}}\eta_s \tag{5.25}$$

where ΔH_{Isen} is the isentropic enthalpy drop across the steam turbine and η_{Isen} is the isentropic efficiency of the turbine. For this section we will assume ΔH_{Isen} is constant and can be calculated offline using

$$\Delta H_{\text{isen}} = H_{\text{in}} - \texttt{HPS}(P_{\text{out}}, \texttt{SPH}(P_{\text{in}}, H_{\text{in}})) \tag{5.26}$$

This leaves the isentropic efficiency requiring calculation as a function of load, which varies nonlinearly. Following early work by Mavromatis in [212] it is shown that the Willans line can account for this nonlinear characteristic of turbine efficiency versus load in a linear fashion, thus enabling simpler optimization-ready models. Figure 5.28 shows the typical relationship between the Willans line and turbine efficiency for a 1MW steam turbine model running at 50 bar, 400°C and an outlet pressure of 10 bar.

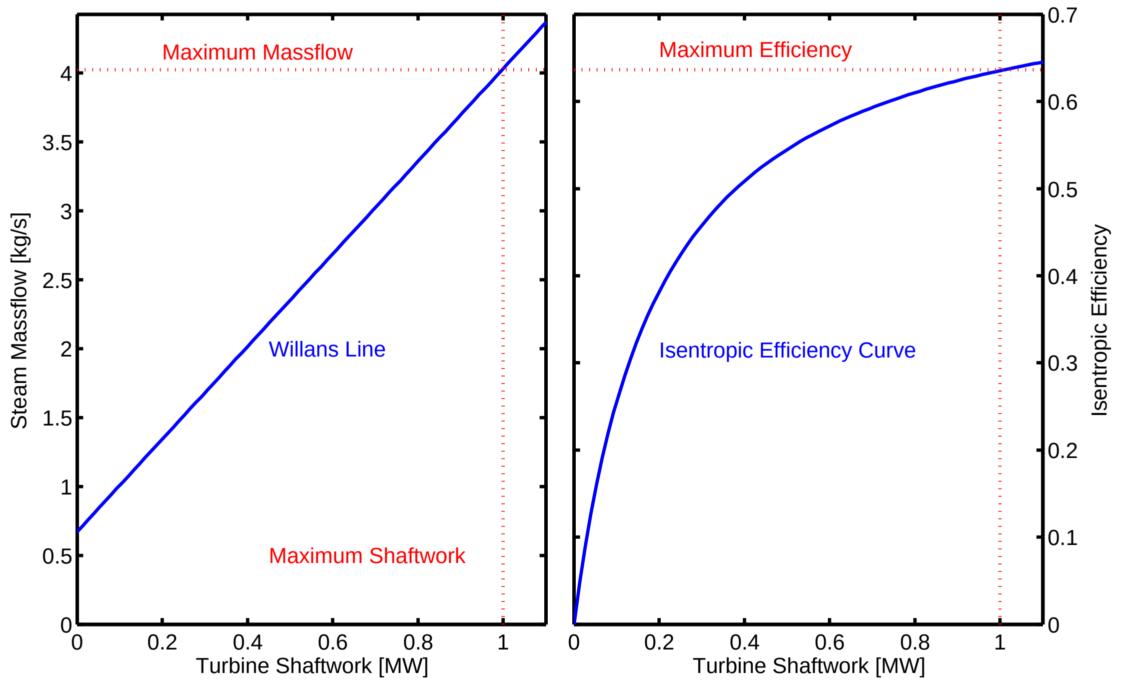


Figure 5.28: Willans line versus isentropic efficiency curve. Note the Willans line does not cross through the origin due to spinning losses (and other internal losses) within the steam turbine.

In order to determine the Willans line (and thus a linear model) for an arbitrary steam turbine (without measurement data), we can use the efficiency curves presented in 5.4.1 as the basis for the underlying power versus efficiency relationship.

Following Appendix B in [212], it is shown that the linear relationship

$$\Delta H_{\text{isen}} M_{\max} = \frac{W_{\max}}{\eta_{\text{isen(max)}}} = \alpha_s + \beta_s W_{\max} \tag{5.27}$$

can be built by replotting the efficiency curves as maximum output divided by maximum efficiency versus maximum power, as shown in Figure 5.29.

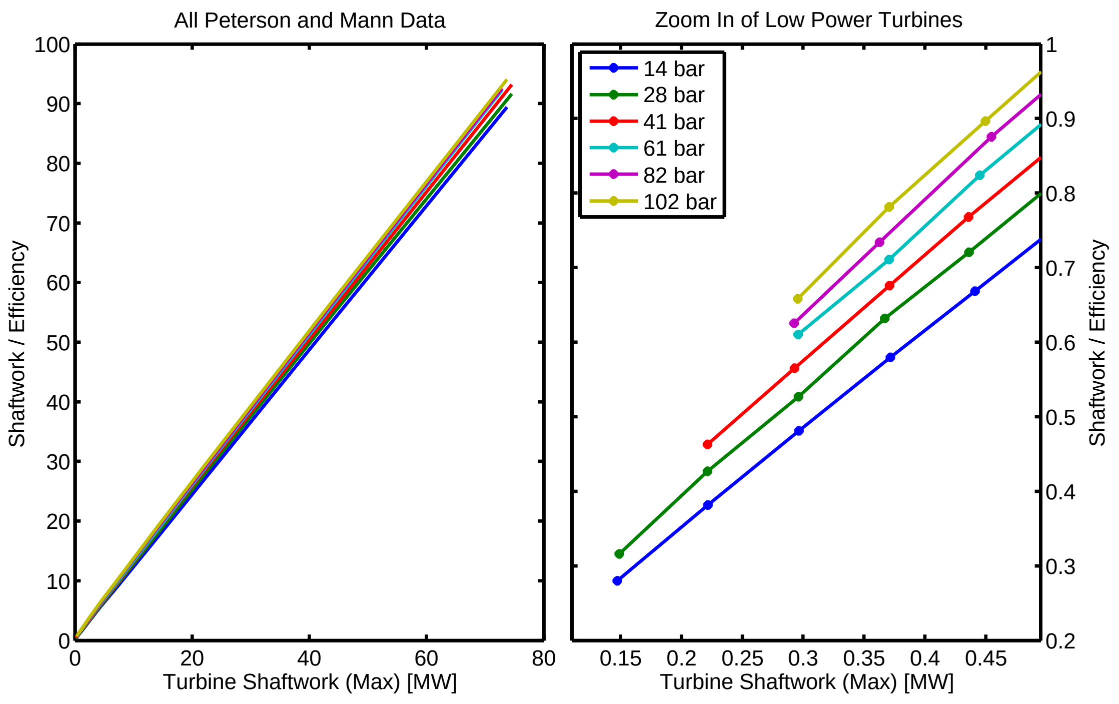


Figure 5.29: Linearization step of the Peterson and Mann data.

In the original reference (as well as in other later work, such as [315]), the coefficients α_s and β_s are found as a function of the saturation temperature, e.g.

$$\begin{aligned} \alpha_s &= a_0 + a_1 \texttt{TsatP}(P_{\text{In}}) \\ \beta_s &= b_0 + b_1 \texttt{TsatP}(P_{\text{In}}) \end{aligned} \tag{5.28}$$

beacuse each line in Figure 5.29 represents a different pressure level. In order to increase the accuracy of this linear approximation (noting the lines look straight, but there are variations which lead to errors of up to 30% as noted in the original reference), the range of turbine power outputs is split into two in order to account for small (< 1.2MW) and larger turbines.

The advantage in this work is that since we are not designing a new utility system, we can afford to calculate the constants α_s and β_s offline (before optimization). In order to maximize the accuracy of the regressed coefficients, the calculation procedure starts from the power function fits presented in the last section. Using these functions, a table is created over the power region of interest of the turbine, which can then be used with a gridded interpolation scheme to derive a new curve at the specified inlet pressure. The table for the turbine shown in Figure 5.28 can be viewed in Figure 5.30 showing the region of interest is from the maximum output power (1MW in this case) down to 40% of maximum load, 0.4MW. The 50 bar efficiency line is a result of a spline interpolated vertically across each of the power function fitted efficiency lines, derived from the original data. Note the original data is not used directly as this technique requires the data points to be aligned vertically.

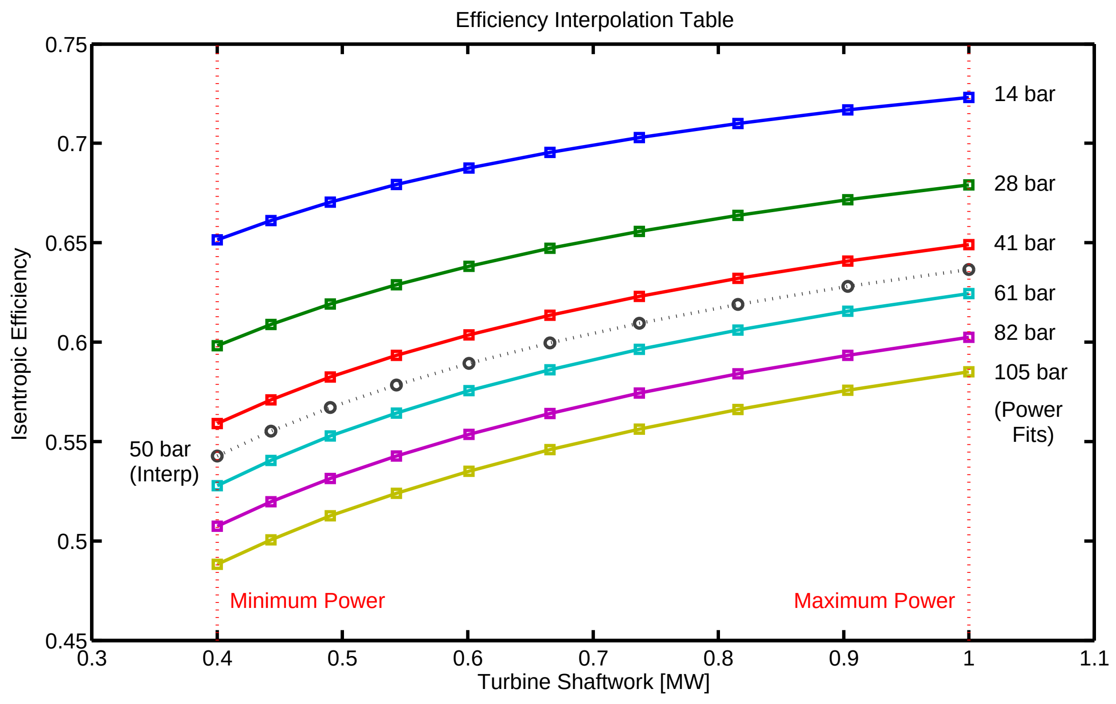


Figure 5.30: Efficiency interpolation table for a 1MW, 50 bar inlet turbine. Square points are created from the power function fits, while circles are spline interpolated points.

Using the fitted efficiency curve at the correct pressure, the linearization step can be performed over just this curve, maximizing the accuracy of the fit to just this turbine at this pressure level. From here, a simple straight line fit can be performed to find the coefficients α_s and β_s from Equation 5.27, noting from Figure 5.31 that the sum of squared errors is very close to 0, which indicates a virtually linear response.

Again following Appendix B in [212], a model for output power as a function of the Willans line can be derived as

$$W_{\text{shaftwork}} = nM - W_{\text{loss}} \tag{5.29}$$

where from early studies, and as described in this paper, the loss is typically char-

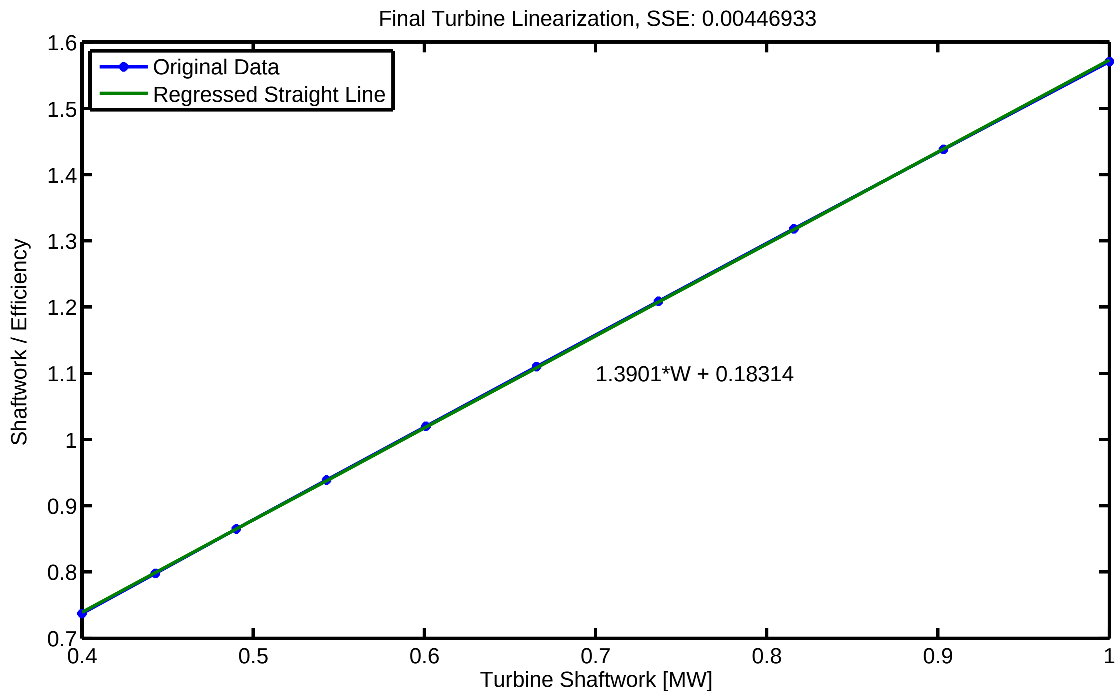


Figure 5.31: Straight line fit of turbine shaft work over isentropic efficiency for a 1MW, 50 bar inlet turbine.

acterized as

$$W_{\text{loss}} = 0.2 W_{\text{max}} \tag{5.30}$$

thus resulting in

$$W_{\text{max}} = \frac{5}{6} n M_{\text{max}} \tag{5.31}$$

where n is the slope of the Willans line. By substituting Equation 5.31 into Equation 5.27 and solving for the Willans line slope, we get the following expression

$$n = \frac{6}{5\beta_s}\left(\Delta H_{\text{isen}} - \frac{\alpha_s}{M_{\text{max}}}\right) \tag{5.32}$$

and similarly the intercept (or turbine loss)

$$W_{\text{loss}} = \frac{1}{5\beta_s}\left(\Delta H_{\text{isen}} M_{\text{max}} - \alpha_s\right) \tag{5.33}$$

By substituting Equations 5.32 and 5.33 into Equation 5.29, we can derive the final expressions for power output as a function mass flow and isentropic enthalpy drop

$$W_{\text{shaftwork}} = \frac{6}{5\beta_s}\left(\Delta H_{\text{isen}} - \frac{\alpha_s}{M_{\text{max}}}\right)\left(M - \frac{1}{6} M_{\text{max}}\right) \tag{5.34}$$

and isentropic efficiency

$$\eta_{\text{isen}} = \frac{6}{5\beta_s}\left(1 - \frac{\alpha_s}{\Delta H_{\text{isen}} M_{\max}}\right)\left(1 - \frac{M_{\max}}{6M}\right) \tag{5.35}$$

To show the effect of both turbine size (related to $M_{\max}$) and inlet pressure on isentropic efficiency, Figure 5.32 illustrates 6 models generated using the model presented in this chapter. Note this figure is similar to Figure 14 in [212], however it is recreated using the framework described so far. It is expected the curves in this work would follow the original Peterson and Mann data more accurately due to the extra interpolation step.

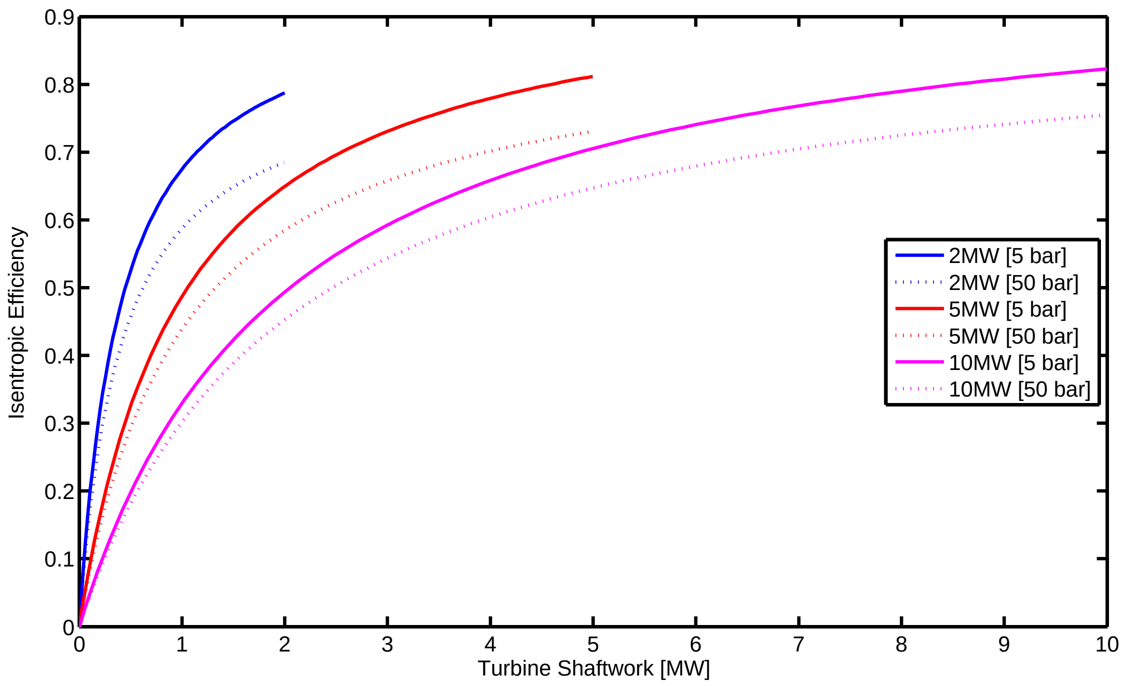


Figure 5.32: Isentropic efficiency curves for a range of turbines.

For problems where the output enthalpy is also required, one can simply use Equation 5.7 and substitute the derived expression for isentropic efficiency, Equation 5.35. Do note however that this function is no longer linear because the gradient now has the form

$$\frac{\partial H_{\text{out}}}{\partial M} = \frac{\gamma}{\sigma M^2} \tag{5.36}$$

This is not an issue in this work because the resulting formulation is expected to be nonlinear.

5.4.3 Incorporating Varying Input Enthalpy

For optimization studies where the input enthalpy may vary, such as when the header enthalpies are free to adjust based on an energy balance, it may be desirable not to fix ΔH_{Isen} to a constant, as performed in the last subsection.

The main effect of a varying input enthalpy is on the isentropic enthalpy drop across the turbine, which features as ΔH_{Isen} in both the output shaftwork calculation and the isentropic efficiency calculation. Therefore modelling this variation can assist with maintaining the accuracy of the model.

Using Equation 5.26, which incorporates the IAPWS thermodynamic routines within the JSteam engine, it is possible to generate the curve for ΔH_{Isen} for the area around the nominal operating point. As shown in Figure 5.33, the curve is a weak quadratic for typical operating regions; however we have chosen to model it as a straight line, weighted about the nominal operating point. The reason a linear fit is

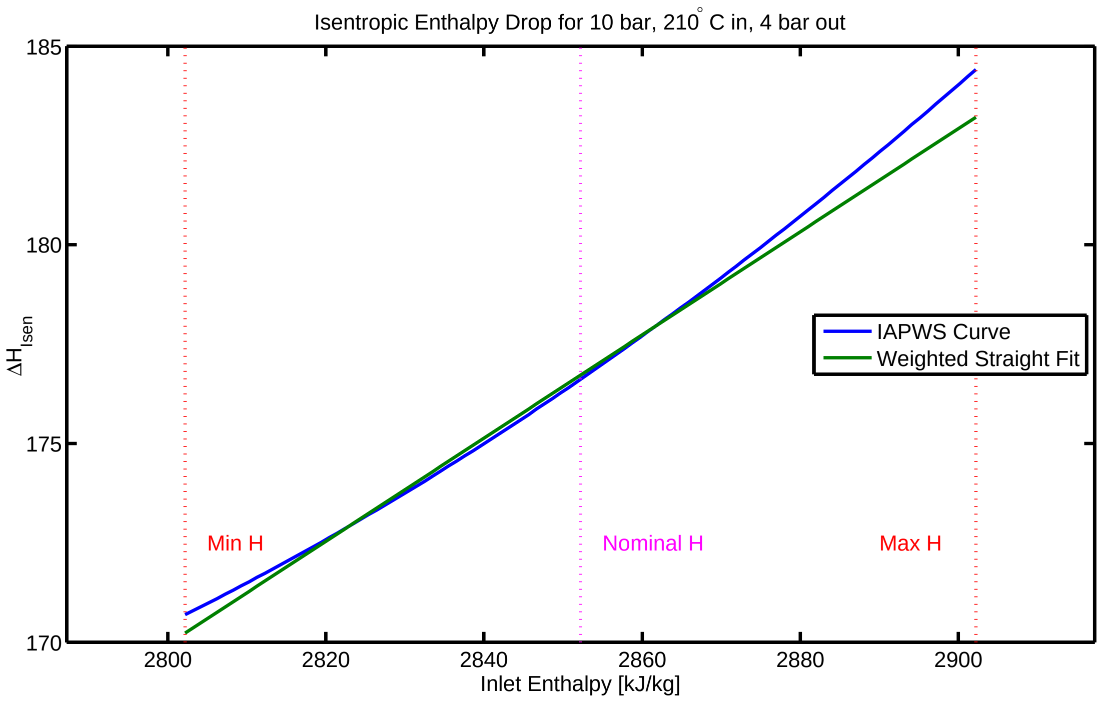


Figure 5.33: Isentropic enthalpy drop linear regression.

chosen is based on the typical range of ΔH_{Isen}, which as shown on the y-axis, varies only around 15% of the range of the x-axis (inlet enthalpy). Therefore an accurate fit would not gain significantly increased ΔH_{Isen} accuracy.

For turbines operating close to the saturation point of the inlet header, the minimum enthalpy is chosen as slightly above the saturation enthalpy of the inlet pressure, to ensure the sharp change in enthalpy is not modelled.

5.4.4 Complete Steam Turbine Model

To summarize the modelling presented within this section, the complete steam turbine model is as follows

$$W_{\text{shaftwork}} = \frac{6}{5\beta_s}\left(\Delta H_{\text{isen}} - \frac{\alpha_s}{M_{\max}}\right)\left(M - \frac{1}{6}M_{\max}\right) \tag{5.37}$$

$$H_{\text{out}} = H_{\text{in}} - \frac{6\Delta H_{\text{isen}}}{5\beta_s}\left(1 - \frac{\alpha_s}{\Delta H_{\text{isen}} M_{\max}}\right)\left(1 - \frac{M_{\max}}{6M}\right) \tag{5.38}$$

where

$$\Delta H_{\text{isen}} = \alpha_h + \beta_h H_{\text{in}} \tag{5.39}$$

$$M_{\max} = \frac{W_{\max}}{\Delta H_{\text{isen}} \eta_{\max}} \tag{5.40}$$

and the remaining parameters are listed in Table 5.17. Note the expression for output shaftwork is linear, while the output enthalpy expression is nonlinear. Maximum mass flow is typically assumed as a constant using the value of ΔH_{Isen} which is taken at the nominal point (but could be substituted as a function of H_{In}, if required).

Table 5.17: Steam turbine model parameters.

$W_{\text{shaftwork}}$	kW	Output Shaftwork
$W_{\max}$	kW	Maximum Output Shaftwork (Manufacturer Specifcation)
$\eta_{\max}$	fraction	Maximum Isentropic Efficiency (from Peterson and Mann)
M	kg/s	Mass flow through turbine
$M_{\max}$	kg/s	Maximum mass flow through turbine
H_{in}	kJ/kg	Input Enthalpy
H_{out}	kJ/kg	Output Enthalpy
ΔH_{Isen}		Isentropic Enthalpy drop across turbine
α_s		Willans Line regression intercept
β_s		Willans Line regression gradient
α_h		Isentropic Enthalpy drop regression intercept
β_h		Isentropic Enthalpy drop regression gradient

Section B.2.1 provides an example of the use of the detailed steam turbine model.

5.5 Detailed Steam Boiler Model

For a typical utility system where one or more steam boilers produce the bulk of working steam, these boilers represent the largest managed cost of the system. For two of the utility systems used for validation of iUO in Malaysia, boiler fuel gas made up between 80 and 95% of the total operational expense of the system, eclipsing fuel-gas costs of multiple furnaces and gas turbines, and the relatively insignificant demineralized water production. Therefore an accurate model of the steam boiler is paramount to ensuring realistic optimized operating points. This section develops a part-load model for the JSteam Fired Boiler and adds a variable efficiency from operational data reported in literature.

5.5.1 Part-Load Modelling

Using the JSteam Fired Boiler model from Section 5.3.5.2, one of the assumptions made is that the boiler efficiency is independent of load, and rather that it is a function of losses due to characteristics such as input air requiring heating by the boiler, as well as stack heat losses. These losses, while a function of the required fuel energy, are directly proportional and thus result in a constant efficiency. An example JSteam Fired Boiler fired with 100% methane is shown in Figure 5.34 with input conditions shown in Table 5.18. Note the JSteam model is independent of boiler size, and therefore the x-axis is absolute.

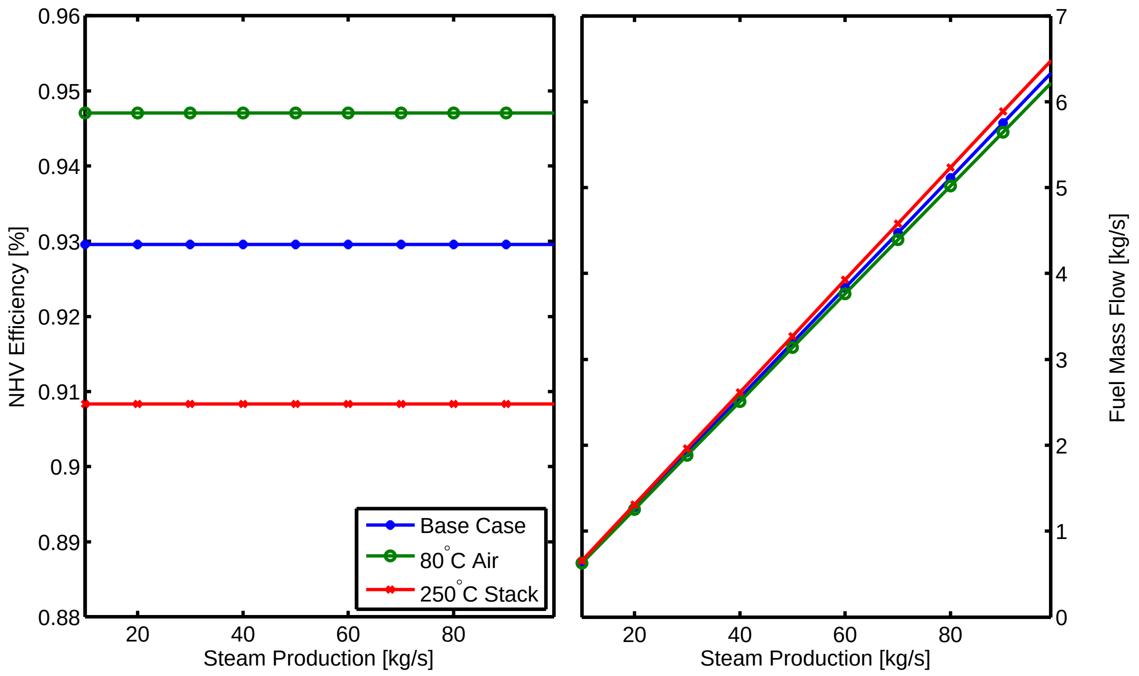


Figure 5.34: JSteam Fired Boiler efficiency as a function of steam production.

In order to account for part-load performance of a specific boiler we must introduce a load-dependent efficiency curve, as the JSteam model is effectively ideal. Following recent papers in the modelling of utility systems, (such as [4, 315]) the general efficiency model used for gas-fired boilers is based on earlier work by Shang

Table 5.18: JSteam Fired Boiler base case example input data.

H_{BFW}	250 kJ/kg	Boiler Feed Water Enthalpy
P_{Steam}	40 bar	Steam Pressure
T_{Steam}	400°C	Steam Temperature
T_{Air}	30°C	Air Temperature
T_{Fuel}	30°C	Fuel Temperature
T_{Stack}	200°C	Stack Temperature
F_{BD}	1%	Blowdown Ratio
F_{O_2}	1%	Inlet Excess O_2

and Kokossis [294] where they derive

$$\frac{Q_{\text{loss}}}{Q_{\text{steam}}}\frac{M}{M_{\text{max}}} = \alpha + \beta\frac{M}{M_{\text{max}}}, \quad 10\% \leq \frac{M}{M_{\text{max}}} \leq 100\% \tag{5.41}$$

to describe the ratio of energy loss to the production of steam, as a function of the ratio of mass flow of steam to the maximum mass flow. In order to fit the coefficients (α, β) they used early work by Pattison & Sharma [244] which provided the data reproduced in Figure 5.35. This data was collected by the British Gas Corporation in the 1970s and 1980s and when linearized, as shown in the bottom plot, results in the linear fit with coefficients $\alpha = 0.0126, \beta = 0.2156$. Using these values it is possible to estimate the loses that result from both boiler surface and flue gas losses for the boilers originally surveyed.

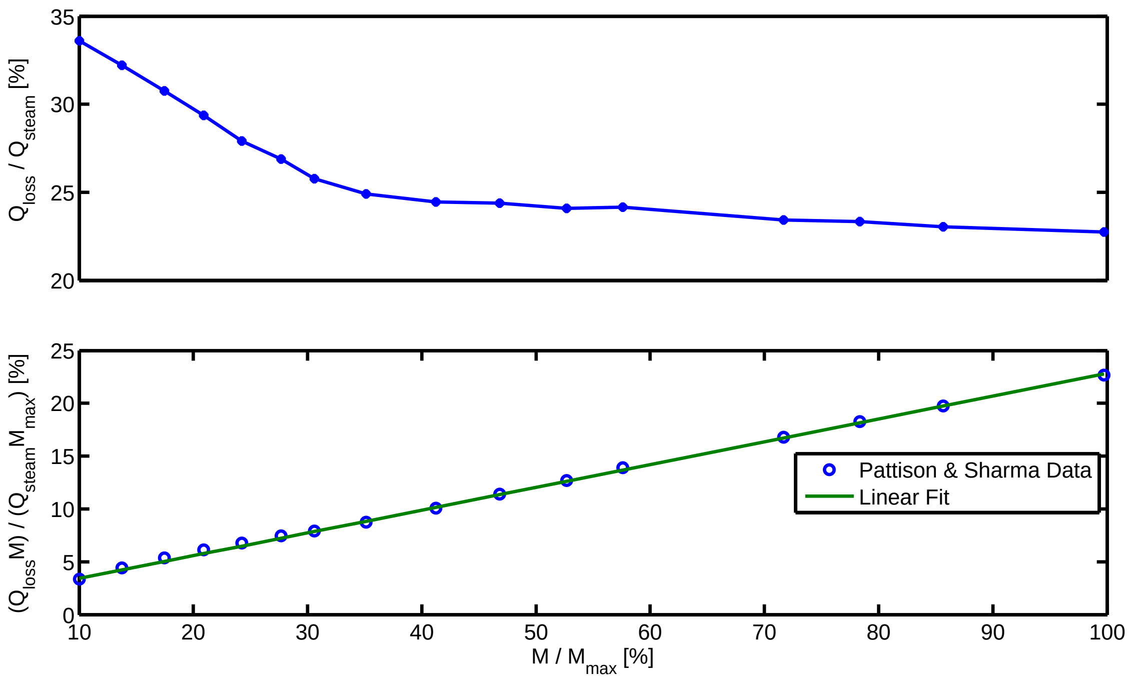


Figure 5.35: Boiler losses as a function of load (digitized from [294]).

Rearranging Equation 5.41 we can solve the losses as

$$Q_{\text{loss}} = \frac{Q_{\text{steam}}}{M} \left(\alpha M_{\text{max}} + \beta M\right) \tag{5.42}$$

which forms a linear expression for losses as a function of the ratio of mass flow of steam to maximum mass flow. This can be further rearranged to solve for the boiler efficiency

$$\eta = \frac{Q_{\text{steam}}}{Q_{\text{fuel}}} = \frac{\frac{M}{M_{\text{max}}}}{\left(1+\beta\right)\frac{M}{M_{\text{max}}} + \alpha} \tag{5.43}$$

Do note however this model is somewhat limited because it does not account for blowdown losses, nor does it specifically take into account the operating condition of the boiler (such as air and fuel temperature, O_2 specifications, etc). Varbanov et al [315] corrected for the blowdown loss in their model of fuel energy required as a function of mass flow

$$Q_{\text{fuel}} = \Delta H_{\text{Boiler}} \left[\left(1+\beta\right) M + \alpha M_{\text{max}}\right] + F_{\text{BD}} M \Delta H_{\text{Econ}} \tag{5.44}$$

They did however still retain the same coefficients from the work of Shang and Kokossis. Later work by Aguilar et al [4] proposed a slightly different boiler model to account for multi-component fuels

$$M\left(\Delta H_{\text{Boiler}} + \Delta H_{\text{Econ}} F_{\text{BD}}\right) = \left(\frac{Q_{\text{f1}}}{B_{\text{f1}}} + \frac{Q_{\text{f2}}}{B_{\text{f2}}} + \cdots + \frac{Q_{\text{fm}}}{B_{\text{fm}}}\right) - D_{\text{avg}} \tag{5.45}$$

however they also reference back to the Shang and Kokossis paper as the underlying relationship used when solving for the model coefficients.

Following a similar technique of Aguilar et al, the proposed part-load boiler model will be based on the relationships derived by Shang and Kokossis, but the coefficients α, β will be re-regressed based on the extra information present in the JSteam model. This decision is based on the original coefficients being effectively independent of varying conditions such as boiler air-preheaters, varying excess oxygen specifications, and other operational specifications included in the JSteam model. Re-examining Equation 5.42, we can see that the α coefficient represents constant losses associated with the size of the boiler, while the β term represents load dependent losses. Currently the JSteam model assumes α is 0 (i.e. the intercept of Figure 5.35) indicating there are no fixed losses (this results in a constant efficiency), while the β term is specification dependent (this results in the maximum achievable efficiency).

In order to find α, β for the JSteam model, Equation 5.41 is implemented and uses the results obtained from a series of model outputs and the following equations

$$Q_{\text{loss}} = Q_{\text{fuel}} - Q_{\text{steam}} \tag{5.46}$$

$$Q_{\text{steam}} = -\left(H_{\text{BFW}} - H_{\text{steam}} - F_{\text{BD}}H_{\text{BD}} + F_{\text{BD}}H_{\text{BFW}}\right) M_{\text{steam}} \tag{5.47}$$

$$Q_{\text{fuel}} = M_{\text{fuel}}\text{NHV}_{\text{fuel}} \tag{5.48}$$

where M_{fuel} is calculated from the JSteam model based on the fuel, air, O_2 and other operational specifications for the desired steam production. By evaluating this set of equations and regressing a straight line (as done in Figure 5.35), we can find the Shang and Kokossis model β_{J} coefficient (where the subscript $_{\text{J}}$ indicates the JSteam fitted coefficient). As stated, it is not possible to find α_{J} using the JSteam model so this must be estimated from the original coefficient. As a first approximation α_{J} is calculated as

$$\alpha_{\text{J}} = \alpha\frac{\beta_{\text{J}}}{\beta} \tag{5.49}$$

noting α_{J} is found as the ratio of difference between the new JSteam β_{J} and the original Shang and Kokossis β. Where actual boiler plant data is available this should obviously be used to regress $\alpha_{\text{J}}, \beta_{\text{J}}$, however this approximation is sufficient for the purposes of this work. Section B.2.2 provides an example of the use of the detailed steam turbine model, including the modification in Equation 5.49.

5.5.2 Complete Steam Boiler Model

To summarize the modelling presented in this section, the complete steam boiler model is as follows

$$M_{\text{fuel}} = \lambda + \gamma M \tag{5.50}$$

$$\eta = \frac{M}{\left(1 + \beta_{\text{J}}\right) M + \alpha_{\text{J}} M_{\max}} \tag{5.51}$$

where

$$\lambda = \frac{-\alpha_{\text{J}} M_{\max} \Delta H_{\text{boiler}}}{\text{NHV}_{\text{fuel}}} \tag{5.52}$$

$$\gamma = \frac{-\left(1 + \beta_{\text{J}}\right)\Delta H_{\text{boiler}}}{\text{NHV}_{\text{fuel}}} \tag{5.53}$$

$$\Delta H_{\text{boiler}} = H_{\text{BFW}} - H_{\text{steam}} - F_{\text{BD}}H_{\text{BD}} + F_{\text{BD}}H_{\text{BFW}} \tag{5.54}$$

$$\alpha_{\text{J}} = \alpha\frac{\beta_{\text{J}}}{\beta} \tag{5.55}$$

and the remaining parameters are listed in Table 5.19. Note the expression for required fuel mass flow versus steam production is linear, as is standard in literature.

Table 5.19: Boiler model parameters.

η	fraction	Boiler NHV Efficiency
NHV_{fuel}	kJ/kg	Fuel Net Heating Value
M	kg/s	Steam Production Mass flow (Manufacturer Spec)
M_{max}	kg/s	Maximum Steam Production
M_{fuel}	kg/s	Fuel Gas Mass Flow
H_{BFW}	kJ/kg	Boiler Feed Water Enthalpy
H_{steam}	kJ/kg	Steam Enthalpy
H_{BD}	kJ/kg	Blowdown Enthalpy (saturated water)
F_{BD}	fraction	Blowdown Fraction
ΔH_{boiler}		Enthalpy change across boiler (econ + superheater)
λ		Boiler fuel prediction model intercept
γ		Boiler fuel prediction model gradient
α_{J}		JSteam regressed intercept
β_{J}		JSteam regressed gradient
α		Shang and Kokossis intercept: 0.0126
β		Shang and Kokossis gradient: 0.2156

5.6 Detailed Gas Turbine Model

Common within newer utility systems is the ability to generate electricity on-site, and for larger systems where more than a few MW is required, a Gas Turbine Generator (GTG) is an efficient means of generating power. The JSteam GTG model presented in Section 5.3.5.3 implements a very simple method of calculating fuel consumption when the efficiency of the unit is known. However, as with the other detailed models presented so far, this work assumes no efficiency data is given and that we only know the unit size and fuel supplied. This section expands the simple model by using two pieces of industrial gas turbine data to derive an expression for maximum GTG efficiency as a function of size, and then to derive part-load equations for variable efficiency as a function of load.

5.6.1 Maximum Efficiency Regression

The maximum open-cycle (i.e. no HRSG) efficiency of a gas turbine is primarily a function of the turbine size, as well as of the manufacturer's design. In the absence of manufacturer specifications, two pieces of literature provide an estimate of the maximum efficiency of an open-cycle gas turbine, based on two sets of operational data. For large gas turbines, (26MW $\leq$ Q $\leq$ 255MW) Varbanov et al [315] provide a linear regression based on General Electric operational data. For small turbines, (2MW $\leq$ Q $\leq$ 8MW) Aguilar et al [4] provide a 'rated efficiency trend' based on surveyed gas turbines which has been digitized and regressed for this work.

The small turbine regression equation is as follows

$$\eta_{\max} = \alpha_s \exp(\beta_s W_{\max}) + \gamma_s \exp(\delta_s W_{\max}), \quad 1.5\text{MW} \leq W_{\max} \leq 8.5\text{MW} \tag{5.56}$$

where $\alpha_s = 0.3308, \beta_s = -0.00572, \gamma_s = -0.09133$, and $\delta_s = -0.3098$, noting the subscript 's' stands for small. For the larger turbines the regression equation is

$$\eta_{\max} = \frac{W_{\max}}{\alpha_l + \beta_l W_{\max}}, \quad 26.1\text{MW} \leq W_{\max} \leq 255.6\text{MW} \tag{5.57}$$

where $\alpha_l = 21.9917, \beta_l = 2.6683$. In order to create a smooth curve which connects both regressions, a power fit has been used to approximate the entire operational region as follows

$$\eta_{\max} = \alpha_{\text{J}} W_{\max}^{\beta_{\text{J}}} + \gamma_{\text{J}} \tag{5.58}$$

where $\alpha_{\text{J}} = 1.116$, and $\beta_{\text{J}} = 0.01389$ and $\gamma_{\text{J}} = -0.8416$, noting the subscript 'J' infers JSteam regressed coefficients, and W is specified in MW.

It is acknowledged that this complete fit, as shown in Figure 5.36, does not follow the trend for the larger gas turbines below around 100MW. This has however been sacrificed in order to match the smaller gas turbine curve, as well as a smooth transition for the area between both sets of data. It is noted in multiple papers such as [315] that "Coefficients differ between different gas turbine manufacturers and even different gas turbine types from the same manufacturer", therefore this curve is simply an estimate, and manufacturer data should always be substituted if available.

5.6.2 Part-Load Modelling

Similarly to the steam turbine, gas turbine part-load models are often based on a Willans line approximation. As shown in [4] the gas turbine part load performance is approximately linear when viewed as a relationship between fuel duty and electric power output, and thus a Willans line is a reasonable approximation. Following the derivation in [315], a part-load model based on the Willans line is described as

$$W_{\text{shaftwork}} = nM_{\text{fuel}} - W_{\text{loss}} \tag{5.59}$$

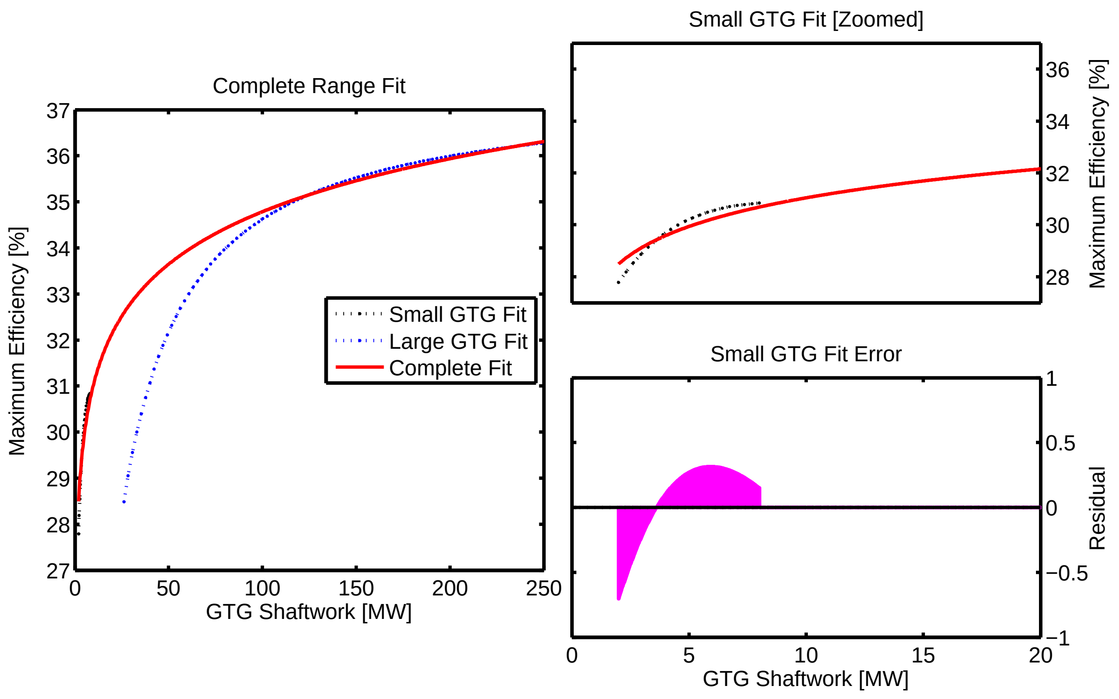


Figure 5.36: Gas turbine maximum efficiency as a function of gas turbine power output.

where n is the gradient of the Willans line, M is specified in kg/s and W is specified in kW. The loss term is approximated as a function of the maximum gas turbine output

$$W_{\text{loss}} = LW_{\text{max}} \tag{5.60}$$

where L is a parameter which must be regressed for each turbine. For the purposes of simplifying the design, L is assumed constant at 20%, as with steam turbines, however it is noted for small gas turbines this value can be much larger.

When operating at full load Equation 5.59 is written as

$$W_{\text{max}} = nM_{\text{fuel(max)}} - 0.2W_{\text{max}} \tag{5.61}$$

$$= \frac{5}{6}nM_{\text{fuel(max)}} \tag{5.62}$$

which is the same as with the steam turbine, and substituting steam with fuel. In order to relate the the mass flow of fuel to available duty, the Net Heating Value (NHV_{fuel}, specified in kJ/kg) is substituted for ΔH_{isen} in Equation 5.27 and solved for the maximum shaftwork

$$W_{\text{max}} = \frac{1}{\beta_{gt}}\left(Q_{\text{fuel(max)}} - \alpha_{gt}\right) \tag{5.63}$$

where

$$Q_{\text{fuel(max)}} = \text{NHV}_{\text{fuel}} M_{\text{fuel(max)}} \quad \text{(in kW)} \tag{5.64}$$

$$M_{\text{fuel(max)}} = \frac{W_{\text{max}}}{\eta_{\text{max}} \text{NHV}_{\text{fuel}}} \tag{5.65}$$

noting α_{gt}, β_{gt} are regressed coefficients, described below, and η_{max} is solved from Equation 5.58. Following the same derivation as per the steam turbine, expressions for the gradient and loss can be derived as

$$n = \frac{6}{5\beta_{gt}} \left(\text{NHV}_{\text{fuel}} - \frac{\alpha_{gt}}{M_{\text{fuel(max)}}} \right) \tag{5.66}$$

$$W_{\text{loss}} = \frac{1}{5\beta_{gt}} \left(Q_{\text{fuel(max)}} - \alpha_{gt} \right) \tag{5.67}$$

which when substituted back into Equation 5.59 and solved for the mass flow of fuel required, results in

$$M_{\text{fuel}} = -\frac{M_{\text{fuel(max)}} \left(Q_{\text{fuel(max)}} - \alpha_{gt} + 5\beta_{gt} W_{\text{shaftwork}} \right)}{6\alpha_{gt} - 6Q_{\text{fuel(max)}}} \tag{5.68}$$

and similarly for the efficiency of the gas turbine

$$\eta = -\frac{W_{\text{shaftwork}} \left(6\alpha_{gt} - 6Q_{\text{fuel(max)}} \right)}{Q_{\text{fuel(max)}} \left(Q_{\text{fuel(max)}} - \alpha_{gt} + 5\beta_{gt} W_{\text{shaftwork}} \right)} \tag{5.69}$$

In order to derive the two constants, α_{gt}, β_{gt}, a similar approach as was done for the steam turbine in Section 5.4.2 has been performed, whereby Equation 5.58 is linearized over the operating region of interest. Typically this is between 10% and 100% of rated gas turbine output, and the linearization is performed by fitting a straight line to W versus W/η_{max}, as shown in Figure 5.37.

For an example of the part-load performance described by this model, Section B.2.3 provides a comparison with literature as well as three hypothetical models.

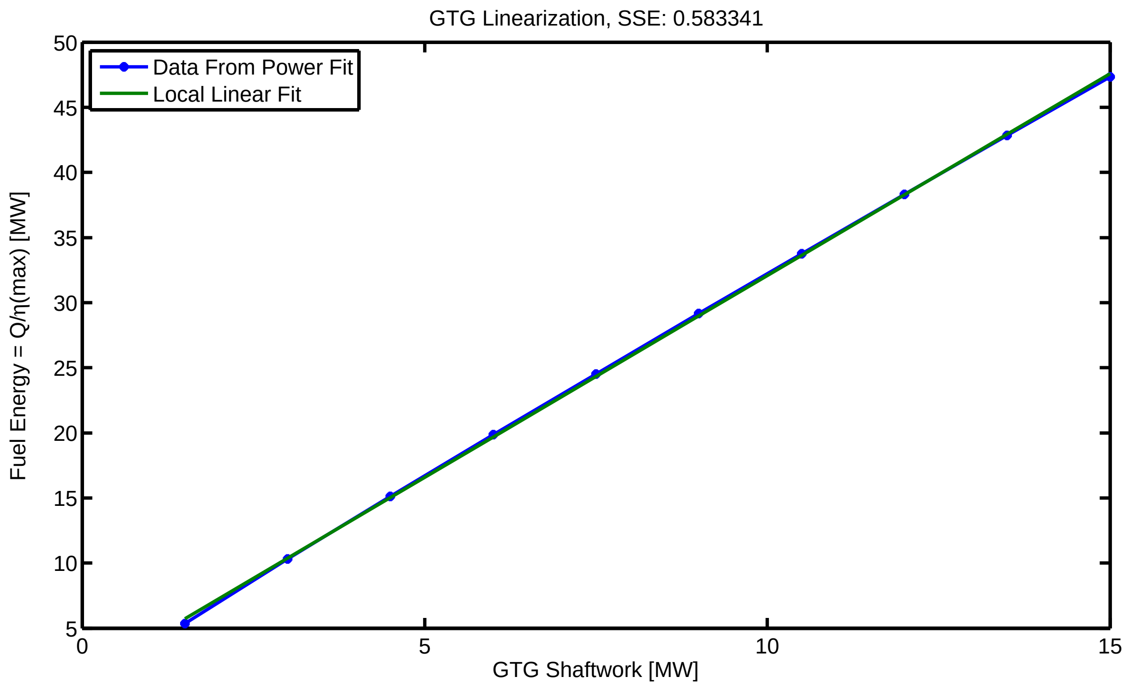


Figure 5.37: Straight line fit of gas turbine shaft work over efficiency for a 15MW unit.

5.6.3 Complete Gas Turbine Model

To summarize the modelling presented in this section, the complete gas turbine model is as follows

$$M_{\text{fuel}} = -\frac{M_{\text{fuel(max)}}\left(Q_{\text{fuel(max)}} - \alpha_{gt} + 5\beta_{gt}W_{\text{shaftwork}}\right)}{6\alpha_{gt} - 6Q_{\text{fuel(max)}}} \tag{5.70}$$

$$\eta = -\frac{W_{\text{shaftwork}}\left(6\alpha_{gt} - 6Q_{\text{fuel(max)}}\right)}{Q_{\text{fuel(max)}}\left(Q_{\text{fuel(max)}} - \alpha_{gt} + 5\beta_{gt}W_{\text{shaftwork}}\right)} \tag{5.71}$$

$$\tag{5.72}$$

where

$$Q_{\text{fuel(max)}} = \text{NHV}_{\text{fuel}}M_{\text{fuel(max)}} \tag{5.73}$$

$$M_{\text{fuel(max)}} = \frac{W_{\text{max}}}{\eta_{\text{max}}\text{NHV}_{\text{fuel}}} \tag{5.74}$$

and the remaining parameters are listed in Table 5.20.

Table 5.20: Gas turbine model parameters.

$W_{\text{shaftwork}}$	kW	Output Shaftwork
$W_{\max}$	kW	Maximum Output Shaftwork (Manufacturer Spec)
$Q_{\text{fuel(max)}}$	kW	Fuel Duty at Maximum Output
NHV_{fuel}	kJ/kg	Fuel Net Heating Value
$\eta_{\max}$	fraction	Maximum Open-Cycle Efficiency (from Equation 5.58)
η	fraction	Open-Cycle efficiency
M_{fuel}	kg/s	Mass Flow of Fuel
$M_{\text{fuel(max)}}$	kg/s	Maximum Mass Flow of Fuel
α_{gt}		Willans Line regression intercept
β_{gt}		Willans Line regression gradient

5.7 Approximate Heat Recovery Steam Generator (HRSG) Model

We have opted not to develop a detailed model based on industry data regressions, simply due to the complexity of the unit for the HRSG. As the JSteam HRSG model described in Section 5.3.5.4 already accounts for a multitude of operational specifications, it was decided to approximate this model and use it in conjunction with the detailed gas turbine model from the last section. As shown below, the approximated model performs as expected when constrained within sensible operating limits.

5.7.1 Base Load Model

For an unfired HRSG, (base load only) the steam production is a function of both the gas turbine and HRSG boiler, including operating specifications for both units. The main variable that controls the steam production potential is the available duty in the gas turbine exhaust so this must be predicted by the gas turbine model. Using the current JSteam Gas Turbine model, the exhaust temperature of the gas turbine is specified, so that an energy balance across the unit can be calculated to determine the exhaust mass flow, and therefore its duty. This is a reasonable assumption because the exhaust temperature is one of the controlled variables by a gas turbine control system, and thus expected to be fairly constant (at full load).

It is however acknowledged the exhaust temperature and mass flow will vary as a function of part-load conditions, and as noted in [315], without manufacturer or operating data, predicting the exhaust mass flow (or temperature) requires turbine specific regressions, which is beyond the scope for this model. For the HRSG model therefore, the gas turbine output temperature is assumed constant across part-load conditions.

The method used to develop an approximate HRSG model utilizes the JSteam Gas Turbine and HRSG models, as well as a detailed gas turbine model from the last section. Leveraging the fact we are targeting operational optimization, we can generate an expression to relate steam production versus power output about the current operating point, which turns out to be approximately linear. Model building is completed as follows:

1. Using the rated GTG output (W_{max}), together with the fuel gas duty and the detailed gas turbine model, an expression which relates efficiency versus part-load power output is derived.
2. The JSteam Gas Turbine and HRSG models are run over a series of intermediate points within the expected power output range of the GTG. The gas turbine efficiency is specified from part (1), while the remaining specifications are entered as per the current operating point. This enables the HRSG model to estimate the output steam production as a function of HRSG specifications such as minimum stack temperature, steam temperature and pressure, blowdown ratio, as well as GTG specifications such as fuel composition, fuel and air temperature and compressor pressure.
3. A straight line is regressed through the steam production versus gas turbine output data from (2). This is then used to predict base load steam production for this unit.

As long as the HRSG is operated away from operational constraints, such as approach temperatures for each of the exchangers, the relationship between input energy and output steam production is virtually linear, as found in Section 5.5. For an example of a combined cycle gas turbine with unfired HRSG, see Section B.2.4.1.

5.7.2 Modelling Secondary Firing

When modelling an HRSG with secondary (or supplementary) firing, the model now has an extra degree of freedom, which is the specified amount of steam to produce, in addition to the gas turbine power output. This means our model to approximate the required fuel flow must be a surface (2D) rather than a line (1D), as per all other models in this work. In addition, the model must be able to predict the base steam load (as above) as well as the maximum steam production, in order for the optimizer to select a valid steam mass flow within the limits of the unit.

As with the base-load HRSG, the secondary fired HRSG surface of total fuel flow (GTG fuel + HRSG supplementary fuel) with respect to gas turbine output and steam demand is approximately linear. However, due to operational constraints, such as minimum stack temperature, minimum exchanger approach temperatures and minimum stack O_2, the operating region of the HRSG is also subject to two further constraints. Looking at the surface of total fuel flow, viewed in Figure 5.38, there are two effects within this example that demonstrate the constraints: The top left of the surface is missing as the steam demand exceeds the maximum steam production available given the GTG exhaust gas duty, and the bottom right of the surface is missing due to the steam demand being less than the base load steam production.

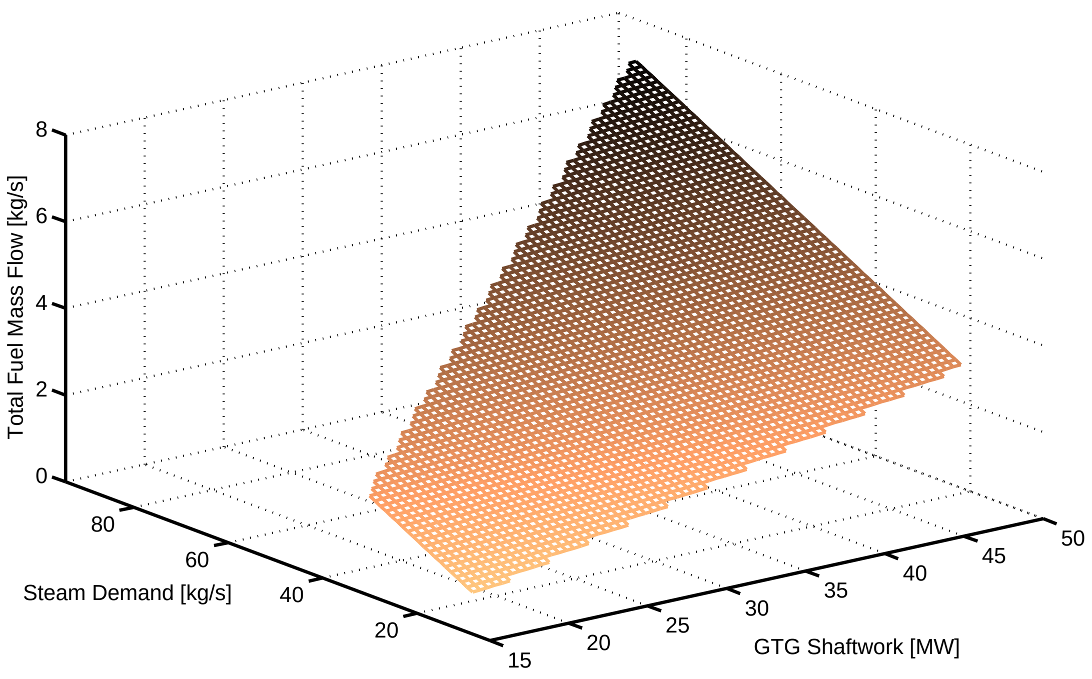


Figure 5.38: Fuel flow surface using JSteam gas turbine and HRSG models.

As stated, this surface is approximately linear and can therefore be regressed with a 2D surface

$$M_{\text{fuel(total)}} = \alpha_{\text{hrsg}} + \beta_{\text{hrsg}} W_{\text{gtg}} + \gamma_{\text{hrsg}} M_{\text{steam}} \tag{5.75}$$

where W_{gtg} is in kW, and M_{steam} and $M_{\text{fuel(total)}}$ are in kg/s. Using the same procedure as for the unfired HRSG, the JSteam Gas Turbine and HRSG models are run over a series of evaluation points to build up the required data to fit the surface to. Points which fall outside the operating constraints are removed, leaving only the linear surface. As well as building up the expression for total fuel flow, expressions for base steam production and maximum steam production are fitted to data reported by the JSteam models, thus building up expressions for the constraints. This again creates an accurate approximation of the GTG + HRSG at the current operating point, however it now includes part load models for both the gas turbine power as well as the HRSG steam production.

Section B.2.4.2 provides an example of the use of the part-load fired HRSG model, together with typical operational constraints.

5.8 Summary

This chapter has introduced the utility system as a large and complex system of interconnected equipment models, and shown the level of modelling detail required to begin to estimate the performance of such a plant. It includes an accurate library of thermodynamic routines for both steam (the working fluid), as well as combustion and fuel gas routines for calculating the economics of the operation of these systems. The JSteam library has been introduced, which provides a new, high-speed, high-accuracy modelling engine for estimating the design performance of a suite of common utility models, including turbines, boilers, gas turbines and other standard equipment. It has been shown that while some of these models are well known thermodynamic relationships, incorporating knowledge of how these models are used in industry has meant they are now more suited to predicting industrial conditions, as well as suitable for use in optimization, given the emphasis on computational efficiency. A case-study showing the use of the modelling package used for predicting the response of a combined cycle power-plant showed a good approximation, within 5% across a range of equipment operating conditions.

In addition to the design models, four new part-load models of critical utility system equipment have been developed. These include a back-pressure steam turbine, steam boiler, gas turbine and heat recovery steam generator, all key components when predicting the economic impact of varying the operating point of a utility system. Each model combines knowledge from industry, insight from the thermodynamics and regressed performance curves from literature, leading to a set of flexible, wide-range models suitable for use in operational optimization of a utility system. Together, these four models are a significant contribution to the field of utility system modelling, specifically when looking at practical off-design analysis.

The following chapter will exploit the structure and underlying relationships of both the full-load and part-load models developed here, and tailor an optimization solver to best solve the operational optimization of steam utility systems.

Chapter 6

Utility System Optimization

This chapter follows on from the modelling presented in the proceeding one, and presents a practical framework for operational optimization of industrial utility systems. As identified in the introduction, a steam utility system is a large, interconnected, multi-faceted optimization problem, and as will be shown in this chapter, is nonlinear, non-convex and contains discrete constraints. This renders the optimization problem as one of the hardest known to solve, a non-convex MINLP (the relaxed problem is non-convex, not just due to the integer constraints), a problem which is currently known to be $\mathcal{NP}$-hard. However using the OPTI Toolbox, a new framework developed in this work, globally optimal results to the operational optimization of these systems will be shown to be achieved in less than 5 seconds for a range of industrially significant case studies, all using a standard laptop computer.

The chapter begins by highlighting a deficiency in the optimization solvers currently available via MATLAB, namely the omission of a free rigorous mixed integer nonlinear programming solver. This leads into the development of our own optimization framework for MATLAB, OPTI, which combines a suite of optimization solvers from the open-source community with the 'glue' via a library of MATLAB MEX files and an object-orientated approach to building and solving optimization problems. The OPTI framework not only enables MINLPs to be solved, but linear, quadratic and general nonlinear problems as well, automatically identifying the problem entered and tailoring the problem for the selected solver. Using OPTI, it is shown that simply wrapping a nonlinear optimizer around a simulation model obtains poor results, while exploiting the structure of the model via an algebraic description can result in significant performance enhancements. To exploit the structure of these models, SymBuilder, a new modelling package developed in this work, is used to construct algebraic models of three utility system case studies, including one real petrochemical system, which are then optimized using OPTI.

6.1 Introduction

Following the core focus of practical industrial optimization in this thesis, this chapter deals with the operational optimization of industrial utility systems. By employing the suite of models developed in the last chapter, together with the JSteam package for thermodynamic calculations, it will be shown that a purpose-built utility system optimization model can be built, validated against a rigorous thermodynamic model, then optimized in less than five seconds to minimize operating costs.

In order to complete this aim, a suitable optimization environment was to be chosen. One of the primary requirements was the ability to investigate mixed integer optimization, both linear and nonlinear, because the operational optimization problem requires multiple binary variables. Given the rapid prototyping ability of MATLAB, together with the suite of tools it provides built in, this was an obvious first choice. It is however *very* limited in its mixed integer optimization options: `bintprog` [207] solves BILPs, and the genetic algorithm solver [204] can solve inequality constrained MINLPs, however its performance is very problem dependent.

In addition, as we wanted to utilize JSteam for constructing approximated models, the environment had to support a method for interfacing to external code. Comparing packages such as GAMS and AMPL, this requirement was clearly going to be difficult. It would also most likely require another software platform to be used to interface to the code, then generate a GAMS/AMPL model for optimization (such as done with the software STEAM [45] detailed in [46]).

Therefore to keep development within a single environment, MATLAB was retained, but it was clear a lot of development would be required to robustly optimize mixed integer utility systems. The next section of this chapter details the development of the OPTI Toolbox, a framework we developed initially for solving MINLPs resulting from this work, but soon grew to a toolbox used by several thousand international users. Succeeding sections detail the modelling and optimization methodology employed, including development of our own algebraic modelling system, then concluding with 3 case studies to demonstrate the effectiveness of the proposed approach.

6.2 OPTI Toolbox

The OPTimization Interface (OPTI) Toolbox (`http://www.i2c2.aut.ac.nz/Wiki/OPTI/`) is a free, open source MATLAB Toolbox developed as part of this work. It is currently used by several thousand people internationally (5000 as of January

2014), ranging from academics and students, to engineers, scientists, economists and a multitude of other users (we track downloads and maintain a full database of users). In addition it has undergone substantial development over the course of the past two years, and is still in active development. This section will detail its development and functionality, and provides examples of how it is used within this work.

6.2.1 Toolbox Conception

Midway through 2011 it was decided that MATLAB's inability to handle integer constraints within nonlinear optimization problems was becoming a hurdle to this research, and we were going to either have to switch development environments, use alternative 3rd party software, or develop something ourselves. Given the limited budget for this work, most commercial platforms were out of range (such as GAMS and AMPL) but we were able to evaluate software for fixed time periods.

A piece of software that appeared to meet all (initial) requirements was TOMLAB [130], a commercial optimization platform for MATLAB that provided a suite of optimization solvers with interfaces to MATLAB. In addition, it provided an advanced automatic differentiation tool that could automatically generate first and second derivatives for simple to moderate complexity MATLAB functions, as well as a suite of utility functions. The concept of the platform was quite attractive because it provided a suite of C/C++/Fortran solvers already compiled and ready to use, we did however find the interface somewhat difficult to use, especially when trying to swap between linear, quadratic and nonlinear problems. This, coupled with a high cost for the TOMLAB, and additional costs for each solver meant this tool was not going to be possible to use. It however prompted thinking about what was freely available in the open source community, which turned out to be highly active.

An initial survey of the open source optimization landscape showed there was a large amount of activity, with key players such as COmputational INfrastructure for Operations Research (COIN-OR) [15], a project to develop open source for the operations research community, NonLinear Optimization (NLopt) [157], a suite of C and C++ optimization solvers converted and coded by Steven Johnson, as well as Python based frameworks such as OpenOpt [175] and pyOpt [246] just to name a few. A key finding of this survey was that open source solvers were predominantly written for Linux/Unix based compilation (i.e. using makefiles and auto-tools), with many stating Windows compatibility was either untested, unavailable or not compatible with 64bit Windows. Furthermore, solver interfaces to high-level languages such as MATLAB were typically either immature or non-existent, with Python being the preferred high-level interface language (due most likely to its open source nature).

Given that we were using Windows and MATLAB and wanted to leverage the large amount of work that existed within the open source OR community, it was decided to start our own project. It was to be designed in a way to easily convert between linear and nonlinear problems, as was common with the utility system work we were doing, as well as supplying solvers from the open source community with MATLAB interfaces. However this task would require compiling the solvers from scratch, writing our own MATLAB interfaces (or fixing up existing ones, where applicable) and integrating it all into a framework, much of which was very time consuming with little to no research benefit. The upside was that the framework would hopefully be useful to other researchers facing the same challenges as ourselves, and we planned to release it for no cost once it reached maturity. After 2 months of development, OPTI Toolbox (typically referred to as OPTI) was released via our research centre website in late August 2011.

6.2.2 Optimization Solvers

The original plan was to compile and interface solvers that were required as part of this research. However the toolbox quickly grew and now includes 24 solvers compiled and supplied with the toolbox. The author (or manager) of each solver has been individually contacted and agreed to the distribution of their work in binary form within OPTI, provided the licence terms were met. For a complete list of solvers supplied with the toolbox, together with problem examples, see Section C.1.1.

6.2.3 Problem Identification and Construction

As identified earlier, one of the main design ideas behind the toolbox was to be able to easily create linear and nonlinear optimization problems , including the addition of integer constraints, without having to rewrite the problem construction. The concept was to create a single object that could automatically identify the problem entered, and then choose the best solver available. This was a variation on the GAMS / AMPL approach where an engine is instructed to solve the optimization problem using a specific solver given the problem type, or the TOMLAB approach where the problem is entered given a specific category.

To develop this functionality, an object orientated approach has been taken, where the user creates their specific problem of interest using the OPTI class constructor

```
OptProb = opti('obj',my_objective,'bounds',lb,ub,'x0',x0)
```

The single line above creates a bounded nonlinear optimization problem, identified only by the arguments supplied. In addition, it automatically checks all user arguments for errors, as well as problem formulation problems, and then, based on the information provided, decides on the best solver available to solve the problem. To solve this problem, a user can simply call solve on the OPTI object

```
[x,fval,exitflag,info] = solve(OptProb)
```

noting the method returns the optimal decision variable vector (`x`), the objective value at this optimum (`fval`), the reason why the solver exited (`exitflag`) and information on the solver run (`info`).

To demonstrate the functionality of the object without listing every possible option, a number of illustrative examples are presented below. For examples of solving other problem types using OPTI, please see the Wiki, available at `http://www.i2c2.aut.ac.nz/Wiki/OPTI/`.

6.2.3.1 Linear Programming

Consider the following small linear program

$$\begin{aligned}
\min_{\mathbf{x}} \quad & -6x_1 - 5x_2 \\
\text{subject to:} \quad & x_1 + 4x_2 \leq 16 \\
& 6x_1 + 4x_2 \leq 28 \\
& 2x_1 - 5x_2 \leq 6 \\
& 0 \leq \mathbf{x} \leq 10
\end{aligned}$$

The problem can be entered into a script file and an OPTI object created using

```
% Objective (min f'*x)
f = -[6 5]';
% Linear Constraints (A*x <= b)
A = [1,4; 6,4; 2, -5];
b = [16;28;6];
% Bounds (lb <= x <= ub)
lb = [0;0]; ub = [10;10];

% Build OPTI Object
OptProb = opti('grad',f,'ineq',A,b,'bounds',lb,ub)
```

In as little as 6ms, the OPTI constructor has:

1. Based on the arguments supplied, determined we are solving an LP.
2. Ensured all arguments are valid, including checking for invalid numbers or missing argument pairs.
3. Determined that CLP is the best available LP solver on this PC for this problem, and converted the user's problem to suit CLP (note CLP uses row/range constraints, as shown below).
4. Set up the problem and all required intermediate data for immediate calling of CLP.

As we did not suppress command output, OPTI prints the problem description to the command window, as shown below

```
Linear Program (LP) Optimization
 min  f'x
 s.t. rl <= Ax <= ru
      lb <= x <= ub
------------------------------------------------------
   Problem Properties:
# Decision Variables:        2
# Constraints:               7
  # Linear Inequality:       3
  # Bounds:                  4
------------------------------------------------------
  Solver Parameters:
Solver:                     CLP
```

To solve the problem the solve method is called

```
>> [x,fval,exitflag,info] = solve(OptProb)
```

which CLP solves in less than 1ms for the correct optimum of -31.4. In addition to solving the problem with OPTI, we can also plot the solution together with the objective and constraints using the overloaded `plot` method

```
>> plot(OptProb)
```

where as shown in Figure 6.1, the linear objective is shown as the dashed plane, and each linear inequality constraint and bound represents one of the shaded areas and lines. The optimum is shown as the red dot.

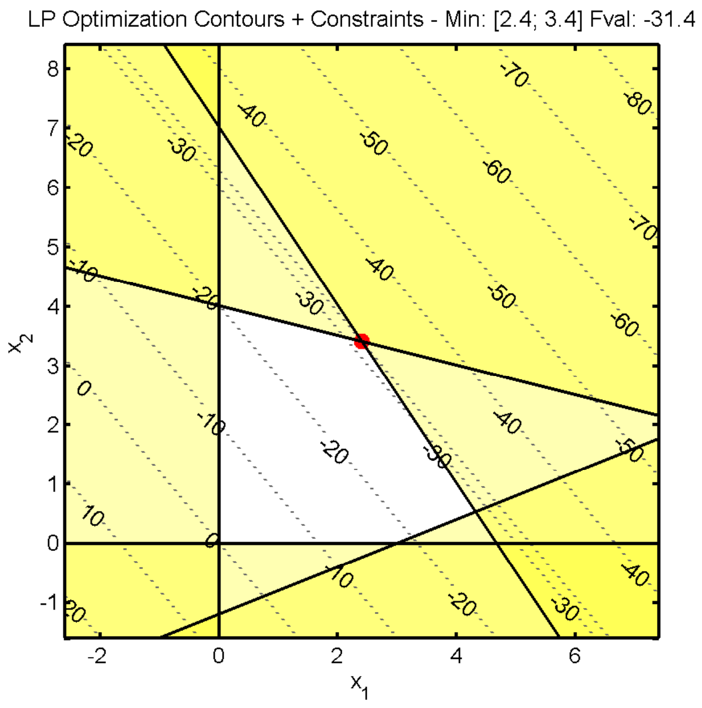


Figure 6.1: Example LP plot automatically generated by OPTI. The dashed lines are the contours of the linear objective, while infeasible areas are shown highlighted in yellow. Rectangular constraints are due to variable bounds, while the linear inequality constraints are (in this problem) the slanted lines.

6.2.3.2 Quadratic Programming

Consider the following non-convex quadratically constrained quadratic program

$$\begin{aligned}
\min_{\mathbf{x}}\ & 0.5x_1^2 + 0.5x_2^2 - 2x_1 - 2x_2 \\
\text{subject to: }& 3 \leq x_1^2 + x_2^2 - 2x_2 \leq 5 \\
& x_1^2 + x_2^2 - 2x_1 + 2x_2 = 1 \\
& -x_1 + x_2 \leq 2 \\
& x_1 + 3x_2 \leq 5 \\
& 0 \leq \mathbf{x}
\end{aligned}$$

This problem is particularly challenging because it contains both a non-convex quadratic constraint, as well as a quadratic equality, which is also non-convex. This is entered into MATLAB as following the code below:

```
% Objective (min 0.5*x'*H*x + f'*x)
H = eye(2);
f = -[2 2]';
% Quadratic Constraints (qrl <= x'*Q*x + l'*x <= qru)
```

```
Q = {[1 0; 0 1]
     [1 0; 0 1]};
l = {[0;-2]; [-2;2]};
qrl = {3; 1};   %QC1 is double sided, QC2 is an equality
qru = {5; 1};
% Linear Constraints (A*x <= b)
A = [-1,1; 1,3];
b = [2;5];
% Bounds (lb <= x)
lb = [0;0];

% Build OPTI Object
OptProb = opti('qp',H,f,'qcrow',Q,l,qrl,qru,'ineq',A,b,'lb',lb);
```

With the argument supplied, OPTI recognises this problem as a QCQP and will inspect both the objective and quadratic constraints for convexity. If it detects a non-convex term OPTI will automatically use a non-convex solver. In this case SCIP is chosen because the problem is non-convex. Note this functionality only exists for quadratic problems because detecting non-convexity of general nonlinear problems is very computationally expensive. The solution found by SCIP is shown by the OPTI generated plot in Figure 6.2, which looks optimal.

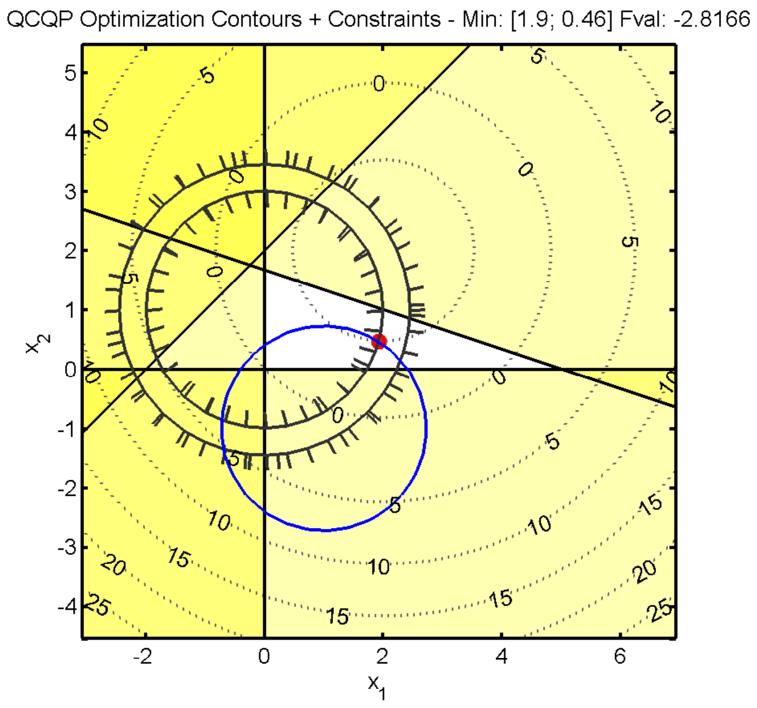


Figure 6.2: Example QCQP plot automatically generated by OPTI. As in the previous plot, the dashed curves indicate the objective function contours, in this case the underlying quadratic function. The annulus is the result of the double sided quadratic inequality constraint, with hatches indicating the infeasible side. The blue circle is the quadratic equality constraint, noting the solution (red dot) lies on this circle, within the annulus and linear constraints, and at the minimum.

To illustrate the ability to switch to a general nonlinear solver, the user can specify the solver as an option

```
% Set OPTI Options
opts = optiset('solver','ipopt');
% Rebuild OPTI Object
OptProb = opti('qp',H,f,'qcrow',Q,l,qrl,qru,'ineq',A,b,'lb',lb,...
               'x0',[2;0],'options',opts);
% Re-Solve
x = solve(OptProb)
```

noting the NLP solver IPOPT has been chosen by the user, and passed to the OPTI constructor. The toolbox will then automatically convert the QCQP to a NLP, including generating both first and second derivatives and their sparsity patterns, and will set problem specific options within IPOPT if applicable to the problem being solved. This functionality enables a user to enter the problem in a problem specific format, yet solve the problem using a range of different solvers without having to account for solver specific conversions.

6.2.3.3 Nonlinear Programming

Consider the following nonlinear program

$$\begin{aligned} \min_{\mathbf{x}} \;& \log(1+x_1^2) - x_2 \\ \text{subject to: } & \left(1+x_1^2\right)^2 + x_2^2 = 4 \end{aligned}$$

As this is a nonlinear problem, the problem is specified as a collection of MATLAB functions

```
% Objective (min f(x))
fun = @(x) log(1 + x(1)^2) - x(2);
% Nonlinear Constraint
nlcon @(x) (1 + x(1)^2)^2 + x(2)^2;
nlrhs = 4;
nle = 0; % 0 indicates an equality

% Build OPTI Object
OptProb = opti('obj',fun,'nlmix',nlcon,nlrhs,nle,'x0',[2;2])
```

If we inspect the resulting object

```
Nonlinear Program (NLP) Optimization
 min  f(x)
 s.t. cl <= c(x) <= cu
```

```
------------------------------------------------------
   Problem Properties:
# Decision Variables:        2
# Constraints:               1
  # Nonlinear Equality:      1
------------------------------------------------------
  Solver Parameters:
Solver:                     IPOPT
Objective Gradient:         @(x)mklJac(prob.fun,x,1) [numdiff]
Constraint Jacobian:        @(x)mklJac(prob.nlcon,x,nnl) [numdiff]
Jacobian Structure:         Not Supplied
Lagrangian Hessian:         Not Supplied
Hessian Structure:          Not Supplied
```

we see the OPTI has chosen to use IPOPT, and approximated the required derivatives using a finite difference routine (`mklJac`). This allows the user to quickly attempt to solve this problem. Note however that for larger problems, this will most likely cause convergence issues. OPTI provides a suite of routines for calculating the derivatives, described further on in Section 6.2.4 which can be used to overcome this issue, or alternatively the user can supply routines to calculate the derivatives, or to specify a derivative-free solver. Note that even with approximated derivatives IPOPT solves this problem in less than 8ms to the correct minimum of -1.7321.

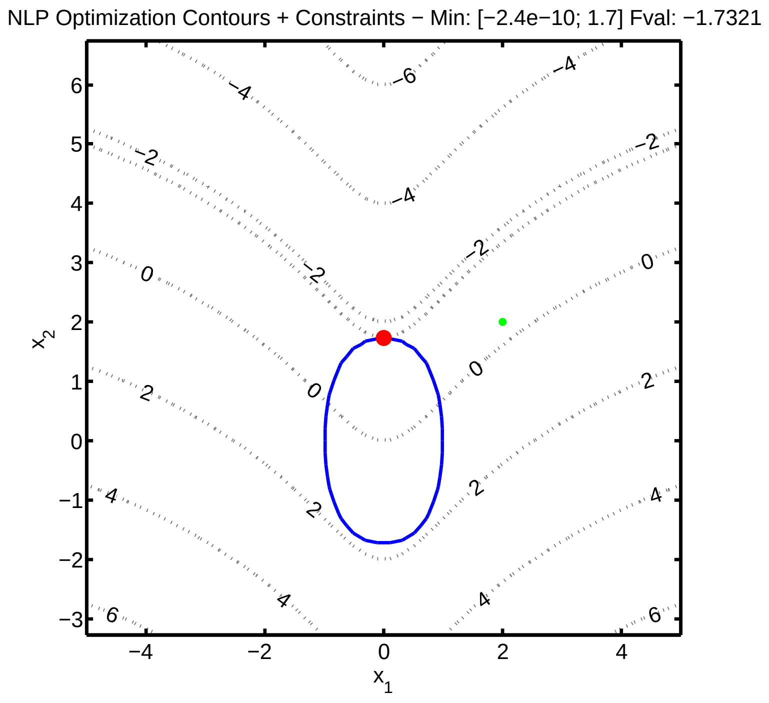


Figure 6.3: Example NLP plot automatically generated by OPTI. The green dot is the initial solution guess, while the blue closed curve is the nonlinear equality constraint.

For solvers such as SCIP, OPTI will automatically convert the blackbox nonlinear functions into an algebraic description of the problem. This functionality is described in Chapter 7.

6.2.4 Obtaining Accurate Derivatives

As identified, solving nonlinear optimization problems, which is the focus of this work, requires generally at least first derivatives for both the objective and constraints, and for challenging problems, second derivatives. The accuracy of these derivatives directly affects both the accuracy of the solution obtained, and also in many cases, the solution time required. It therefore becomes obvious why restricted algebraic modelling programs such as GAMS and AMPL are so attractive, given their built-in functionality for generating both first and second derivatives using tools such as automatic differentiation.

Given MATLAB is not an algebraic modelling language, one of its downfalls is therefore the inability to provide robust derivatives of general MATLAB code. To combat this, OPTI provides several tools for obtaining both accurate and estimates of derivatives.

Centred Finite Difference (`mklJac`) Finite difference is one of the most common methods for obtaining derivatives and is the default method used with OPTI. Its particular advantage is that it will work for any smooth function, regardless of whether the function is purely MATLAB based or calls external functions (such as thermodynamic routines in JSteam). The downside though is that finite difference only provides an estimate of the derivatives, typically only down to around 10^{-7} before the routine becomes too computationally expensive to run or runs into numerical issues (i.e. trying to find the difference between two very big numbers, also known as subtractive roundoff). As this value hovers around the convergence tolerance of most optimizers, it can cause convergence issues. OPTI implements the Intel MKL routine `djacobi` [148] which in turn implements centred finite difference, and this provides the most accurate estimate of the finite difference algorithms. Calculating the Hessian via finite difference is particularly inaccurate and is not supported in OPTI.

Automatic Differentiation (`autoJac`) Automatic differentiation is the standard method for calculating derivatives among commercial software given it provides derivatives to numerical precision and can be quite efficient. In order to do this, automatic differentiation uses the object-orientated principles of function and operator overloading (within this work, we are not using source-code transformation) over the original function, and replaces each original numerical calculation with its corresponding derivative. The chain rule is then used to combine each calculation, resulting in required derivative(s), as described in [233]. As with each technique, automatic differentiation has a downside which is that only functions that have been overloaded with their respective derivatives can be used. Therefore calling external functions, or even Simulink simulations, means derivatives cannot be calculated. Within OPTI the open source package `adiff` [216] is used to provide basic forward mode automatic differentiation. It is however, quite limited with respect to both available functions, as well as being reasonably slow.

Complex Step Differentiation (`cstepJac,cstepHess`) A surprisingly simple yet incredibly powerful approach to obtaining first derivatives is by using complex step differentiation [191], a technique which uses the step size from finite difference as a complex component of the evaluation point (the point at which the derivative is calculated at). A remarkable feature of this method allows the complex step method to obtain first derivatives to numerical precision, virtually independently of the step size (once below around 10^{-8}, function dependent). The catch is that the language the complex step differentiation is implemented in must support complex numbers natively. For MATLAB this is simple, given its Fortran roots, however there are complications with any simple functions such as `abs` and relationship operators, as well as the use of complex conjugate (`'` in MATLAB) and transpose (`.'`), noting most users incorrectly use `'`. In addition, the increased accuracy does not cross over to second derivatives, although heuristics have been proposed. OPTI supplies complex step routines for calculating both first and second derivatives, using second derivative heuristics described in [181].

Symbolic Differentiation (`symJac,symHess`) The final method implemented by OPTI utilizes the MATLAB Symbolic Toolbox [209] to symbolically differentiate the user's objective and constraints to generate both first and second derivative expressions. The advantage is that this process is done once only before the optimization begins, and then the expressions can be used for any number of optimization studies, as opposed to calculating the derivatives at each evaluation point. Once again, a downside exists, and in this case the Symbolic Toolbox does not natively differentiate MATLAB functions; they must be written using Symbolic variables instead. OPTI overcomes this issue by providing a very simple parsing system for building Symbolic expressions from simple MATLAB functions, provided a set of variable naming rules is used, and without using vectorized code (a large drawback of this method). Another issue is that the analytical expressions for derivatives, especially second derivatives, can be very complex and in some cases take longer to execute than alternative approaches such as automatic differentiation. OPTI provides two simple routines for symbolically differentiating MATLAB anonymous functions for both first and second derivatives.

6.2.5 Toolbox Summary

By utilizing the OPTI Toolbox together with JSteam, the operational optimization of any typical utility system model can be performed. For linear and quadratic problems, the toolbox will automatically handle all data manipulation to supply the problem to a compatible solver, while for nonlinear problems, derivative generation (if required) and option tuning is automatically performed. In addition, including binary and integer constraints within the model is a single argument, allowing both continuous and discrete optimization studies to be performed. Furthermore, auxiliary solvers such as nonlinear root solvers can be used for converging utility flowsheet models, as described in the next section.

The following sections will detail the methodology used for modelling and optimizing utility systems using MATLAB, OPTI and JSteam.

6.3 Optimization Methodology

6.3.1 Optimization of Process Flow Diagrams

Once a model of a system to be optimized has been completed within a flowsheet-based process simulator (such as HYSYS, JSteam, etc), it would be natural to assume that this model would also be suitable for optimization. This could be based on the process simulator model being of high fidelity, so that it represents an accurate view of the real system. In addition, given that the user has already invested considerable time modelling the system, it would make sense to exploit the model for optimization studies. As shown in this section, optimizing a *simulation model* is typically the slowest and least robust method, as opposed to a purpose built *optimization model* [70].

To provide a benchmark, the test utility system shown in Figure 6.4 will be used. The system includes three variable efficiency boilers of different sizes as well as two variable efficiency turbo generators connected between different headers. In addition, two back pressure turbines with fixed loads and redundant electric motors are connected between headers, as well as a condensate collection system with LP flash to recover useful steam. The three headers are at 40, 11 and 4 bar for the HP, MP and LP headers respectively. Table 6.1 lists the operating specifications of each unit operation.

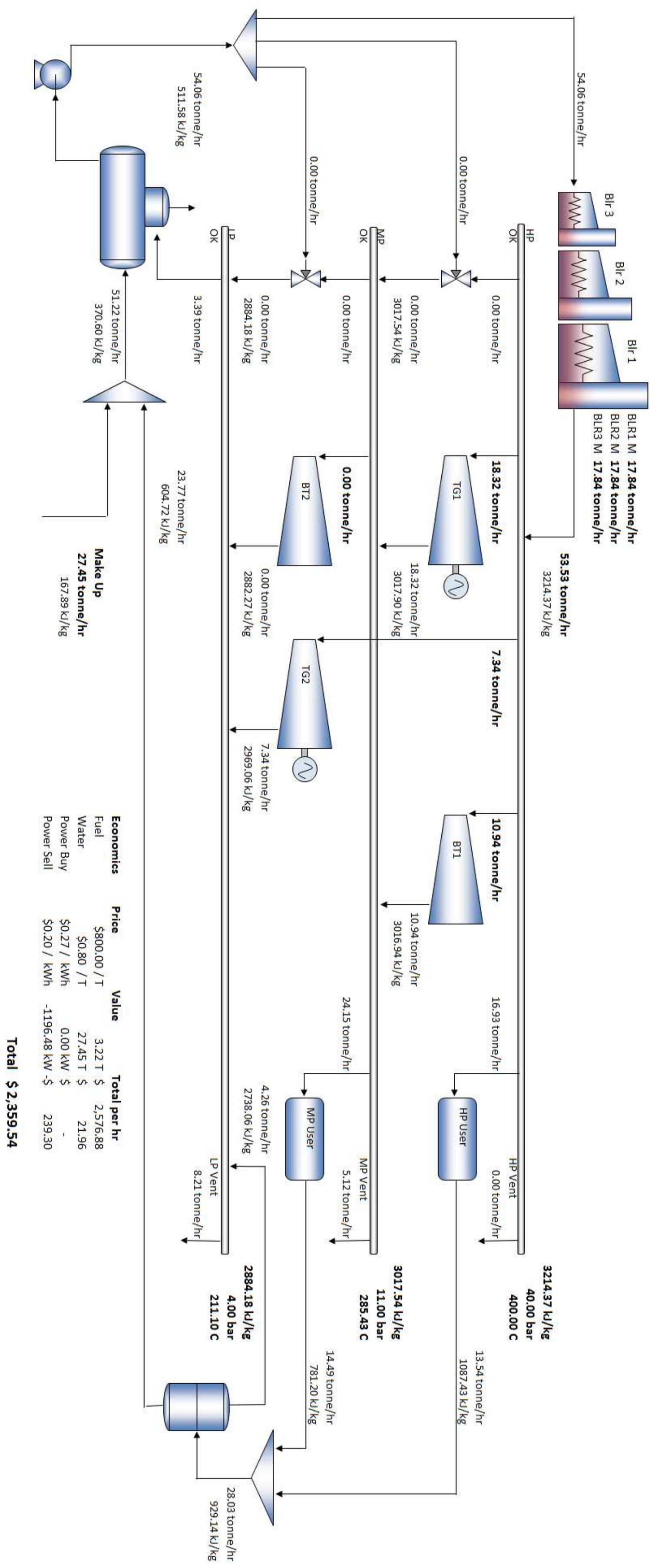
Table 6.1: Hypothetical 3 header model equipment operating conditions.

Boiler 1 (BLR1)	20-60 tonne/hr of 400°C steam at 40 bar. 200°C minimum stack temperature and 1% minimum mole fraction of stack O_2. Run on 100% 30°C Methane with 30°C ambient air.
Boiler 2 (BLR2)	10-40 tonne/hr, remainder of specifications identical to Boiler 1.
Boiler 3 (BLR3)	5-20 tonne/hr, remainder of specifications identical to Boiler 1.
Turbo Generator 1 (TG1)	2000kW maximum.
Turbo Generator 2 (TG2)	2000kW maximum.
Back Pressure Turbine 1 (BT1)	600kW rated, 60% isentropic efficiency, with backup electric motor.
Back Pressure Turbine 2 (BT2)	200kW rated, 60% isentropic efficiency, with backup electric motor.
BFW Pump (PMP)	47 bar outlet pressure, 70% isentropic efficiency.
HP Steam User (HPU)	10MW duty based user, returns 80% of steam as saturated condensate.
MP Steam User (MPU)	15MW duty based user, returns 60% of steam as saturated condensate.
Deaerator	2 bar with 0.01 continuous vent ratio.
Make Up Water	4 bar, 40°C.

In order to optimize this system, the utility model must be modelled again within MATLAB. This is primarily due to the JSteam Excel Add-In not supporting a second optimization layer over the nonlinear root-solver used to solve the system recycle. However by using MATLAB we are also able to use OPTI and the solvers supplied with it, so that a more detailed optimization study can be performed. An example snippet of the resulting model in MATLAB is shown below

```
% MP-LP Desuperheater
LP_T = JStm.TPH(LP_P,LP_H);
[~,MPDsp_WM,MPDsp_SM,MPDsp_H] = JStm.UnitOp_DesuperheaterMout(MP_H,BFW_H,...
                                LP_P,LP_T,LP_Feed);
% Update LP H & Condensate H
LP_H = (MPDsp_H*LP_Feed+BT2_H*BT2_M+TG2_H*TG2_M+FVapH*FVapM)/(LP_Feed+...
        BT2_M+TG2_M+FVapM);
Con_H = (FLiqH*FLiqM+(Drtr_CondM-FLiqM)*MU_H)/Drtr_CondM;
% MP Fixed Inputs
[~,BT1_M,BT1_H] = JStm.UnitOp_Turbine1Q(HP_H,HP_P,MP_P,BT1_Eff,BT1_Q*BT1);
[~,TG1_Q,TG1_H] = JStm.UnitOp_Turbine1M(HP_H,HP_P,MP_P,TG1_Eff(TG1_M),TG1_M);
```

noting that the model is not particularly legible nor easy to create. It is written in a 'Sequential Modular' form [35, 285] wherein the solving order of the unit operations



Economics	Price	Value	Total per hr
Fuel	$800.00 / T	3.22 T	$ 2,576.88
Water	$0.80 / T	27.45 T	$ 21.96
Power Buy	$0.27 / kWh	0.00 kW	$ -
Power Sell	$0.20 / kWh	-1196.48 kW	-$ 239.30

Total $ 2,359.54

Figure 6.4: Example 3 header utility system simulation model.

is fixed by the sequential flow of process material. However once the model is created and converged, it should return an identical solution to the JSteam Excel model, given that they both rely on the same modelling strategy.

To converge a sequential modular model that includes recycle loops (of which multiple loops are present in a utility system with a closed energy balance, as discussed in [71]), the model must be successively iterated until convergence criteria has been met. This problem is posed mathematically as a system of nonlinear equations (or multivariable root solving), and is solved in Excel using the built-in Excel Solver. However for the MATLAB model, a nonlinear root-solver must be implemented deliberately by the user to solve each of the system recycles. For a typical utility system the enthalpy of each header must be solved, as well as the enthalpy of the condensate entering the deaerator and total mass flow of boiler feed water. These variables were identified within our work constructing iUO and are described further in [71]. For the example system, these 5 variables are solved in MATLAB using either `fsolve` or an OPTI supplied nonlinear root solver, as detailed in Section C.1.1.4.

An issue common to all sequential and non-sequential simulation packages with a recycle solver (as opposed to equation based simulators such as gPROMS [257] and others) is the availability of gradients, which, for a typical classical multi-variable root solver, is a requirement. Most packages utilize finite-difference to obtain an estimate of the derivatives, because the complex thermodynamics and interior model iterations which would make deriving analytical derivatives virtually impossible. However, while finite-difference is a simple method in practice, it can be an expensive exercise.

To provide an example, converging the example 3 header utility system from Figure 6.4 using the Intel MKL nonlinear equation solver [148] requires just two solver iterations, but this requires four function evaluations and two gradient evaluations. Breaking these numbers down further, the model is actually called 24 times which indicates each gradient evaluation requires 10 model calls. This can be very computationally expensive for larger models, especially considering we are only obtaining an estimate of the actual derivatives.

6.3.1.1 Gradient-Based Optimizer Pitfalls

In practice, converging a simulation model with recycle is not an expensive exercise, given the 3 header system only takes 30ms to solve on a standard laptop. The problem is that solving the system recycle does not optimize the system, it only converges the model to a physically realisable point. In order to find an operating point that improves the operational expenditure of the plant, the optimizer will need to converge the recycle loops multiple times at each evaluation point, as chosen by the optimizer. Furthermore, as we cannot obtain analytical gradients of either the objective or constraints, we are once again left with finite difference approximations. This last step is the source of the major disadvantage of using a gradient-based optimizer with a simulation model that involves recycles; gradients of the optimizer objective and constraints (i.e. the gradient and Jacobian functions) will use a finite difference approxmation of the results of a multivariable root solver, which also uses a finite-difference approximation for its solution. Figure 6.5 shows the situation in a graphical form.

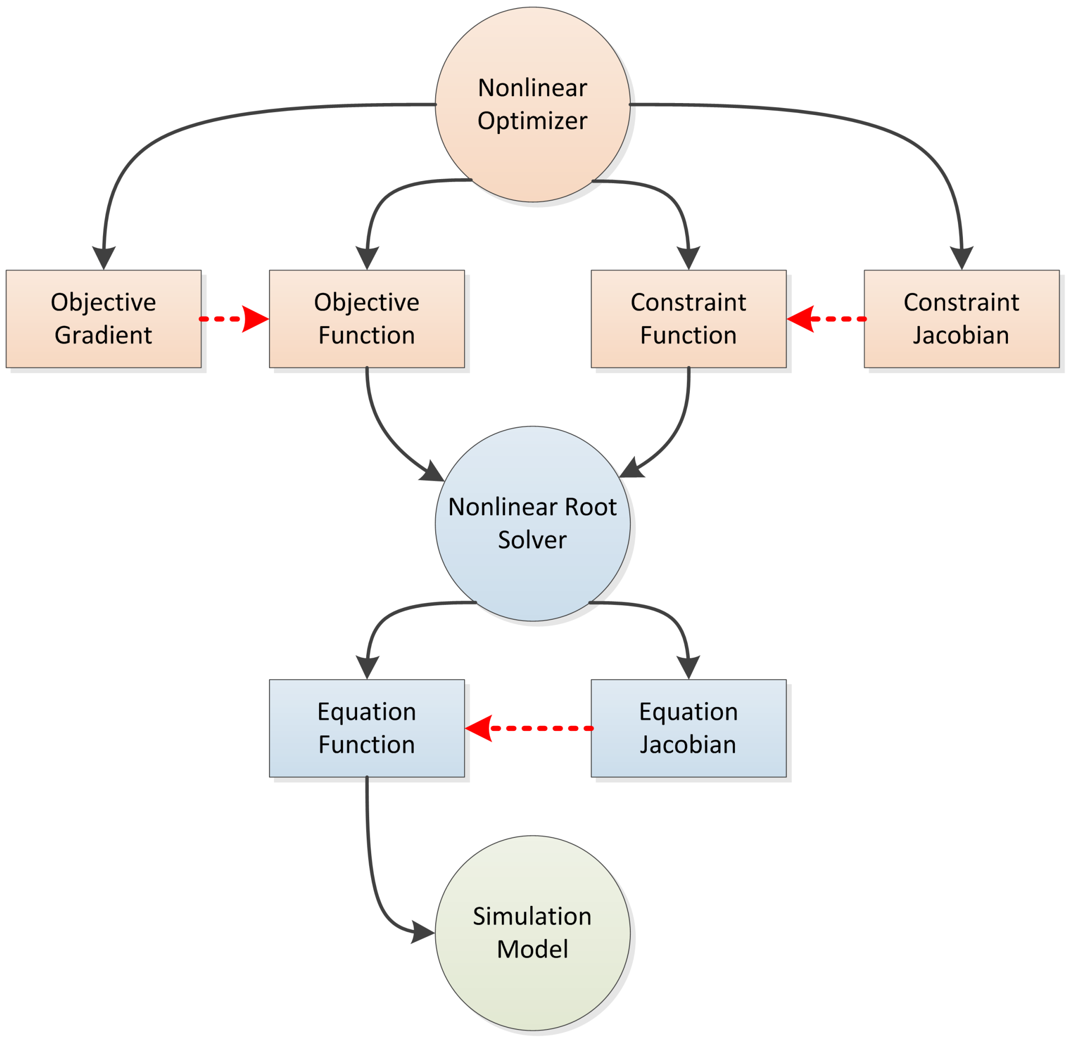


Figure 6.5: Gradient-based nonlinear optimization of a simulation model with recycle. Dashed arrows indicate derivatives obtained via finite difference, while all arrows indicate the calling hierarchy of each function.

Note that this situation requires the simulation model be evaluated for *both* the optimization objective and constraints. This requirement is based on the implementation of equipment constraints such as maximum power output of a turbo-generator, which requires the model to converge in order to determine the power output (due to for example, varying header enthalpies) and must therefore be modelled as general nonlinear constraints.

The result of this formulation strategy is that not only does the simulation model get called many more times and thus the optimization process takes longer, but also due to inaccuracies within the method of obtaining derivatives, the optimizer is less likely to converge to the optimal solution, or in some cases, to any solution at all.

6.3.2 Simulator Model Optimization Case-Study

To illustrate the issues described so far, this subsection will attempt to optimize the base case of the utility model in Figure 6.4. The objective function we will minimize is

$$\mathcal{J} = C_{\text{fuel}} \sum_{n=1}^{3} M_{\text{fuel,Boiler}_n} + C_{\text{water}} M_{\text{water}} - C_{\text{elec}} W_{\text{total}} \tag{6.1}$$

where C represents a cost in dollars. The electricity cost is defined as

$$C_{\text{elec}} = \begin{cases} C_{\text{sell}} & \text{if } W_{\text{total}} \geq 0 \\ C_{\text{buy}} & \text{if } W_{\text{total}} < 0 \end{cases} \tag{6.2}$$

which models different electricity prices for buying and selling. The power balance of the system is given by

$$W_{\text{total}} = \sum_{n=1}^{2} W_{\text{turbogen}_n} - \sum_{n=1}^{2} W_{\text{bpt}_n} (1 - b_n) - W_{\text{pump}} \tag{6.3}$$

where W_{turbogen_n} is the output shaftwork of turbo generator n, W_{bpt_n} is the shaftwork requirement of back pressure turbine n and where b_n indicates whether the turbine is connected ($b = 1$) or a redundant electric motor is used ($b = 0$).

In order to keep the system within operating limits, 6 nonlinear inequality constraints are implemented

$$\begin{aligned} W_{\text{turbogen}_1} &\leq 2000\text{kW} \\ W_{\text{turbogen}_2} &\leq 2000\text{kW} \\ M_{\text{steam,Boiler}_1} &\leq 60 \text{ tonne/hr} \\ M_{\text{steam,Boiler}_2} &\leq 40 \text{ tonne/hr} \\ M_{\text{steam,Boiler}_3} &\leq 20 \text{ tonne/hr} \\ M_{\text{steam,HPvent}} &\geq 0 \text{ tonne/hr} \end{aligned} \tag{6.4}$$

where the only non-intuitive constraint is the HP vent constraint (the final constraint), which is required to ensure steam is not supplied via the vent to the header, but rather supplied by the boilers. As discussed in the last subsection, both the objective function and the constraints require the simulator model to be evaluated in order to obtain function outputs. By supplying a single `mode` argument to the simulator function, the function can be instructed to act as an objective and calculate the cost, or constraints, and calculate the violations.

The final design choice for this optimization run is the selection of decision variables. Table 6.2 lists the variables for this study together with their bounds where the first two variables are the mass flows through the turbo generators, the

Table 6.2: Simulation model optimization variables.

	Decision Variable	Model Variable	Lower Bound	Upper Bound
Continuous	x_1	M_{turbogen_1}	0	31.37 tonne/hr
	x_2	M_{turbogen_2}	0	19.55 tonne/hr
	x_3	F_{Boiler_1}	0	1
	x_4	F_{Boiler_2}	0	1
	x_5	F_{Boiler_3}	0	1
Binary	x_6	b_{bpt_1}	0	1
	x_7	b_{bpt_2}	0	1

second three are the fraction of the steam demand supplied by each boiler and the last two are binary variables which select whether the back pressure turbine is connected. Given that this problem has only 7 decision variables (of which 5 will be optimized as initially the binary variables will be fixed) and 6 constraints, it could be expected to be quite simple. However, referencing back to the complexities in Figure 6.5, the solution times will demonstrate this complexity.

Due to initially using a continuous optimizer, the binary variables associated with the back pressure turbines will be fixed as $b_{\text{bpt}_1} = 1$, $b_{\text{bpt}_2} = 0$. Therefore the optimization problem is to decide what is the optimal allocation of steam between the boilers, and how much electricity should be generated from each turbogenerator, in order to minimize operating cost. The initial condition for the optimization is taken as the base-case operating point, as described in Figure 6.4. Solution times for a range of solvers are shown below in Table 6.3. Details of the solvers used can be found in Section C.1.1.6, except for `fmincon` which is a commercial general nonlinear solver supplied with the MATLAB Optimization Toolbox.

Each solver obtained the same solution where turbo generator 1 was decreased to 595.1kW, turbo generator 2 switched off and the boilers split with 21.7, 11.77 and 7.42 tonne/hr of steam generated for boilers 1, 2 and 3 respectively. However the

Table 6.3: Continuous simulation model optimization results (gradient based solvers).

Solver	Cost ($/hr)	Time	Iters	Obj Evals	Grad Evals	Sim Evals
IPOPT	$1937.74	26.3s	29	109	30	29497
FILTERSD	$1937.74	33.2s	2	82	43	37488
`fmincon`	$1937.74	35.8s	15	36	-	39162

solution times between solvers varied, but more importantly were of a significant magnitude given that this is a *tiny* nonlinear program. However due to the large number of simulation evaluations, it is not surprising these times were recorded. As reported by IPOPT, over 97% was spent within function evaluations, leaving the small fraction remaining within the solver itself. It is results like these that validate the considerable work that went into optimizing the speed of the underlying JSteam thermodynamic routines, as described earlier in Section 5.3.1.1.

Taking an alternative approach and using a derivative-free solver, one can avoid the need to calculate the derivatives of both the objective and constraints using finite-difference. Using the OPTI Toolbox we are free to try one of the compatible solvers without any changes to the optimization problem formulation. Results from three derivative-free solvers are presented in Table 6.4, noting `patternsearch` is another commercial algorithm but this time from the MATLAB Global Optimization Toolbox.

Table 6.4: Continuous simulation model optimization results (derivative-free solvers).

Solver	Cost ($/hr)	Time	Iters	Obj Evals	Sim Evals
NOMAD	$1896.34	33.1s	41	315	33248
NLOPT COBYLA	$1937.74	12.2s	-	123	12030
`patternsearch`	$1937.74	44.0s	3	543	46266

Two interesting results are observed using derivative-free solvers: First that most solvers so far have been falling into a local solution (as shown by NOMAD finding a better solution), and second that a derivative-free solver can out-perform gradient based solvers for these types of problems, in this case COBYLA which is supplied with NLOPT. It is worth pointing out that the global solution to this problem is $1866.34 (found using techniques in Section 7.3), which no solver, 'global' or local, has yet found.

Pushing this problem further and enabling the back pressure turbine binary variables, we can attempt to solve the problem as a MINLP. Now the nonlinear optimizer in Figure 6.5 will be called multiple times as the branch and bound solver attempts to find the integer optimal solution. Table 6.5 shows the results when solved with BONMIN, which as expected, finds a better solution than IPOPT's continuous solution by switching off both back pressure turbines.

Table 6.5: Integer simulation model optimization results.

Solver	Cost ($/hr)	Time	Iters	Obj Evals	Grad Evals	Sim Evals
BONMIN	$1903.2	174s	0	-	-	182059
NOMAD	$1908.8	40.7s	33	356	-	40538

With respect to optimality of the MINLP solutions, we again find that neither NOMAD nor BONMIN has found the global minimum of $1831.87. In fact NOMAD, which found the best solution in the continuous problem has now found a worse solution in the discrete problem. This is an unfortunate result of the added complexity that comes with mixed integer problems, together with the added degrees of freedom. In addition, BONMIN required tuning of multiple parameters in order to solve this problem, many of which would be beyond a novice user's ability to adequately set. Also note the computation times for all of these solvers are substantial, which once again leads to the conclusion that straight optimizing of a simulation model is neither efficient nor robust.

With regards to derivative-free solvers, these are not going to be pursued within the remainder of this work because all the problems of interest will contain binary constraints. The only derivative-free solvers that had a facility for binary constraints (and were available for use) were NOMAD and a genetic algorithm solver supplied with the MATLAB Global Optimization Toolbox, and both did not substantially outperform the gradient based solvers. In addition, we were unlikely to improve on these algorithms given that the underlying simulation model is already highly optimized C++ code, so that other than using algorithmic heuristics, the optimization process itself cannot be easily improved for the problems of interest. Furthermore, as described in the next subsection, by exploiting the model structure we can obtain exact derivatives and thus achieve much faster solution times using gradient based solvers. Moreover, this technique results in a number of equality constraints of which are not supported by either NOMAD or the genetic algorithm solver (typical of derivative-free global optimization solvers).

6.3.3 Exploiting Model Structure

Two issues are evident given the optimization strategy used so far: The global optimum has not been found, and secondly the time taken to solve for this suboptimal solution has taken far too long. Each of these issues will be tackled individually, with Chapter 7 describing the method of proving a global optimum, and the remainder of this chapter dealing with the speed problem.

As described in Section 6.3.1.1, the main issue with optimization of a simulation model is that it is computationally expensive to find derivatives, and even then the derivatives are likely only to be an approximation by finite-difference or similar. In order to be able to obtain accurate derivatives quickly, the simulation model will need to be 'opened up' and the underlying equations used to generate the derivatives via one of the methods described in Section 6.2.4. The implication here is that JSteam (and its thermodynamic engine) can no longer realistically be used, because it is effectively a 'black box' with the internal model structure and equations 'unknown'.

To open up a JSteam model involves re-implementing the basic model equations from Section 5.3.3 within an environment that can then utilize (exploit) this extra information to generate the derivatives. Furthermore, by writing out all model equations for the system, it allows a range of solution techniques to be used to either improve the solution, or aid finding it. To illustrate the process, consider the desuperheater from Section 5.3.3.5 as shown in Figure 6.6.

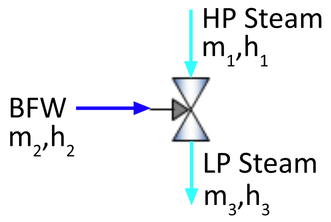


Figure 6.6: Desuperheater with mass and enthalpy variables listed.

Writing out fundamental equations for the model results in two expressions: the mass balance

$$m_3 = m_1 + m_2 \tag{6.5}$$

and the energy balance

$$m_3 h_3 = m_1 h_1 + m_2 h_2 \tag{6.6}$$

Writing both these equations as standard optimization constraints

$$m_1 + m_2 - m_3 = 0 \tag{6.7}$$
$$m_1 h_1 + m_2 h_2 - m_3 h_3 = 0 \tag{6.8}$$

we can begin to develop an equation based model (termed equation or optimizer model from here on). This model of the desuperheater returns the same solution as the JSteam model. Now however we are not limited to supplying certain inputs and calculating certain outputs, rather the model works for any set of arguments provided the degrees of freedom are met.

To implement the above model and solve it with an optimizer, a fundamental shift in modelling paradigm is required. First of all, the model above currently has six decision variables, given that enthalpy and mass flow of each stream is allowed to vary. Compared with the simulation model optimization problem in the last section, which only had 7 decision variables for the entire MINLP formulation, the desuperheater by itself already has 6 variables. Clearly this modelling strategy is going to substantially increase both the number decision variables, and the number of constraints. Generally this is ill-advised, as larger problems can take exponentially longer to solve. However, as will be shown later in this chapter, this is actually the key to obtaining an optimal solution much faster.

Looking at a larger modelling example, Figure 6.7 shows a section of a hypothetical utility system. Within the Figure the model has been labelled as per the

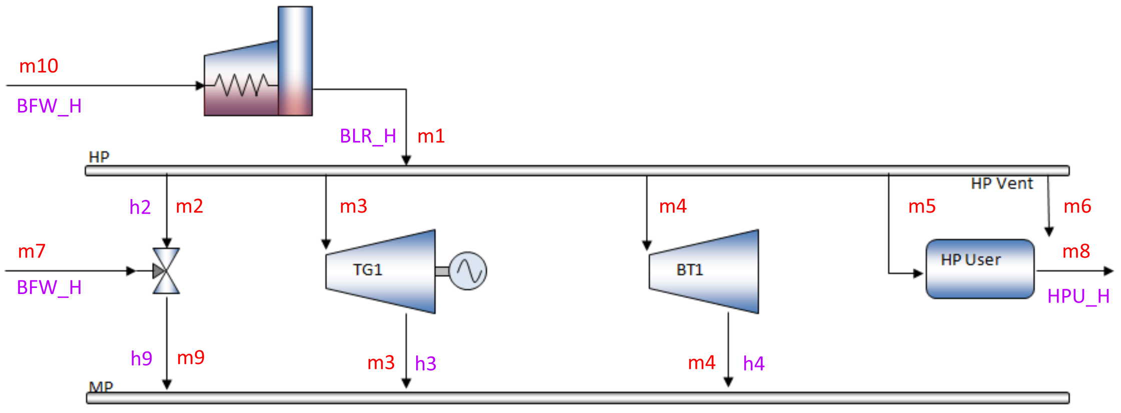


Figure 6.7: Desuperheater with mass and enthalpy variables listed.

optimization variables that will exist in the equation based model. By intelligently selecting the decision variables, it is possible to reduce the number required. Within this model the following hand optimizations have been performed:

- Both turbo generators (TG1) and back pressure turbines (BT1) are assumed to be lossless in terms of steam, thus only one mass flow variable is required through the turbine.
- The boiler feed water is assumed to be of constant enthalpy, given that the pump model within this work assumes a constant isentropic efficiency. This reduces the enthalpy decision variables because it is replaced by a constant.
- The boiler steam temperature and pressure are assumed constant thus their output enthalpy can also be modelled as a constant.

While further optimizations could be performed, such as m_1 being written as a function of m_{10} (given we know the blowdown ratio of the boiler), or applying a similar approach to the HP User, the remaining variables are chosen in order to keep the model readable. Note in the above example h_2 is the enthalpy of the HP header, while h_9 is the enthalpy of the MP header. In this example, the HP header enthalpy will be fixed at the boiler output enthalpy, but it remains a variable because it is common to have multiple inlets at different enthalpies in a real example.

Given the constants and variables identified, the mass and energy balance equations for the HP header can be written as

$$m_1 - m_2 - m_3 - m_4 - m_5 - m_6 = 0 \tag{6.9}$$
$$m_1 h_{\text{blr}} - h_2 (m_2 + m_3 + m_4 + m_5 + m_6) = 0 \tag{6.10}$$

noting again the mass balance is linear, while the energy balance is bilinear. This bilinear equation is particularly difficult in that it is non-convex, which can be shown by examining the eigenvalues of the Hessian of the energy balance equation

$$\operatorname{eig}\left(\nabla^2 Q\right) = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ \sqrt{5} \\ -\sqrt{5} \end{bmatrix} \tag{6.11}$$

which shows a saddle point because it is an indefinite matrix. This feature, combined with being an equality constraint, infers that we are now solving a non-convex problem which, depending on other constraints, will exhibit multiple minima.

With the added constraints resulting from the system mass and energy balances, the optimizer now has two roles: To find a set of mass flows and enthalpies (the decision variables) that satisfy the constraints (i.e. acting as a nonlinear root solver), and working as a nonlinear optimizer, to select the available degrees of freedom which minimizes a supplied cost function. In this way there does not have to be n equations and n unknowns (i.e. a square system), because by being underdetermined the optimizer has 'room to move' and thus optimize the operating point of the model. The catch is that the model must still be suitably constrained so that the problem is not unbounded, as well as requiring a *good* initial guess of all variables.

6.3.3.1 Initial Solution Guess

A common problem with the equation based modelling approach is the requirement of a feasible, or at least realistic, initial solution guess (optimizer starting point). Given the equation model will contain many more decision variables, a number of which are intermediate flows and duties, to start with a vector of **0** is unlikely to converge, or at least it will take longer to solve. This problem was well known in the early days of equation orientated simulators, as Westerberg said in [328] with respect to variable initialization "we believe this step must be done with extreme care, as doing it poorly can often lead to no chance to solve the problem even though a solution exists".

Modern equation based simulators such as gPROMS have a robust initialization procedure, which as described during a plenary at FOCAPO 2012, can actually take significantly longer than the solution itself. For this work a simple approach is available, given that we already have a sequential simulator model. The initial solution guess for the optimizer can be taken directly from the results of the simulator model, which typically provides a point that is either feasible, or very nearly feasible within the approximated equation model.

6.3.4 Integrating Purpose-Built Unit Operation Approximations

For simple unit operations such as a desuperheater or deaerator, the required mass and energy balance constraints are easily obtained by inspection. Together with the other required unit operations (headers, steam users), the fundamental equations that describe the steam side of a utility system are relatively easy to derive. However the relationships that describe the required fuel flow for a boiler or gas turbine cannot be described as easily.

To illustrate, consider the steam boiler shown in Figure 6.8. While the mass

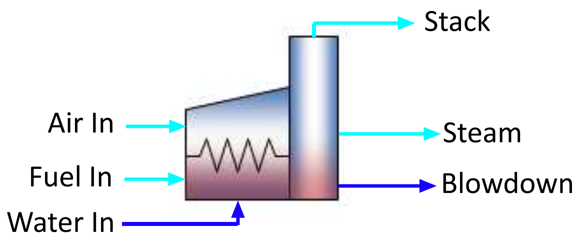


Figure 6.8: Steam boiler unit operation.

and energy balance of the steam side can be simply described by the steam boiler model equations in Section 5.3.3.1, the equations governing the combustion thus consumption of fuel and air, together with stack constraints, are too complex if we were to follow the furnace model in Section 5.3.5.1.

To overcome this, the detailed models developed at the end of Chapter 5 can be substituted for the JSteam models within the optimizer model. In this way the unit operation equations for the fuel-gas side are approximated by simple equations, while the underlying regression coefficients are based on both literature, plant data and JSteam models (for thermodynamic values and operational specifications). Furthermore, as the energy balance of the fuel-gas side is only of interest in calculating the required fuel consumption, it does not have to be explicitly calculated if we approximate the solution using properties of the steam side and a regression which implicitly takes into account this energy balance. This allows the unit operation equations to be reduced to just a mass balance and relationship of duty/fuel to mass flow, which in most cases is linear.

Taking for example the detailed steam boiler model in Section 5.5.2, the final relationship between steam production and fuel gas consumption is linear, even with a varying efficiency, so that optimization of this model is much simpler than a detailed JSteam black box model. Note the JSteam unit operation model and the approximation of the detailed unit operation will calculate the same result if both use the same efficiency correlations and input data, therefore simplifying validation.

Completing the steam boiler example, the following two linear equations completely describe the unit operation for the purposes of optimization (ignoring steam production limits for now)

$$M_{\text{fuel}} = \lambda + \gamma M_{\text{steam}} \tag{6.12}$$

$$M_{\text{BFW}} - (1 + F_{\text{BD}})\, M_{\text{steam}} = 0 \tag{6.13}$$

where the first equation is used in the objective function (noting λ and γ are regressed coefficients as detailed in 5.5.2) and the second equation is the mass balance constraint for the unit.

Further details of the implementation of unit operations with regressed coefficients are described in Section 6.5.2.

6.3.5 Linear Approximations

A common approach amongst literature within the utility system optimization field [5, 134, 149, 197, 213] is the reduction of the optimization problem to a set of linear constraints which are then solved as a mixed integer linear program. While admittedly many of these papers are solving a synthesis problem (the design of a new utility system or evaluating retrofit options), which may be much larger, they all essentially take the same approach to linearization by fixing the steam header enthalpy. Revisiting the energy balance equation from Section 6.3.3

$$m_1 h_{\text{blr}} - h_2 (m_2 + m_3 + m_4 + m_5 + m_6) = 0 \tag{6.14}$$

shows that by setting both h_{blr} and h_2 as constants, the energy balance equation results in a linear expression. By applying a similar method across the energy balance equations for the rest of the model, together with further approximations (such as piecewise-linear), the entire model can be reduced to a MILP. While an MILP is attractive for optimization purposes, it does not accurately represent header temperature within the model, which an operator familiar with the 'steam hammer' effect (described well in [311]), knows that condensing steam within headers can cause major problems and that it is important that header enthalpy should be accurately modelled.

Varbanov et al [315] proposed a solution to this problem by successively solving a series of MILPs and rigorous simulations. As shown in Figure 6.9, after each MILP optimization a rigorous nonlinear simulation was run, and the change in header and turbine enthalpies compared. The process was repeated until satisfactory convergence was reached, which according to the author, was typically around 5 iterations.

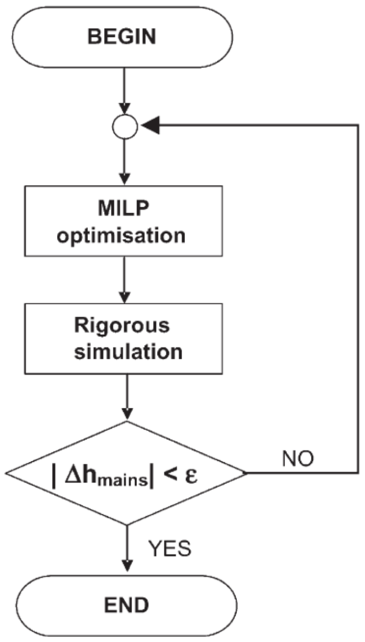


Figure 6.9: Successive MILP method (Figure 21 in [315]).

A point made in [315] was that "... the resulting optimization formulations would be mixed integer non-linear programs (MINLPs). This usually leads to computation problems, inherent to the currently available MINLP solution algorithms.", so the reasoning for applying the successive MILP approach. However it is now close to 10 years later, and the algorithms for solving MINLPs have matured to the point where they are both fast, robust and can guarantee global optimum, if required. For example, the solver BONMIN has been available in the open-source community for over 7 years, and continues to be in active development by a range of world experts in discrete optimization. Therefore by using today's technology it is possible to avoid this linearization step, or a successive approach, and solve the full MINLP directly, and still obtain competitive, if not more accurate results. It is for this reason that this work is concerned only with nonlinear optimization.

6.3.6 Discrete Nonlinear Optimization

When optimizing the operation of a utility system, whether this be a simulator model or optimizer model, there are two methods of solving the problem. The first method uses a standard nonlinear optimizer, such as IPOPT or MATLAB's `fmincon`, which solves the *continuous* optimization problem. For this problem, decisions such as whether a unit should be turned on, or switched to a particular state, are not made by the optimizer, but rather fixed by the user. An example of a decision is the ability to select whether to use steam (via a back pressure turbine) or electricity to drive fixed loads.

The second method employs state-of-the-art optimizers such as BONMIN to solve a *discrete* optimization problem. A discrete optimizer can solve problems with binary and integer constraints, and thus take control of back pressure turbines and other units in order to find the optimum. This method provides a much more automated approach to optimization of a utility system, where, apart from operational constraints and current prices and demands, the optimizer has full control over the solution process. As shown in both Section 6.3.2 and our work in [70], the discrete optimizer has a greater chance of finding the optimum solution given the extra degrees of freedom, and is therefore the preferred approach.

In addition to better optimization potential, utilizing binary variables allows the optimization model to include common operational constraints such as minimum turn-downs, fixed costs and other simple logical constraints. These allow the model to better represent actual operation and thus aid the optimizer to find physically realisable results. The following section details common binary and logical constraints that are used within this work.

A final note most likely obvious to the reader, is that a discrete optimization solver is much more complex and computationally intensive, and thus can require a longer time to solve. However by exploiting the model structure as described so far, as well as avoiding binary variable 'bad practices' (e.g. unnecessarily multiplying binary variables with continuous ones), the addition of binary and integer constraints does not substantially impact solution time for our problems.

6.4 Discrete Constraints Common In Utility Systems

As already identified in Section 5.2.3, utility systems are discrete in nature and therefore the optimization of a utility system inherently requires discrete variables. Furthermore, to efficiently model logical operational constraints common in utility systems, binary variables should be utilized. The techniques described in this section have been derived from the informative AIMMS modelling guide [37], which provides a clear tutorial for new users of integer programming.

6.4.1 Equipment Selection Problem

One of the most common opportunities for economic savings within a utility system is to exploit the redundancy in many fixed mechanical loads. These loads include equipment such as compressors, fans, pumps and other process requirements where mechanical shaft work can be utilized. Part of the design of a utility system includes both steam turbines sized to drive these loads, but also redundant electric motors which can be switched in when starting the plant, if the turbine were to fail or if economically attractive. This provides an opportunity to exploit a binary decision on which driver to use, based on factors such as fuel and electricity prices, as shown in Figure 6.10.

This situation can be modelled using a single binary variable, b_{bpt}, and single linear constraint

$$W_{\text{shaftwork}} b_{\text{bpt}} - \eta M_{\text{steam}} \Delta H_{\text{isen}} = 0 \tag{6.15}$$

where $W_{\text{shaftwork}}$ is the required mechanical shaftwork (in this example it is constant). When b_{bpt} is 1 (on), the steam turbine is connected, and when 0, the electric motor is used. The utility model mass and energy balance will automatically take care of the situation when the steam turbine is connected, but a modification to the power

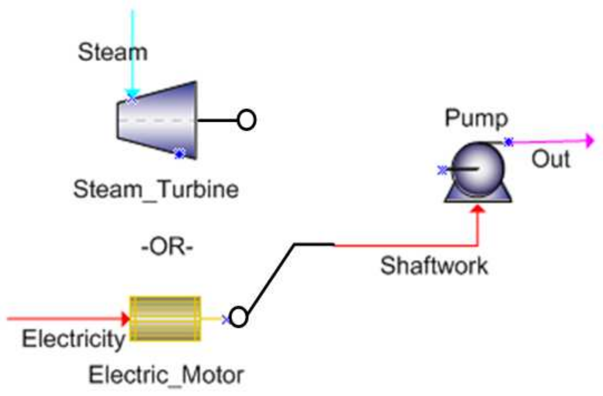


Figure 6.10: Equipment selection via binary variables.

balance is required to account for the use of the electric motor

$$W_{\text{elec}} = W_{\text{shaftwork}} \left(1 - b_{\text{bpt}}\right) \tag{6.16}$$

The above equation adds the shaft work to the electricity requirement of the site when the steam turbine is disconnected, so this needs to be included in the objective economic calculation.

Using this modelling strategy requires only one binary variable per steam turbine, and within this work, only for back pressure turbines connected to a fixed load. Turbo generators are modelled differently due to a varying load, however a similar approach can be taken and is described in Section C.2.2.6.

6.4.2 Logical Operating Constraints

Logical constraints which involve situations such as either-or, conditional or fixed cost are powerful methods which when implemented correctly, can add realistic operating constraints to the optimizer model. Taking the steam boiler as an example, there are two common constraints associated with its operation which can be described as a piecewise function

$$M_{\text{steam}} = \begin{cases} 0 & \text{if (Boiler is Shutdown)} \\ M_{\min} \leq M_{\text{steam}} \leq M_{\max} & \text{if (Boiler is Running)} \end{cases} \tag{6.17}$$

As described by this expression, when the boiler is operational it has both minimum and maximum steam production limits, but when turned off it must produce no steam. When viewed as a relationship between steam production and fuel consumption, as shown graphically in Figure 6.11, a discontinuous function results.

In order to model this function without introducing ‘if’ statements, either-or

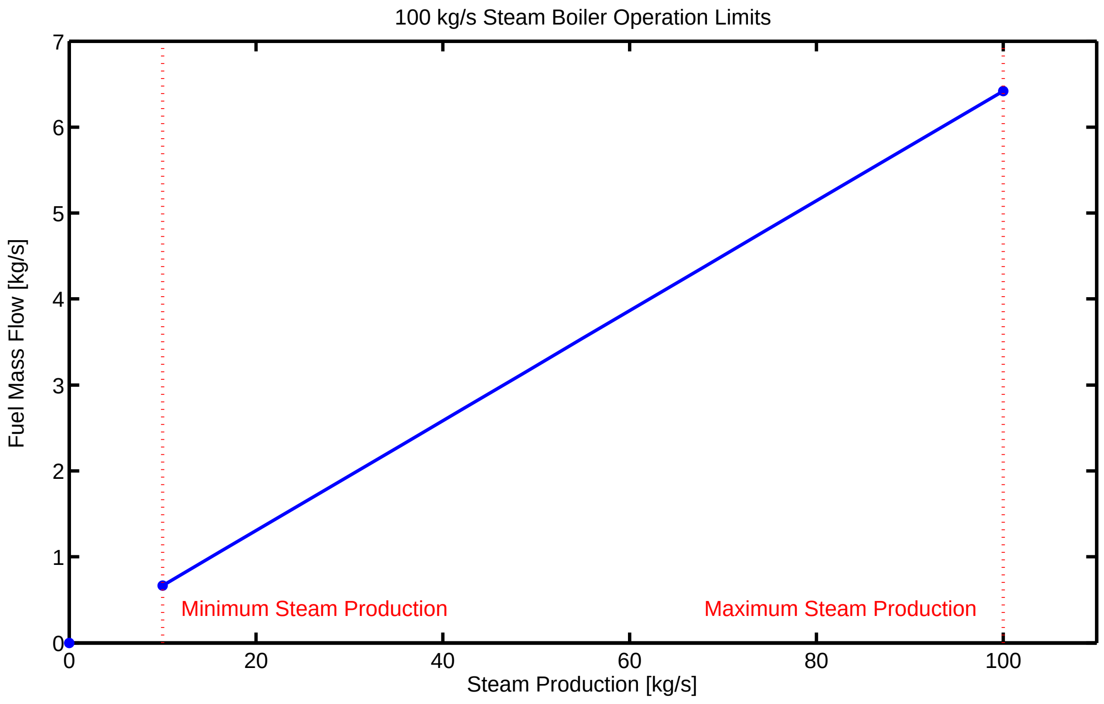


Figure 6.11: Steam boiler discontinuous fuel flow function.

constraints from Section 7.3 of [37] can be used. By introducing a binary variable to represent whether the boiler is running (1) or shutdown (0), the logical condition of only one constraint being active is enforced. When the boiler is shutdown, the first case in Equation 6.17 states the steam mass flow must equal 0. To implement this, we replace it with an inequality of the form

$$a_1 x \leq b_1 + M_{\text{big}} y \tag{6.18}$$

where a_1 and b_1 are linear inequality coefficients, x is the continuous decision variable, y is the binary variable and M_{big} is a 'big M' constraint set so $a_1 x$ is always less than $b_1 + M_{\text{big}}$. Written with respect to the steam boiler, this constraint becomes

$$M_{\text{steam}} \leq M_{\text{big}} y \tag{6.19}$$

where we have set $a_1 = 1$ and $b_1 = 0$. When the boiler is off ($y = 0$), this constraint forces the steam production to less than or equal to 0 (a further lower bound on the M_{steam} decision variable keeps it at 0), while when on, the constraint is non-binding (has no effect) when M_{big} is set as greater than $M_{\text{steam(max)}}$.

When the boiler is running, a second constraint ensures that the steam production is above the minimum turndown (noting the coefficient signs are flipped as this is a $\geq$ constraint)

$$-a_2 x \leq -b_2 + M_{\text{big}} (1 - y) \tag{6.20}$$

Again written with respect to the boiler

$$-M_{\text{steam}} \leq -M_{\text{steam(min)}} + M_{\text{big}}\,(1-y) \tag{6.21}$$

the coefficient $a_2 = 1$ and b_2 is set as the minimum turndown. With this constraint, when the boiler is off, we see the constraint is non-binding due to the big M value. However when the boiler is on, it enforces the minimum mass flow. Similar to the first constraint, the maximum mass flow is set as a variable bound on M_{steam}.

This method of using either-or constraints is used within the turbo generator, gas turbine, HRSG and boiler models, and therefore a binary variable is required for each of these units.

6.4.3 Piecewise Linear Approximations

In our early work on the optimization of these systems, [70] we investigated the opportunity of keeping the entire model formulation linear. In order to do this, nonlinear expressions had to be converted to linear approximations. Taking the output enthalpy of a steam turbine as an example, Figure 6.12 shows an 'optimal' piecewise linear approximation of the nonlinear response.

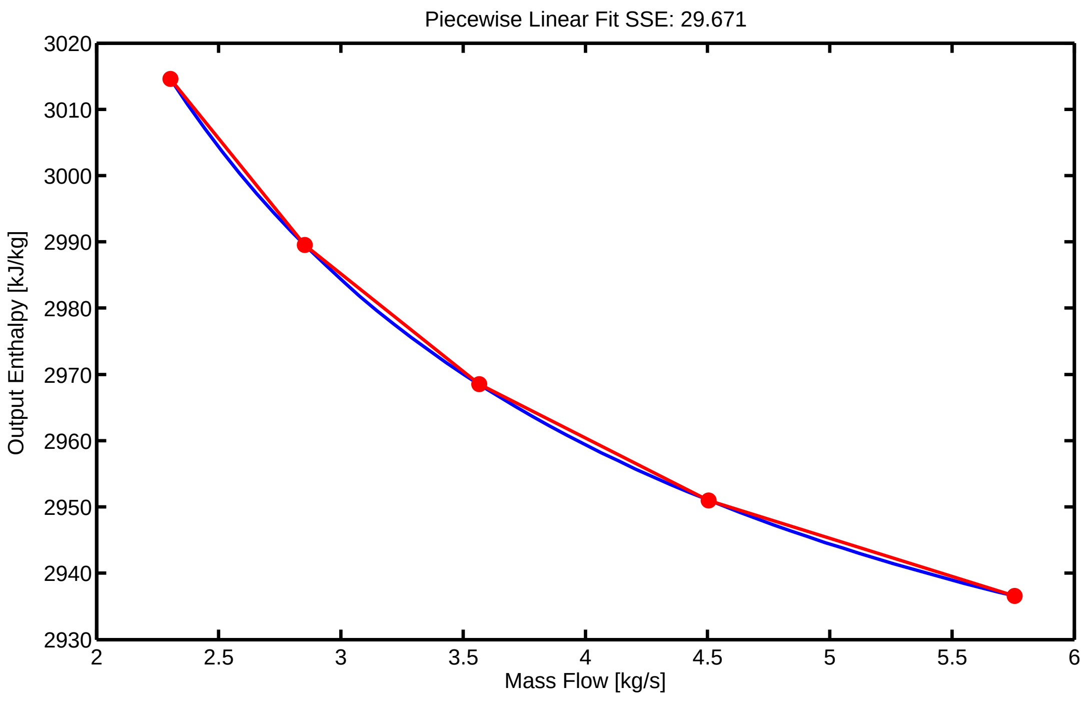


Figure 6.12: Piecewise linear fit to output enthalpy of a variable efficiency steam turbine.

Using a nonlinear least squares optimizer, each red dot indicates the start/end of a linear segment chosen by the solution to an optimization problem setup to minimize the sum of squared errors. The number of segments is set by the user, and once their location has been optimized, a custom routine automatically generates the required Type 2 Special Ordered Set (SOS) (see Section 7.6 in [37]) and equality constraints required to implement the approximation within a MILP.

It is worth noting that this approach can only be applied to separable nonlinear functions (i.e. functions without products or ratios of variables), and that each dot in Figure 6.12 effectively requires a binary variable. In practice however, modern solvers have a facility for treating SOS separately so that an explicit binary formulation by the user is not required. For the purposes of this work, the piecewise linear approximation is not useful, beacuse most functions are not separable given they involve the product of mass and enthalpy, and therefore cannot be modelled without further linear approximations (such as described in Section 6.3.5). However the functionality remains built-in for future work.

6.4.4 Modelling Power Usage

The final discrete constraint required for modelling utility systems is when calculating the cost of buying or selling electricity. As described further on in Section C.2.2.10, it is expected that the prices for buying and selling electricity will be different, and so the objective function must be able to accommodate the price change as a function of the power balance (i.e. whether we are importing or exporting electricity).

To model this scenario a binary variable is introduced to describe whether the power balance is positive (selling power, binary 0), or negative (buying power, binary 1). Using the big M strategy together with either-or constraints, as described in the logical operating constraints section, the binary variable is used to modify the cost constants in the objective. Further details, including implementation, are described in Section C.2.2.10.

6.5 SymBuilder Framework

As detailed earlier in Section 6.3.1.1, the issue of obtaining accurate gradients of a utility system simulation model is impossible using the normal method via finite difference. However by explicitly writing out the mass and energy balances, as described in Section 6.3.3, it is much easier to obtain derivatives. To automate the process of obtaining derivatives of these explicit models, we developed the SymBuilder framework, a MATLAB framework built on the Symbolic Toolbox which was added to OPTI in June 2012. As with OPTI, it is an object-orientated framework which uses a single object to pose an optimization problem. Given the requirement to use the Symbolic Toolbox, the optimization problem is entered into MATLAB as a series of strings, whilst also using methods to indicate their functionality within the problem description. Repeating the linear programming example from Section 6.2.3.1, one would build it in SymBuilder as

```
%Create a Blank SymBuilder Object
B = SymBuilder();
%Add Objective
B.AddObj('-6*x1 - 5*x2');
%Add Constraints
B.AddCon('x1 + 4*x2 <= 16');
B.AddCon('6*x1 + 4*x2 <= 28');
B.AddCon('2*x1 - 5*x2 <= 6');
B.AddBound('0 <= x <= 10');
```

where at each declaration the object stores the string within a class property, keeping note of whether it were an objective, constraint, bound or other declaration. Once the problem definition has been entered, the object is then 'built' using

```
>> Build(B)

Generating Symbolic Representation of Equations...Done
Generating Symbolic Jacobian...Done
Generating Symbolic Hessian...Done

SymBuilder Object
 BUILT in 0.024s with:
 -  2 variables
 -  1 objective
      -  1 linear
      -  0 quadratic
      -  0 nonlinear
 -  3 constraint(s)
      -  3 linear
      -  0 quadratic
      -  0 nonlinear
 -  4 bound(s)
 -  0 integer variables(s)
```

which, as per the status messages printed, turns the user entered strings into a system of symbolic equations, and generates the system derivatives. From the derivative information, the object can then determine whether each equation is linear, quadratic or nonlinear, and thus what type of problem has been entered. To solve the problem the `Solve` method is called

```
>> Solve(B)
```

which internally generates an OPTI object, then solves it. For linear and quadratic problems the process of turning a SymBuilder object into an OPTI problem is very quick, because all problem information is stored as numeric data in matrices. For nonlinear problems this process is quite different. Consider the Hock & Schittkowski nonlinear test problem #71 [322]

$$\begin{aligned} \min_{\mathbf{x}} \; & x_1 x_4 \left(x_1 + x_2 + x_3\right) + x_3 \\ \text{subject to: } & x_1 x_2 x_3 x_4 \geq 25 \\ & x_1^2 + x_2^2 + x_3^2 + x_4^2 = 40 \\ & 1 \leq \mathbf{x} \leq 5 \end{aligned}$$

which is then entered into SymBuilder using the following script

```
B = SymBuilder();
% Add Objective
B.AddObj('x1*x4*(x1 + x2 + x3) + x3');
% Add Constraints
B.AddCon('x1*x2*x3*x4 >= 25');
B.AddCon('x1^2 + x2^2 + x3^2 + x4^2 = 40');
% Add Bounds
B.AddBound('1 <= x <= 5');
```

When `Build` is called on the object, the output below is printed

```
SymBuilder Object
 BUILT in 0.028s with:
 -  4 variables
 -  1 objective
      -  0 linear
      -  0 quadratic
      -  1 nonlinear
 -  2 constraint(s)
      -  0 linear
      -  1 quadratic
      -  1 nonlinear
 -  8 bound(s)
 -  0 integer variables(s)
```

illustrating SymBuilder has correctly processed our problem with a nonlinear objective, single nonlinear constraint and single quadratic constraint. As this is a nonlinear problem, numeric data cannot be used to describe it and functions must therefore be generated. If we request to view the OPTI object, we can access the object details

```
>> GetOPTI(B)
```

```
Generating Objective & Gradient....Done
Generating Hessian....Done
Generating Constraints & Jacobian....Done

Nonlinear Program (NLP) Optimization
 min  f(x)
 s.t. lb <= x <= ub
      cl <= c(x) <= cu
------------------------------------------------------
   Problem Properties:
# Decision Variables:        4
# Constraints:              10
  # Bounds:                  8
  # Nonlinear Inequality:    1
  # Nonlinear Equality:      1
------------------------------------------------------
  Solver Parameters:
Solver:                    IPOPT
Objective Gradient:        symb_grad
Constraint Jacobian:       symb_nljac
Jacobian Structure:        Supplied
Lagrangian Hessian:        symb_hess
Hessian Structure:         Supplied
```

and it can be seen that SymBuilder has constructed functions for not only the objective and constraints, but also for the gradient, Jacobian and Hessian, as well as both required sparsity patterns. As per design, SymBuilder writes MATLAB function files containing the nonlinear function callbacks, as shown for the Hessian of this problem below:

```matlab
function H = symb_hess(x,sigma,lambda)
% SYMB_HESS
%
% Hessian of the Lagrangian Callback

% Symbolic Builder Auto-Generated Callback Function
% Generated 09-Nov-2013 22:44:10

% Preallocate
H = spalloc(4,4,10);

% Sparse Matrix:
H(1,1) = 2*lambda(2) + 2*sigma*x(4);
H(2,1) = sigma*x(4) + lambda(1)*x(3)*x(4);
H(3,1) = sigma*x(4) + lambda(1)*x(2)*x(4);
H(4,1) = sigma*(2*x(1) + x(2) + x(3)) + lambda(1)*x(2)*x(3);
H(2,2) = 2*lambda(2);
H(3,2) = lambda(1)*x(1)*x(4);
H(4,2) = sigma*x(1) + lambda(1)*x(1)*x(3);
H(3,3) = 2*lambda(2);
H(4,3) = sigma*x(1) + lambda(1)*x(1)*x(2);
H(4,4) = 2*lambda(2);
```

The decision to auto-generate MATLAB functions is based on being able to exploit efficient sparse descriptions within the language, as viewed above, as well as giving MATLAB the opportunity to optimize the code further during its Just-In-Time (JIT) compilation phase. In addition, it allows the user to view the derivative functions and run them as a normal MATLAB function which could be useful, for example, during sensitivity analysis or validation.

This technique is sufficient for problems up to a few hundred variables and constraints, which is common within this work, however larger problems quickly increase in size. For larger problems we have experimented with auto-generating C files to contain the function callbacks which are then compiled into a single callback function. While this technique requires a longer time initially, function execution times can be substantially reduced, and there is scope for parallelizing the work. This is currently a work in progress.

6.5.1 Comparison with Other Algebraic Modelling Packages

The reader at this point will have noticed that SymBuilder appears to have reinvented the wheel when compared to packages such as GAMS and AMPL. The reasons why we have developed this functionality are detailed below, including major differences in the design philosophy between these packages.

Single Package Design By developing SymBuilder within MATLAB the user has the ability to build and define optimization problems using standard MATLAB syntax, then generate the required expressions as strings using standard MATLAB syntax (detailed in the next subsection). In addition, this allows the full use of domain specific routines within the problem formulation, such as the JSteam thermodynamic routines and curve fitting routines to generate the required approximated models. This is opposed to a technique such as in [46] whereby an external software package is used to describe the problem, which then generates a GAMS model, and then extracts the results back. Leveraging SymBuilder, OPTI and JSteam all within MATLAB means the problem can be posed, solved and inspected all within MATLAB.

Symbolic Simplification Using a symbolic modelling engine allows the model to be symbolically simplified once it is created, reducing unnecessary terms and optimizing the code performance. For example, setting a variable to a constant value of 0 will automatically remove any associated equations from the model which are not required.

Code Optimization As detailed in the previous subsection, all nonlinear callback functions are auto-generated as MATLAB function files. The aim is that the MATLAB JIT compiler will then optimize these files, leveraging threaded linear algebra and element-wise routines for faster execution.

Derivative Complexity For the problems of interest in this work, the first and second derivative analytical functions are not especially complex or dense, and can therefore be described succinctly in MATLAB functions. It is acknowledged that for complex and dense derivative functions, automatic differentiation as used by alternative packages would be a better option.

Inspection and Validation of Results One of the major advantages of MATLAB is the ability to inspect and visualize data easily, either by simply printing it to the command window or by plotting it in an intelligent fashion. As optimization problems often contain hundreds to thousands of variables, inspecting the result by scrolling through each element one by one is time consuming and error-prone. On the other hand validation against a simulation model also built in MATLAB is quick and easy. Furthermore, results can be automatically propagated to a JSteam Excel model if required.

6.5.2 Modelling Utility Systems using SymBuilder

Building utility models with explicit mass and energy balance equations within SymBuilder is simple, given the natural handling of variables and mathematical equations. For example, consider the desuperheater example from Section 6.3.3 (Figure 6.6). From standard mass and energy balances, this model can be entered into SymBuilder as

```
% Mass Balance
B.AddCon('m1 + m2 - m3 = 0');
% Energy Balance
B.AddCon('m1*h1 + m2*h2 - m3*h3 = 0');
```

noting the variable name convention `m` for mass flow and `h` for enthalpy. This allows us to provide a set of global variable bounds (effectively vectorized) such as

```
B.AddBound('0 <= m <= 100');
B.AddBound('0 <= h <= 800');
```

which can then be further customized for each unit. These two lines provide a simple mechanism to bound all optimizer variables within sensible limits (i.e. no negative mass flows or enthalpies), without tediously assigning bounds to each variable.

It is acknowledged that this method of entering the mass and energy balances of a model is ‘slow-going’, however even an industrial utility model does not require more than 50 simple linear or bilinear mass and energy balance equations, all of which can be read directly from a PFD. These manual equations are predominantly used to describe simple units such desuperheaters, steam header balances, mixers and splitters. For more complex units which involve regressed models, a derived class (`SymUtility`) has been written which extends the functionality of the SymBuilder class to modelling steam turbines, boilers, gas turbines and other units. Each of the methods implemented is described in Section C.2.2.

6.6 Utility System Case Studies

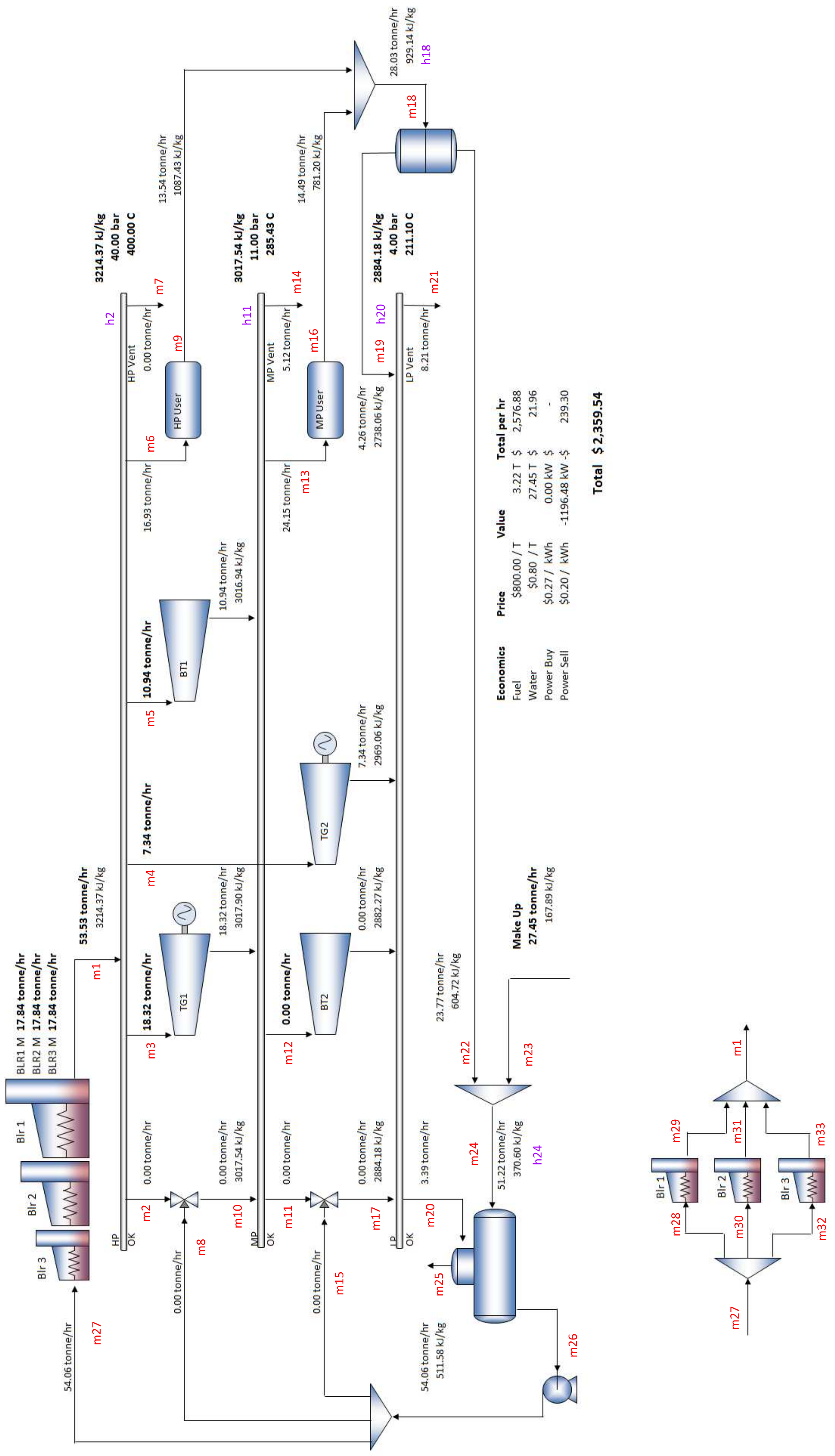
To illustrate the effectiveness of the proposed combined SymBuilder and JSteam approach to building and optimizing utility systems, three case studies are presented. Each case study will present a base-case operating scenario, then examine optimization opportunities over a range of hypothetical operating conditions. Where possible, comparisons are made with literature, however this is not always possible, given the multitude of specifications required (and often not listed in entirety within Journal papers) and variations in modelling approach.

6.6.1 Three Header Hypothetical System

The hypothetical three header system is a simple utility system designed to demonstrate the ease of optimization within the SymBuilder framework. This model is the same as described in Section 6.3.1, but this time will be optimized by exploiting the model structure within SymBuilder. The model, including optimization variables, is shown in Figure 6.13.

The base-case operating point splits the steam demand equally between the three boilers, as well as using both turbo generators to generate excess electricity to sell back to the grid. As the LP header is already venting steam, BT2 is switched off. Following the stream names as set out in Figure 6.13, the model is entered into a SymBuilder object. For a listing of the model code, including mass and energy equations, see Section C.3.1. Once built, SymBuilder reports the following model statistics

```
SymBuilder Object
 BUILT in 0.785s with:
 - 46 variables
 -  1 objective
```



Economics	Price	Value	Total per hr
Fuel	$800.00 / T	3.22 T $	2,576.88
Water	$0.80 / T	27.45 T $	21.96
Power Buy	$0.27 / kWh	0.00 kW $	-
Power Sell	$0.20 / kWh	-1196.48 kW -$	239.30

Total $2,359.54

Figure 6.13: Hypothetical 3 header model with base case operating data

```
      -  0 linear
      -  0 quadratic
      -  1 nonlinear
 - 45 constraint(s)
      - 28 linear
      - 13 quadratic
      -  4 nonlinear
 - 92 bound(s)
 -  8 integer variables(s)
      -  0 integer
      -  8 binary
```

which as expected, results in a Mixed Integer Nonlinear Program (MINLP). The 46 variables consist of the 33 modelled mass flows, 5 mixer/header enthalpies, 7 binary variables for each piece of equipment and 1 binary variable for the power balance calculation. Linear constraints result from mass balances and linear model approximations, while quadratic constraints result from energy balances and bilinear model approximations. The nonlinear constraints result from energy balances around the lower two headers, as well as from the duty expressions for each of the steam users. Finally, the nonlinear objective is a result of multilinear expressions which result from calculating the power balance. It should be noted that further approximations could be made to reduce the problem to a MIQCQP, however as shown below, the results of this formulation are quite sufficient.

In addition to optimizing the base case, three further operational points are used to validate the model and optimization results. In total, the four operational points are shown in Table 6.6. The three cases explore opportunities where electricity pricing becomes very attractive to export electricity, as well as when increased process demands require more steam from the utility system. The final case combines both higher electricity prices and increased demands to drive the system to its design limit.

Table 6.6: Hypothetical 3 header model optimization cases.

	Base Case	Case 1	Case 2	Case 3
Electricity Price (Buy)	$0.27	$1.50	$0.27	$1.50
Electricity Price (Sell)	$0.20	$2.50	$0.20	$2.50
HP User Demand	10 MW	10 MW	20 MW	20MW
MP User Demand	15 MW	15 MW	25 MW	45MW

Each case is formulated in SymBuilder using SymUtility models, and the resulting optimization problems are minimzed using the free, open-source solver BONMIN and the outer-approximation algorithm [88, 97]. Table 6.7 lists the results for each of the cases together with key operating variables within the system. The build time includes both the symbolic generation of the derivatives, as well as the generation of the MATLAB callback functions, while 'nodes' refers to the number of solving nodes (not iterations) searched by the mixed integer solver.

Table 6.7: Hypothetical 3 header model optimization results.

	Base Case	Case 1	Case 2	Case 3
Starting Cost	\$2299.7	\$-452.22	\$3580.2	\$2509.7
Optimized Cost	\$1831.9	\$-5889.5	\$3204.8	\$-1982.6
Build Time	4.6s	4.6s	4.5s	4.6s
Solve Time	0.11s	0.1s	0.09s	0.1s
Nodes	0	0	0	0
TG1 W	1460.5 kW	2000 kW	1868.9 kW	2000 kW
TG2 W	(Off)	2000 kW	(Off)	1493.8 kW
BT1 M	(Off)	10.9 T/h	10.9 T/h	10.9 T/h
BT2 M	(Off)	5.4 T/h	(Off)	(Off)
BLR1 M	(Off)	59.4 T/h	56.8 T/h	60 T/h
BLR2 M	30 T/h	(Off)	(Off)	40 T/h
BLR3 M	11.3 T/h	19.5 T/h	17.7 T/h	20 T/h

As can be seen, the key results viewable are the solution times for this model, noting the average solve time is only 100ms, even for a MINLP with 45 constraints and 46 variables, of which 8 are binary. This impressive solution time is common across all cases, and when compared to the simulator model optimized in Section 6.3.2 (the same utility system), we see a speed up of over 1500x *and* an improved optimized result. The large speed up is due primarily to the fact that now the non-linear functions do not require a root solver, but rather are described using algebraic relationships only. Moreover, by using an algebraic model, the derivatives are much easier (and more accurate) to obtain, even if approximated by finite difference. Table 6.8 shows a comparison of solution and model generation times with varying degrees of analytical derivatives.

Table 6.8: Hypothetical 3 header model optimization results (derivative comparison).

	Base Case	Base Case[1]	Base Case[2]
Optimized Cost	\$1831.9	\$1831.9	\$1831.9
Build Time	4.6s	3.1s	2.9s
Solve Time	0.11s	0.13s	0.32s
Nodes	0	0	0

[1] No Analytical Second Derivatives
[2] No Analytical First or Second Derivatives

Another benefit of the algebraic model is the ability to determine a-priori the sparsity structure of the model derivatives. As shown in Figure 6.14, these are significantly sparse with the Jacobian only 8.3% dense and the Hessian 1.6%. By utilizing this information, the sparse linear solver (in this case the Harwell MA57 routine [87]) is not only more efficient in terms of memory, but is also faster due to being able to factorize and solve exploiting the sparsity pattern. Note the Hessian pattern, being symmetric, is displayed as lower triangular only (following the IPOPT/BONMIN convention).

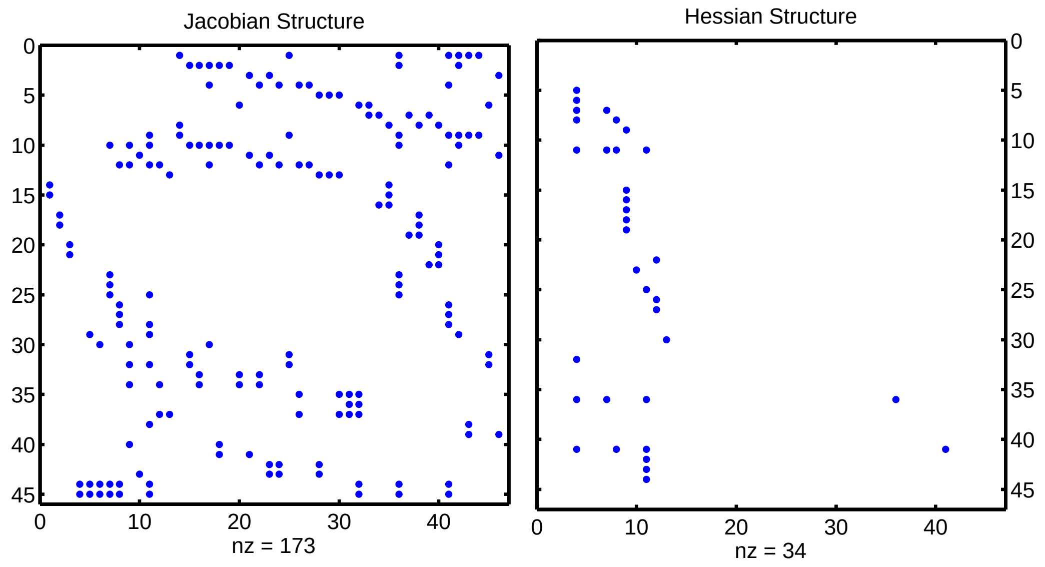
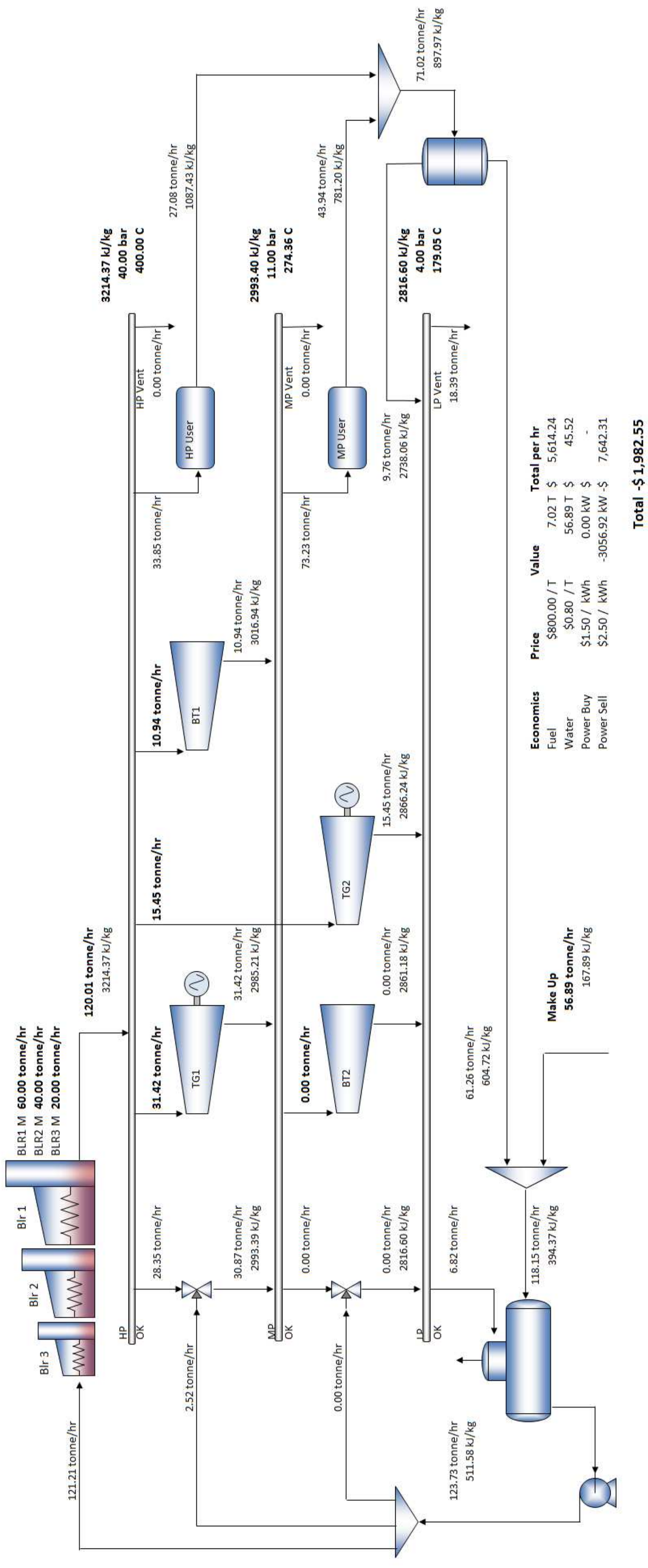


Figure 6.14: Hypothetical 3 header model sparsity patterns.

Comparing the results of the optimized model with the simulator model for Case 3, the optimizer model returns $-1982.58 while the simulator model returns $-1982.55 for the same operating point. Similar differences in the range of a few cents can been seen across the other cases, indicating the approximated equation-based model matches very closely to the rigorous simulation model over a range of operating points. To visualize the optimized solution, the results for Case 3 are exported to the JSteam Excel model, and can be viewed in Figure 6.15. Comparing the LP header temperature to the base case in Figure 6.13, we see the temperature has dropped by over 30°C. This demonstrates that an approximation which fixes the header enthalpy would not be accurate.

6.6.2 Four-Header Hypothetical System

The hypothetical four header system is based on the superstructure described by Bruno and Grossman in [46], noting that it was originally proposed in [242]. Both articles are concerned with the optimal synthesis (i.e. grass-root design) of utility



Economics	Price	Value	Total per hr
Fuel	$800.00 / T	7.02 T	$ 5,614.24
Water	$0.80 / T	56.89 T	$ 45.52
Power Buy	$1.50 / kWh	0.00 kW	$ -
Power Sell	$2.50 / kWh	-3056.92 kW	-$ 7,642.31

Total -$ 1,982.55

Figure 6.15: Hypothetical 3 header model with case 3 optimized data.

systems for specific heating, mechanical and electrical loads, so the results cannot be directly compared with this work. However the superstructure and resulting utility models provide a realistic platform that can be compared for optimal operation. The four headers of the system are at 45, 17, 4.5 and 0.2 bar for the HP, MP, LP and VLP headers respectively.

To enable a comparative study, the system superstructure has been fixed to the configuration shown in Figure 6.16. This configuration allows examples 1,1b,2 and 3 within [46] to be implemented and the optimal operation compared. For each example, the heating, mechanical and electrical demands are changed, however a common set of operating conditions are detailed in Table 6.9. Note that there are a number of assumptions in these examples which are not realistic, but are modelled for the sake of comparison with the original work. These include:

- 100% of the available duty within the steam consumed by a process user is available for use, inferring condensate is returned at 0 kJ/kg. While this is modelled in order to match the mass flow, the condenser header is set at 1.43 bar, 110°C as per the original paper. Note however that this is a broken energy balance.
- 100% of the steam consumed by a process user is returned as condensate, which is unrealistic in practice for all steam users.
- 100% of the available waste heat duty can be used to generate steam.

There are also a number of differences that will not be modelled in the same manner within this work, as described below:

- The cooling water circuit is not modelled explicitly, however the power required by the cooling water pumps (as reported by the reference) is included in the power balance.
- Fans for forcing air into the boilers are not modelled, however as above, the power required is included in the power balance.
- Variable efficiency models (as described in Chapter 5) are used in place of models such as turbo generators and boilers within the original reference. This means these units will report different operating conditions, such as fuel required by boilers.

In order to compare the solution from this work with the optimal solutions reported in [46], four examples are set up and solved based on the superstructure in

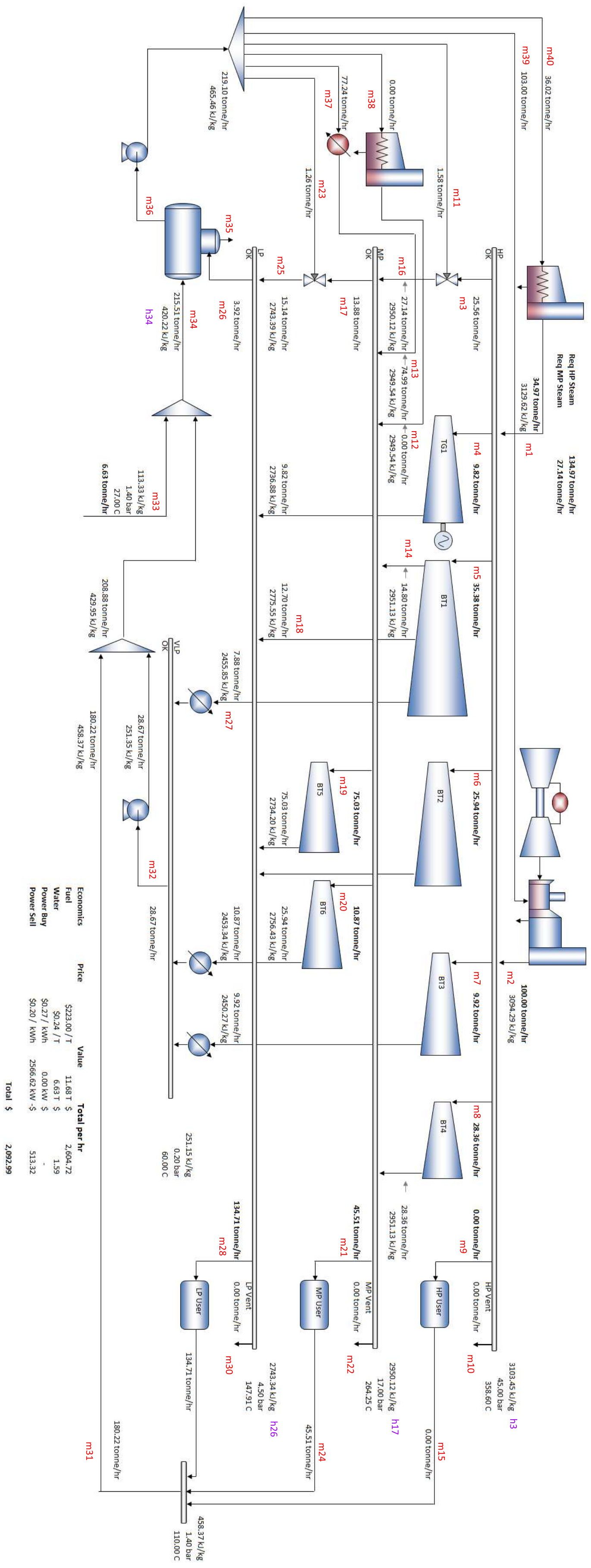
Figure 6.16. For each example, specifications in the SymUtility model are modified to match the reference (such as turbine shaft work and efficiency), also units not connected are disabled (such as turbines not required for the specified utility demands). The resulting SymUtility model is then optimized using techniques described in this chapter, and the resulting operating point compared against the optimal solution within the original reference.

Table 6.9: Hypothetical 4 header model equipment operating conditions.

HP Boiler (BLRH)	15-150 tonne/hr of 369°C steam at 45 bar. 180°C minimum stack temperature and 1% minimum mole fraction of stack O_2. Run on 72.9% Methane, 25.9% Ethane and 1.2% Nitrogen at 30°C with 30°C ambient air, 3% continuous blowdown.
MP Boiler (BLRM)	8-40 tonne/hr of 264°C steam at 17 bar, remainder of specifications identical to HP Boiler.
Waste Heat MP Boiler (WHB)	264°C, 17 bar steam with mass flow set by available duty, 100% efficient.
Gas Turbine + HRSG	5-40MW gas turbine exhausting 500°C gas into an HRSG with secondary firing, generating 355°C steam at 45 bar. Remainder of specifications identical to HP Boiler.
Turbo Generator 1 (TG1)	6000kW maximum.
BFW Pump (PMP)	45 bar outlet pressure, 65% isentropic efficiency.
Vacuum Pump (VACPMP)	1.4 bar outlet pressure, 65% isentropic efficiency.
Deaerator	1.4 bar with 0.0015 continuous vent ratio.
Make Up Water	4.5 bar, 27°C.

The utility system is entered into a SymBuilder object, following naming conventions detailed in Figure 6.16. The complete model implementation is detailed in Section C.3.2, and the resulting model shown below.

```
SymBuilder Object
 BUILT in 1.742s with:
 - 56 variables
 -  1 objective
      -  0 linear
      -  0 quadratic
      -  1 nonlinear
 - 50 constraint(s)
      - 32 linear
      - 15 quadratic
      -  3 nonlinear
 - 112 bound(s)
 - 11 integer variables(s)
      -  0 integer
```



Economics	Price	Value	Total per hr
Fuel	$223.00 / T	11.68 T	$ 2,604.72
Water	$0.24 / T	6.63 T	$ 1.59
Power Buy	$0.27 / kWh	0.00 kW	$ -
Power Sell	$0.20 / kWh	2566.62 kW	-$ 513.32
		Total	$ 2,092.99

Figure 6.16: Hypothetical 4 header model with base case operating data, adapted from [46].

```
    - 11 binary
```

For this system, the 56 variables consist of the 40 modelled mass flows, 4 mixer/-header enthalpies, the gas turbine output power, 10 binary variables for each piece of equipment and 1 binary variable for the power balance calculation. As with the first utility system, the model is quite sparse with 7.6% non-zero entries in the Jacobian and 1.6% non-zero entries in the Hessian. The structure of this model remains the same for each of the following examples, however the numerical constants and available equipment are modified based on the example utility requirements.

Note also that the optimized cost of the model is not the point of this case study, which is about whether the optimization model in this work converges to the same operating point as in the original reference. To check this, key variables such as shaft work, power balance, make up water flow, and mass flows through turbines will be compared. In addition, installed equipment is still free to 'switch off', so the optimizer must also choose to correctly switch on all equipment to match the original reference. Finally, as the original work did not exploit electricity prices, the price of selling electricity is reduced to zero, so the optimizer has no incentive to generate extra electricity.

As well as comparing the two optimizer model solutions, each example is built in JSteam and the SymUtility operating point inserted. This provides a rigorous simulation (including full thermodynamic engine) to validate that the solution obtained is in fact feasible.

All optimization examples are solved using BONMIN.

6.6.2.1 Example 1

The optimal solution for the first example in [46] is shown in Figure 6.17 with utility demands listed in Table 6.10. Optimization results are shown in Table 6.11, noting that a similar result is shown in the original reference. Observable differences include an extra 3 tonne/hr drawn through the turbo generator, because the SymUtility model calculates a lower efficiency (74% versus 77%). This has a flow on effect throughout the model because the system requires more HP steam and demineralized water than the reference system, however the optimal point is effectively the same.

Table 6.10: Hypothetical 4 header model example 1 utility demands and resources.

HP User	0 kW
MP User	22000 kW
LP User	55000 kW
Electricity	5737 kW
Mechanical 1	1500 kW
Mechanical 2	1200 kW
WHB Duty	0 kW

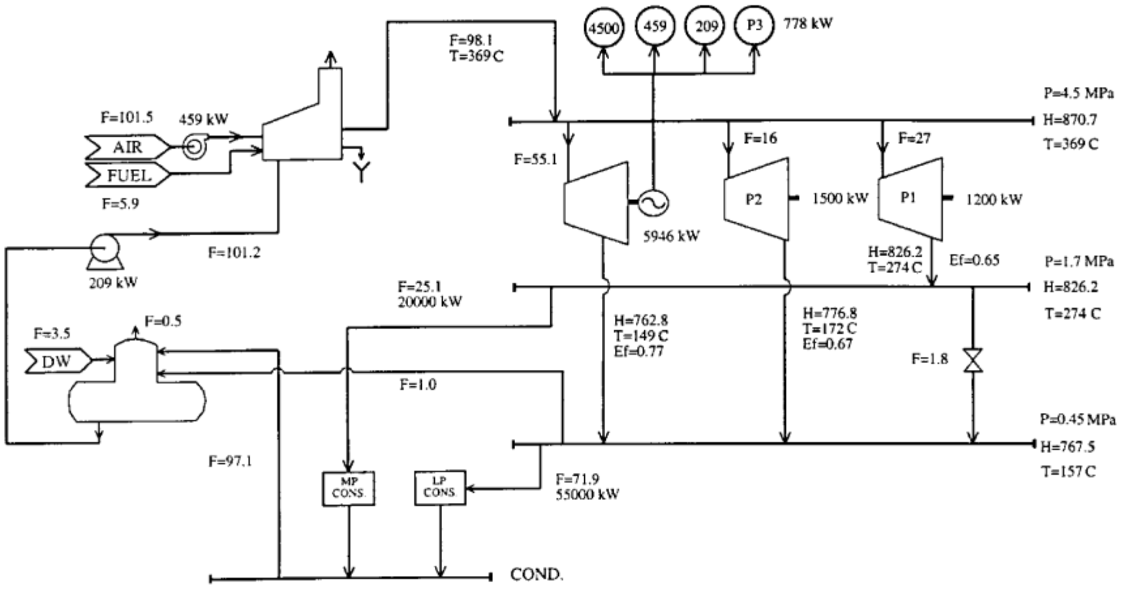


Figure 6.17: Hypothetical 4 header model example 1 system (Figure 4 in [46]).

Table 6.11: Hypothetical 4 header model example 1 optimization results.

	SymUtility	JSteam	Bruno et al
Build Time	8s	-	-
Solve Time	0.1s	-	-
Nodes	0	-	-
HP Steam Demand	102 tonne/hr	102 tonne/hr	98.1 tonne/hr
Boiler Fuel	6.1 tonne/hr	6.1 tonne/hr	5.9 tonne/hr
Power Balance	0 kW	0 kW	0 kW
TG1 Shaft work	5944 kW	5944 kW	5946 kW
TG1 Mass Flow	58.1 tonne/hr	58.1 tonne/hr	55.1 tonne/hr
P1 Mass Flow	16.2 tonne/hr	16.2 tonne/hr	16 tonne/hr
P2 Mass Flow	27.7 tonne/hr	27.7 tonne/hr	27 tonne/hr
Water Mass Flow	8.1 tonne/hr	8.1 tonne/hr	3.5 tonne/hr
HP Header T	369°C	369°C	369°C
MP Header T	274.3°C	274.3°C	274°C
LP Header T	158.8°C	158.9°C	157°C

6.6.2.2 Example 1b

The optimal solution for example 1 with lower heating demands is shown in Figure 6.18, with utility demands listed in Table 6.12. Note that the mass flow for the LP user reported in the original reference appears to be incorrect, because it does not match process user models used within the remainder of the examples (i.e 100% of available duty used). For this reason the return enthalpy of the LP user has been set to 735 kJ/kg, in order to match the required mass flow.

In addition, the reference solution has the MP header temperature fed via a letdown valve without cooling, thus raising the header temperature above the turbine exhaust temperature. This is unusual in practice, and will be modelled within this work as a desuperheater with cooling water. The result is that the header temperature will be lower than the reference work, however this has only a small impact on the final result.

Shown in Table 6.13 are the results for the optimization of this system, where once again the SymUtility solution approximately matches the Bruno et al solution.

Table 6.12: Hypothetical 4 header model example 1b utility demands and resources.

HP User	0 kW
MP User	16000 kW
LP User	30000 kW
Electricity	5763 kW
Mechanical 1	1200 kW
Mechanical 2	1500 kW
WHB Duty	0 kW

6.6.2.3 Example 2

The optimal solution for the second example is shown in Figure 6.19 with utility demands listed in Table 6.14. This example includes the gas turbine with secondary fired HRSG for HP steam production, as well as a large waste heat boiler supplying MP steam. In addition, two large process steam users draw over 150 tonne/hr of steam from the lower two headers.

The optimization results for this example are shown in Table 6.15, illustrating again that the SymUtility solution closely matches the reference solution.

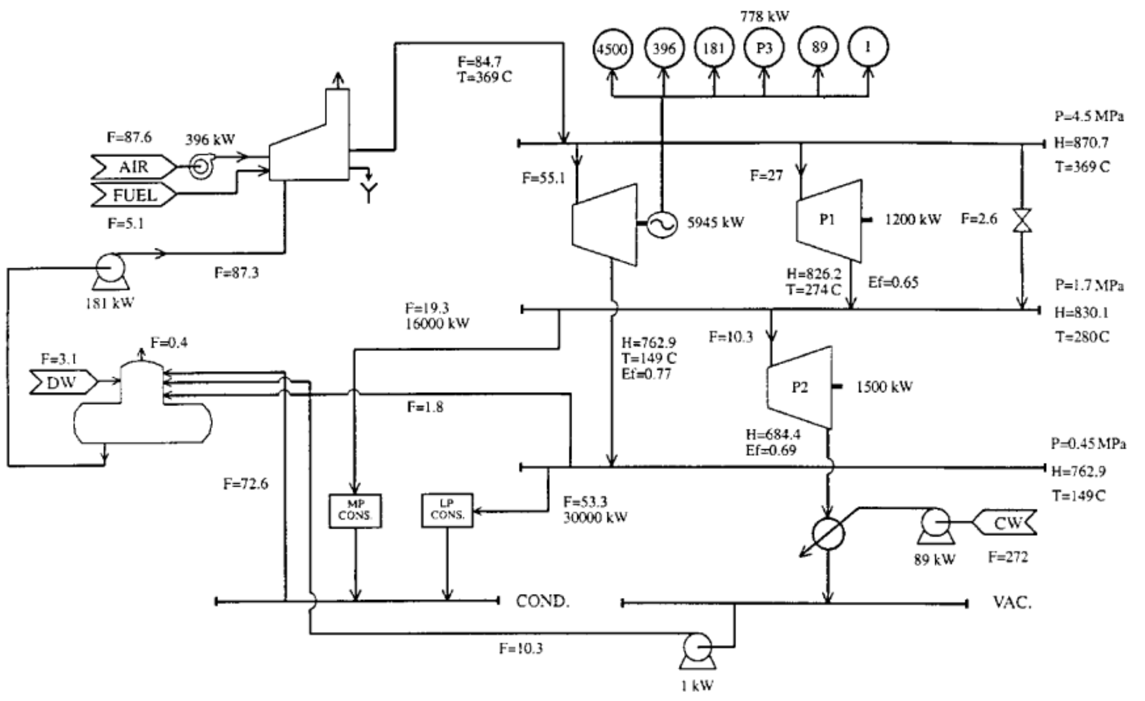


Figure 6.18: Hypothetical 4 header model example 1b system (Figure 5 in [46]).

Table 6.13: Hypothetical 4 header model example 1b optimization results.

	SymUtility	JSteam	Bruno et al
Build Time	8s	-	-
Solve Time	0.1s	-	-
Nodes	0	-	-
HP Steam Demand	87.7 tonne/hr	87.7 tonne/hr	84.7 tonne/hr
Boiler Fuel	5.3 tonne/hr	5.3 tonne/hr	5.1 tonne/hr
Power Balance	0 kW	0 kW	0 kW
TG1 Shaft work	5941.9 kW	5941.9 kW	5945 kW
TG1 Mass Flow	58 tonne/hr	58.1 tonne/hr	55.1 tonne/hr
P1 Mass Flow	27.7 tonne/hr	27.7 tonne/hr	27 tonne/hr
P2 Mass Flow	10.4 tonne/hr	10.4 tonne/hr	10.3 tonne/hr
Water Mass Flow	5.6 tonne/hr	5.6 tonne/hr	3.1 tonne/hr
HP Header T	369°C	369°C	369°C
MP Header T	274.3°C	274.3°C	280°C
LP Header T	155.5°C	155.6°C	149°C

Table 6.14: Hypothetical 4 header model example 2 utility demands and resources.

HP User	0 kW
MP User	31500 kW
LP User	85500 kW
Electricity	36648 kW
Mechanical 1	1800 kW
Mechanical 2	4540 kW
Mechanical 3	3120 kW
WHB Duty	52000 kW

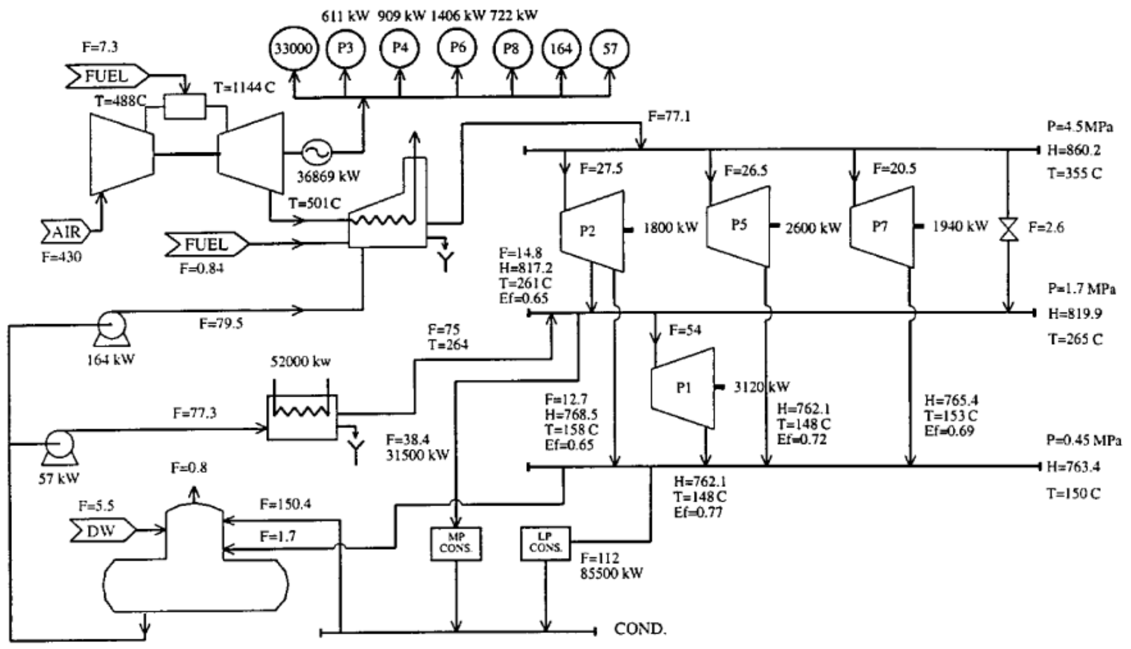


Figure 6.19: Hypothetical 4 header model example 2 system (Figure 6 in [46]).

Table 6.15: Hypothetical 4 header model example 2 optimization results.

	SymUtility	JSteam	Bruno et al
Build Time	8s	-	-
Solve Time	0.75s[1]	-	-
Nodes	10	-	-
HP Steam Demand	76.7 tonne/hr	76.8 tonne/hr	77.1 tonne/hr
GTG Shaft work	36956 kW	36956 kW	36869 kW
GTG + HRSG Fuel	8.76 tonne/hr	8.74 tonne/hr	8.1 tonne/hr
Power Balance	0 kW	0 kW	0 kW
P1 Mass Flow	54.1 tonne/hr	54.1 tonne/hr	54 tonne/hr
P2 S1 Mass Flow	17.5 tonne/hr	17.5 tonne/hr	14.8 tonne/hr
P2 S2 Mass Flow	11.2 tonne/hr	11.2 tonne/hr	12.7 tonne/hr
P5 Mass Flow	47.7 tonne/hr	47.7 tonne/hr	47 tonne/hr
Water Mass Flow	4.8 tonne/hr	4.8 tonne/hr	5.5 tonne/hr
HP Header T	355°C	355°C	355°C
MP Header T	263.5°C	263.5°C	265°C
LP Header T	149.9°C	150°C	150°C

[1] Solved with BONMIN's Branch and Bound Algorithm

6.6.2.4 Example 3

The optimal solution for the third example is shown in Figure 6.20 with utility demands listed in Table 6.16. Note that the HRSG steam temperature is lower than the other examples, so this has also been modified in the SymUtility model. The optimization results for this example are shown in Figure 6.17 where again the solutions match quite well.

The largest difference is the allocation of steam between the stages of dual stage turbine P1, where the SymUtility model has chosen to extract the most steam from the second stage. This operational point prevents the need for a letdown between the MP and LP header, as included in the reference work, with the sacrifice of less steam available in the MP header. However, because as the power production potential from HP-LP is greater than HP-MP, extracting most of the steam via stage 2 appears to be an optimal operating point.

Table 6.16: Hypothetical 4 header model example 3 utility demands and resources.

HP User	0 kW
MP User	15000 kW
LP User	40000 kW
Electricity	36903 kW
Mechanical 1	2200 kW
Mechanical 2	1600 kW
Mechanical 3	1200 kW
Mechanical 4	4000 kW
Mechanical 5	700 kW
WHB Duty	0 kW

6.6.2.5 Summary

The 4 examples contained within this case study have verified that the optimized solutions returned by the SymUtility modelling framework, together with BONMIN as the MINLP solver, approximately match solutions obtained in earlier literature. In all cases the optimizer has chosen to enable the same set of equipment as per the Bruno paper, and has matched the mass flows through key pieces of equipment, within tolerances expected by the different modelling strategies. Furthermore, the solution is obtained in an average of 400ms, indicating even semi-complex utility models with multiple pieces of equipment can be solved robustly and within fractions of a second.

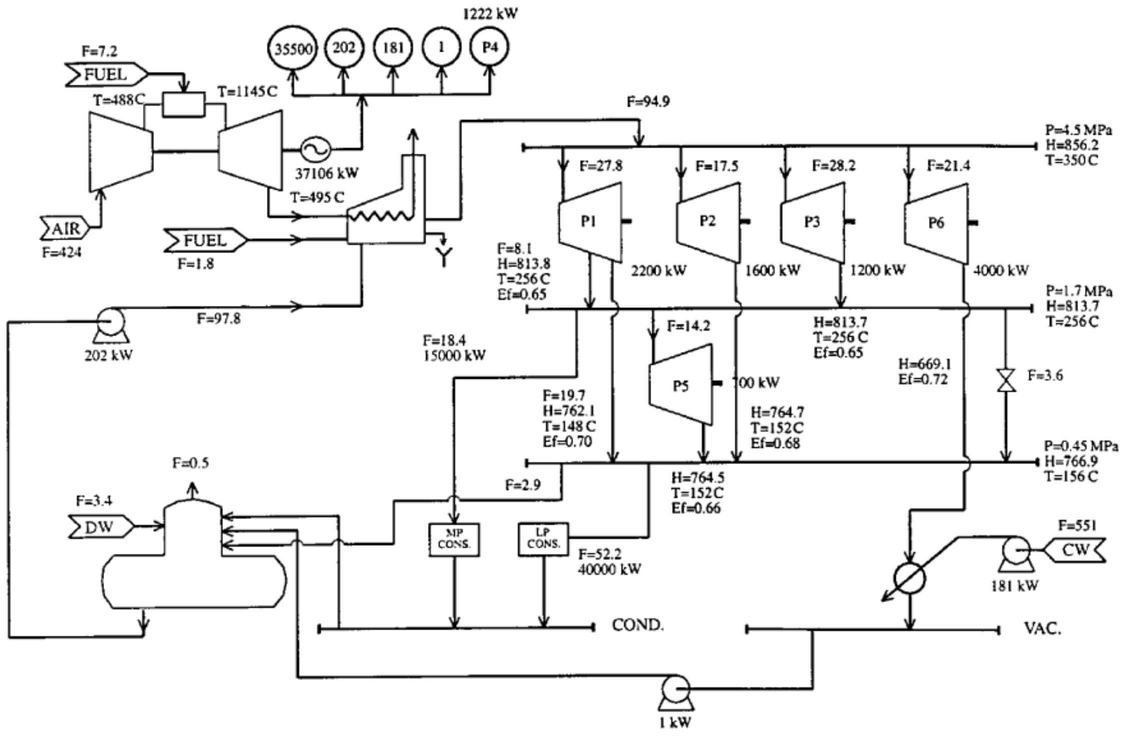


Figure 6.20: Hypothetical 4 header model example 3 system (Figure 7 in [46]).

Table 6.17: Hypothetical 4 header model example 3 optimization results.

	SymUtility	JSteam	Bruno et al
Build Time	8s	-	-
Solve Time	0.68s[1]	-	-
Nodes	6	-	-
HP Steam Demand	94.5 tonne/hr	94.5 tonne/hr	94.9 tonne/hr
GTG Shaft work	37096 kW	37096 kW	37106 kW
GTG + HRSG Fuel	9.73 tonne/hr	9.72 tonne/hr	9 tonne/hr
Power Balance	0 kW	0 kW	0 kW
P1 S1 Mass Flow	1.4 tonne/hr	1.4 tonne/hr	8.1 tonne/hr
P1 S2 Mass Flow	23 tonne/hr	23 tonne/hr	19.7 tonne/hr
P2 Mass Flow	17.6 tonne/hr	17.6 tonne/hr	17.5 tonne/hr
P3 Mass Flow	28.9 tonne/hr	28.9 tonne/hr	28.2 tonne/hr
P5 Mass Flow	14.4 tonne/hr	14.4 tonne/hr	14.2 tonne/hr
P6 Mass Flow	21.4 tonne/hr	21.4 tonne/hr	21.4 tonne/hr
Water Mass Flow	3 tonne/hr	3 tonne/hr	3.4 tonne/hr
HP Header T	350°C	350°C	350°C
MP Header T	256.7°C	256.7°C	256°C
LP Header T	151.2°C	151.2°C	156°C

[1] Solved with BONMIN's Branch and Bound Algorithm

6.6.3 Industrial Petrochemical Utility System

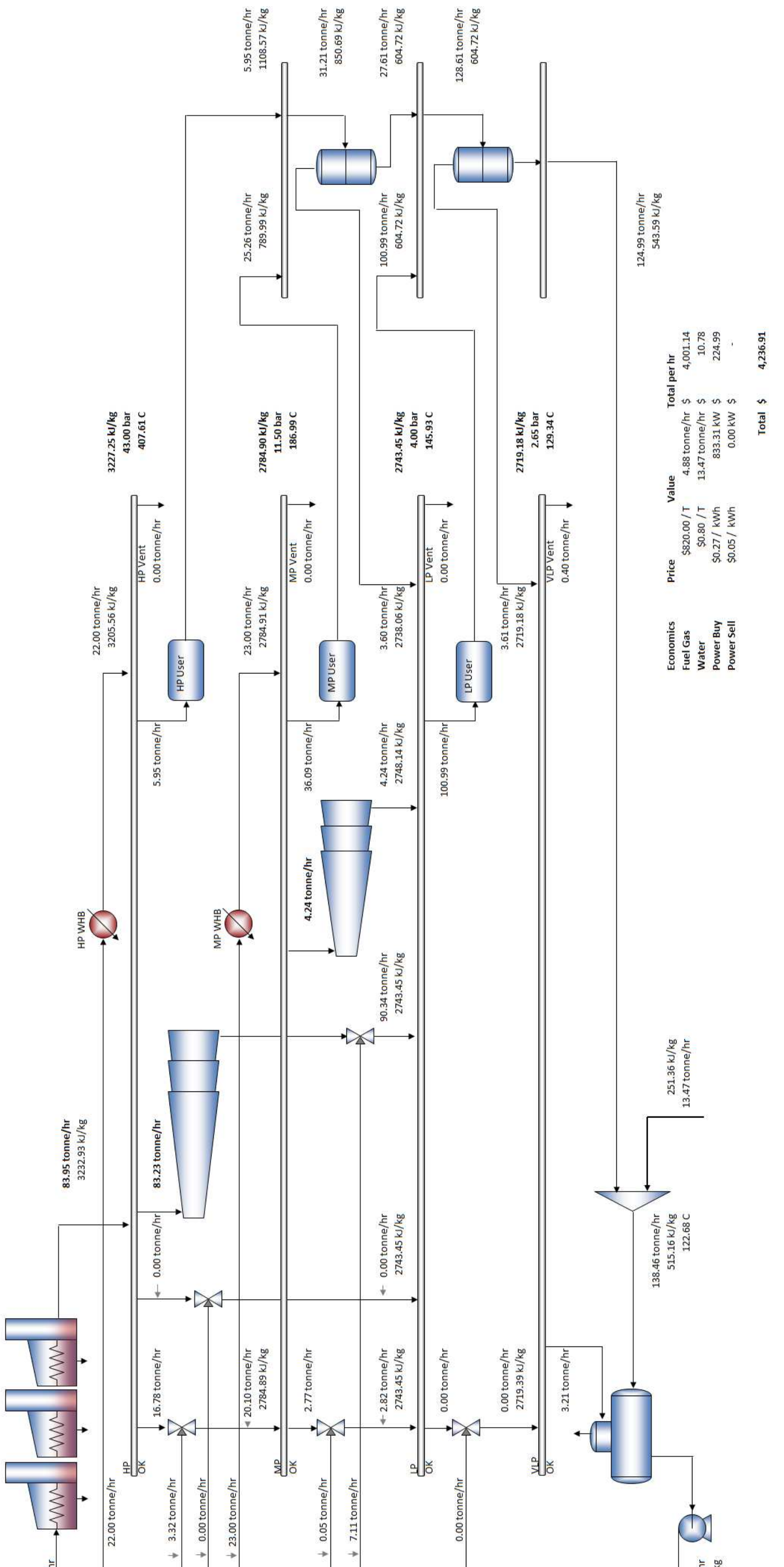
The final case study presented is an actual industrial utility system for a petrochemical plant in Malaysia. This particular system was one of the validation systems used within the iUO project [71] (Section 5.2), and its operation can be validated against original PETRONAS engineering models.

As this is a commercial utility system, the exact specifications of the plant cannot be published. However we were granted permission to publish a case study of this system in our early work [69], which included a PFD, and specifications which can be gleaned from that paper are listed in Table 6.18. The four system headers operate at 43, 11.5, 4 and 2.65 bar for the HP, MP, LP, and VLP headers respectively, and the system includes a comprehensive condensate collection with two flash drums to recover useful steam for the lower headers, as shown in Figure 6.21.

Table 6.18: Industrial utility system specifications.

HP Boilers	3x identical small steam boilers (rated at less than 50 tonne/hr) run on natural gas composed of primarily Methane and Hydrogen. All three boilers generate 410°C, 43 bar steam with a 1% continuous blowdown ratio.
HP WHB Boiler	Supplies 22 tonne/hr of 400°C, 43 bar steam.
MP WHB Boiler	Supplies 23 tonne/hr of 187°C, 11.5 bar steam.
HP Turbines	14x small (less than 500 kW) and 2x medium (less than 3MW) back pressure turbines, each with redundant electric motor.
MP Turbines	3x small (less than 20 kW) back pressure turbines, each with redundant electric motor.
BFW Pump (PMP)	47 bar outlet pressure, 70% isentropic efficiency.
HP Steam User (HPU)	3.5MW duty based user, returns 100% of steam as saturated condensate.
MP Steam User (MPU)	20MW duty based user, returns 70% of steam as saturated condensate.
LP Steam User (LPU)	60MW duty based user, returns 100% of steam as saturated condensate.
Deaerator	2.65 bar with 0.01 continuous vent ratio.
Make Up Water	2.65 bar, 60°C.

The optimization opportunity for this particular utility system is quite limited, given that there is no cogeneration on site. This is typical of older systems which are designed to supply heat and mechanical demands only. To realise economic savings with this particular system becomes predominantly a driver selection problem, whereby given the current electricity price, a choice must be made between which



Economics	Price	Value	Total per hr	
Fuel Gas	$820.00 / T	4.88 tonne/hr	$	4,001.14
Water	$0.80 / T	13.47 tonne/hr	$	10.78
Power Buy	$0.27 / kWh	833.31 kW	$	224.99
Power Sell	$0.05 / kWh	0.00 kW	$	-
		Total	$	4,236.91

Figure 6.21: Industrial utility system base case.

mechanical loads to drive via electric motors, and which to drive via steam turbines, as described in Section 6.4.1.

With the model entered into SymBuilder, the following statistics are reported

```
SymBuilder Object
 BUILT in 4.357s with:
 - 96 variables
 -  1 objective
      -  0 linear
      -  0 quadratic
      -  1 nonlinear
 - 72 constraint(s)
      - 35 linear
      - 34 quadratic
      -  3 nonlinear
 - 192 bound(s)
 - 23 integer variables(s)
      -  0 integer
      - 23 binary
```

noting that due to the large number of steam turbines, there is also a larger number of binary variables required. The 96 variables are composed of 66 mass flow variables, 7 enthalpy variables, 19 steam turbine binary variables, 3 binary variables for the boilers and the power balance binary variable (which could be left out because no power will be exported, however it has been retained for future retrofit opportunity testing).

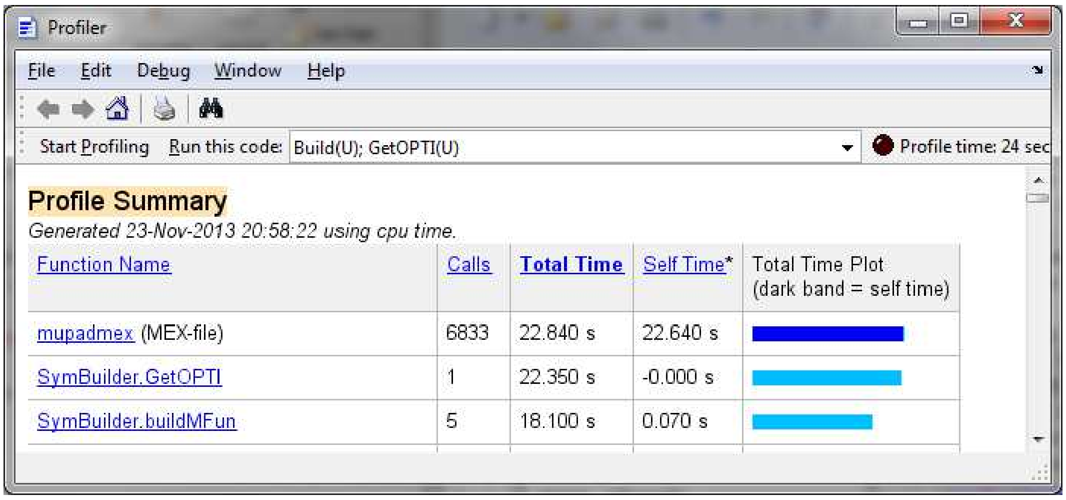
Optimizing the SymBuilder model using BONMIN's Branch and Bound solver returns the results listed in Table 6.19. As expected, the small and inefficient steam turbines (less than 50 kW) have been switched off by the optimizer, and two boilers are now run closer to full load, with the third switched off in order to maximize efficiency. The optimal point reduces the hourly operating cost by \$82.90, which is around 2%, in only a few seconds. This figure aligns with operational optimization savings reported in literature (such as [80, 94]), where savings of 1-2% are common.

Table 6.19: Industrial utility system optimization results.

	Base Case	Optimized
Optimized Cost	\$4128	\$4045.1
Build Time	-	24.2s
Solve Time	-	3.6s
Nodes	-	114

It is worth commenting on the relatively long build time observed within this example. By profiling the MATLAB code being executed, Figure 6.22 shows that over

94% of the time is spent within the MATLAB Symbolic Toolbox (within MUPAD) substituting variables required for building the MATLAB callback functions. This result shows that further code optimization is required in order to make SymBuilder competitive for generating callback functions for larger problems.


Profiler
File Edit Debug Window Help
Start Profiling Run this code: Build(U); GetOPTI(U)
Profile time: 24 sec

Profile Summary

Generated 23-Nov-2013 20:58:22 using cpu time.

Function Name	Calls	Total Time	Self Time*	Total Time Plot (dark band = self time)
mupadmex (MEX-file)	6833	22.840 s	22.640 s	
SymBuilder.GetOPTI	1	22.350 s	-0.000 s	
SymBuilder.buildMFun	5	18.100 s	0.070 s	

Figure 6.22: MATLAB profiler report for building the industrial utility system symbolic model showing the symbolic engine takes the most time (`mupadmex`).

6.6.4 Case Study Summary

These three case studies have demonstrated the capability of the three software packages developed for this work: JSteam for rigorous thermodynamics and utility modelling in Excel, OPTI for supplying a suite of optimizers and the 'glue' to convert from a standard problem description into solver specific formats, and lastly SymBuilder for creating optimizer targeted algebraic models with analytical derivatives. By using these tools rigorous models of three utility systems have been built, optimized, and validated against results from both literature and industry.

Furthermore, optimization of all but one problem was completed within 1 second, with the worst case taking 3.6s for the industrial utility system, all on a standard laptop computer. This result, combined with the fact we are using both an open-source optimizer and an open-source optimization platform, both at no cost, shows economically significant results can be achieved even with modest software and hardware. Moreover, by utilizing the framework developed in this work to allow accurate off-design models to be regressed, exploiting the structure of the resulting utility system model, and then tailoring the resulting optimization problem to a nonlinear solver, speedups of over 1500 times have been realised by 'simply' optimizing a simulation model, as well as the ability to robustly find better solutions using analytical derivatives.

6.7 Summary

The framework-based approach described in Chapter 1 has been heavily utilized to realise the results presented in this chapter, whereby OPTI has enabled significant results to be obtained from complex, multi-faceted optimization problems. Practically significant results have been achieved with minimal computational time by combining the power of MATLAB as a modelling environment, together with the software developed in this work; JSteam for high-speed thermodynamics and detailed off-design utility models, OPTI which provides a highly efficient interface to 3rd party optimization solvers as well as automating low-level optimization task such as problem identification and solver setup, and finally SymBuilder for algebraic model generation with symbolic derivatives. Furthermore, while JSteam is specifically focused on industrial utility systems, both OPTI and SymBuilder were written with flexibility in mind, meaning they can be applied to most optimization and modelling problems, with industrial models, detailed simulation models or even real equipment connected for optimization. These are therefore a significant contribution to the wider field of optimization and operations research, and not just within the steam utility field.

A final question needs to be answered when analysing the results of the case studies is: "Have we found the global optimum, or are we stuck in a local minimum?" Given that we know the problem is non-convex based on its formulation (see Section 6.3.3), it is therefore quite possible that the solutions found so far are local minima, and that better more economically interesting results may exist. These however would need to be proved. Rather than approach this question using standard 'global' techniques such as genetic algorithms or evolutionary algorithms which give no guarantee of a global optimum, the following chapter will detail white-box optimization strategies that prove global optima.

Chapter 7

Global Optimization of Energy Systems

This chapter aims to prove deterministically that the optimums found in the proceeding chapter are indeed the global optimums. In order to achieve this aim, two new OPTI interfaces are formulated to two advanced white-box global optimization solvers, SCIP and BARON. Within each interface, an algebraic structure of the optimization problem is collected from the MATLAB model and supplied to the solver. By utilizing this algebraic description, a global white-box solver is able to prove a global optimum to a deterministic, general nonlinear problem, and thus we can confidently make the statement: "There is no better solution to this problem".

The chapter details the development of each of the interfaces for SCIP and BARON, and proves their accuracy via a series of benchmark tests against common global optimization problems. It then concludes by repeating the utility system optimization case studies from the previous chapter using each of the new solvers, and compares the results achieved.

7.1 White-Box Optimization

General nonlinear optimization, such as that performed in the previous chapter, is deemed black-box because as far as the optimizer is concerned, the internals of the nonlinear objective and constraint functions are unable to be exploited. This black-box simply accepts a decision variable vector and returns an objective value or vector of constraint evaluations. The underlying equations, structure, and relationships between the decision variables (inputs) and objective or constraints (outputs) are effectively unknown, with the exception of the derivatives, which describe how a change in input produces a change in the output.

In contrast, linear and quadratic optimizers have a rigid mathematical format which the solver is tailored to exploit and solve. Furthermore, because a linear or quadratic program is described natively using numerical matrices and vectors, the solver has a full problem description which includes all problem data. This means that higher levels of pre-processing, such as scaling, redundant constraint removal and bound tightening can be applied. In addition, the standard problem definition of a linear or quadratic program requires the problem to be convex, further assisting in tailoring the internals of the solver to exploit matrix properties (such as positive-definite systems of linear equations). This all combines to make linear and quadratic solvers typically much faster and more robust than their general nonlinear counterparts, and they are therefore capable of solving much larger problems within a reasonable time frame.

To be able to leverage the same level of problem information for a nonlinear problem requires an algebraic description of the problem supplied to the solver. This effectively 'opens-up' the back-box so that the optimizer can see inside and thus results in the term 'white-box'. By exploiting the algebraic description, a white-box solver is able to much more effectively pre-process the problem, which can vastly reduce the search space required. In addition, a white-box solver may recognise mathematical features of the supplied functions, such as identifying monomials or polynomials, linear, bilinear and multilinear relationships or other common nonlinear expressions such as `log` and `exp`. This enables the white-box solver to be able to exploit problem relaxations and solution heuristics, such as outer approximations [88], convex/concave envelopes [102], a Generalized Benders Decomposition (GBD) [1] and/or search space cutting planes [122, 151]. Furthermore, by using a suite of mathematical rules based on the structure of the problem, it can prove deterministically a global optimum to general nonlinear problems, provided that the problem meets certain formulation requirements.

Given the requirement of an algebraic description, both the objective and constraint functions must be deterministic, that is contain no stochastic terms or conditional statements. Furthermore, the library of functions that can be used is typically limited to only a handful, such as `log`, `exp`, `abs` and `sign`, noting the omission of all trigonometric functions at present. This requirement rules out many problems, and especially ones within MATLAB that utilize toolbox functionality such as interpolation, integration, mixed data types, and problems calling external code such as Simulink or JSteam.

With the restrictive space in which these solvers can be applied, especially within an environment such as MATLAB which is much more flexible than a traditional optimization language such as GAMS, we were hesitant to pursue applying either of these solvers to the utility system problems. Moreover, none of the available white-box solvers had a MATLAB interface suitable for this work (the exception was the LINDO Global solver with a rudimentary interface), meaning that once again substantial development work would be required. Note it is acknowledged YALMIP [189] contains a white-box global optimization solver, however it is intended primarily for bilinear problems which limits its use for the general nonlinear problems of interest within this work.

A chance meeting with Nick Sahinidis of Carnegie Melon at FOCAPO 2012 in Savannah, Georgia, provided the opportunity to develop a MATLAB interface to BARON [304], a global MINLP solver being developed by his research team. Nick had noticed our work developing OPTI and acknowledged that a MATLAB interface would complement his current GAMS interface. I accepted the contract that night after studying the documentation and a MATLAB interface was developed over the succeeding months.

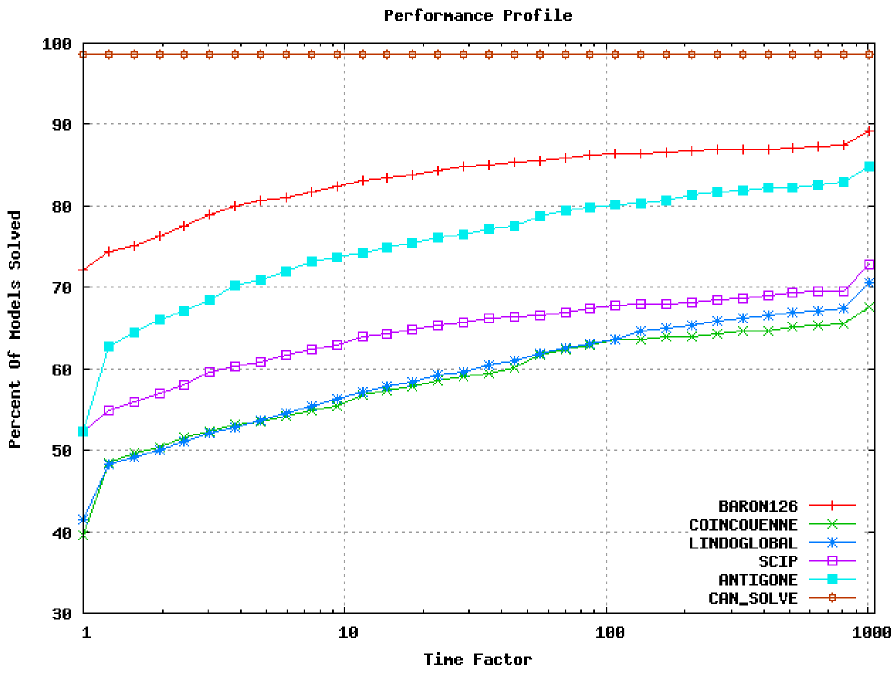


Figure 7.1: A comparison of 1599 global test problems solved with leading deterministic global optimization solvers [290]. Only BARON and SCIP will be used for this work.

7.2 Generating an Algebraic Description from MATLAB

At the time of writing we are unaware of any packages that perform the required task of generating an algebraic description of a problem written in native MATLAB code. This is not to say it cannot be done with the MATLAB Symbolic Toolbox (which SymBuilder attempted to do), however the task is to provide a high-speed toolbox independent implementation that does not use an underlying symbolic engine. The closest package that fits this requirement is YALIMP [189], which provides near-native MATLAB syntax together with a custom class (`sdpvar`) to derive an algebraic model description.

While modification of the sdpvar class was a possibility, YALMIP is not released for commercial use. In addition, the `sdpvar` class was *too* flexible in its implementation and it supported functions not supported by a global optimizer. As a result it was decided that it would be easier to create a new, custom class object.

It is worth noting that string parsing of raw MATLAB functions was investigated, however it was deemed this approach would be more time consuming than writing a custom class object, considering the multitude of ways a MATLAB program can be written.

7.2.1 MATLAB – BARON Interface

Given that BARON is a commercially available solver, it already had a robust parser for reading in optimization models described in BARON format. This meant interfacing at the code level to the solver was not required, rather that the interface was primarily a convertor from MATLAB code to BARON format, with the added ability to call the solver and retrieve the results. To illustrate the differences between formats, consider the nonlinear optimization problem from Section 6.2.3.3,

$$\begin{aligned} \min_{\mathbf{x}} \ & \log(1+x_1^2) - x_2 \\ \text{subject to:} \ & \left(1+x_1^2\right)^2 + x_2^2 = 4 \end{aligned}$$

This problem could be entered into MATLAB as two anonymous functions

```
% Objective
obj = @(x) log(1+x(1)^2) - x(2);

% Nonlinear Equality Constraint
```

```
nlcon = @(x) (1+x(2)^2)^2 + x(2)^2;
cl = 4; cu = 4;
```

while the same problem entered in BARON format could look like (a simplified GAMS language)

```
MODULE: NLP;

VARIABLES x1,x2;

EQUATIONS e1;

e1: (1 + x2^2)^2 + x2^2 == 4;

OBJ: minimize log(1 + x1^2) - x2;
```

The primary difference between the two formats is the method for declaring variables. In MATLAB a vector is used, and each variable indexed into the expression, while in BARON each variable is a scalar, and each must be individually declared. In addition, there are subtle model changes required for terms such as described in Table 7.1.

Table 7.1: MATLAB/BARON format differences.

MATLAB	BARON
x^y	$\exp(y\log(x))$
$\lvert x\rvert$	$(x^2)^{0.5}$
$\log_{10} x$	$0.434294482\log x$
$x\hat{}y\hat{}z = (x\hat{}y)\hat{}z$	$x\hat{}y\hat{}z = x\hat{}(y\hat{}z)$

In order to provide the required transformations in Table 7.1, as well as to convert MATLAB vectors to BARON scalars, a MATLAB class was written. The class overloads all common matlab operators, as well as all supported functions for use with BARON. At each operation, the class converts the MATLAB operation (whether matrix, vector or scalar) into an equivalent scalar operation and then saves the result as a string (or vector/matrix of strings, operation dependent). This is best illustrated using the BARON vector (`barvec`) class

```
x = barvec(1,1,'x');
y = barvec(1,1,'y');
z = barvec(2,2,'z');

e1 = x^y
e2 = abs(magic(2)*z)
```

where `e1` returns a scalar expression and `e2` returns a matrix

```
>> e1
Scalar BARVEC Object
Eq : exp(y1*log(x1))

>> e2
Vectorized BARVEC Object: 2 x 2
Eq(1,1) : ((1*z1 + 3*z2)^2)^0.5
Eq(2,1) : ((4*z1 + 2*z2)^2)^0.5
Eq(1,2) : ((1*z3 + 3*z4)^2)^0.5
Eq(2,2) : ((4*z3 + 2*z4)^2)^0.5
```

To use the `barvec` class object for optimization problems, it is simply declared as a vector with the same size as the numerical decision variable vector, and then passed to the objective and constraint functions

```
% Declare BARVEC object
x = barvec(2,1);

% Pass To Earlier Nonlinear Optimization Problem
bobj = obj(x)
bnlcon = nlcon(x)
```

and returned as BARON compatible equation declarations

```
>> bobj
Scalar BARVEC Object
Eq : log(1 + x1^2) - x2

>> bnlcon
Scalar BARVEC Object
Eq : (1 + x2^2)^2 + x2^2
```

The `barvec` class effectively performs the conversion as each operator and function is called, automatically deciding where brackets are required and performing the required transformations to ensure the problem is compatible with BARON. Moreover, as the class overloads most common MATLAB operators, the user is still free to use vectorized constructs to describe their problem, and the class will generate a scalarized implementation

```
n = 3; % Any length vector
x = barvec(n,1);

% Vectorized Rosenbrock Objective
j = sum((1-x).^2) + sum(100*(x(2:n) - x(1:n-1).^2).^2);


>> j
```

```
Scalar BARVEC Object
Eq : (1 - x1)^2 + (1 - x2)^2 + (1 - x3)^2 + 100*((x2 - x1^2)^2) +
     100*((x3 - x2^2)^2)
```

In addition, users are free to implement for-loops to describe their problem as long as the termination of the loop is not dependent on a decision variable. Functions can be declared as anonymous or general MATLAB functions, and can return a scalar or vector and still be compatible, given the object-orientated approach used. The complete list of supported functions is listed in Table C.1 in Section C.4.1, noting that considerable attention has been paid to supporting matrix operations. The object supports operations up to 2D, and replicates MATLAB functionality (including optional second arguments) as closely as possible.

With the comprehensive list of functions available, the aim is that the user does not have to modify their optimization functions in order to solve their problem using BARON. The interface also self validates the generated BARON function against the original MATLAB function, ensuring the representation passed to BARON is correct.

It is acknowledged that this interface does not (by definition) generate an algebraic description of the MATLAB function, but, by converting to a format which has an algebraic structure which can be parsed by BARON, it has achieved this aim. The remainder of the MATLAB to BARON Interface developed simply writes the generated expressions to a `.bar` file, including generating the required variable and equation definitions, calls the BARON executable, then parses the BARON results file to return the results to MATLAB. Considering the amount of overhead involved in generating and concatenating equation strings, the process is remarkably quick, as detailed in Section C.4.2.

Solving optimization problems using the MATLAB - BARON Interface is a simple single function call, similar in form to standard Optimization Toolbox functions

```
[x,fval,ef,info] = baron(fun,A,rl,ru,lb,ub,nlcon,cl,cu,xtype,x0,opts)
```

where `fun` is the MATLAB objective function, `A,rl,ru` are the linear constraints, `lb,ub` are the decision variable bounds, `nlcon,cl,cu` are the nonlinear constraints, `xtype` is the decision variable integrality, `x0` is an optional initial solution guess and `opts` are BARON options, available via `baronset`. The function uses the `barvec` class to generate the BARON compatible functions internally, and then solves the problem and returns the results via the output arguments. In addition, BARON is interfaced via OPTI so any compatible OPTI problem can be solved to global optimality using BARON, as shown below.

```
% Non-convex Polynomial Objective
fun = @(x) x^6 - 2.08*x^5 + 0.4875*x^4 + 7.1*x^3 - 3.95*x^2 - x + 0.1;
% Bounds
lb = -1.5; ub = 1.5;
% Initial Solution
x0 = 0.5;
% Options
opts = optiset('solver','baron');

% Create OPTI Object
Opt = opti('fun',fun,'bounds',lb,ub,'options',opts)
% Solve using BARON
[x,fval,exitflag,info] = solve(Opt,x0)
% Plot Solution within Problem Bounds
plot(Opt,[lb ub])
```

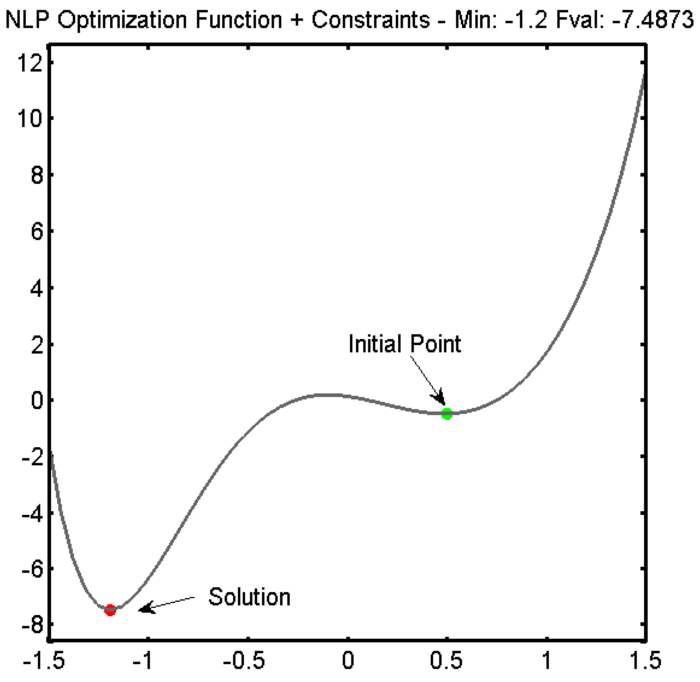


Figure 7.2: Non-convex polynomial solved with BARON using OPTI Toolbox.

Substantial verification of the MATLAB - BARON Interface was undertaken by both Nick Sahinidis and myself, with well over 500 global nonlinear and mixed integer nonlinear problems written in MATLAB (or parsed from existing models) and then solved with the interface. Currently the interface is in use by a few users dedicated to finding global optima, however it has not been officially released. This is expected in early 2014.

If the reader is interested in trying out the MATLAB - BARON Interface a demonstration BARON executable and MATLAB Interface can be obtained from `www.minlp.com/download`. Instructions on the OPTI Wiki `www.i2c2.aut.ac.nz/Wiki/OPTI/index.php/Solvers/BARON` detail how to install the solver and interface. Note that the demonstration executable utilizes IPOPT with MUMPS as the linear solver, and CLP as the linear programming solver. This is the same version as will be used for benchmarking global results later in this chapter (BARON can also utilize CPLEX, SNOPT, MINOS and other commercial solvers if purchased).

7.2.2 MATLAB – SCIP Interface

The second global optimization solver used within this work is SCIP, which is an open source solver framework for finding the global solutions to MINLPs. While the original focus for this solver was on solving MIQCQPs, experimental support has also been added for general nonlinear problems. The framework is maintained by a large team of researchers at Zuse Institute Berlin (ZIB), as well as by contributors from various other German universities. The project is released for at no cost under an academic licence, and under a licensing agreement for commercial users.

Work on the interface for SCIP began after a request from Johan Löfberg (of YALMIP [189]) to add the ability to solve MILPs in SCIP within MATLAB. A rudimentary interface did exist but it appeared this was not a priority for the developers. Interestingly enough, SCIP is perhaps best known as a MILP solver, and reportedly the fastest non-commercial MILP solver when benchmarked by Hans Mittelmann [220]. To be able to solve MILPs, SCIP includes its own LP solver, SoPlex, which is an implementation of the revised simplex algorithm. To solve quadratic and nonlinear problems, SCIP leverages IPOPT for solving relaxed problems together with CppAD for generating derivatives.

Beginning in January 2013, we began to investigate what would be required to upgrade the MATLAB - SCIP Interface to solve NLPs and MINLPs, which at the time was limited to MIQCQPs only. In order to supply a nonlinear model to SCIP, the model must be described by an algebraic expression tree within C code. This is in contrast to BARON, which provided a parser that automatically decoded and created its own internal algebraic expression. This meant an interface from MATLAB to SCIP would have to be able to generate a full algebraic description of the MATLAB function, and then using this description, generate an SCIP expression tree for each nonlinear function.

To illustrate the requirements of entering an expression tree into SCIP, consider the following bilinear expression

$$2x_0x_1$$

which is entered within SCIP as (ignoring error catching constructs)

```
//Declare SCIP Expressions and Variables
SCIP_EXPR *exp0, *exp1, *var0, *var1;

//Create Variables and assign indices
SCIPexprCreate(SCIPblkmem(scip), &var0, SCIP_EXPR_VARIDX, 0);
SCIPexprCreate(SCIPblkmem(scip), &var1, SCIP_EXPR_VARIDX, 1);

//Create '2*x0' and store in exp0
SCIPexprCreateLinear(SCIPblkmem(scip), &exp0, 1, &var0, 2.0, 0.0);

//Create (2*x0)*x1 and store in exp1
SCIPexprCreate(SCIPblkmem(scip), &exp1, SCIP_EXPR_MUL, exp0, var1);

//Finalize Expression Tree
SCIPexprtreeCreate(SCIPblkmem(scip), &exprtree, exp1, 2, 0, NULL);
```

As can be seen, each variable is declared within SCIP and then a series of functions are used to generate a series of intermediate expressions. Once the final expression has been completed, it is saved into an expression tree and then later loaded as a constraint within the SCIP framework. What is clear from this example is that a very low level description of the nonlinear function is required, which includes every numerical constant, operator, function, and the order in which they are all used.

Based on this requirement, I decided to develop what I termed an 'instruction list' based approach to describing MATLAB functions. By once again utilizing a MATLAB class with overloaded operators and functions, the SCIP interface would build up a list of instructions that described the function, instead of building up equation strings (as per the BARON Interface). Incidentally after 6 weeks of development my supervisor informed me that I had inadvertently (re)invented Reverse Polish Notation (RPN) [50], an established method of entering functions into portable calculators common in the 1970s and 1980s. Even though the method was not novel I continued to develop the interface, including a C based interpreter for converting the instruction list to an SCIP expression tree (in reality, an RPN/postfix interpreter).

The following code snippet demonstrates the functionality of the SCIP instruction list object, `scipvar`, for converting MATLAB code to an RPN-like instruction list,

```
% Declare SCIP converter object
x = scipvar(5,1)
% Convert to instruction list (RPN)
obj = x(5) + ((x(1) + x(2)) * x(4)) - x(3)

Scalar SCIPVAR Object
VAR   : 4
VAR   : 0
VAR   : 1
ADD   : NaN
VAR   : 3
```

```
MUL   : NaN
ADD   : NaN
VAR   : 2
SUB   : NaN
```

noting that the order of instructions is exactly the same (coincidentally) as if described in RPN. Variable indices are deliberately decremented because the index will be used by the C interpreter directly, while `NaN` is simply a place holder for the second column.

Within the object the instructions are stored as a column vector of numbers

$$\begin{bmatrix} 1 & 4 & 1 & 0 & \ldots & 1 & 2 & 6 & \texttt{NaN} \end{bmatrix}^T$$

where the order is (instruction,value/index,instruction,value/index,...) and so forth.

Table 7.2: `scipvar` operation numbering scheme.

0	Constant
1	Variable
2	-
3	Multiplication
4	Division
5	Addition
6	Subtraction
7	Square (^{2})
8	Square root
9	Power
10	Exponential
11	Natural Logarithm

A simple numbering scheme relates each instruction number to an instruction, as detailed in Table 7.2. This single column vector contains a complete algebraic description of any compatible MATLAB function, albeit in a format hard to manually convert to the original equation.

Complementing the `scipvar` object is a state-machine implemented in C which interprets the instruction list vector, creates the required intermediate SCIP expressions, and then finally creates the complete expression tree. This interpreter, `scipnlmex.cpp`, which is supplied with OPTI in Solvers/scip/Source, maintains two stacks as it processes the instruction list, one for expressions and one variables, resulting in a complex system of pushing and popping based on what was last saved.

To ensure the MATLAB function was correctly converted to an instruction list, and then correctly converted to an SCIP expression tree, the resulting expression tree is evaluated at a random (or user supplied) test point, and the result compared to the same test point applied to the MATLAB function. This provides a simple validation method which indicates the conversion process was successful, but it is noted this method is not failsafe.

This conversion process is applied to each nonlinear function and an individual expression tree is added for each constraint and objective. Because SCIP only solves problems with a linear objective, a nonlinear objective is added as a constraint and a connecting variable links it back to the objective.

The MATLAB - SCIP Interface described so far is still in beta, with no code optimizations performed and no memory management within the MATLAB object. Table C.2 in Section C.4.3 lists the functions currently available, with more to be added as time allows.

7.2.3 Global Optimization Solver Interface Validation

To validate the interfaces developed to BARON and SCIP, 16 small test problems have been taken from GLOBALLib [103], a collection of global optimization problems, and 16 from MINLPLib [51], a collection of mixed integer nonlinear problems. These problems represent the types of problems (albeit smaller) that could be expected within the optimization of utility systems, and thus form a representative set. Table C.3 in Section C.4.4 lists the results of the interface validation, indicating all 32 problems for both solvers are correctly parsed and supplied to each solver.

7.3 Comparison of Global and Local Solutions of Utility Optimization Case Studies

By utilizing the two global optimization solvers now available via MATLAB and OPTI, the solutions to the utility optimization case studies in Section 6.6 can be reviewed for global optimality. This step is required if we want to categorically prove that the solutions obtained are indeed the global optimum, given that the problems of interest are both non-convex and contain integer variables (which also infer the problem is non-convex). Recalling that solutions obtained in the previous chapter were obtained using BONMIN and IPOPT, both of which are designed for convex problems only (BONMIN solves mixed integer problems where the underlying nonlinear problem is convex), the results so far may be local minima only.

7.3.1 The Three Header Hypothetical System Global Results

Repeating the three header example from Section 6.6.1, Table 7.3 lists the optimization results when solved with BARON and SCIP. A notable observation when comparing the optimized costs between BONMIN (local solutions, global if lucky) and BARON (deterministically global) is that the global optimum has been achieved in every case using a local optimizer. This result is discussed further in the summary (Section 7.4). Furthermore, solution times averaging 10 seconds are required to solve this problem to global optimality using BARON, or around 1 second using SCIP.

It is also noted that the time to build the SymBuilder problem for both SCIP and BARON is reduced, because SymBuilder recognises that both these solvers do not require derivatives (they are automatically solved internally), and therefore callback functions do not need to be generated.

Table 7.3: Hypothetical 3 header model global optimization results.

Solver		Base Case	Case 1	Case 2	Case 3
BONMIN	Optimized Cost	\$1831.9	\$-5889.5	\$3204.8	\$-1982.6
(Local)	Build Time	4.6s	4.6s	4.5s	4.6s
	Solve Time	0.11s	0.1s	0.09s	0.1s
	Nodes	0	0	0	0
SCIP	Optimized Cost	\$1831.9	\$-5889.5	\$3204.8[1]	\$-1982.6
(Global)	Build Time	3.2s	3.2s	3.2s	3.2s
	Solve Time	1.1s	1.76s	0.99s	0.97s
	Nodes	421	888	103	166
BARON	Optimized Cost	\$1831.9	\$-5889.5	\$3204.8	\$-1982.6
(Global)	Build Time	3.2s	3.2s	3.2s	3.2s
	Solve Time	9.9s	12.35s	10.7s	5.13s
	Nodes	113	391	891	135

[1] Obtained used SCIP + MATLAB 32bit with a solver tolerance of 1×10^{-10}

One result which took quite a bit of effort to obtain was the SCIP result for the second case. All other results in this Chapter were obtained using 64bit SCIP together with a linear program feasibility tolerance of 1×10^{-7} (a linear program is used to solve the relaxed problems generated by SCIP), which for the majority of problems works sufficiently well. However this problem would only solve with the 32bit version, together with an increased tolerance. This problem of differing results between architectures had been raised in the past with Stefan Vigerske, one of the authors of the MINLP component of SCIP. His response was "In the 32bit version, the primal heuristics seem to have become more lucky, so it found a feasible solution at node 181. It could be that the difference in platforms can lead to such a difference and there isn't much I can do about it - it's not a bug." [317]. For now, this discrepancy is left open for further investigation.

7.3.2 The Four Header Hypothetical System Global Results

The hypothetical model from Section 6.6.2 is re-optimized using both SCIP and BARON and the results presented in Table 7.4. Once again, the solutions obtained via BONMIN are globally optimal, however do recall this is not guaranteed. In addition, SCIP has correctly solved all problems within this example using the default settings, and done so much faster than BARON.

Table 7.4: Hypothetical 4 header model global optimization results.

Solver		Example 1	Example 1b	Example 2	Example 3
BONMIN	Optimized Cost	$1371.6	$1180.5	$1954.4	$2171.0
	Build Time	8s	8s	8s	8s
	Solve Time	0.1s	0.1s	0.75s	0.68s
	Nodes	0	0	10	6
SCIP	Optimized Cost	$1371.6	$1180.5	$1954.4	$2171.0
	Build Time	5s	5s	5s	5s
	Solve Time	0.9s	0.98s	1.16s	4.19s
	Nodes	201	274	180	2995
BARON	Optimized Cost	$1371.6	$1180.5	$1954.4	$2171.0
	Build Time	5s	5s	5s	5s
	Solve Time	31.3s	17.8s	16.2s	60.0s
	Nodes	400	391	437	1027

7.3.3 The Industrial Utility System Global Results

The industrial petrochemical utility system model from Section 6.6.3 is also re-optimized using both SCIP and BARON and the results presented in Table 7.5. What is interesting is this time BARON is much faster than SCIP, a result that has not been seen in any of the other problems. It is expected that BARON would begin to out-perform SCIP as the problem sizes grow, such as in this problem, and given that this is typically a feature observed when comparing commercial solvers to open-source versions, as demonstrated in Figure 7.1. In addition, BONMIN has once again found the global minimum, and as expected, in less time than the global solvers.

Table 7.5: Industrial utility model global optimization results.

Solver		Base Case
BONMIN	Optimized Cost	\$4045.1
	Build Time	24.2s
	Solve Time	3.6s
	Nodes	114
SCIP	Optimized Cost	\$4045.1
	Build Time	11.8s
	Solve Time	134.3s
	Nodes	101879
BARON	Optimized Cost	\$4045.1
	Build Time	11.8s
	Solve Time	40.1s
	Nodes	827

7.4 Global Optimization Summary

This chapter has described two deterministic global optimization solvers that have been interfaced to MATLAB via two purpose built interfaces. Each interface has been developed to exploit the algebraic relationships within common MATLAB functions, enabling a deterministic global minimum to be robustly obtained from arbitrary nonlinear functions. For robust global solutions BARON has proved it is the best choice within the solvers surveyed. It is however the slower of the two for the problems tested and requires a commercial license.[1] However SCIP has also performed very well, solving all problems to global optimality and on average 12.7x faster than BARON. These results, combined with being freely available to academics, makes SCIP an attractive option that can be easily tried on any compatible OPTI problem.

The most interesting result from this study was that BONMIN found the global minimum in 9 out of 9 cases, and across 3 quite different utility models. This result was not expected, given that we had already identified that these models consisted of non-convex terms, as detailed in Section 6.3.3. The hypothesis is that predominantly the non-convexities are saddle points in hyperspace (due to the indefinite Hessian), and with the realistic starting guess provided by the JSteam model, BONMIN is starting on the 'right side of the saddle' and converging to the global solution.

It is not the expectation of this work that SymUtility will always find the global solution, but we can help the optimizer converge to the global solution by exploiting the model structure and providing a sensible starting guess.

[1]Nick Sahinidis was kind enough to supply a license during development of the interface.

Chapter 8

Conclusions and Further Work

8.1 Conclusion

This thesis has described a number of techniques to increase the performance of industrial optimization algorithms. Two industrial problems are presented which vary widely in physical size and dynamic speed, however by leveraging common frameworks developed within this work, both problems can be described at a high-level, yet retain the low-level performance improvements designed in this work. This is achieved by efficiently formulating the problem in a manner best suited to the solver, tailoring the solvers to the problems being solved, and lastly by utilizing the frameworks developed to provide the 'glue' between the high-level problem description and complexities of the implementation. For each industrial problem of interest, optimization performance using techniques developed in this work exceed performance results reported in existing literature, and where possible, results have been applied to real industrial systems to validate the solution.

In Section 1.3 we referred to four research questions within the field of industrial optimization that formed the significance of this work. The following subsections summarize the contribution of this thesis in addressing each of these questions, while Section 8.2 presents a critique of the questions to validate they have been answered successfully and reinforce the significance of this work.

8.1.1 Quadratic Programming Solver Results

Two primal-dual interior-point quadrating programming solvers have been developed, `quad_wright` and `quad_mehrotra`, which are based on algorithms and modifications described by Stephen Wright and Sanjay Mehrotra respectively. Both solvers have been specifically developed to solve the quadratic programming problems that result from a model predictive control formulation for small to medium constrained multivariable control problems. With respect to the dimensions of the resulting quadratic program, this bounds the problems of interest to an upper limit of 40 decision variables and 320 constraints, with the nominal problem size consisting of 20 decision variables and 160 constraints. These limits allow the model predictive controller to be tuned with larger prediction horizons ($N_c > 10, N_p > 30$) even for multivariable problems, allowing for improved control performance which is after all, the point of a controller. Moreover, based on the control formulation, the problem is predominantly dense (typically greater than 70%), and therefore sparse techniques are not useful.

Given the focus on developing an embedded model predictive controller, both algorithms have been developed with an end goal of deployment on embedded hardware, which for this work is a single precision hardware floating point microcontroller. This goal dramatically changes the design process for both algorithms because there are no longer multi-threaded linear algebra libraries, GHz clock speeds or GBs of RAM available, but rather all resources are considerably limited. It is for this reason a fully-unrolled implementation as is done in CVXGEN would not be applicable, and furthermore an explicit MPC formulation would not fit for the problems of interest.

Both solvers developed have been tailored to solve a series of consecutive (and similar) small, dense, linear inequality constrained quadratic problems in single precision floating point, and as fast and accurately as possible whilst using as little memory as possible. Techniques used have included benchmarking common algorithmic modifications such as exploiting rectangular bounds, utilizing higher derivative information to reduce iterations, and simplifying the internal linear system to determine the best combination of modifications that suit the problems of interest. From this investigation it was found the added complexity of treating rectangular bounds separately was less efficient for the problems of interest, however Mehrotra's predictor-corrector modification demonstrated reliable performance improvements. Furthermore, given that the quadratic programs to be solved result from consecutive optimal control problems, heuristics have been developed to warm-start the solver to reliably reduce the number of iterations required by approximately 20%, thus increasing the power efficiency of the algorithm.

Both solvers were profiled to identify potential hot-spots, and this identified the formulation of the linear system as the most computationally intensive step. This step was targeted with a symbolic engine to better understand the way the matrix of equations was generated, and an algorithm was developed to accelerate the computation by reducing the number of required operations. This modification increased the performance of the solver by over 5 times the traditional (mathematical) implementation, with no loss in accuracy or extra memory required. This was achieved by only calculating the lower triangular elements, as well targeting efficient code to access and process the numeric data.

In addition, an investigation has been carried out into the behaviour of the solvers when presented with an infeasible problem, which is not unreasonable to expect within a constrained dynamic controller. Heuristics were developed to recognise both typical infeasibility, as described in the literature, and numerical problems associated with infeasibility, such as negative pivots within the Cholesky factorization routine. One could therefore exit the routine before unrealistic control moves were calculated. A case study showed that even when presented with consecutive infeasible problems, the `quad_wright` algorithm maintained partial control of the system, whereas a traditional quadratic solver lost complete control.

To combat reduced precision on an embedded platform, a scaling heuristic was developed which substantially improved the performance of both algorithms within single precision. A case study demonstrated the heuristic reduced required solver iterations by over 30%, as well as preventing most solver failures caused by numerical errors. Furthermore, termination conditions were proposed that took into account the reduced precision, to ensure the solver would not attempt to reduce iterates below the native numerical precision of the embedded system.

Referring back to the first research question posed in Section 1.3, the techniques described in this section have enabled the `quad_mehrotra` solver to solve literature model predictive control problems at sampling rates exceeding 10kHz on modest embedded hardware. This required only 9.1KB to store the entire controller in memory, and achieved a solution accuracy within a fraction of a percent of Cplex (when compared in double precision). For problems with the nominal size of 20 decision variables and 160 constraints, both `quad_wright` and `quad_mehrotra` outperform in terms of solution speed all surveyed interior-point quadratic programming solvers, including those which utilize industry standard BLAS and LAPACK linear algebra libraries. Furthermore, an implementation of this size requires less than 75KB of memory to store, including the algorithm and all problem data, which allows for deployment on a range of small hardware packages.

8.1.2 Embedded MPC Results

By utilizing the rapid-prototyping nature of MATLAB, together with the object-orientated functionality, the jMPC Toolbox provides a framework for describing, tuning, simulating and then generating high-speed linear embedded model predictive controllers. The framework enables an MPC controller to be described succinctly in a high-level language, allowing the user to focus on the modelling and tuning, rather than on the intricacies of how the controller is formulated. In addition, the framework provides a simulation and validation platform which allows controllers to be simulated within both linear and nonlinear environments, within code or Simulink, and then validated against a reference controller to establish sensible tuning parameters. Furthermore, the framework provides the ability to quickly generate an embedded controller, automatically optimized for the type of control problem entered, which can then been validated on real hardware via a processor-in-the-loop validation study.

In order to leverage the work developing the quadratic programming solvers `quad_wright` and `quad_mehrotra`, the framework includes an automatic code generator which can automatically generate a deployable MPC controller in ANSI C, including one of these solvers which have been tailored for the problem entered. The code generator combines the best of the high-level MATLAB language, which allows the user to complete the problem description using the `jMPC` object, with high-speed hand-optimized C code templates used to implement both the quadratic programming solver and controller engine. In this way the controller description remains succinct, while the resulting implementation is still hand-optimized, and allows a memory efficient and high performance solver to be generated with minimal code required. Furthermore, given the problems are in a standard format, the code generator customizes both the solver and controller engine based on the problem description. Examples include removing redundant constraint calculations, precalculating the Cholesky factorization of the unconstrained system, and removing any unused functionality such as disturbance prediction modelling. In total, the time required to build and then subsequently generate a embedded MPC controller is typically less than 1 second. Using Texas Instrument's Code Composer Studio, the controller can then be compiled in approximately 8 seconds, which means that a new controller can be built and tuned from scratch, then generated, compiled and deployed in as little as 10 seconds, allowing rapid re-tuning iterations. This is opposed to an FPGA development route, where re-tuning may require a complete recompilation process that may take 100 times longer, severely limiting this rapid re-tuning.

To validate the automatically generated controller, the framework includes two methods: Automatic validation on the development PC and secondly, a processor-in-the-loop (PIL) module. By default, when the controller is generated, both the C code quadratic solver and the complete controller are automatically compiled into MATLAB MEX functions, and the results of the functions compared against reference implementations. Because the reference implementations are selected to match the functionality within the generated controller and solver, compiled with the same compiler and settings, and then run on the same development PC, bit accurate results are obtained. This first method quickly indicates the generated code and problem data is correct on a PC, however it does not guarantee the same performance when run on an embedded target. To validate the controller on the target hardware, the framework provides a second method whereby a PIL implementation can be automatically run, where the controller and solver are both implemented on the target hardware, and the development computer acts as the plant to control. This enables any plant model, including a full nonlinear model, to be used in the validation step and provides the most accurate estimate of the performance of the generated controller before connecting it to a real plant. To illustrate this functionality, a simulated case study of the control of an inverted pendulum on a moving cart was implemented, with the controller run on a Texas Instruments C28343, and a nonlinear simulation of the pendulum run on the development computer. The embedded controller obtained a sampling rate of close to 300Hz, while maintaining nearly identical control performance to that run on the development computer alone.

Concluding the investigation into high-speed embedded controllers, an actual hardware case study was presented which applied an automatically generated MPC controller to a physical laboratory helicopter. This study encapsulated all of the contributions of this work to high-speed embedded optimization. These included utilizing the high-level jMPC Toolbox for automatically deriving a linear model for the controller, automatic code generation of a quadratic programming solver tailored for the resulting control problem, a PIL validation step to validate the generated code, and finally proof that the system worked on an actual multivariable system. The controller was implemented with a long prediction horizon of 80 samples (which is significant for an embedded MPC implementation), achieved a sampling rate exceeding 100Hz on hardware costing less than US$160, required just 56KB of RAM to store, and achieved the expected control performance. This proved that the combination of techniques and tools developed are a significant contribution to this field of work.

8.1.3 Design of a Common Optimization Framework for MATLAB

The concept for the OPTI Toolbox was born from recognising that the open-source optimization community was substantially under-utilized by the majority of industrial users. The toolbox aimed to bridge the gap between the 'industrial MATLAB user', deemed a competent engineer or technologist whose primary interest and expertise is in their own specific domain, and those users being able to solve real world industrial optimization problems. To fulfill this aim, MATLAB had to be able to solve optimization problems with binary and integer variables, common within typical industrial problems, which meant that new optimization solvers would have to be made available within MATLAB. At the time of writing, the OPTI Toolbox adds 24 solvers from the open source community to MATLAB users, including 6 solvers able to handle binary and integer constraints.

The majority of solvers added did not have an existing MATLAB interface, or if they did, they were either very limited in functionality, or broken due to changes in the MATLAB MEX API. One of the contributions of the toolbox was not only to collect together and compile the solvers for Windows MATLAB users, but also to develop MEX interfaces to access as much of the functionality of the solver as possible. The problem with this approach was that every solver expected a different problem format, which when further complicated by hundreds of unique configuration options, would have rendered the toolbox complex and restricted to specialist users.

The solution here was to define a common problem format which provided a conversion layer between an easy to use, easy to describe optimization format, and the configuration requirements of each solver. The framework developed introduced the `opti` object, a MATLAB class that allowed any one of the 16 supported optimization problems to be entered (such as linear, quadratic or nonlinear programs), then automatically identify the problem entered, select the best solver available to solve it, and configure the problem to match the requirements of the solver. This approach is very similar to that used in the development of the jMPC Toolbox, whereby a high-level problem description is entered, and the object automatically configures itself to tailor the selected solver to best solve the problem, using low-level code. Furthermore, the toolbox provides a suite of auxiliary tools for assisting in the solution of common industrial optimization problems. These include solution inspection via automatically generated plots, solution validation against the supplied constraints, automatic derivative generation if not available, file reading and writing routines and numerous other tools.

One issue identified when modelling problems natively in MATLAB was that given its flexible language, it was very easy to poorly formulate optimization problems. Furthermore, given the language does not contain algebraic constructs, problem simplification and analysis is not easily achievable. This complicates solving nonlinear problems because the generation of derivatives is limited to numerical approaches, which can affect the accuracy of the solution. This issue was addressed within this work by creating an extension to the Symbolic Toolbox called Sym-

Builder, which is a language extension to allow optimization problems to be posed more succinctly. The advantage was that by utilizing a symbolic engine, the optimization model could be symbolically simplified and analytical derivatives obtained, which then allowed a full picture of the problem being solved to be found. Using this information, an optimization solver can exploit much more information than if posed as a MATLAB function 'black-box', such as constraint linearity, sparsity patterns, and accurate derivatives. This technique was utilized to solve industrial scale utility optimization problems in fractions of a second.

In addition to the solver framework and symbolic modelling extensions, the toolbox provided MATLAB users with the ability to deterministically calculate a global optimum to general non-convex (mixed integer) nonlinear problems. Two methods of generating algebraic representations of MATLAB functions were developed, together with two interfaces to leading global optimization solvers, BARON and SCIP. By utilizing the OPTI Toolbox a user can obtain global solutions to problems without changing their problem description or format, thereby substantially reducing the barrier to obtaining the best possible solution to an industrial problem.

8.1.4 Energy Optimization Results

The second industrial optimization problem focuses on the optimal operation of combined heat and power utility systems. When considering optimizing the operation of a utility system, a key requirement is the availability of part-load models of major unit operations, such as steam boilers, turbines, gas turbines and heat recovery steam generators. All of these units exhibit both varying efficiencies as a function of the rated output, and as a function of the unit size and manufacturer specifications. Therefore manufacturer design data, rated outputs or even the current state alone cannot be used to predict the operation of a piece of equipment over a range of loads/outputs, which is critical to being able to move the operating point within the optimizer.

To address this issue, this work has built on the work of Mavromatis, Shang, Kokossis, Varbanov, Aguilar and others, re-regressing published data to models proposed by these authors, and combining it with the JSteam rigorous thermodynamic engine and a library of mass and energy balance models. Utilizing the JSteam models and thermodynamics, unit operations can be fitted to a specific operating region, while industrial data can be used to determine the part load performance curves. Furthermore, the JSteam fuel gas models have the facility to include operational constraints such as minimum stack temperature, exchanger approach temperatures, and minimum excess O_2. It is therefore argued that combining the literature models, industrial data, and combustion models developed in this work results in a more accurate representation of the real equipment.

A derived class based on SymBuilder has been developed, SymUtility, which provides the same symbolic optimization modelling functionality but which is further customized for modelling steam utility systems. Methods have been added to automatically add most common pieces of utility equipment, including the part-load models which are automatically generated from supplied operational specifications, which allows the user to concentrate on modelling the complete system. The result is an industrial utility system can be described in as little as 50 lines of MATLAB code. This includes a complete mass and energy balance of the system, detailed part-load models of all equipment, and being based on SymBuilder, is automatically simplified, sparse first and second derivatives are generated, and a chosen solver tailored to suit the problem entered (such as identifying mass balance equations as linear, energy as bilinear, etc).

This once again comes back to the common framework approach, whereby a high-level language is used to describe the problem, while the framework manages the detail of the optimization problem and automatically exploits features present in the problem to aid the optimizer solving it. Using SymUtility together with OPTI and JSteam, utility systems from both literature and a real petrochemical system are optimized, with significant economic savings obtained in typically a fraction of a second. Recalling that the systems being optimized are nonlinear, non-convex, contain binary variables in addition 50 or more continuous variables and 100 or more constraints, solutions obtained on a standard 2.8GHz i7 laptop within 0.7 seconds to these systems is a significant contribution, especially when the solutions are both economically significant and feasible when validated in a rigorous nonlinear model. Furthermore, when compared to the status-quo of optimizing a process simulation model, speed-ups of over 1500 times are observed, indicating that considerable effort has been made to tailoring the problem for the optimizer.

Further investigation into the optimality of the solutions obtained was carried out, specifically looking at whether a local minima had been found. By utilizing the global optimization framework developed as part of the OPTI Toolbox, the same utility system models were re-optimized and the global optimum proven for each. In all nine case studies undertaken, it was proved that the solution found by BONMIN (a local solver) was indeed the global optimum, which was not expected, considering the models had been identified as non-convex. However this result does add to the argument that tailoring the problem to the optimizer, as has been done here, can not only increase solver performance with respect to solution time, but also increase the chance of the solver correctly finding the global minimum, which is an attractive benefit. It is not to say this will always be the case when using BONMIN with these models, but it is more likely when a realistic starting guess is available from the JSteam model, in conjunction with a tailored formulation.

8.2 A Critique of the Research Questions Originally Posed

To conclude the contributions resulting from this work, a review of the research questions from Section 1.3 is presented below. Each research question is addressed in the original order, and described within the context of the results of this work.

1. By using the `quad_mehrotra` solver and MPC algorithm developed in this work, together with the USD$159 Texas Instruments C28343 microcontroller, Section 4.5.3.2 demonstrated a processor-in-the-loop implementation of embedded MPC that achieved a sampling rate of over 10kHz, while requiring less than 10KB of memory, all in single precision. Furthermore a case-study where the predictive control was applied to a laboratory-scale helicopter was demonstrated (Section 4.6), proving that both the software and hardware developed in this work could robustly control real-world systems. This result exceeded all aims of this research question.

2. Developed within the course of this research, the jMPC Toolbox (Section 3.5.2) allows a high-level description of a linear model predictive controller to be formulated in MATLAB. Using the `jMPC` object, the controller could be tuned, simulated and validated within a rapid-prototyping development environment, before being automatically-coded into an ANSI C controller. By utilizing a combination of both hand-coded and hand-optimized code templates (Section 4.3), together with automatically generated problem specific routines and data structures, the performance and memory benefits of hand-optimized code could be realised within a high-level controller description. In addition, the framework designed is both easier to use and provides better performance for the problems of interest, than recent overlapping research, as demonstrated in Section 4.4.5.

3. The OPTI Toolbox, together with the packaged SymBuilder platform (Sections 6.2 and 6.5), allows complex optimization problems to be posed succinctly within MATLAB, yet be symbolically simplified and then solved with state-of-the-art solvers. By extending the functionality of the MATLAB Symbolic Toolbox to better handle optimization problems, together with a suite

of routines for simplifying and generating optimization models, SymBuilder allows industrially significant mixed integer nonlinear programs to be solved in fractions of a second. Furthermore, by utilizing the common optimization framework developed within OPTI Toolbox, an optimization problem can be posed independently of the solver used, yet still exploit known structural properties of the problem when solved by a compatible solver.

4. Based on industrial experience, JSteam (Section 5.3) allows an industrial user to rigorously model industrial-scale steam utility models using established mass and energy balances, together with a powerful thermodynamic engine. To then apply this technology to optimization, Sections 5.4-5.7 develop part-load unit operation models based on industrial data and regressions from industry, allowing operational optimization to be performed. Moreover, by including the part-load models within SymUtility, a subclass of SymBuilder, large-scale utility system models could be symbolically generated, then optimized using freely available solvers to achieve economically significant results in less than 5 seconds. All optimizer solutions were validated against equivalent rigorous JSteam thermodynamic results, indicating the optimizer was correctly solving physically realisable operating points.

8.3 Recommended Further Work

There are several areas where further development could improve the performance of both the optimization speed and robustness, as well as the functionality of the frameworks developed. However, given the large scope of this work across two significant optimization problems, it has not been possible to include every idea within the scope of this research. Therefore the areas described below are left as further work, and are intended to be completed as time allows.

8.3.1 QP Algorithm

Two quadratic programming algorithms have been developed within this work, however the decision when to use each algorithm has not been fully explored. As shown in Section 4.4.4.1, it is clear that `quad_wright` is faster for smaller problems, while `quad_mehrotra` is faster for larger problems. While a threshold could be developed that decides which algorithm to implement based purely on size, it would be worthwhile to examine further properties such as problem numerical conditioning, potential for infeasibility, termination tolerances, and decision variables versus constraints ratio, in order to make the best possible decision. Furthermore, a quadratic programming solver which implements Gondzio's higher order correction steps should be designed and added to the solver selection strategy. It is anticipated that for larger problems, Gondzio's modification would assist in the reduction of the number of iterations required, as demonstrated when comparing the `qpip` solver to the `quad_mehrotra` solver in Figure 3.16.

In addition, the active-set solution strategy has not been investigated at all within this work, based on the reasons set out in Section 3.4. This decision was made very early on in this research, and it is anticipated that recent advancements in the active-set algorithm may make it quite competitive for this work, especially given that it is better suited to solving consecutive similar quadratic programs, as found in an MPC controller.

A more advanced scaling algorithm would examine the conditioning of the original model state-space matrices and correctly scale the as the quadratic program is formulated, rather than afterwards, such as done in this work. An example of two strategies include one by Liuping Wang where she describes an exponentially weighted cost function modification in [326], while Rossiter et al proposed an inner-loop state feedback stabilization strategy in [150]. In addition, using the current heuristic, the selection of $\text{scale}_{\text{fac}}$ does impact the robustness of the algorithm and thus should be investigated further. This work is currently being undertaken by a Master's student based on this work.

A further idea yet to be investigated is what information can be utilized between consecutive solver runs, given that only the objective linear term ($\mathbf{f}$) and constraint right hand side ($\mathbf{b}$) change. This is a remarkable property that must be able to be exploited to either aid finding a solution, or at least used in a heuristic for choosing initial values to warm-start the solver.

8.3.2 Embedded MPC Implementation

A noticeably absent topic within this work is a focus on stability of the controller, given that this is typically the focus of most control studies. It was decided early on in this work that ensuring closed-loop stability of the resulting controller would detract from the aim of the development of the associated optimization software, and given both the MPC algorithm and descriptions of stability are well known in literature [164, 214, 273], it would not be investigated. However, it would be interesting to investigate the effect of reduced precision on the closed-loop stability of an MPC controller, given that the target implementation will typically be in single precision, at best. Furthermore, given that the controller communicates to the outside world via analog to digital and digital to analog convertors, what effect varying levels of quantisation would have on ensuring stability, such as discussed in [348].

While infeasibility of the control problem was 'handled' by the quadratic programming solver, in reality the solver would simply exit with the last 'least-infeasible' solution found. A more robust strategy should be developed, such as automatically applying soft constraints when an infeasible problem is detected, in order to retain control of the system. There is also room for further work on developing techniques to handle what happens when the quadratic solver runs out of time, based on the sampling rate being faster than the solution rate.

There is also considerable recent research into nonlinear model predictive control [120] which given advances in nonlinear programming algorithms [323], is now competitive for sub-minute sampling rates [76]. While implementing a full nonlinear model predictive controller on the hardware used within this work may be unreasonable, by utilizing modern ARM processors [346] such as those found within smart phones (with GHz clock rates and ample off-chip memory), it is not unrealistic to expect to implement a full nonlinear programming solver and collocation formulated nonlinear program within an embedded controller. However, without requiring a full nonlinear MPC, there may still be significant advantages in allowing bilinear models [40], and solving via a linear matrix inequality strategy [39, 44], which is much more realistic on the same embedded hardware used within this work.

Another area of recent research is the high-speed MPC control of motor driver circuits [109], specifically those which use multiple FETs to generate poly-phase currents. As each FET can only be on or off (it is a discrete electronic device), the control problem is now an integer quadratic program where all decision variables are binary (also known as a Hybrid MPC problem). This now becomes a combinatorial problem which, as shown in the energy optimization work, is significantly harder to solve. However, given that this work encompasses optimization of both continuous and discrete systems, only simple modifications would be required to allow binary variables within the MPC framework, which could then be solved via one of the OPTI solvers. It is however much harder to implement a MIQP solver on embedded hardware, because it is likely a branch-and-bound solver would need to be written, and when implemented, would substantially limit the achievable sampling rate. This is an active area of research [25], and could be assisted by the tools and algorithms developed in this work.

A further idea is leveraging the now common dual-core hardware available in modern microcontrollers. Rather than completely changing architecture to an FPGA, parallelism is available in much easier to use and program microcontrollers and should be exploited within the MPC algorithm. While traditional approaches have targeted the matrix multiplication and linear algebra routines for parallelism, we investigated an alternative hybrid of PID and MPC which we called 'interpolated MPC' in [67]. By utilizing a PID controller in parallel with the MPC controller, the sampling rate of the combined controller could be accelerated. This is achieved by using the PID controller to provide intermediate control moves, while the MPC controller is calculating the next major control moves. Further work is required to determine whether this is a feasible (and stable) strategy.

8.3.3 Optimization Framework

There are a range of topics identified for further work within the OPTI Toolbox, especially in conjunction with other researchers. These are summarized in the file `OPTI_ToDo.m`, located in the OPTI Toolbox/Help directory supplied on the Appendix DVD. These include projects such as the the development of a new automatic differentiation framework, the development of a new symbolic engine for SymBuilder and the development of optimal control and dynamic optimization modules.

With respect to derivative generation with MATLAB, an additional method not yet explored is to use a C++ automatic differentiation framework such as CppAD [30] together with an algebraic representation of a MATLAB function. This follows ideas presented in the package BLOM [165], which solves large-scale nonlinear programs resulting from Simulink models and uses IPOPT together with an automatic sparse derivative representation.

A useful piece of functionality missing in the OPTI Toolbox is the ability to automatically identify linear and quadratic models, when supplied as general nonlinear functions. While this functionality exists via the SymBuilder framework, it is much harder to do with native MATLAB code. Further work could expand on the algebraic transformations used for the global optimization solvers, to automatically identify structure within the function being processed and allow improved decision making on the type of problem entered.

8.3.4 Energy Optimization

The most important future improvement to the energy optimization work would be to include multiple time periods into the optimization problem. This was intended to be included in the original scope of this work, however due to time constraints, was not able to be completed. By factoring in multiple time periods, the optimization problem could take into account costs of start-up and shut-down, as well as the forecasting of both weather and electricity prices. This would result in a much more accurate model and one which has real industrial opportunity.

In addition, the detailed unit operation models developed have been based on industrial data presented by a range of authors, much was of which was collected 20 to 30 years ago. As the technology within plant equipment has improved, the regressions used within this work will no longer accurately model modern utility systems, and it is therefore recommended that a new set of regressions be developed. Furthermore, the framework should include a facility for automatically generating optimization ready models from plant data, based on a data-reconciliation approach. This would allow the SymUtility models to be connected to a live distributed control system or Process Integration (PI) historian with access to real-time data, which would enable the user to conduct real-time optimization studies.

Further work investigating the combustion process is currently tasked for a masters or summer student, specifically looking at implementing the GRI-Mech [106] reaction database. This would provide a means to calculate the production of Nitric Oxide (NO), a combustion by-product which gas turbine operators are at pains to minimize.

Finally, we suggest further work could be undertaken in reducing the MINLP optimization model into a MIQCQP model. This would simplify the optimization problem to a form where the algebraic description is implicit in the formulation and aiding the solver finding a solution. The model would however remain non-convex, so that applicable solvers from the OPTI Toolbox would still be limited to the same SCIP and BARON solvers used in this work.

Bibliography

[1] A. M. Geoffrion. Generalized Bednders Decomposition. *Journal of Optimization Theory and Applications* (1972), 237–260.

[2] Achterberg, T. SCIP: Solving Constraint Integer Programs. *Mathematical Programming Computation 1*, 1 (2009), 1–41.

[3] Agha, M. H., Thery, R., Hetreux, G., Hait, A., and Le Lann, J. M. Integrated production and utility system approach for optimizing industrial unit operations. *Energy 35*, 2 (2010), 611–627.

[4] Aguilar, O., Perry, S. J., Kim, J. K., and Smith, R. Design and Optimization of Flexible Utility Systems Subject to Variable Conditions: Part 1: Modelling Framework. *Chemical Engineering Research and Design 85*, 8 (2007), 1136–1148.

[5] Aguilar, O., Perry, S. J., Kim, J. K., and Smith, R. Design and Optimization of Flexible Utility Systems Subject to Variable Conditions: Part 2: Methodology and Applications. *Chemical Engineering Research and Design 85*, 8 (2007), 1149–1168.

[6] Aguilera, A., Rodríguez, M., Villegas, T., and Colmenares, W. Explicit Implementation of Predictive Control Based on FPGA. In *Proceedings of the 2nd European Conference of Control, and Proceedings of the 2nd European Conference on Mechanical Engineering* (Stevens Point, Wisconsin, USA, 2011), ECC'11/ECME'11, World Scientific and Engineering Academy and Society (WSEAS), pp. 101–106.

[7] Åkesson, J. MPCtools v1.0. `http://www.control.lth.se/user/johan.akesson/mpctools/index.html`, 2006.

[8] Alessio, A., and Bemporad, A. A survey on explicit model predictive control. In *Nonlinear Model Predictive Control*, L. Magni, D. M. Raimonodo, and F. Allgöwer, Eds., vol. 384 of *Lecture Notes in Control and Information Sciences*. Springer Berlin Heidelberg, 2009, pp. 345–369.

[9] Allgöwer, F. Model Predictive Control: A Success Story Continues. Presentation at APACT, Bath, April 2004.

[10] Allgöwer, F., Badgwell, T., Qin, J., Rawlings, J., and Wright, S. Nonlinear Predictive Control and Moving Horizon Estimation - An Introductory Review. *Advances in Control, Highlights of ECC'99* (1999), 391.

[11] Allgöwer, F., Findeisen, R., and Nagy, Z. K. Nonlinear Model Predictive Control: From Theory to Application. *J. Chin. Inst. Chem. Engrs. 35*, 3 (2004), 299–315.

[12] Alsop, N., and Ferrer, J. M. What Dynamic Simulation brings to a Process Control Engineer: Applied Case Study to a Propylene/Propane Splitter, 2004.

[13] Amestoy, P. R., Duff, I. S., Koster, J., and L'Excellent, J.-Y. A fully asynchronous multifrontal solver using distributed dynamic scheduling. *SIAM Journal of Matrix Analysis and Applications 23*, 1 (2001), 15–41.

[14] Anderson, E., Bai, Z., Bischof, C., Blackford, S., Demmel, J., Dongarra, J., Du Croz, J., Greenbaum, A., Hammarling, S., McKenney, A., and Sorensen, D. *LAPACK Users' Guide*, third ed. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1999.

[15] Anon. Computational Infrastructure for Operations Research (COIN-OR). `http://www.coin-or.org/`, Accessed November 2013.

[16] Applegate, D., Cook, W., Dash, S., and Mevenkamp, M. QSOPT v080725. `http://www.math.uwaterloo.ca/~bico//qsopt/`, 2008.

[17] Asante, N. D. K., and Zhu, X. X. An Automated and Interactive Approach for Heat Exchanger Network Retrofit. *Chemical Engineering Research and Design 75*, 3 (1997), 349–360.

[18] AspenTech. Aspen Plus. `http://www.aspentech.com/products/aspen-plus.aspx`, 2013.

[19] AspenTech. HYSYS. `http://www.aspentech.com/hysys/`, 2013.

[20] Association, M. Modelica - A Unified Object-Orientated Language for Systems Modeling. Language specification, Modelica Association, May 2012.

[21] Atmel. *AT32UC3C Technical Reference Manual*, 32002f ed., March 2010. `http://www.atmel.com/Images/doc32002.pdf`.

[22] Atmel. *AVR32907: AT32UC3C-EK Getting Started Guide*, 2010. `http://www.atmel.com/Images/doc32137.pdf`.

[23] Atmel. *8-bit Atmel Microcontroller with 128KBytes In-System Programmable Flash*, 2467x-avr-06/11 ed., 2011. `http://www.atmel.com/Images/doc2467.pdf`.

[24] Atmel. *AT32UC3C Series Summary*, 32117ds ed., January 2012. `http://www.atmel.com/Images/32117S.pdf`.

[25] Axehill, D., and Hansson, A. A preprocessing algorithm for MIQP solvers with applications to MPC. In *43rd IEEE Conference on Decision and Control* (2004), vol. 3, pp. 2497–2502.

[26] B., D. G. *Linear Programming and Extensions*. Princeton University Press, 1963.

[27] Bartlett, R. A., Wachter, A., and Biegler, L. T. Active Set vs. Interior Point Strategies for Model Predictive Control . In *Proceedings of the American Control Conference* (Chicago, Illinois, 2000), pp. 4229–4233.

[28] Barton, P. I. The equation orientated strategy for process flowsheeting. Tech. rep., MIT, 2000.

[29] Baudet, P., Castelain, P., and Baudouin, O. Process Simulation & Optimization: Sequential Modular Approach or Global Approach? And Why Not Both? In *18th European Symposium on Computer Aided Process Engineering (ESCAPE)* (2008).

[30] Bell, B. M. CppAD: A Package for Differentiation of C++ Algorithms. `https://projects.coin-or.org/CppAD/wiki`, December 2013.

[31] Bemporad, A., and Filippi, C. Suboptimal explicit receeding horizon control via approximate multiparametric quadratic programming. *Journal of Optimization Theory and Applications 117*, 1 (2003), 9–38.

[32] Benson, S. J., Ye, Y., and Zhang, X. DSDP: Solving Large-Scale Sparse Semidefinite Programs for Combinatorial Optimization. *SIAM Journal on Optimization 10*, 2 (2000), 443–461.

[33] Berkelaar, M., Eikland, K., and Notebaert, P. LP_SOLVE v5.5.2.0. `http://lpsolve.sourceforge.net/5.5/index.htm`, 2013.

[34] Bezanson, J., Karpinski, S., Shah, V. B., and Edelman, A. Julia: A Fast Dynamic Language for Technical Computing. `http://arxiv.org/pdf/1209.5145v1.pdf`, 2012.

[35] Biegler, L. T. Chemical Process Simulation. *Chemical Engineering Progress 85*, 10 (1989), 50–61.

[36] Biegler, L. T. Efficient Solution of Dynamic Optimization and NMPC Problems. In *Nonlinear Model Predictive Control*, F. Allgöwer and A. Zheng, Eds., vol. 26 of *Progress in Systems and Control Theory*. Birkhäuser Basel, 2000, pp. 219–243.

[37] Bisschop, J. *AIMMS: Optimization Modeling*. Paragon Decision Technology B.V., 2012, ch. Integer Linear Programming Tricks, p. 287.

[38] Bleris, L. G., and Kothare, M. V. Real-Time Implementation of Model Predictive Control. In *American Control Conference* (2005), pp. 4166–4171.

[39] Bloemen, H., Cannon, M., and Kouvaritakis, B. Closed-Loop Stabilizing MPC for Discrete-Time Bilinear Systems . *European Journal of Control 8*, 4 (2002), 304 – 314.

[40] Bloemen, H. H. J., van den Boom, T. J. J., and Verbruggen, H. B. Optimization Algorithms for Bilinear Model-Based Predictive Control Problems. *AIChE Journal 50*, 7 (2004), 1453–1461.

[41] Bonami, P., Biegler, L. T., Conn, A. R., Cornuejols, G., Grossmann, I. E., Laird, C. D., Lee, J., Lodi, A., Margot, F., , and Wäechter, A. An Algorithmic Framework for Convex Mixed Integer Nonlinear Programs. *Discrete Optimization 5*, 2 (2008), 186–204.

[42] Bonami, P., Biegler, L. T., Conn, A. R., Cornuejols, G., Grossmann, I. E., Laird, C. D., Lee, J., Lodi, A., Margot, F., and Waechter, A. An Algorithmic Framework for Convex Mixed Integer Nonlinear Programs. *Discrete Optimization 5*, 2 (2008), 186–204.

[43] Borchers, B. CSDP: A C Library for Semidefinite Programming. *Optimization Methods and Software 11*, 1 (1999), 613–623.

[44] Boyd, S., Ghaoui, L. E., Feron, E., , and Balakrishnan, V. *Linear Matrix Inequalities in System and Control Theory.* Studies in Applied Mathematics 15. SIAM, 1994.

[45] Bruno, J. C. STEAM: Optimal Synthesis And Optimization of Utility Plants Software.

[46] Bruno, J. C., Fernandez, F., Castells, F., and Grossmann, I. E. A Rigorous MINLP Model for the Optimal Synthesis and Operation of Utility Plants. *Chemical Engineering Research and Design 76*, 3 (1998), 246–258.

[47] Budiansky, S. *Blackett's War: The Men Who Defeated the Nazi U-Boats and Brought Science to the Art of Warfare.* Knopf Doubleday Publishing Group, 2013.

[48] Burcat, A., and Ruscic, B. Third Millenium Ideal Gas and Condensed Phase Thermochemical Database for Combustion with Updates from Active Thermochemical Tables. Tech. rep., Argonne National Laboratory, 2005.

[49] Burchard, J. *M.I.T in World War II.* John Wiley and Sons, 1948.

[50] Burks, A. W., Warren, D. W., and Wright, J. B. An Analysis of a Logical Machine Using Parenthesis-Free Notation. *Mathematical Tables and Other Aids to Computation 8*, 46 (1954), 53–57.

[51] Bussieck, M. R., Drud, A. S., and Meeraus, A. MINLPLib - A Collection of Test Models for Mixed-Integer Nonlinear Programming. *INFORMS Journal on Computing 5*, 1 (2003), 114–119.

[52] Bussieck, M. R., and Pruessner, A. Mixed-integer nonlinear programming. Tech. rep., SIAG/OPT Newsletter: Views and News, 2003.

[53] Cameron, D., Clausen, C., and Morton, W. *Chapter 5.3: Dynamic simulators for operator training*, vol. 11. Elsevier, 2002, pp. 393–431.

[54] Cannon, M. Closed-loop Properties of Model Predictive Control. Oxford University Lecture Notes, February 2014.

[55] Carl L. Yaws. *Thermophysical Properties of Chemicals and Hydrocarbons*, revised ed. William Andrew Inc, 2008.

[56] Carl L. Yaws. *Transport Properties of Chemicals and Hydrocarbons.* William Andrew Inc, 2009.

[57] Cengel, Y. A., and Boles, M. A. *Thermodynamics: An Engineering Approach*, 4th ed. McGraw-Hill Series in Mechanical Engineering. McGraw-Hill, 2002.

[58] Church, E. *Steam Turbines*, 3 ed. McGraw-Hill, New York, 1950.

[59] Contact. Power Stations: Thermal Generation. `http://www.contactenergy.co.nz/web/shared/powerstations?vert=au`, Accessed November 2013.

[60] Cota, R. Development of an Open Source Process Simulator. Master's thesis, University of Calgary, 2003.

[61] Currie, J. jMPC Toolbox v3.11 User's Guide. User's guide, AUT University, July 2011.

[62] Currie, J. JSteam Excel Add In v2.05. `http://www.inverseproblem.co.nz/Software/JSteamExcel.html`, 2012.

[63] Currie, J. jMPC Toolbox v3.20. `http://www.i2c2.aut.ac.nz/Resources/Software/jMPCToolbox.html`, 2013.

[64] Currie, J. OPTI Toolbox v2.05. `http://www.i2c2.aut.ac.nz/Wiki/OPTI/index.php`, 2013.

[65] Currie, J., Prince-Pike, A., and Wilson, D. Auto-Code Generation for Fast Embedded Model Predictive Controllers. In *19th International Conference on Mechatronics and Machine Vision in Practice* (Auckland, New Zealand, 28–30 November 2012), pp. 122–128.

[66] Currie, J., and Wilson, D. I. Lightweight Model Predictive Control Intended for Embedded Applications. In *9th International Symposium on Dynamics and Control of Process Systems (DYCOPS)* (Leuven, Belgium, 5–7 July 2010), pp. 264–269.

[67] Currie, J., and Wilson, D. I. Interpolated Model Predictive Control: Having Your Cake and Eating it Too. In *Australian and New Zealand Annual Chemical Engineering Conference, Chemeca* (Sydney Australia, 18–21 September 2011).

[68] Currie, J., and Wilson, D. I. OPTI: Lowering the Barrier Between Open Source Optimizers and the Industrial MATLAB User. In *Foundations of Computer-Aided Process Operations* (Savannah, Georgia, USA, 8–11 January 2012), Nick Sahinidis and Jose Pinto, Eds.

[69] Currie, J., and Wilson, D. I. Rigorously Modelling Steam Utility Systems for Mixed Integer Optimization. In *10th International Power and Energy Conference* (Ho Chi Minh City, Vietnam, 12–14 December 2012), IEEE, pp. 526–531.

[70] Currie, J., and Wilson, D. I. The Efficient Modelling of Steam Utility Systems. In *Australian and New Zealand Annual Chemical Engineering Conference, Chemeca* (Wellington, New Zealand, 23–26 September 2012), Engineers Australia.

[71] Currie, J., Wilson, D. I., Depree, N., Young, B., Azmanai, S., and Karim, L. Steam Utility Systems are not "Business As Usual" for Chemical Plant Simulators. In *American Institute of Chemical Engineers (AIChE) Spring Meeting* (Chicago, USA, 13–17 March 2011), pp. 63d:1–6.

[72] Cutler, C., Morshedi, A., and Haydel, J. An industrial perspective on advanced control. In *AIChE Annual Meeting* (October 1983).

[73] Cutler, C. R., and Ramakar, B. L. Dynamic Matrix Control - Computer Control Algorithm. In *AIChE National Meeting* (April 1979).

[74] Cutler, C. R., and Ramakar, B. L. Dynamic Matrix Control - Computer Control Algorithm. In *Proceedings of the Joint Automatic Control Conference* (1980).

[75] D. W. Clarke and C. Mohtadi and P. S. Tuffs. Generalized Predictive Control - Part I. The Basic Algorithm. *Automatica 23*, 2 (1987), 137–148.

[76] D'Amato, F., Kumar, A., Lopez-Negrete, R., and Biegler, L. T. Fast Nonlinear Model Predictive Control: Optimization Strategies and Industrial Process Applications. Presentation at CPC VIII, January 2012.

[77] Dantzig G. B., Orden A., W. P. The generalized simplex method for minimizing a linear form under linear inequality constraints. *Pacific J. Math. 5* (1955), 183–195.

[78] Dennis, J., Gay, D., and Welsch, R. Algorithm 573: An Adaptive Nonlinear Least-Squares Algorithm. *ACM Transactions on Mathematical Software 7*, 3 (1981), 367–383.

[79] Dhole, V. R., and Linnhoff, B. Total site targets for fuel, co-generation, emissions, and cooling. *Computers & Chemical Engineering 17*, Supplement 1 (1993), 101–109.

[80] Diego Fernandez Polanco, Alan Eastwood, and Nicola Knight. Real-Time Utility System Optimization. *Energy Engineering 102*, 3 (2005), 63–78.

[81] Digabel, S. L. Algorithm 909: NOMAD: Nonlinear Optimization with the MADS Algorithm. *ACM Transactions on Mathematical Software 37*, 4 (2011), 44:1–44:15.

[82] Domahidi, A. FORCES: Fast optimization for real-time control on embedded systems. `http://forces.ethz.ch`, Oct. 2012.

[83] Domahidi, A. *Methods and Tools for Embedded Optimization and Control*. PhD thesis, ETH Zürich, 2013.

[84] Domahidi, A., Zgraggen, A., Zeilinger, M., Morari, M., and Jones, C. Efficient interior point methods for multistage problems arising in receding horizon control. In *IEEE Conference on Decision and Control (CDC)* (Maui, HI, USA, Dec. 2012), pp. 668 – 674.

[85] Douglas, P. L., and Young, B. E. Modelling and simulation of an AFBC steam heating plant using ASPEN/SP. *Fuel 70*, 2 (1991), 145–154.

[86] Dua, P., Kouramas, K., Dua, V., and Pistikopoulos, E. N. MPC on a chip - Recent advances on the application of multi-parametric model-based control. *Computers and Chemical Engineering 32*, 4-5 (2008), 754–765.

[87] Duff, I. S. Ma57—a code for the solution of sparse symmetric definite and indefinite systems. *ACM Trans. Math. Softw. 30*, 2 (2004), 118–144.

[88] Duran, M. A., and Grossmann, I. E. An outer-approximation algorithm for a class of mixed-integer nonlinear programs. *Mathematical Programming 36*, 3 (1986), 307–339.

[89] Eastwood, A. ProSteam - A Structured Approach to Steam System Improvement. *Energy Engineering 100*, 1 (2003), 59–79.

[90] Eastwood, A., and Bealing, C. Optimizing the Day-to-Day Operation of Utility Systems. *Energy Engineering 101*, 1 (2004), 7–25.

[91] Eaton, J. W., Bateman, D., and Hauberg, S. *GNU Octave Manual Version 3*. Network Theory Limited, 2008.

[92] Fernandez Polanco, D., and Richard, A. T. OptiSteam reduces operating costs at Spanish Refinery. *Driving Competitive Advantage 1Q04* (2004).

[93] FICO. Xpress-Optimizer. Reference manual, Fair Isaac Corporation (FICO), June 2009.

[94] Fien, G. J. Make the most of it. *The Chemical Engineer* (2008).

[95] Flanders, S. W., and Davis, W. J. Scheduling a Flexible Manufacturing System with Tooling Constraints: An Actual Case Study. *Interfaces 25*, 2 (1995), 42–54.

[96] Fletcher, R. FilterSD. `https://projects.coin-or.org/filterSD`, 2013.

[97] Fletcher, R., and Leyffer, S. Solving Mixed Integer Nonlinear Programs by Outer Approximation. *Mathematical Programming 66* (1994), 327–349.

[98] Forrest, J. Personal Communication, May 2013.

[99] Forrest, J. Coin-OR Branch and Cut (CBC) v2.8.5. `https://projects.coin-or.org/Cbc`, 2013.

[100] Forrest, J. Coin-OR Linear Programming (CLP) v1.15.3. `https://projects.coin-or.org/Clp`, 2013.

[101] Fourer, R., Gay, D. M., and Kernighan, B. W. *AMPL: A Modeling Language for Mathematical Programming*, 2nd ed. Duxbury Press / Brooks/-Cole Publishing Company, 2002.

[102] G. P. McCormick. Computability of global solutions to factorable non-convex programs: Part I - Convex underestimating problems. *Mathematical Programming 10* (1976), 147–175.

[103] GAMSWorld. GLOBAL Library (GLOBALLib). `http://www.gamsworld.org/global/globallib.htm`, Accessed November 2013.

[104] Garcìa, C. E., and Morshedi, A. M. Quadratic programming solution of dynamic matrix control (QDMC). *Chemical Engineering Communications 46* (1986), 73–87.

[105] Garcìa, C. E., Prett, D. M., and Morari, M. Model predictive control: Theory and practice - a survey. *Automatica 25*, 3 (1989), 335 – 348.

[106] Gas Research Institute (GRI). GRI-Mech. `http://www.me.berkeley.edu/gri-mech/overview.html`, Accessed October 2013.

[107] Gertz, E. M., and Wright, S. J. Object-Orientated Software for Quadratic Programming, Optimization Technical Report 01-02. Technical report, University of Wisconsin-Madison, 2001.

[108] Gertz, E. M., and Wright, S. J. Object Orientated Software for Quadratic Programming. *ACM Transactions on Mathematical Software 29* (2003), 58–81.

[109] Geyer, T., Papafotiou, G., and Morari, M. Model Predictive Control in Power Electronics: A Hybrid Systems Approach. In *Proceedings of the 44th IEEE Conference on Decision and Control, and the European Control Conference* (December 2005), pp. 5606–5611.

[110] Geyer, T., Torrisi, F. D., and Morari, M. Optimal complexity reduction of polyhedral piecewise affine systems. *Automatica 44* (2008), 1728–1740.

[111] Gilbert, J., and Lemarechal, C. Some numerical experiments with variable-storage quasi-Newton algorithms. *Mathematical Programming 45* (1989), 407–435.

[112] Giselsson, P. Execution time certification for gradient-based optimization in model predictive control. In *Proc. 51st IEEE Conf. on Decision and Control* (2012).

[113] Goldfarb, D., and Idnani, A. A Numerical Stable Dual Method for Solving Strictly Convex Quadratic Programs. *Mathematical Programming 27* (1983), 1–33.

[114] Gomory, R. Outline of an algorithm for integer solutions to linear programs. *Bull. Amer. Math. Soc. 64* (1958), 275–278.

[115] Gondzio, J. Multiple Centrality Corrections in a Primal-Dual Method for Linear Programming. *Computational Optimization and Applications 6* (1996), 137–156.

[116] Grant, M., and Boyd, S. CVX: Matlab Software for Disciplined Convex Programming, version 2.0 beta. `http://cvxr.com/cvx`, September 2012.

[117] Grossmann, I. E. Mixed-integer programming approach for the synthesis of integrated process flowsheets. *Computers & Chemical Engineering 9*, 5 (1985), 463–482.

[118] Grossmann, I. E. *MINLP Optimization Strategies and Algorithms for Process Synthesis.* Foundations of Computer-Aided Process Design. Cache-Elsevier, Amsterdam, 1990.

[119] Grossmann, I. E., and Daichendt, M. M. New trends in optimization-based approaches to process synthesis. *Computers & Chemical Engineering 20*, 6-7 (1996), 665–683.

[120] Grüne, L., and Pannek, J. *Nonlinear Model Predictive Control*, 1st ed. Springer-Verlag, London, 2011.

[121] Gupta, A. WSMP: Watson sparse matrix package (Part-I: Direct solution of symmetric sparse systems). Technical Report RC 21886, IBM T. J. Watson Research Center, Yorktown Heights, NY, November 2000.

[122] H. Tuy. Concave Programming Under Linear Constraints. *Soviet Mathematics* (1964), 1437–1440.

[123] Halasz, L., Nagy, A. B., Ivicz, T., Friedler, F., and Fan, L. T. Optimal retrofit design and operation of the steam-supply system of a chemical complex. *Applied Thermal Engineering 22*, 8 (2002), 939–947.

[124] Hans Hasse, Bessling, B., and Bottcher, R. OPEN CHEMASIM: Breaking Paradigms in Process Simulation. In *16th European Symposium on Computer Aided Process Engineering* (2006), Elsevier, pp. 255–260.

[125] Hennig, A., and Balas, G. J. MPC supervisory flight controller: A case study to flight EL AL 1862. In *AIAA Guidance, Navigation and Control Conference* (2008).

[126] Henson, M., and Seborg, D. *Nonlinear Process Control.* Prentice Hall PTR, 1997.

[127] Herceg, M., Kvasnica, M., Jones, C., and Morari, M. Multi-Parametric Toolbox 3.0. In *Proc. of the European Control Conference* (Zürich, Switzerland, July 17–19 2013), pp. 502–510. `http://control.ee.ethz.ch/∼mpt`.

[128] Hindmarsh, A. C., Brown, P. N., Grant, K. E., Lee, S. L., Serban, R., Shumaker, D. E., , and Woodward, C. S. SUNDIALS: Suite of Nonlinear and Differential/Algebraic Equation Solvers. *ACM Transactions on Mathematical Software 31*, 3 (2005), 363–396.

[129] Holmgren, M. X Steam, 2010.

[130] Holmström, K. TOMLAB – An Environment for Solving Optimization Problems in MATLAB. In *Proceedings for the Nordic MATLAB Conference* (1997), pp. 27–28.

[131] Honeywell. Profit Controller. `https://www.honeywellprocess.com/en-US/explore/products/advanced-applications/advanced-control-and-optimization/control-performance-solutions/Pages/Profit-Controller.aspx`, Accessed December 2013.

[132] Houska, B., Ferreau, H., and Diehl, M. ACADO Toolkit – An Open Source Framework for Automatic Control and Dynamic Optimization. *Optimal Control Applications and Methods 32*, 3 (2011), 298–312.

[133] Houska, B., Ferreau, H., and Diehl, M. An Auto-Generated Real-Time Iteration Algorithm for Nonlinear MPC in the Microsecond Range. *Automatica 47*, 10 (2011), 2279–2285.

[134] Hui, C.-W., and Natori, Y. An industrial application using mixed-integer programming technique: A multi-period utility system model. *Computers & Chemical Engineering 20*, Supplement 2 (1996), S1577–S1582.

[135] Hutchison, H. P., Jackson, D. J., and Morton, W. The development of an equation-oriented flowsheet simulation and optimization package–I. The quasilin program. *Computers & Chemical Engineering 10*, 1 (1986), 19–29.

[136] IAPWS. Supplementary Release on the Backward Equations for Pressure as a Function of Enthalpy and Entropy $p(h,s)$, 2001.

[137] IAPWS. Revised Supplementary Release on Backward Equations for the Functions $T(p,h)$, $V(p,h)$ and $T(p,s)$, $V(p,s)$ for Region 3, 2004.

[138] IAPWS. Supplementary Release on Backward Equations p(h,s) for Region 3, Equations as a Function of h and s for the Region Boundaries, and an Equation $T\text{sat}(h,s)$ for Region 4, 2004.

[139] IAPWS. Supplementary Release on Backward Equations for Specific Volume as a Function of Pressure and Temperature $V(p,t)$ for Region 3, 2005.

[140] IAPWS. Revised Release on the IAPWS Industrial Formulation 1997 for the Thermodynamic Properties of Water and Steam, 2007.

[141] IAPWS. Release on the IAPWS Formulation 2008 for the Viscosity of Oridinary Water Substance, 2008.

[142] IAPWS. Revised Release on the IAPWS Formulation 1995 for the Thermodynamic Properties of Ordinary Water Substance for General and Scientific Use, 2009.

[143] IAPWS. Release on the IAPWS Formulation 2011 for the Thermal Conductivity of Ordinary Water Substance , 2011.

[144] IBM. CPLEX v12.5.0. `http://www-01.ibm.com/software/commerce/optimization/cplex-optimizer/`, 2013.

[145] IBM. IBM ILOG CPLEX Optimization Studio: CPLEX. User's manual, IBM Corporation, 2013.

[146] IEEE Standards Committee 754. IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Standard 754-1985. Standard, Institute of Electrical and Electronics Engineers, New York, 1985.

[147] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual. Technical manual, Intel, 2013.

[148] Intel. Math Kernel Library v11.1 R0. `http://software.intel.com/en-us/intel-mkl`, 2013.

[149] Iyer, R. R., and Grossmann, I. E. Synthesis and operational planning of utility systems for multiperiod operation. *Computers & Chemical Engineering 22*, 7-8 (1998), 979–993.

[150] J. A. Rossiter and B. Kouvaritakis and M. J. Rice. A numerically stable robust state-space approach to stable-predictive control strategies. *Automatica 34* (1998), 65–73.

[151] J. E. Kelley. The cutting plane method for solving convex programs. *Journal of SIAM 8*, 4 (1960), 703–712.

[152] Jasper van Baten. CAPE-OPEN to CAPE-OPEN (COCO) Simulation Environment v2.06. `http://www.cocosimulator.org/`, 2013.

[153] Jerez, J., Goulart, P., Richter, S., Constantinides, G., Kerrigan, E., and Morari, M. Embedded Online Optimization for Model Predictive Control at Megahertz Rates. *IEEE Transactions on Automatic Control (Submitted)* (2013).

[154] Jerez, J., Goulart, P., Richter, S., Constantinides, G., Kerrigan, E., and Morari, M. Embedded Predictive Control on an FPGA using the Fast Gradient Method. In *European Control Conference (Accepted)* (2013).

[155] Jerez, J. L., Constantinides, G. A., and Kerrigan, E. C. An FPGA implementation of a sparse quadratic programming solver for constrained predictive control. In *Proceedings of the ACM Symposium on Field Programmable Gate Arrays* (2011).

[156] Johansen, T. A., Jackson, W., Schreiber, R., and Tondel, P. Hardware Synthesis of Explicit Model Predictive Controllers. *IEEE Transactions on Control Systems Technology 15*, 1 (2007), 191–197.

[157] Johnson, S. G. The NLopt nonlinear-optimization package. `http://ab-initio.mit.edu/nlopt`, Accessed November 2013.

[158] Jones, E., Oliphant, T., Peterson, P., et al. SciPy: Open source scientific tools for Python. `http://www.scipy.org/`, 2001–.

[159] Kalman, R. E. A new approach to linear filtering and prediction problems. *Transactions of ASME, Journal of Basic Engineering 87* (1960), 35–45.

[160] Kalman, R. E. Contributions to the theory of optimal control. *Bulletin de la Societe Mathematique de Mexicana 5* (1960), 102–119.

[161] Karmarkar, N. A New Polynomial-Time Algorithm for Linear Programming. *Combinatorics 4* (1984), 373–395.

[162] KBC Advanced Technologies. ProSteam. `http://www.kbcat.com/energy-utilities-software/prosteam`, 2013.

[163] KBC Advanced Technologies. SuperTarget. `http://www.kbcat.com/energy-utilities-software/supertarget`, 2013.

[164] Keerthi, S., and Gilbert, E. Optimal infinite-horizon feedback laws for a general class of constrained discrete-time systems: Stability and moving-horizon approximations. *Journal of Optimization Theory and Applications 57*, 2 (1988), 265–293.

[165] Kelman, A., Vichik, S., and Borrelli, F. BLOM: The Berkeley Library for Optimization Modeling and Nonlinear Model Predictive Control. Technical Report, Department of Mechanical Engineering, University of California, Berkeley, CA 94720-1740, USA, 2012.

[166] Kenney, W. *Energy Conservation in Process Industries*. Elseiver, 1984.

[167] Khachiyan, L. G. A Polynomial Algorithm in Linear Programming. *Soviet Mathematics Doklady 20* (1979), 191–194.

[168] Kim, J. H., Lee, M. H., Han, C., Kim, S. H., and You, S. H. Multiperiod planning for utility systems using dynamic programming. *Computers & Chemical Engineering 23*, Supplement 1 (1999), S519–S522.

[169] Kimura, H., and Zhu, X. X. R-Curve Concept and Its Application for Industrial Energy Management. *Industrial & Engineering Chemistry Research 39*, 7 (2000), 2315–2335.

[170] Kögel, M., and Findeisen, R. A Fast Gradient Method for Embedded Linear Predictive Control. In *Proceedings of the 18th IFAC World Congress* (2011), pp. 1362–1367.

[171] Kögel, M., and Findeisen, R. Fast predictive control of linear, time-invariant systems using an algorithm based on the fast gradient method and augmented lagrange multipliers. In *IEEE International Conference on Control Applications (CCA)* (2011), pp. 780–785.

[172] Kögel, M., and Findeisen, R. Parallel solutions of model predictive control using the alternating direction method of multipliers. In *Proc. 4th IFAC Conf. on Nonlinear Model Predictive Control* (2012), pp. 369–374.

[173] Koh, S. L. Solving interior point method on a FPGA. Master's thesis, Nanyang Technological University, Singapore, 2009.

[174] Kothare, M. V., Balakrishnan, V., and Morari, M. Robust Constrained Model Predictive Control Using Linear Matrix Inequalities. *Automatica 32*, 10 (1996), 1361–1379.

[175] Kroshko, D. OpenOpt: Free scientific-engineering software for mathematical modeling and optimization. `"http://www.openopt.org/"`, 2007.

[176] Kuon, I., Tessier, R., and Rose, J. FPGA Architecture: Survey and Challenges. *Foundations and Trends in Electornic Design Automation 2*, 2 (2007), 135–253.

[177] Kvasnica, M., Grieder, P., and Baotić, M. Multi-Parametric Toolbox (MPT). `http://control.ee.ethz.ch/~mpt/`, 2004.

[178] Kvasnica, M., Rauova, I., and Fikar, M. Automatic code generation for real-time implementation of Model Predictive Control. In *IEEE International Symposium on Computer-Aided Control System Design (CACSD)* (2010), pp. 993–998.

[179] Kyle, B. G. *Chemical and Process Thermodynamics*, vol. 3. Prentice Hall, 1999.

[180] Laganier, F. Dynamic process simulation trends and perspectives in an industrial context. *Computers & Chemical Engineering 20*, Supplement 2 (1996), S1595–S1600.

[181] Lai, K. L., Crassidis, J. L., Cheng, Y., and Kim, J. R. New Complex-Step Derivative Approximations with Application to Second-Order Kalman Filtering. In *AIAA Guidance, Navigation, and Control Conference* (August 2005).

[182] Lau, M. S. K., Yue, S. P., Ling, K. V., and Maciejowski, J. M. A Comparison of Interior Point and Active Set Methods for FPGA Implementation of Model Predictive Control. In *Proceedings of the European Control Conference* (2009), pp. 156–161.

[183] Lawson, C., Hanson, R., Kincaid, D., and Krogh, F. Basic Linear Algebra Subprograms for FORTRAN Usage. *ACM Trans. Math. Soft. 5* (1979), 308–323.

[184] Levenberg, K. A Method for the Solution of Certain Non-Linear Problems in Least Squares. *Quarterly of Applied Mathematics 2* (1944), 164–168.

[185] Lim, H., Currie, J., and Wilson, D. Validating a Thermodynamic Model of the Otahuhu B Combined Cycle Gas Turbine Power Station. In *Australian and New Zealand Annual Chemical Engineering Conference, Chemeca* (Brisbane, Australia, 29 Sept–2 October 2013), Engineers Australia.

[186] Ling, K. V., Wu, B. F., and Maciejowski, J. M. Embedded Model Predictive Control (MPC) using a FPGA. In *The 17th World Congress of the International Federation of Automatic Control (IFAC)* (2008), pp. 15250–15255.

[187] Ling, K. V., Yue, S. P., and Maciejowski, J. M. A FPGA Implementation of Model Predictive Control. In *American Control Conference* (2006), pp. 1930–1935.

[188] Linnhoff, B., and Hindmarsh, E. The pinch design method for heat exchanger networks. *Chemical Engineering Science 38*, 5 (1983), 745–763.

[189] Löfberg, J. YALMIP : A toolbox for modeling and optimization in MATLAB. In *Proceedings of the CACSD Conference* (Taipei, Taiwan, 2004).

[190] Lourakis, M. I. A. Levenberg-Marquardt Nonlinear Least Squares Algorithms in C/C++. `www.ics.forth.gr/~lourakis/levmar/`, July 2004.

[191] Lyness, J. N., and Moler, C. B. Numerical Differentiation of Analytic Functions. *SIAM Journal of Numerical Analysis 4* (1967), 202–210.

[192] Maciejowski, J. M. *Predictive Control with Constraints.* Pearson Education, 2002.

[193] Maia, L. O. A., and Qassim, R. Y. Synthesis of utility systems with variable demands using simulated annealing. *Computers & Chemical Engineering 21*, 9 (1997), 947–950.

[194] Makhorin, A. GNU Linear Programming Kit (GLPK) v4.48. `http://www.gnu.org/software/glpk/`, 2013.

[195] Manikonda, V., Arambel, P. O., Gopinathan, M., Mehra, R. K., and Hadaegh, F. Y. A model predictive control-based approach for spacecraft formation keeping and attitude control. In *American Control Conference* (1999), p. 4258.

[196] Maplesoft. Maple 17. `http://www.maplesoft.com/products/maple/`, 2013.

[197] Marèchal, F., and Kalitventzeff, B. Process integration: Selection of the optimal utility system. *Computers & Chemical Engineering 22*, Supplement 1 (1998), S149–S156.

[198] Marèchal, F., and Kalitventzeff, B. Targeting the optimal integration of steam networks: Mathematical tools and methodology. *Computers & Chemical Engineering 23*, Supplement 1 (1999), S133–S136.

[199] Marèchal, F., and Kalitventzeff, B. Targeting the integration of multi-period utility systems for site scale process integration. *Applied Thermal Engineering 23*, 14 (2003), 1763–1784.

[200] Mark Mikofski. IAPWS_IF97 Functional Form with No Slip. `http://www.mathworks.com/matlabcentral/fileexchange/35710`, 2012.

[201] Marquardt, D. An Algorithm for Least-Squares Estimation of Nonlinear Parameters. *SIAM Journal on Applied Mathematics 11*, 2 (1963), 431–441.

[202] Mason, A. J., and Dunning, I. OpenSolver - Open Source Optimisation for Excel. In *Proceedings of the 45th Annual Conference of the Operations Research Society of New Zealand* (2010), pp. 181–190.

[203] MathWorks. Embedded coder v6.5. `http://www.mathworks.com.au/products/embedded-coder/`, 2013.

[204] MathWorks. Global Optimization Toolbox v3.2.3. `http://www.mathworks.com.au/products/global-optimization/`, 2013.

[205] MathWorks. MATLAB v8.1.0 (R2013a). `http://www.mathworks.com.au/`, 2013.

[206] MathWorks. MPC Toolbox v4.1.2. `http://www.mathworks.com.au/products/mpc/`, 2013.

[207] MathWorks. Optimization Toolbox v6.3. `http://www.mathworks.com.au/products/optimization/`, 2013.

[208] MathWorks. Simulink v8.1. `http://www.mathworks.com.au/products/simulink/`, 2013.

[209] MathWorks. Symbolic Toolbox v5.10. `http://www.mathworks.com.au/products/symbolic/`, 2013.

[210] Mattingley, J., and Boyd, S. CVXGEN: A Code Generator for Embedded Convex Optimization. *Optimization and Engineering 13*, 1 (2012), 1–27.

[211] Mattsson, S. E., and Elmqvist, H. Modelica - An International Effort to design the next generation modeling language, 1997.

[212] Mavromatis, S. P., and Kokossis, A. C. Conceptual optimisation of utility networks for operational variations – I. Targets and level optimisation. *Chemical Engineering Science 53*, 8 (1998), 1585–1608.

[213] Mavromatis, S. P., and Kokossis, A. C. Conceptual optimisation of utility networks for operational variations – II. Network development and optimisation. *Chemical Engineering Science 53*, 8 (1998), 1609–1630.

[214] Mayne, D., Rawlings, J., Rao, C., and Scokaert, P. Constrained model predictive control: Stability and optimality. *Automatica 36*, 6 (2000), 789 – 814.

[215] Mayne, D. Q., and Michalska, H. Receding Horizon Control of Nonlinear Systems. *IEEE Transactions of Automatic Control 35*, 7 (1990), 814.

[216] McIlhagga, W. Automatic Differentiation with MATLAB Objects. `http://www.mathworks.com/matlabcentral/fileexchange/26807-automatic-differentiation-with-matlab-objects`, March 2010.

[217] Mehrotra, S. On the Implementation of a Primal-Dual Interior Point Method. *SIAM Journal on Optimization 2*, 4 (1992), 575–601.

[218] Microsoft. Parallel Patterns Library (PPL). `http://msdn.microsoft.com/en-us/library/dd492418.aspx`, Accesed October 2013.

[219] Mittelmann, H. Benchmarks for Optimization Software. `http://plato.asu.edu/bench.html`, Accessed November 2013.

[220] Mittelmann, H. Mixed Integer Linear Programming Benchmark (MIPLIB2010). `http://plato.asu.edu/ftp/milpc.html`, Accessed November 2013.

[221] Morari, M. Model Predictive Control: Multivariable Control Technique of Choice in the 1990s? In *In Advances in Model-based Predictive Control* (1990), Oxford University Press, pp. 22–37.

[222] Morari, M. Fast Model Predictive Control (MPC). Presentation held under the auspices of the ACROSS Project, `http://divf.eng.cam.ac.uk/cfes/pub/Main/Presentations/Morari.pdf`, 2013.

[223] Morari, M., and Lee, J. H. Model predictive control: past, present and future. *Computers & Chemical Engineering 23*, 4-5 (1999), 667–682.

[224] More, J., Garbow, B., and Hillstrom, K. MINPACK Levenberg Marquardt (LMDER). `http://www.netlib.org/minpack/`, 1999.

[225] More, J., Garbow, B., and Hillstrom, K. MINPACK Powell Hybrid (HYBRJ). `http://www.netlib.org/minpack/`, 1999.

[226] Morris, C., Sim, W., and Vysniauskas, T. Process Simulation on a Personal Computer, An Interactive Approach. AIChE.

[227] Morris, C. G., Sim, W. D., and Vysniauskas, T. Interactive Approach to Process Simulation. *Chemical Engineering Progress 81*, 9 (1985), 40–44.

[228] Naess, L., Mjaavatten, A., and Li, J. O. Using dynamic process simulation from conception to normal operation of process plants. *Computers & Chemical Engineering 17*, 5-6 (1993), 585–600.

[229] NASA. NASA Glenn ThermoBuild. `http://cearun.grc.nasa.gov/cea/index_ds.html`, Accessed October 2013.

[230] Nasr, K. Module 3: Gas Turbine Power Cycles - Brayton Cycle. Tech. rep., Kettering University, 2003.

[231] Nath, R., and Holliday, J. Optimizing a process plant utility system. *Mechanical Engineering 107*, 2 (1985), 44–50.

[232] Nath, R., Marcinkowska, A. R., Skarke, S., Thomasson, M., and Worsham, B. Experiences with On-Line STEAMPOP. In *Industrial Energy Technology Conference (IETC)* (1988).

[233] Neidinger, R. D. Introduction to Automatic Differentiation and MATLAB Object-Orientated Programming. *SIAM Review 52*, 3 (2010), 545–563.

[234] Nesterov, Y. A method for solving a convex programming problem with convergence rate 1/k^2. *Soviet Math. Dokl. 27*, 2 (1983), 372–376.

[235] Nocedal, J., and Wright, S. *Numerical Optimization*, 2nd ed. Springer Series in Operations Research and Financial Engineering. Springer, 2006.

[236] Oliphant, T. E. Python for scientific computing. *Computing in Science and Engineering 9*, 3 (2007), 10–20.

[237] Ong'iro, A., Ugursal, V. I., Al Taweel, A. M., and Lajeunesse, G. Thermodynamic simulation and evaluation of a steam CHP plant using ASPEN Plus. *Applied Thermal Engineering 16*, 3 (1996), 263–271.

[238] Pachter, M., and Cjhandler, P. R. Challenges of autonomous control. *IEEE Control Systems Magazine 18*, 4 (1998), 92–97.

[239] Pannocchia, G., Rawlings, J. B., and Wright, S. J. Fast, large-scale model predictive control by partial enumeration. *Automatica 43* (2007), 852–860.

[240] Pantelides, C. C. SpeedUp–recent advances in process simulation. *Computers & Chemical Engineering 12*, 7 (1988), 745–755.

[241] Papoulias, S. A., Buchheim, M. E., and Zilova, K. S. UPLAN - A Utility Planning and Optimization Program. In *AICHE Annual Meeting* (1985).

[242] Papoulias, S. A., and Grossmann, I. E. A structural optimization approach in process synthesis–I : Utility systems. *Computers & Chemical Engineering 7*, 6 (1983), 695–706.

[243] PATRICK BANGERT. *Optimization for Industrial Problems.* Springer, 2012.

[244] PATTISON, J. R., AND SHARMA, V. *Selection of Boiler Plant and Overall System Efficiency.* No. 3 in Studies in Energy Efficiency in Buildings. Conservation Co-ordination, British Gas Corporation, 1980.

[245] PENG, D.-Y., AND ROBINSON, D. B. A New Two-Constant Equation of State. *Industrial & Engineering Chemistry Fundamentals 15*, 1 (1976), 59–64.

[246] PEREZ, R. E., JANSEN, P. W., AND MARTINS, J. R. R. A. pyOpt: A Python-based object-oriented framework for nonlinear constrained optimization. *Structures and Multidisciplinary Optimization 45*, 1 (2012), 101–118.

[247] PERKINS, AND SARGENT. SPEEDUP: A computer program for steady-state and dynamic simulation and design of chemical processes. *AIChE Symposium Series 78* (1982), 1–11.

[248] PERRY, R., GREEN, D., AND MALONEY, J., Eds. *Perry's Chemical Engineers' Handbook*, 7 ed. McGraw Hill, 1997.

[249] PETERSON, J., AND MANN, W. Steam system design: how it evolves. *Chemical Engineering 92*, 21 (1985), 62–74.

[250] PETERSON, K. B., AND PEDERSON, M. S. The Matrix Cookbook. Handbook, MIT, 2006.

[251] PHILPOTT, A., AND EVERETT, G. Supply Chain Optimization in the Paper Industry. In *Proceedings of the 34th Annual Conference of the Operational Research Society of New Zealand* (1999).

[252] PIELA, P. C., EPPERLY, T. G., WESTERBERG, K. M., AND WESTERBERG, A. W. ASCEND: an object-oriented computer environment for modeling and analysis: The modeling language. *Computers & Chemical Engineering 15*, 1 (1991), 53–72.

[253] POWELL, M. On the Quadratic Programming Algorithm of Goldfarb and Idnani. *Mathematical Programming Study 25* (1985), 46–61.

[254] PRETT, D. M., AND GILLETTE, R. D. Optimization and constrained multivariable control of a catalytic cracking unit. In *Proceedings of the Joint Automatic Control Conference* (1980).

[255] PRICE, T., AND MAJOZI, T. *An Effective Technique for the Synthesis and Optimization of Steam System Networks*, vol. Volume 27. Elsevier, 2009, pp. 477–482.

[256] PRICE, T., AND MAJOZI, T. *Using Process Integration for Steam System Network Optimization with Sustained Boiler Efficiency*, vol. Volume 26. Elsevier, 2009, pp. 1281–1286.

[257] PROCESS SYSTEMS ENTERPRISE. gPROMS. `http://www.psenterprise.com/gproms.html`, Accessed November 2013.

[258] PROSIM. Ariane, 2010.

[259] ProSim. Ariane: Utilities Management and Power Plant Optimization Brochure, 2010.

[260] Pye, J. freesteam v2.0. `http://freesteam.sourceforge.net/`, January 2010.

[261] Qin, J., and Badgwell, T. A. An overview of nonlinear model predictive control applications. In *Nonlinear Model Predictive Control*, F. Allgöwer and A. Zheng, Eds., vol. 26 of *Progress in Systems and Control Theory*. Birkhäuser Basel, 2000, pp. 369–392.

[262] Qin, S. J., and Badgwell, T. A. An overview of industrial predictive control technology. In *Proceedings of the 5th International Conference on Chemical Process Control (CPC-V)* (1996), pp. 232–256.

[263] Qin, S. J., and Badgwell, T. A. A survey of industrial model predictive control technology. *Control Engineering Practice 11* (2003), 733–764.

[264] Quanser. IP02 Self-Erecting Inverted Pendulum: User's Guide, 1996. `www.quanser.com`.

[265] Quanser. Position Control of a 3-DOF Helicopter: Reference Manual, 2013. `www.quanser.com`.

[266] Quanser. Quanser 3 DOF Helicopter Platform Helps Develop High-Speed Embedded MPCs. `http://quanser.blogspot.co.nz/2013/08/quanser-3-dof-helicopter-platform-helps.html`, August 2013.

[267] Quanser. Linear Control Challenge - IP02 Inverted Pendulum, Accessed September 2013. `http://www.quanser.com/english/html/products/fs_product_challenge.asp?lang_code=english&pcat_code=exp-lin&prod_code=L2-invpen&tmpl=1`.

[268] Quanser. Specialty Control Challenge - 3 DOF Helicopter, Accessed September 2013. `http://www.quanser.com/english/html/products/fs_product_challenge.asp?lang_code=english&pcat_code=exp-spe&prod_code=S1-3dofheli&tmpl=1`.

[269] Rahim, M. A., Amirabedin, E., Yilmazoglu, M. Z., and Durmaz, A. Analysis Of Heat Recovery Steam Generators In Combined Cycle Power Plants. In *The Second International Conference on Nuclear and Renewable Energy Resouces* (Ankara, Turkey, July 2010), pp. 788–798.

[270] Rajgopal, J. *Maynard's Industrial Engineering Handbook*, 5th ed. McGraw Hill Professional, 2001, ch. Principles and Applications of Operations Research, pp. 11.27–11.44.

[271] Rawlings, J. B. Tutorial Overview of Model Predictive Control. *IEEE Control Systems Magazine 20* (2000), 38–52.

[272] Rawlings, J. B., Meadows, E. S., and Muske, K. R. Nonlinear Model Predictive Control: A Tutorial Survey. In *Proc. Int. Symp. Adv. Control of Chemical Processes, ADCHEM* (Kyoto, Japan, 1994), p. 185.

[273] Rawlings, J. B., and Muske, K. R. Stability of constrained receeding horizon control. *IEEE Transactions on Automatic Control 38*, 10 (1993), 1512–1516.

[274] Redlich, O., and Kwong, J. N. S. On The Thermodynamics of Solutions. V. An Equation Of State. Fugacities of Gaseous Solutions. *Chem. Rev. 44*, 1 (1949), 233–244.

[275] Richalet, J., and O'Donovan, D. *Predictive Functional Control.* Advances in Industrial Control. Springer, 2009.

[276] Richalet, J., Rault, A., Testud, J. L., and Papon, J. Algorithmic Control of Industrial Processes. In *Proceedings of the 4th IFAC Symposium on Indentification and System Parameter Estimation* (1976), pp. 1119–1167.

[277] Richalet, J., Rault, A., Testud, J. L., and Papon, J. Model predictive heuristic control: Applications to industrial processes. *Automatica 14* (1978), 413–428.

[278] Richards, A., Stewart, W., and Wilkinson, A. Auto-coding Implementation of Model Predictive Control with Application to Flight Control. In *European Control Conference* (2009), pp. 150–155.

[279] Richter, S., Jones, C., , and Morari, M. Computational complexity certification for real-time MPC with input constraints based on the fast gradient method. *IEEE Transactions on Automatic Control 57*, 6 (2012), 1391–1403.

[280] Richter, S., Jones, C., and Morari, M. Real-time input-constrained MPC using fast gradient methods. In *Proceedings of the 48th IEEE Conference on Decision and Control, held jointly with the 28th Chinese Control Conference (CDC/CCC)* (2009), pp. 7387–7393.

[281] Richter, S., Morari, M., , and Jones, C. Towards computational complexity certification for constrained MPC based on lagrange relaxation and the fast gradient method. In *Proc. 50th IEEE Conf. on Decision and Control* (2011), pp. 5223–5229.

[282] Rigby, B., L. S. L., and Waren, A. D. The Evolution of Texaco's Blending Systems: From OMEGA to StarBlend. *Interfaces 25*, 5 (1995), 64–83.

[283] Robertson, J. L. The ideal process simulator. *Chemical Engineering Progress 85*, 10 (1989), 62–66.

[284] Roelofs, M., and Bisschop, J. AIMMS - The User's Guide. User's guide, Paragon Decision Technology, October 2012.

[285] Rosen, E. M. *Steady State Chemical Process Simulation: A State-of-the-Art Review*, vol. 124. ACS, 1980, ch. 1, pp. 3–36.

[286] Rosenthal, R. E. GAMS - A User's Guide. User's guide, GAMS Development Corporation, July 2013.

[287] Rossiter, J. A. *Model Based Predictive Control - A Practical Approach.* CRC Press, 2004.

[288] Ryan, D. M. The solution of massive generalised set partitioning problems in aircrew rostering. *Journal of the Operational Research Society 43* (1992), 459–467.

[289] Ryan, D. M. Optimization Earns Its Wings. *OR/MS Today* (2000).

[290] Sahinidis Optimization Group. Baron software. `http://archimedes.cheme.cmu.edu/?q=baron`, Accessed November 2013.

[291] Schneider, D. F. Steady State Simulators for Your Project, August 1998.

[292] Scilab Enterprises. *Scilab: Free and Open Source software for numerical computation.* Scilab Enterprises, Orsay, France, 2012.

[293] Shacham, M., Macchieto, S., Stutzman, L. F., and Babcock, P. Equation oriented approach to process flowsheeting. *Computers & Chemical Engineering 6*, 2 (1982), 79–95.

[294] Shang, Z., and Kokossis, A. A transhipment model for the optimisation of steam levels of total site utility system for multiperiod operation. *Computers & Chemical Engineering 28*, 9 (2004), 1673–1688.

[295] Shang, Z., and Kokossis, A. A systematic approach to the synthesis and design of flexible site utility systems. *Chemical Engineering Science 60*, 16 (2005), 4431–4451.

[296] Shrader, C. R. *History of Operations Research In The United States Army, Volume 1: 1942-1962.* The United States Army, 2006.

[297] Smith, S. W. *Digital Signal Processing: A Practical Guide for Engineers and Scientists*, 1st ed. California Technical Pub, 2002, ch. 28: Digital Signal Processors, p. 626.

[298] Spinu, V., Oliveri, A., Lazar, M., and Storace, M. FPGA implementation of optimal and approximate model predictive control for a buck-boost DC-DC converter. In *2012 IEEE International Conference on Control Applications (CCA)* (2012), pp. 1417–1423.

[299] Strouvalis, A. M., Heckl, I., Friedler, F., and Kokossis, A. C. Customized solvers for the operational planning and scheduling of utility systems. *Computers & Chemical Engineering 24*, 2-7 (2000), 487–493.

[300] Strouvalis, A. M., Heckl, I., Friedler, F., and Kokossis, A. C. An accelerated Branch-and-Bound algorithm for assignment problems of utility systems. *Computers & Chemical Engineering 26*, 4-5 (2002), 617–630.

[301] Strouvalis, A. M., and Kokossis, A. C. *A conceptual optimisation approach for the multiperiod planning of utility networks*, vol. Volume 9. Elsevier, 2001, pp. 919–924.

[302] Svrcek, W. Y., Mahoney, D., and Young, B. *A real-time approach to process control.* Wiley, 2000.

[303] Svrcek, W. Y., and Satyro, M. A. Process Simulation - From Large Computers and Small Solutions to Small Computers and Large Solutions. *Chemical Product and Process Modeling 1*, 1 (2006).

[304] Tawarmalani, M., and Sahinidis, N. A polyhedral branch-and-cut approach to global optimization. *Mathematical Programming 103*, 2 (2005), 225–249.

[305] Texas Instruments. *TMS320C28343 Experimenter's Kit Overview*, 2009. http://www.atmel.com/Images/doc32137.pdf.

[306] Texas Instruments. *C238x Floating Point Unit fastRTS Library: Module User's Guide*, 1.0 ed., 2010.

[307] Texas Instruments. C28x Compiler - Understanding Linking. Wiki, 2011. http://processors.wiki.ti.com/index.php/C28x_Compiler_-_Understanding_Linking.

[308] Texas Instruments. Delfino C28343 Experimenter's Kit, 2013. http://www.ti.com/tool/tmdsdock28343.

[309] Texas Instruments. *TMS320C28346, TMS320C28345, TMS320C28344, TMS320C28343, TMS320C28342, TMS320C28341 Delfino Microcontrollers Data Manual*, SPRS516D ed., 2012. http://www.ti.com/lit/ds/sprs516d/sprs516d.pdf.

[310] TI E2E Community Forum. Array Alignment Seems to Leave Large Holes. Support Forum, Accessed August 2013. http://e2e.ti.com/support/development_tools/compiler/f/343/p/61913/222193.aspx.

[311] TLV. Water Hammer / Steam Hammer. http://www.tlv.com/global/TI/steam-theory/what-is-waterhammer.html, Accessed November 2013.

[312] Ullmann, F. A Matlab Toolbox for C-Code Generation of First Order Methods. Master's thesis, ETH Zürich, 2011.

[313] VanLoon, K. L., Hoage, P. R., Facker, M. L., and Gaines, L. Computer-Based Steam Management Pays Off. *Oil & Gas Journal* (1987).

[314] Varbanov, P., Perry, S., Makwana, Y., Zhu, X. X., and Smith, R. Top-level Analysis of Site Utility Systems. *Chemical Engineering Research and Design 82*, 6 (2004), 784–795.

[315] Varbanov, P. S., Doyle, S., and Smith, R. Modelling and Optimization of Utility Systems. *Chemical Engineering Research and Design 82*, 5 (2004), 561–578.

[316] Vaz, A. I. F., and Vicente, L. N. A Particle Swarm Pattern Search Method for Bound Constrained Global Optimization. *Journal of Global Optimization 39* (2007), 197–219.

[317] Vigerske, S. Personal Communication, November 2013.

[318] Virtual Materials Group (VMG). VMGSim. http://www.virtualmaterials.com/node/6, 2013.

[319] Virtual Materials Group (VMG). VMGThermo. http://www.virtualmaterials.com/vmgthermo, 2013.

[320] Vouzis, P., Bleris, L., Arnold, M., and Kothare, M. A System-on-a-Chip Implementation for Embedded Real-Time Model Predictive Control. *IEEE Transactions on Control Systems Technology 17*, 5 (2009), 1006–1017.

[321] W. H. Press and S. A. Teukolsky and W. T. Vetterling and B. P. Flannery. *Numerical Recipes: The Art of Scientific Computing*, vol. 3. Cambridge University Press, 2007.

[322] W. Hock and K. Schittkowski. *Test examples for nonlinear programming codes*, vol. 187 of *Lecture Notes in Economics and Mathematical Systems*. Springer, Berlin-Heidelberg-New York, 1981.

[323] Wächter, A., and Biegler, L. T. On the Implementation of a Primal-Dual Interior Point Filter Line Search Algorithm for Large-Scale Nonlinear Programming. *Mathematical Programming 106*, 1 (2006), 25–57.

[324] Wagner, W., Cooper, J. R., Dittmann, A., Kijima, J., Kretzschmar, H.-J., Kruse, A., Mares, R., Oguchi, K., Sato, H., Stocker, I., Sifner, O., Takaishi, Y., Tanishita, I., Trubenbach, J., and Willkommen, T. The IAPWS Industrial Formulation 1997 for the Thermodynamic Properties of Water and Steam. *ASME Journal of Engineering for Gas Turbines and Power 122* (2000), 150–182.

[325] Wagner, W., and Pruss, A. The IAPWS Formulation 1995 for the Thermodynamic Properties of Ordinary Water Substance for General and Scientific Use. *Journal of Physical and Chemical Reference Data 31* (2002), 387–535.

[326] Wang, L. *Model Predictive Control System Design and Implementation using MATLAB*. Springer, 2009.

[327] Wang, Y., and Boyd, S. Fast Model Predictive Control Using Online Optimization. *IEEE Transactions on Control Systems Technology 18*, 2 (2010), 267–278.

[328] Westerberg, A., and Benjamin, D. Thoughts on a future equation-orientation flowsheeting system. Technical Report Paper 92, Department of Chemical Engineering, Carnegie Melon, 1983.

[329] Whaley, R. C., Petitet, A., and Dongarra, J. J. Automated empirical optimization of software and the ATLAS project. *Parallel Computing 27*, 1–2 (2001), 3–35.

[330] Wills, A. QPC v2.0. `http://sigpromu.org/quadprog/index.html`, 2010.

[331] Wills, A., Bates, D., Fleming, A., Ninness, B., and Moheimani, S. Application of MPC to an Active Structure using Sampling Rates up to 25kHz. Published Internal Technical Report EE05035, 2005.

[332] Wills, A., Bates, D., Fleming, A., Ninness, B., and Moheimani, S. R. Model Predictive Control Applied to Constraint Handling in Active Noise and Vibration Control. *IEEE Transactions on Control Systems Technology 16*, 1 (Dec 2008), 3–12.

[333] Wills, A., Mills, A., and Ninness, B. FPGA Implementation of an Interior-Point Solution for Linear Model Predictive Control. In *Proceedings of the 18th IFAC World Congress* (2011), pp. 14527–14532.

[334] Wilson, D. I., and Young, B. R. The Seduction of Model Predictive Control. *Electrical & Automation Technology* (2006), 27–28.

[335] Wolfram. Mathematica 9. `http://www.wolfram.com/mathematica/`, 2013.

[336] Wong, E. *Active-Set Methods for Quadratic Programming.* PhD thesis, University of California, San Diego, California, USA, 2011.

[337] Wright, S. J. A Path-Following Interior-Point Algorithm for Linear and Quadratic Optimization Problems. *Annals of Operations Research 62* (1996), 103–130.

[338] Wright, S. J. Applying New Optimization Algorithms to Model Predictive Control. In *Chemical Process Control-V, CACHE, AIChE Symposium* (1997), vol. 93, pp. 147–155.

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

[340] Wright, S. J. Modified Cholesky Factorizations in Interior-Point Algorithms for Linear Programming. *SIAM Journal On Optimization 9* (1999), 1159–1191.

[341] Xilinx. *Spartan-3E Starter Kit Board User Guide*, 1.0 ed., 2006.

[342] Xilinx. *Spartan-3E FPGA Family: Complete Data Sheet*, 2008.

[343] Xilinx. AccelDSP Synthesis Tool, Accesed September 2013. `http://www.xilinx.com/tools/acceldsp.htm`.

[344] Xilinx. PandaBoard, Accesed September 2013. `http://pandaboard.org/`.

[345] Yildirim, E. A., and Wright, S. J. Warm-Start Strategies in Interior-Point Methods for Linear Programming. *SIAM Journal on Optimization 12*, 3 (2002), 782–810.

[346] Yiu, J. *The Definitive Guide to the ARM Cortex-M0.* Newnes, 2011.

[347] Zhu, C., Byrd, R. H., and Nocedal, J. L-BFGS-B: Algorithm 778: L-BFGS-B FORTRAN Routines for Large Scale Bound Constrained Optimization. *ACM Transactions on Mathematical Software 23*, 4 (2007), 550–560.

[348] Zometa, P., Kögel, M., and Findeisen, R. Implementation aspects of model predictive control for embedded systems. In *Proc. American Control Conference (ACC), 2012* (Montreal, Canada, 2012), pp. 1205–1210.

[349] Zometa, P., Kögel, M., and Findeisen, R. μAO-MPC: A free code generation tool for embedded real-time linear model predictive control. In *Proc. American Control Conference (ACC), 2013* (Washington D.C., USA, 2013), pp. 5340–5345.

Appendix A

Model Predictive Control Algorithms and Software

This appendix chapter provides additional detail to expand on Chapters 3 and 4, including the source for both quadratic programming solvers, as well as a summary of the jMPC Toolbox and implementation details for embedded MPC.

A.1 Quadratic Programming Solvers

The MATLAB implementations of the quadratic programming solvers developed in this work are listed below, while the C-code implementations are included on the Appendix DVD under jMPC Toolbox/Source.

A.1.1 `quad_wright` Solver

Solves a Quadratic Program (QP) of the form

$$\begin{aligned} \min_{\mathbf{x}} \ & \frac{1}{2}\mathbf{x}^T\mathbf{H}\mathbf{x} + \mathbf{f}^T\mathbf{x} \\ \text{subject to: } & \mathbf{A}\mathbf{x} \le \mathbf{b} \end{aligned}$$

based on an algorithm described by Stephen Wright in [338]. The algorithm implements a primal-dual infeasible-interior-point solver with optional warm-starting.

```
function [z,exitflag,iter,lam,t] = quad_wright(H,f,A,b,maxiter,tol,verbose,
                                                z0,lam0,t0)
```

```
% Solve quadratic programming problem using Wright's (1997) Method
% Minimise 1/2x'Hx + f'x
% Subject to: Ax <= b

% Copyright (C) 2009-2013 Jonathan Currie (www.i2c2.aut.ac.nz)

%Length of constraint matrix
mc = length(b);
%Number of decision variables
ndec = length(f);

%Default Args
if(nargin < 10), t = []; else t = t0; end
if(nargin < 9), lam = []; else lam = lam0; end
if(nargin < 8), z = []; else z = z0; end;
if(nargin < 7 || isempty(verbose)), verbose = 0; end
if(nargin < 6 || isempty(tol)), tol = 1e-6; end
if(nargin < 5 || isempty(maxiter)), maxiter = 200; end
%Test for Warm Start
pmax = max(max(abs([H f; A b])));
if(pmax > 1)
    WARMVAL = sqrt(pmax);
else
    WARMVAL = 0.5;
end
if(isempty(z)) %cold
    z = zeros(ndec,1);
    lam = WARMVAL*ones(mc,1);
    t = WARMVAL*ones(mc,1);
    wmode = 0;
elseif(isempty(lam)) %just primal
    lam = WARMVAL*ones(mc,1);
    t = WARMVAL*ones(mc,1);
    wmode = 1;
elseif(isempty(t)) %primal + dual
    t = WARMVAL*ones(mc,1);
    wmode = 2;
else %all
    wmode = 3;
end

%Default Values
sigma = 0.1; ascale = 1; inftol = tol*10;
exitflag = 1; cholfail = 0;
mu = t'*lam / mc; At = A';
mr2_1 = 100; mr2 = 10;
%Initial Residuals
r1 = -H*z - At*lam - f;
r2 = -A*z + b;
%Linsolve options
opU.UT = true; opUT.UT = true; opUT.TRANSA = true;

if(verbose)
    fprintf('------------------------------------------------\n');
    fprintf('QuadWright QP Solver [MATLAB Double Version]\n');
    switch(wmode)
        case 3, fprintf(' Warm Start: Primal + Dual + Slack\n');
        case 2, fprintf(' Warm Start: Primal + Dual\n');
        case 1, fprintf(' Warm Start: Primal\n');
    end
    fprintf(' %4d Decision Variables\n %4d Inequality Constraints\n',ndec,mc);
```

```
        fprintf('-----------------------------------------------\n');
        fprintf('iter           phi              mu         sigma
                      alpha       max(r1)       max(r2)\n');
    end

    %Begin Searching
    for iter = 1:maxiter
        %Create common matrices
        ilam = 1./lam;
        ilamt = ilam.*t;
        lamt = lam./t;
        mesil = mu*sigma.*ilam;
        IGA = bsxfun(@times,A,lamt); %matrix * diagonal matrix
        igr2 = lamt.*(r2 - mesil);

        %Solve
        [R,p] = chol(H+At*IGA);
        if(~p)
            %exploit matrix properties for solving
            del_z = linsolve (R, linsolve (R, (r1+At*igr2), opUT), opU);
        else %Not Positive Definite
            if(verbose), fprintf(2,'\b (Cholesky Failed)\n'); end
            del_z = (H+At*IGA)\(r1+At*igr2);
            cholfail = cholfail + 1;
            if(cholfail > 2)
                exitflag = -2;
                break;
            end
            %Pull back maximum step size
            ascale = ascale - 0.1;
        end
        del_lam = -igr2 + IGA*del_z;
        del_t = -t + mesil - ilamt.*del_lam;

        %Decide on suitable affine step-length
        duals = [lam;t];
        delta = [del_lam;del_t];
        index = delta < 0;
        if(any(index))
            %solves for min ratio (max value of alpha allowed)
            alpha = 0.9995*min(-duals(index)./delta(index));
        else
            alpha = 0.999995;
        end
        %Check for numerical problems (alpha will go negative)
        if(alpha < eps(1)), exitflag = -3; break; end
        %Local Scaling
        alpha = alpha*ascale;
        %Increment
        lam = lam + alpha*del_lam;
        t = t + alpha*del_t;
        z = z + alpha*del_z;

        %Update residuals
        r1 = (1-alpha)*r1; %equiv to r1 = -H*z - At*lam - f
        r2 = -A*z + b;
        %Complementarity Gap
        muold = mu;
        mu = t'*lam / mc;
        %Infeasibility Phi
        mr2_2 = mr2_1; mr2_1 = mr2;
        mr1 = max(abs(r1)); mr2 = max(abs(r2-t));
        phi = (max([mr1,mr2]) + t'*lam)/pmax;
        if(verbose)
            fprintf('%3d  %13.5g  %13.5g  %11.5g  %11.5g   %11.5g   %11.5g\n',
                    iter,phi,mu,sigma,alpha,mr1,mr2);
        end
        %Check for NaNs
        if(isnan(mu) || isnan(phi)), exitflag = -3; break; end
        %Check Convergence
        if(mu <= tol && phi<=tol)
            exitflag = 1;
            if(verbose)
                fprintf('-----------------------------------------------\n');
                fprintf(' Successfully solved in %d Iterations\n Final phi: %d,
                          mu %g [tol %g]\n',iter,phi,mu,tol);
                fprintf('-----------------------------------------------\n');
            end
            return
        end
        %Check For Primal Infeasible
        if(iter > 6 && mr2/pmax > tol/10)
            if(abs(mr2-mr2_1)/mr2 < inftol && abs(mr2_1-mr2_2)/mr2_1 < inftol)
                if(verbose)
                    fprintf(2,'\b (Primal Infeasibility Detected)\n');
                end
                exitflag = -4;
                break;
            end
        end
        %Solve centering parameter (Mehrotra's Heuristic)
        sigma = min((mu/muold)^3,0.99999);
    end

    %If here, either bailed on Cholesky or Iterations Expired
    if(exitflag==1), exitflag = -1; end %expired iters
    %Optional Output
    if(verbose)
        fprintf('-----------------------------------------------\n');
        switch(exitflag)
            case -1, fprintf(' Maximum Iterations Exceeded\n');
            case -2, fprintf(2,' Failed - Cholesky Factorization Reported Errors\n');
            case -3, fprintf(2,' Failed - Numerical Errors Detected\n');
            case -4, fprintf(2,' Failed - Problem Looks Infeasible\n');
        end
        fprintf(' Final phi: %g, mu %g [tol %g]\n',phi,mu,tol);
        fprintf('-----------------------------------------------\n');
    end
```

A.1.2 quad_mehrotra Solver

Solves a Quadratic Program (QP) of the form

$$\min_{\mathbf{x}} \frac{1}{2}\mathbf{x}^T\mathbf{H}\mathbf{x} + \mathbf{f}^T\mathbf{x}$$
$$\text{subject to: } \mathbf{A}\mathbf{x} \leq \mathbf{b}$$

based on an algorithm described by Stephen Wright in [338] and second order modification by Sanjay Mehrotra [217]. The algorithm implements a predictor-corrector primal-dual infeasible-interior-point solver with optional warm-starting.

```
function [z,exitflag,iter,lam,t] = quad_mehrotra(H,f,A,b,maxiter,tol,verbose,
                                                z0,lam0,t0)
% Solve quadratic programming problem using Wright's Method & Mehrota's
% Predictor - Corrector Method
% Minimise 1/2x'Hx + f'x
% Subject to: Ax <= b

% Copyright (C) 2011-2013 Jonathan Currie (www.i2c2.aut.ac.nz)

%Length of constraint matrix
mc = length(b);
%Number of decision variables
ndec = length(f);

%Default Args
if(nargin < 10), t = []; else t = t0; end
if(nargin < 9), lam = []; else lam = lam0; end
if(nargin < 8), z = []; else z = z0; end;
if(nargin < 7 || isempty(verbose)), verbose = 0; end
if(nargin < 6 || isempty(tol)), tol = 1e-6; end
if(nargin < 5 || isempty(maxiter)), maxiter = 200; end
%Test for Warm Start
pmax = max(max(abs([H f; A b])));
if(pmax > 1)
    WARMVAL = sqrt(pmax);
else
    WARMVAL = 0.5;
end
if(isempty(z)) %cold
    z = zeros(ndec,1);
    lam = WARMVAL*ones(mc,1);
    t = WARMVAL*ones(mc,1);
    wmode = 0;
elseif(isempty(lam)) %just primal
    lam = WARMVAL*ones(mc,1);
    t = WARMVAL*ones(mc,1);
    wmode = 1;
elseif(isempty(t)) %primal + dual
    t = WARMVAL*ones(mc,1);
    wmode = 2;
else %all
    wmode = 3;
```

```
end

%Default Values
ascale = 1; inftol = tol*10;
exitflag = 1; cholfail = 0;
mu = t'*lam / mc; At = A';
mr2_1 = 100; mr2 = 10; phi1 = -Inf; phi = -Inf;
%Initial Residuals
r1 = -H*z - At*lam - f;
r2 = -A*z + b;
%Linsolve options
opU.UT = true; opUT.UT = true; opUT.TRANSA = true;

if(verbose)
    fprintf('------------------------------------------------\n');
    fprintf('QuadMehrotra QP Solver [MATLAB Double Version]\n');
    switch(wmode)
        case 3, fprintf(' Warm Start: Primal + Dual + Slack\n');
        case 2, fprintf(' Warm Start: Primal + Dual\n');
        case 1, fprintf(' Warm Start: Primal\n');
    end
    fprintf(' %4d Decision Variables\n %4d Inequality Constraints\n',ndec,mc);
    fprintf('------------------------------------------------\n');
    fprintf('iter           phi              mu        sigma
                  alpha       max(r1)       max(r2)\n');
end

for iter = 1:maxiter
    %Create common matrices
    ilam = 1./lam;
    ilamt = ilam.*t;
    lamt = lam./t;
    IGA = bsxfun(@times,A,lamt); %matrix * diagonal matrix
    igr2 = lamt.*r2;
    RHS = r1+At*igr2;

    %Solve Linear System
    [R,p] = chol(H+At*IGA);
    if(~p)
        %exploit matrix properties for solving
        del_z = linsolve (R, linsolve (R, RHS, opUT), opU);
    else %Not Positive Definite
        if(verbose), fprintf(2,'\b (Cholesky Failed)\n'); end
        del_z = (H+At*IGA)\RHS;
        cholfail = cholfail + 1;
        if(cholfail > 2)
            exitflag = -2;
            break;
        end
        %Pull back maximum step size
        ascale = ascale - 0.1;
    end
    del_lam = -igr2+IGA*del_z;
    del_t = -t - ilamt.*del_lam;

    %Decide on suitable affine step-length
    duals = [lam;t];
    delta = [del_lam;del_t];
    index = delta < 0;
    if(any(index))
        %solves for min ratio (max value of alpha allowed)
        alpha = 0.9995*min(-duals(index)./delta(index));
    else
        alpha = 0.999995;
    end
    %Check for numerical problems (alpha will go negative)
    if(alpha < eps(1)), exitflag = -3; break; end
    %Local Scaling
    alpha = alpha*ascale;
    %Solve for Centering Variable
    mu1 = dot(lam+alpha*del_lam,t+alpha*del_t)/mc;
    sigma = min((mu1/mu)^3,0.99999);

    %Solve for Correction (2nd Derivative)
    term = (sigma*mu - del_lam.*del_t)./t;
    RHS  = RHS - A'*term;
    if(~p)
        del_z = linsolve (R, linsolve (R, RHS, opUT), opU);
    else
        del_z = (H+At*IGA)\RHS;
    end
    del_lam = -igr2+IGA*del_z + term;
    del_t = -A*del_z + r2 - t;

    %Decide on suitable corrector step-length
    duals = [lam;t];
    delta = [del_lam;del_t];
    index = delta < 0;
    if(any(index))
        %solves for min ratio (max value of alpha allowed)
        alpha = 0.9995*min(-duals(index)./delta(index));
    else
        alpha = 0.999995;
    end
    %Check for numerical problems (alpha will go negative)
    if(alpha < eps(1)), exitflag = -3; break; end
    %Local Scaling
    alpha = alpha*ascale;
    %Sum Increments
    z   = z + alpha*del_z;
    lam = lam + alpha*del_lam;
    t   = t + alpha*del_t;

    %Update residuals
    r1 = (1-alpha)*r1; %equiv to r1 = -H*z - At*lam - f;
    r2 = -A*z + b;
    %Complementarity Gap
    mu = t'*lam / mc;
    %Infeasibility Phi
    mr2_2 = mr2_1; mr2_1 = mr2;
    mr1 = max(abs(r1)); mr2 = max(abs(r2-t));
    phi2 = phi1; phi1 = phi;
    phi = (max([mr1,mr2]) + t'*lam)/pmax;
    if(verbose)
        fprintf('%3d  %13.5g  %13.5g  %11.5g  %11.5g   %11.5g   %11.5g\n',
                iter,phi,mu,sigma,alpha,mr1,mr2);
    end
    %Check for NaNs
    if(isnan(mu) || isnan(phi)), exitflag = -3; break; end
    %Check Convergence
    if(mu <= tol && phi<=tol)
        exitflag = 1;
```

```
        if(verbose)
            fprintf('-----------------------------------------------\n');
            fprintf(' Successfully solved in %d Iterations\n Final phi: %d,
                    mu %g [tol %g]\n',iter,phi,mu,tol);
            fprintf('-----------------------------------------------\n');
        end
        return
    end
    %Check For Primal Infeasible
    if(iter > 4 && mr2/pmax > tol/10)
        if((phi > phi1 && phi1 > phi2) || (abs(mr2-mr2_1)/mr2 < inftol &&
           abs(mr2_1-mr2_2)/mr2_1 < inftol))
            if(verbose), fprintf(2,'\b (Primal Infeasibility Detected)\n'); end
            exitflag = -4;
            break;
        end
    end
end

%If here, either bailed on Cholesky or Iterations Expired
if(exitflag==1), exitflag = -1; end %expired iters
%Optional Output
if(verbose)
    fprintf('-----------------------------------------------\n');
    switch(exitflag)
        case -1, fprintf(' Maximum Iterations Exceeded\n');
        case -2, fprintf(2,' Failed - Cholesky Factorization Reported Errors\n');
        case -3, fprintf(2,' Failed - Numerical Errors Detected\n');
        case -4, fprintf(2,' Failed - Problem Looks Infeasible\n');
    end
    fprintf(' Final phi: %g, mu %g [tol %g]\n',phi,mu,tol);
    fprintf('-----------------------------------------------\n');
end
```

A.2 The jMPC Toolbox

The jMPC Toolbox [63] began as a collection of MATLAB m-files in mid 2009 and has been continuously updated through to 2013. While the existing packages described in Section 3.5 had provided some of the functionality we were looking for, they did not provide the full flexibility required to develop our own embedded model predictive controller, nor test the development of our quadratic programming algorithms.

jMPC was freely released under the BSD License in September 2009 and has since been downloaded over 350 times by a range of international academic and commercial users, for purposes ranging from control of vehicle suspension, UAVs and distillation column control to research in nonlinear MPC and optimal control and teaching in graduate courses. I maintain active support for the project, and it is upgraded with new and improved functionality as time allows. The toolbox is available at `http://www.i2c2.aut.ac.nz/Resources/Software/jMPCToolbox.html`, as well as supplied on the Appendix DVD.

A.2.1 Simulation Functionality

jMPC Toolbox provides the simulation environment shown in Figure A.1. A linear MPC controller can be designed, built and simulated with either a linear or nonlinear plant model, using MATLAB, Simulink or an accelerated jMPC engine in C. The

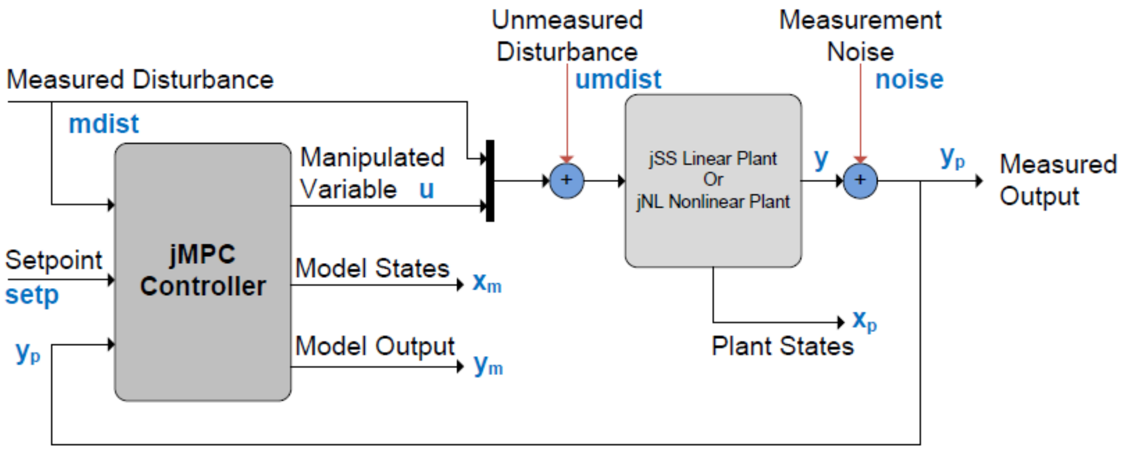


Figure A.1: jMPC Toolbox block diagram.

simulation can accommodate both measured and unmeasured disturbances, as well as as measurement noise. All data (including the actual system and observed states) are automatically saved for post processing and tuning validation. An overloaded `plot` command allows the results to be viewed with a single method call on the `jMPC` object.

A.2.2 Quadratic Programming Solvers

In addition to quadratic programming algorithms developed within this work (`quad_wright` and `quad_mehrotra`), the toolbox also interfaces to a number of 3rd party QP solvers. These are listed below:

1. `quadprog` – QP solver supplied as part of MATLAB's Optimization Toolbox [207].
2. `qpip` – Primal-dual interior point QP solver supplied with QPC [330].
3. `qpas` – Active-set QP solver supplied with QPC [330].
4. `quad_hildreth` – QP solver described by Liuping Wang in [326].

5. OOQP – Stephen Wright's primal-dual interior point QP solver [108].

6. CLP – John Forrest's barrier QP solver [100].

These are interfaced to allow a user to try alternative solution algorithms, or for solving large-scale problems which the supplied solvers are not designed for.

A.2.3 Simulink Implementation

The toolbox includes a Simulink block set for implementing MPC control of Simulink models, and real systems interfaced via an ADC/DAC to Simulink, as done in [66]. The block set includes blocks for MPC and QP solving using both raw Simulink blocks, as well as accelerated blocks written as S Functions in C, again utilizing high-performance linear algebra libraries.

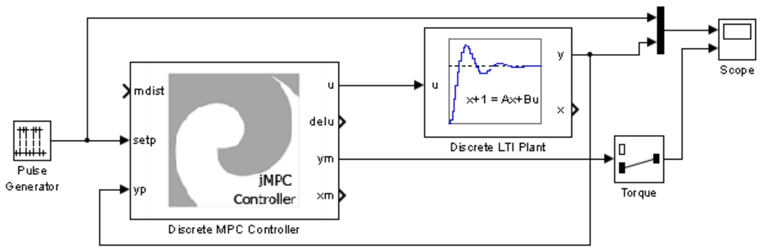


Figure A.2: Example jMPC Simulink block diagram controlling a servo motor.

A.2.4 3D MPC Demo

Supplied with the toolbox are two demos utilizing the Simulink 3D Animation tool for visualizing MPC on a simulated system. As shown in Figure A.3, a SolidWorks model of the Quanser 3DOF Helicopter [265] has been converted to a VRML model, and can be visualized using Simulink. By implementing the nonlinear Equations Of Motion (EOM) together with the visualization, the user can realistically see the result of their control tuning in the real-time animation.

A.2.5 MPC Graphical User Interface

For classroom teaching of MPC, a Graphical User Interface (GUI) provides a live tuning and simulation environment. The GUI has been well received and is used in a number of graduate control courses internationally. The GUI, shown in Figure A.4, provides an easy to use mechanism to load transfer function or state-space models,

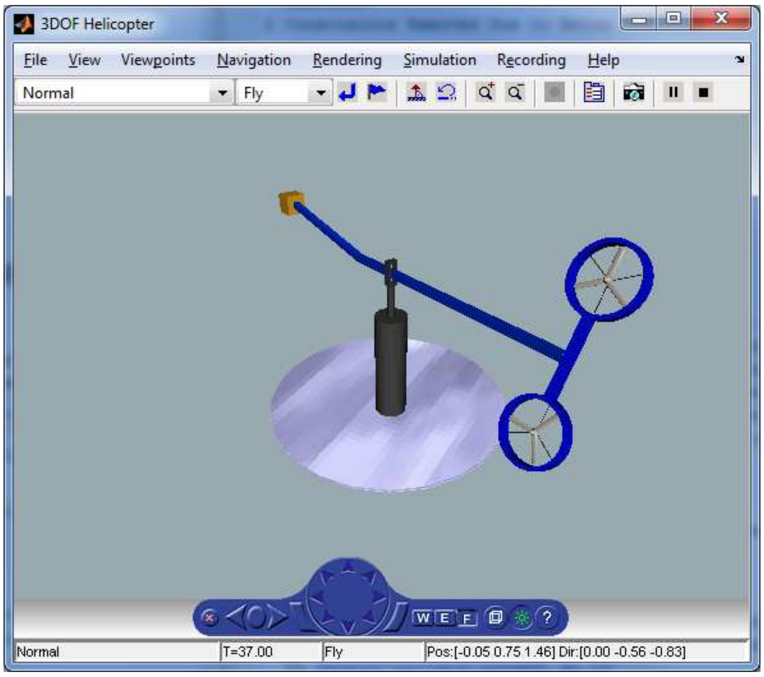

3DOF Helicopter
File View Viewpoints Navigation Rendering Simulation Recording Help
Normal
Fly
W E F
Normal
T=37.00
Fly
Pos:[-0.05 0.75 1.46] Dir:[0.00 -0.56 -0.83]

Figure A.3: 3D visualization of the 3DOF Helicopter.

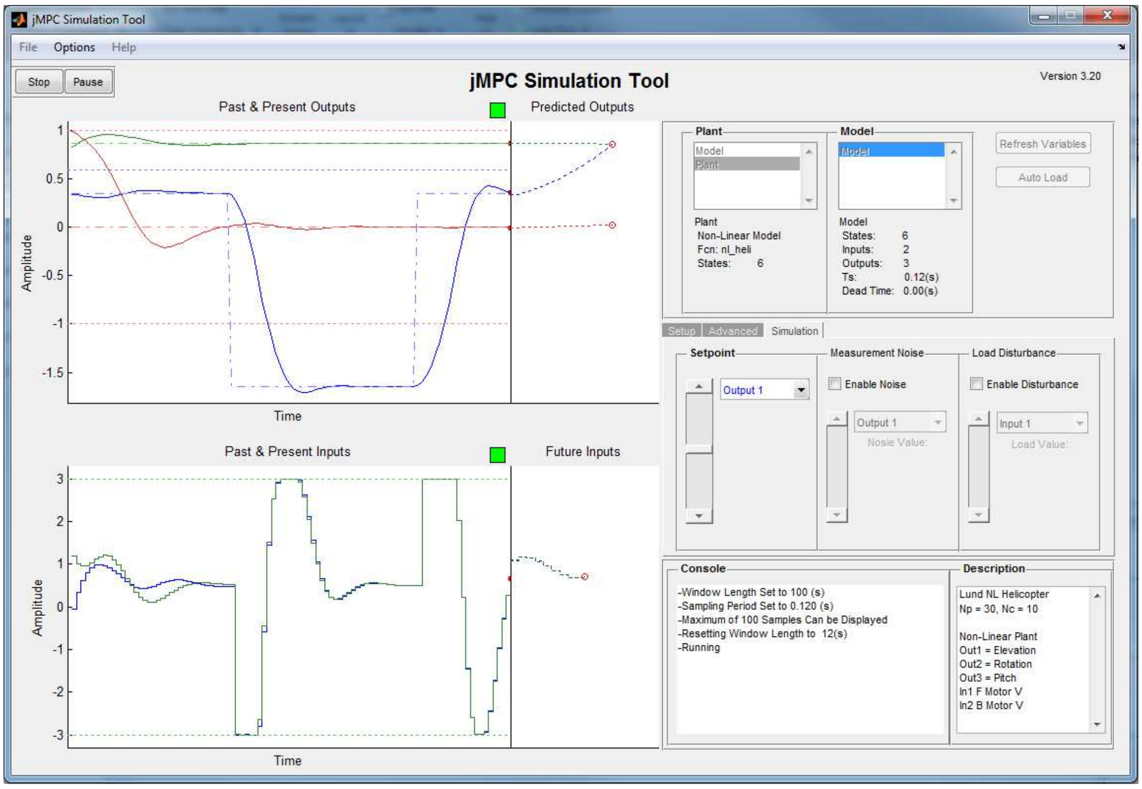
graphically setup the controller tuning parameters and enter system constraints. Once 'Start' is clicked, the controller is built and then simulated against a selected plant model (linear or nonlinear). The simulation is run in real-time, with the process outputs and control inputs displayed on a scrolling figure, and the predicted plant outputs and future control inputs shown to the right of the vertical dashed line. These predicted outputs and future control inputs show what the MPC is 'planning', and aid understanding of the algorithm and its decision making process.

A.2.6 Auto-Code Generation

As detailed in Section 4.3, once the user has confirmed the tuning of their MPC controller is satisfactory, they can then deploy it by automatically generating an embeddable C-code implementation of the controller, in either single or double precision. In addition, a series of test-benches are automatically generated, for both verifying the generated code on the user's computer, as well as on their desired target device.

A.2.7 Algorithm Performance Improvements

The MPC algorithm implemented within the jMPC Toolbox contains a number of performance improvements to increase the achievable sampling rate, lower memory



Plant
Non-Linear Model
Fcn: nl_heli
States: 6

Model
States: 6
Inputs: 2
Outputs: 3
Ts: 0.12(s)
Dead Time: 0.00(s)

```
-Window Length Set to 100 (s)
-Sampling Period Set to 0.120 (s)
-Maximum of 100 Samples Can be Displayed
-Resetting Window Length to  12(s)
-Running
```

Lund NL Helicopter
Np = 30, Nc = 10

Non-Linear Plant
Out1 = Elevation
Out2 = Rotation
Out3 = Pitch
In1 F Motor V
In2 B Motor V

Figure A.4: MPC graphical user interface.

requirements and increase control performance of a deployed controller. These are briefly described below, noting most of these are not novel and are described in standard books or implementations of MPC, however the author cannot find another framework which has collected all of the functionality below in a single package. This makes the jMPC Toolbox unique in its implementation, which when combined with the high-level approach to controller design and simulation, reinforces the valuable contribution it makes to this field of work.

Control Move Blocking Blocking of control moves of arbitrary sized samples blocks is built into the controller formulation, allowing a substantial decrease in required future control moves (and thus decision variables), and still maintain a similar level of control performance.

Disturbance Prediction Modelling Model inputs can be identified as either unmeasured or measured disturbances, and if measured, a model can be used to form a prediction estimation which is used when solving the optimal control moves.

Removing Infinite Constraints As constraints may not exist on every input, input increment or output, constraints which are not required are removed from the control problem. This is also repeated at each sample, when the constraint right hand side **b** vector is recalculated.

Removing Uncontrolled Outputs Outputs which are not controlled but still require constraints (i.e. states where the $\mathbf{C}$ matrix term is 1) can be treated separately in the formulation and updating of the QP problem, speeding up the solution process and reducing memory usage.

Unused Functionality Skipping Within low-level implementations of the controller, functionality that is not required for a particular scenario, such as disturbance prediction modelling or particular categories of constraints are automatically skipped to avoid unnecessary calculation. Furthermore, for controllers generated using the auto-code framework, unused functionality is not built into the controller, further reducing code size.

Global Optimum Checking At every sample the controller checks the global unconstrained minimum to see if it satisfies all constraints. If it does, solving the full QP is skipped and the global solution is used. This approach substantially reduces the number of QP iterations, increasing the power efficiency of the algorithm. Furthermore, as the QP $\mathbf{H}$ matrix is constant throughout the simulation, it can be factorized off-line and thus only a triangular substitution is required at each sample.

Separate Dynamic Memory To reduce the number operations required, dynamic QP variables (i.e. variables which change at each sample, $\mathbf{f}$ and $\mathbf{b}$), are separated into constant components (elements which can be precalculated), and dynamic components (elements which are dependent on the current state). This reduces the number of computations required at each sample, as only the dynamic elements require recalculation.

Soft Constraints In order to maintain feasibility it is generally recommended that all output constraints be implemented as *soft* constraints, meaning they are relaxed via a penalty term in the objective. This allows the constraints to be broken, if say a large disturbance was to effect the system, but penalized so as to minimize the constraint violation.

Warm Starting The warm starting heuristics described in Section 3.4.5.4 are used to automatically initialize the primal, dual and slack variables at each sample.

State Estimation A linear observer is built into the controller algorithm, allowing a Kalman filter to be designed using MATLAB's `dlqe` routine and automatically incorporated. This provides the ability to recover unmeasured states and outputs (for observable systems), required for the implementation of the controller in a real system.

Setpoint Look-ahead While typically academic in application, the controller can utilize future setpoint information to begin moving the controlled outputs before a setpoint has changed.

Automatic Linearization The toolbox can accept a nonlinear model as a collection of first order Ordinary Differential Equations (ODEs) and automatically linearize the system about an input operating point. The routine includes a robust steady-state solution solver, allowing the user to specify only the system inputs and it will automatically find the steady-state point and linearize the model. This enables linear controllers to be developed from first-principle models, simplifying the design flow, as described in Section 4.5.2.

Nonlinear Simulation Following from the above point, if a nonlinear model is supplied to the jMPC constructor, then as well as being automatically linearized, the nonlinear model can be used as the reference plant to control, while the linearized model is used for controller design. This provides a much more challenging simulation scenario for the controller, allowing a more robust validation of tuning and implementation.

In addition to algorithmic improvements, the jMPC controller is implemented in both single and double precision as C MEX functions, allowing high-speed limited precision simulations on a development computer. Using the double precision controller with a small MPC problem (2-3 decision variables, 10-15 constraints), it is not unusual to see sampling rates exceed 1MHz on a standard desktop PC. To see this first-hand, the reader is referred to the Oscillatory SISO MPC Example, located in the `jMPC Toolbox/Examples/Linear_Examples.m` set of examples. By completing MPC simulations at such a high speed, allows rapid tuning and validation of a controller.

A.2.8 jMPC Code Example

To illustrate the functionality of the jMPC Toolbox, the design and implementation of an MPC controller of a highly nonlinear Continuously Stirred Reactor (CSTR) from [126] is presented. A schematic of the system is presented below in Figure A.5.

The control objective is to control the temperature of the reactor (T_r) by adjusting the temperature of the cooling jacket (T_c). Both the concentration and temperature of the feed (C_{af}, T_f) are measured disturbances, and cannot be modified by the controller. The reactor concentration (C_a) is constrained but not controlled to a setpoint.

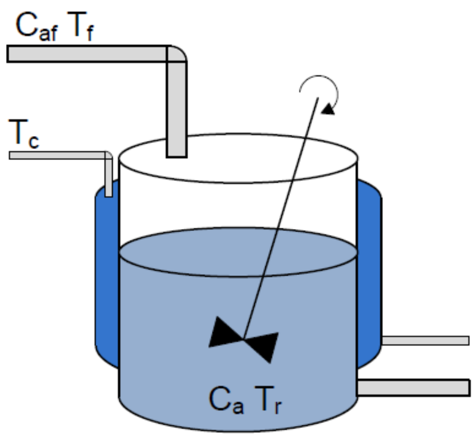


Figure A.5: CSTR example schematic.

The continuous time, ordinary differential equations of the CSTR model are:

$$\begin{aligned}\frac{\partial C_a}{\partial t} &= \frac{q}{V}(C_{af} - C_a) - \sigma \\ \frac{\partial T_r}{\partial t} &= \frac{q}{V}(T_f - T_r) + \frac{H}{C_p\rho}\sigma + \frac{UA}{C_p\rho V}(T_c - T_r)\end{aligned}$$

where

$$\sigma = k_0 e^{\frac{-E}{RT_r}} C_a$$

where the equations represent a continuously stirred tank reactor with a single reaction from A→B, and a complete mass and energy balance.

We have defined the state vector, $\mathbf{x}$, as:

x_1	Concentration of A in reactor (C_a)	[mol/m^3]
x_2	Temperature of reactor (T_r)	[K]

and the output vector, $\mathbf{y}$, as:

y_1	Concentration of A in reactor (C_a)	[mol/m^3]
y_2	Temperature of reactor (T_r)	[K]

and input vector, $\mathbf{u}$, as:

u_1	Concentration of A in feed (C_{af}) (Measured Disturbance)	[mol/m^3]
u_2	Temperature of feed (T_f) (Measured Disturbance)	[K]
u_3	Temperature of cooling jacket (T_c)	[K]

The jacket cooling temperature is limited as:

$278.15 \leq \mathrm{u}_3 \leq 450$ K

and the reactor is constrained to lie between:

$0 \leq y_1 \leq 3$ mol/m^3

$278.15 \leq y_2 \leq 450$ K

A.2.9 Linear MPC with Nonlinear CSTR Simulation

This example will demonstrate the functionality of the jMPC Toolbox for working with nonlinear models. The nonlinear model above will be entered into MATLAB, then converted to a linearized form. From the linearized model, a linear MPC controller will be designed and tuned to control the system within a full nonlinear simulation. The example includes algorithm functionality such as measured disturbance prediction modelling, state constraints and control move blocking, as well as the option to run the simulation automatically within Simulink.

The complete code example is available on the Appendix DVD under `jMPC Toolbox/Examples/Documentation/CSTR_Example.m`.

A.2.9.1 Step 1 - Build Nonlinear ODE Callback Function

The first step is to write an m-file which contains the above expressions, suitable for use with a MATLAB integrator:

```
function xdot = nl_cstr(t,x,u,param)
% Nonlinear CSTR model

% Assign Parameters
[q,V,k0,E,R,H,Cp,rho,UA] = param{:};

r = k0*exp(-E/(R*x(2)))*x(1);

xdot(1,1) = q/V*(u(1)-x(1)) - r;
xdot(2,1) = q/V*(u(2)-x(2)) + (H/(Cp*rho))*r + (UA)/(Cp*rho*V)*(u(3)-x(2));
end
```

The file, `nl_cstr.m`, is saved in a suitable folder on the MATLAB path.

A.2.9.2 Step 2 - Build the `jNL` Object

The next step is to build a `jNL` object, which is a MATLAB class supplied with the jMPC Toolbox for describing nonlinear models. The function above is passed as a function handle to the `jNL` constructor:

```
% Parameters
q = 100;     % Volumetric flow rate [m^3/min]
V = 100;     % Volume in reactor [m^3]
k0 = 7.2e10; % Pre-exponential nonthermal factor [1/min]
E = 7.2752e4;% Activation energy in the Arrhenius Equation [J/mol]
R = 8.31451; % Universal Gas Constant [J/mol-K]
H = 5e4;     % Heat of Reaction [J/mol]
Cp = .239;   % Heat capacity (J/g-K)
rho = 1000;  % Density (g/m^3)
UA = 5e4;    % Heat Transfer * Area [J/min-K]

% Output Matrix
C = eye(2);

% Nonlinear Plant
param = {q,V,k0,E,R,H,Cp,rho,UA};  %parameter cell array
Plant = jNL(@nl_cstr,C,param);
```

Note we have passed the parameters as a cell array, and also retained the linear **C** output matrix for now. This could also be a function handle to a nonlinear output function.

A.2.9.3 Step 3 - Linearize the `jNL` Object

In order to use the nonlinear plant with our linear MPC controller, we must linearize the ODE function and generate the required linear state space model. The system is linearized about an unsteady operating point as specified in the original reference:

```
% Initial U
CAf = 1; % Feed Concentration [mol/m^3]
Tf = 350; % Feed Temperature [K]
Tc = 300; % Coolant Temperature [K]

% Linearize Plant
u0 = [CAf Tf Tc]';
xop = [0.5 350]'; %unstable operating point [Ca Tr]
Model = linearize(Plant,u0,xop,'ode15s')
```

Note in this instance we have specified to use `ode15s` for solving the ODE during linearization. While this is not used in this particular scenario (we are not solving for a steady state here), it will be used for all future simulations of this `jNL` plant (instead of the default `ode45`). This is due to this particular ODE being quite stiff.

A.2.9.4 Step 4 - Create the Controller Model

Returned from the routine `linearize` is a MATLAB `lti` object containing the linearized model of our system. This must be converted to a `jSS` object, and discretized to be used to build an MPC controller:

```
% Build jSS object & discretize @ Ts = 0.05s
Model = jSS(Model);
Ts = 0.05; %sample time
Model = c2d(Model,Ts)
```

In order to assign the measured disturbances in the model, we use the following method:

```
% Set Measured Disturbances (Caf,Tf)
Model = SetMeasuredDist(Model,[1 2]); %Specified indices within B
```

A.2.9.5 Step 5 - Setup MPC Specifications and Options

The MPC Controller design components are specified below, including prediction and control horizons, constraints, tuning and general options. Note the reactor concentration has an output weighting of 0, indicating this output is not controlled to a setpoint, but we still wish to place constraints on it (effectively a state constraint). This modification reduces the size of the QP being solved, increasing the efficiency of the controller.

```
% Horizons
Np = 30;           %Prediction Horizon
Nc = [10 10 10];   %Blocking moves

% Constraints
con.u = [278.15 450 20]; %umin umax delumax
con.y = [0      3;       %ymin1 ymax1
         278.15  450];   %ymin2 ymax2

% Controller Weighting
uwt = 1;
ywt = [0 5]';

% Discrete Observer with W,V = eye()
Kest = dlqe(Model);

% Set Options
opts = jMPCset('InitialU',u0,...  %Set initial control input at lin point
               'InputNames',{'Feed Con','Feed Temp','Jacket Temp'},...
               'InputUnits',{'mol/m^3','K','K'},...
               'OutputNames',{'Reactor Con','Reactor Temp'},...
               'OutputUnits',{'mol/m^3','K'})
```

A.2.9.6 Step 6 - Setup Simulation Options

Next we must set up the simulation environment for the specified MPC controller:

```
% Simulation Length
T = 300;

% Setpoint (CA)
setp = zeros(T,1);
setp(:,1) = xop(2);
setp(50:end,1) = xop(2)+25;
setp(200:end,1) = xop(2)-25;

% Measured Disturbances (Caf Tf)
mdist = zeros(T,2); mdist(:,1) = CAf; mdist(:,2) = Tf;
mdist(130:140,1) = CAf+0.1;              %Step disturbance of Caf
mdist(220:260,2) = Tf-linspace(0,20,41); %Slow cooling of Tf
mdist(261:end,2) = Tf-20;                %Tf final

% Set Initial values at linearization point
Plant.x0 = xop;
Model.x0 = xop;
```

A.2.9.7 Step 7 - Build the MPC Controller and Simulation Options

Now we have specified all we need to build an MPC controller and simulation environment, we call the two required constructor functions:

```
%-- Build MPC & Simulation --%
MPC1 = jMPC(Model,Np,Nc,uwt,ywt,con,Kest,opts);
simopts = jSIM(MPC1,Plant,T,setp,[],mdist);
```

`MPC1` is created using the `jMPC` constructor, where the variables we have declared previously are passed as initialization parameters. Simulation options `simopts` is created similarly, except using the `jSIM` constructor.

A.2.9.8 Step 8 - Run the MPC Simulation and Plot Results

With the controller and environment built, we can run the simulation, and plot the results. We use Simulink as the evaluation environment as it runs significantly faster than raw MATLAB for nonlinear simulations:

```
%-- Simulate & Plot Result --%
simresult = sim(MPC1,simopts,'Simulink');
plot(MPC1,simresult,'detail');
```

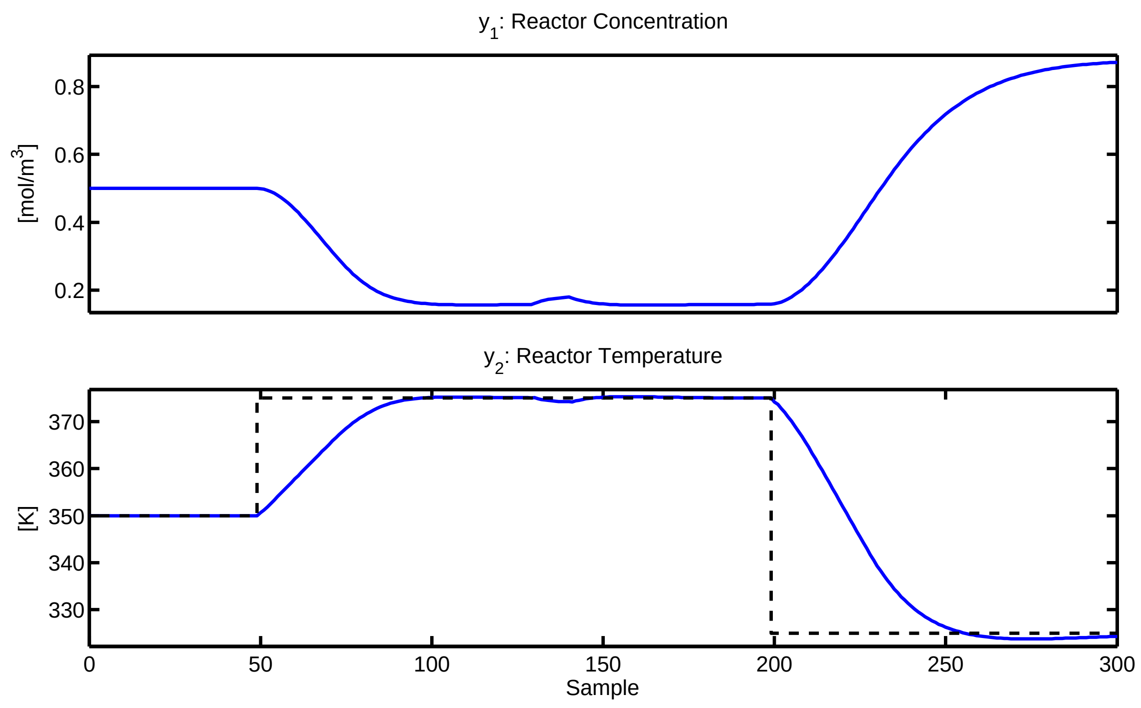


Figure A.6: CSTR simulation output response (see also Figure A.7).

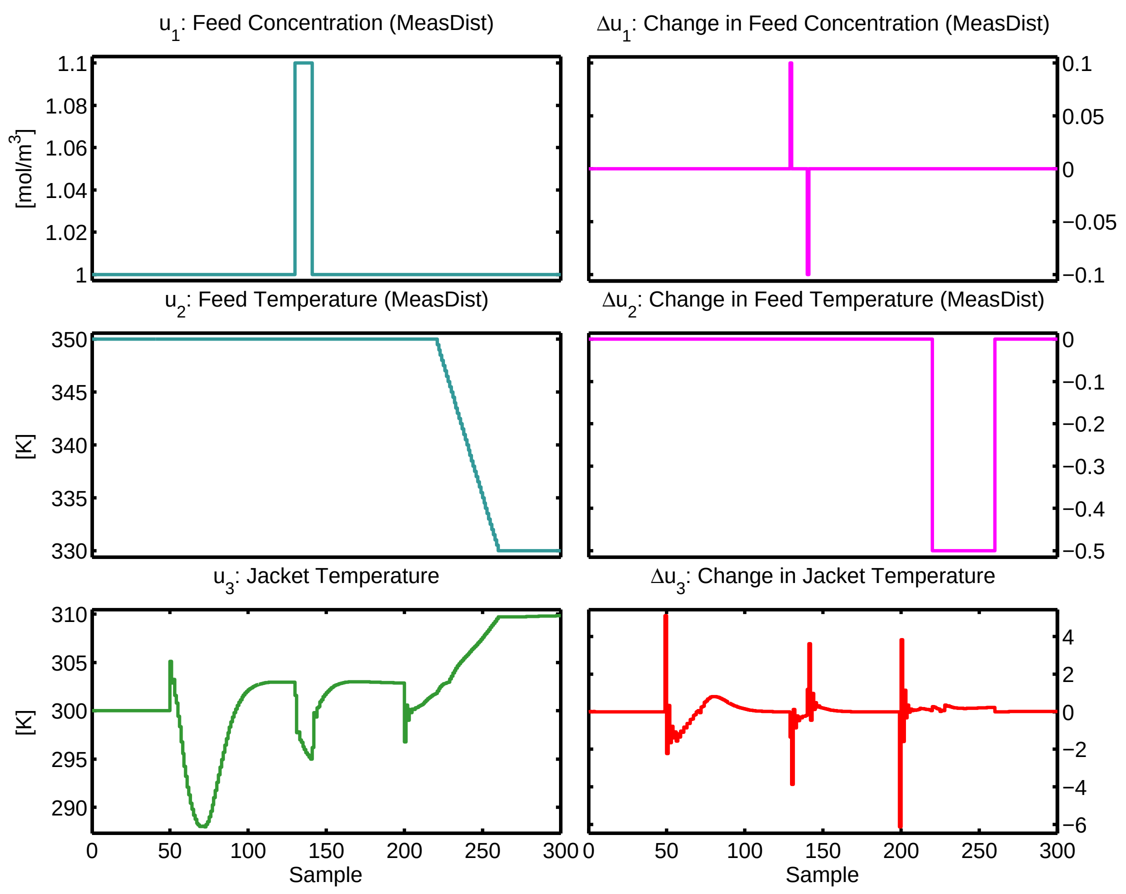


Figure A.7: CSTR simulation inputs (see also Figure A.6).

As shown in Figures A.6 and A.7 the system shows good control with minimal overshoot even for large step changes. This particular problem presents a considerable challenge to linear MPC based on the unstable operating point and significant nonlinearities of the system. However correct tuning can give good results, even when responding to disturbances.

A.3 Random MPC QP Generation

The following function creates an array of quadratic programs resulting from random model predictive controllers, to enable testing of quadratic programming solvers with real MPC optimization problems. Each controller is built with a random stable, discrete model, using the Control Systems Toolbox function `drmodel`, with all inputs and outputs tightly constrained. From the controller definition, a single simulation step is taken, and the resulting quadratic program saved into an array. This allows the function to create any number of arbitrarily sized quadratic programs, that all result from a model predictive control formulation.

```
function [QP,MPC] = mpc_qps(Np,Nc,n_in,n_out,states,no,scale)
%MPC_QPS Create QP problems based on an MPC formulation

if(nargin < 7), scale = 1; end
if(nargin < 6 || isempty(no)), no = 10; end

%Static Constraints
con.u = [-1*ones(n_in,1)  1*ones(n_in,1)  0.1*ones(n_in,1)];
con.y = [0*ones(n_out,1) 1.01*ones(n_out,1)];
%Static Weighting
uwt = ones(n_in,1);
ywt = ones(n_out,1);
%Static Variables
yp = zeros(n_out,1);
setp = ones(1,n_out);
warningstate1 = warning('off', 'jMPCToolbox:SETP');

%Preallocate Return Variable
QP.H = zeros(Nc*n_in,Nc*n_in,no);
QP.f = zeros(Nc*n_in,no);
QP.A = zeros(4*Nc*n_in+2*Np*n_out,Nc*n_in,no);
QP.b = zeros(size(QP.A,1),no);
MPC = cell(no,1);
opts = jMPCset('ScaleSystem',scale);

for k = 1:no
    %Create Random Model
    [A,B,C,D] = drmodel(states,n_out,n_in);
    Model = jSS(A,B,C,zeros(n_in,n_out),0.1);
    Model.x0 = zeros(states,1);
    %-- Build MPC --%
    MPC1 = jMPC(Model,Np,Nc,uwt,ywt,con,[],opts);
    %Allocate initial values
    sModel = MPC1.Model;
    u = MPC1.initial.u;
    del_xm = sModel.x0;

    %State Estimator Update
    del_xm = MPC1.state_est.IKC*del_xm + MPC1.state_est.Kest*yp;
    %Update Dynamic RHS of QP Problem
    [b,f] = MPC1.update_rhs(del_xm,u,setp,0,k);

    %Save dynamic QP variables
    QP.H(:,:,k) = MPC1.QP.H;
    QP.f(:,k) = f;
    QP.A(:,:,k) = MPC1.constraints.A;
    QP.b(:,k) = b;
    MPC{k} = MPC1;
end

%Restore warning state
warning(warningstate1)
```

A.4 Additional Embedded MPC Detail

The following sections expand on the material presented in Chapter 4, providing additional detail on our initial investigation into FPGA based embedded MPC, as well as specifics of the linear algebra library developed and how a processor-in-the-loop implementation is run within the jMPC Toolbox.

A.4.1 Initial Xilinx FPGA Implementation

As described in 4.2.1.5 the initial target decision was to pursue an FPGA route, motivated by a desire to parallelize the MPC algorithm. By exploiting the available resources on the FPGA, specifically the multiple hardware multipliers, we intended to develop a soft-processor with co-processor based architecture as shown in Figure A.8. The soft-processor implemented was a MicroBlaze, a 32bit generic design that is provided at no cost by Xilinx in ISE, and it communicated to a custom hardware accelerator acting as a co-processor for linear algebra functions.

As shown in Figure A.8, the architecture leveraged parallelism in two ways; firstly by effectively providing two co-processors in parallel, and secondly by utilizing parallelized linear algebra routines for functions such as matrix × vector or element-wise operations. By exploiting the fact that not all operations in the `quad_wright` algorithm were required to complete sequentially, it was possible to at least theorize this two-level parallel architecture.

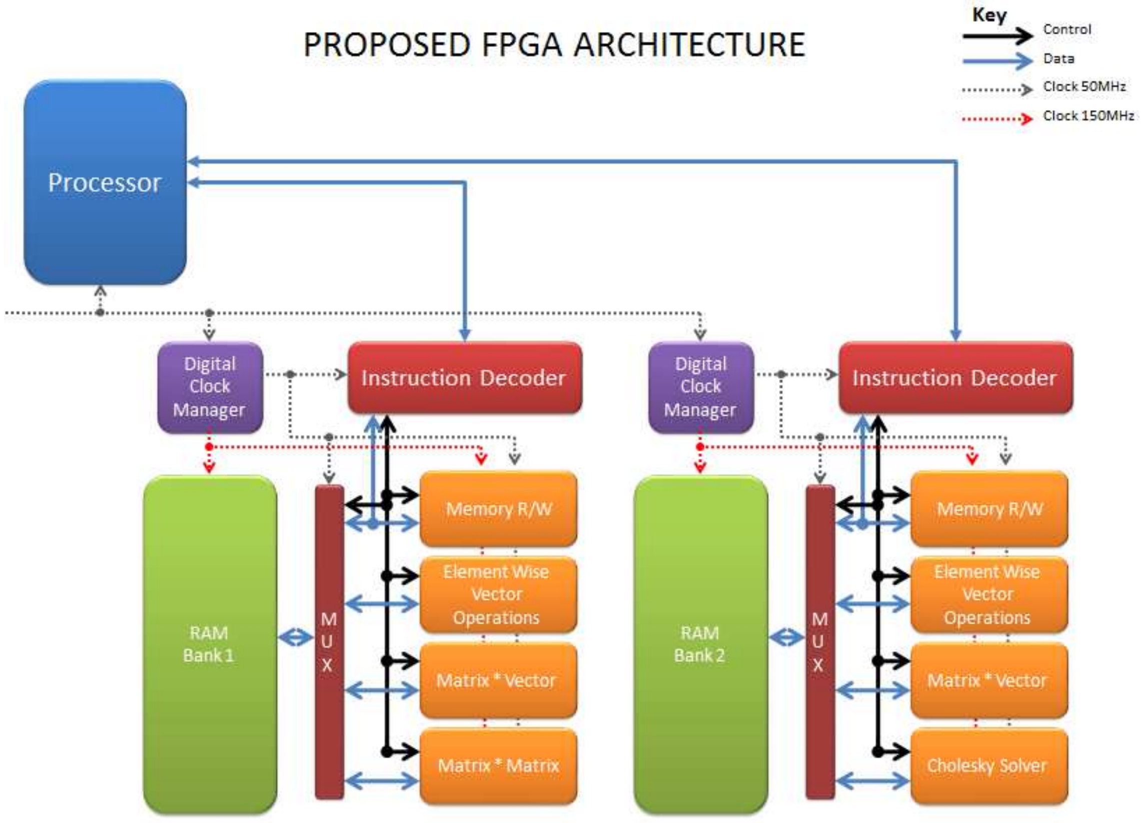
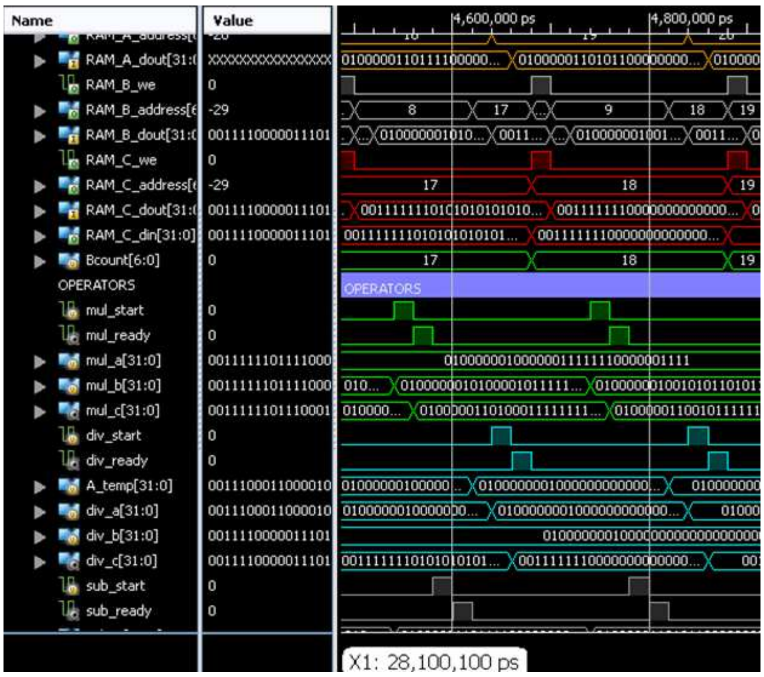


Figure A.8: Initial embedded MPC FPGA architecture design.

In practice the implementation of this algorithm was extremely slow going, with the hardware written manually in Verilog, a language similar to C. An appreciation for the auto-coding tools such as Handel-C used in [186] was quickly gained. After three months development we had completed a parallelized implementation of a matrix × vector module, utilizing a pipelined multiply-accumulate stage and three single precision multipliers and adders. In addition the Cholesky solver was implemented in Verilog, the vectorized element-wise functions written and the memory multiplexing and management auxiliary functions written. The entire system was automatically verified against a MATLAB implementation, by utilizing Verilog test bench files, binary files to provide test data and all automatically called from within a MATLAB script.

In addition to the co-processor development we were also developing the MicroBlaze processor and the Fast-Simplex-Link (FSL) bus which was to act as the communication channel between the soft-processor and the co-processor in hardware. This link was successfully tested and was able to transfer data from the processor at 50MHz to the co-processor memory, which was then operated on and transmitted back to the processor.

A comment made by my supervisor during this development which ultimately shaped the future of my thesis was "life is too short to be looking at simulations in picoseconds", with reference to endless timing diagrams I was using for behavioural simulation and verification (see Figure A.9). It was clear that this development was too time consuming compared to the implementation in C (which took a day), and I was already running out of resources on the Spartan-3E FPGA which meant I was unable to exploit parallelism to the degree I had originally intended.



Name	Value
RAM_A_dout[31:(	XXXXXXXXXXXXXXXX
RAM_B_we	0
RAM_B_address[(	-29
RAM_B_dout[31:(	0011110000011101
RAM_C_we	0
RAM_C_address[(	-29
RAM_C_dout[31:(	0011110000011101
RAM_C_din[31:0]	0011110000011101
Bcount[6:0]	0
OPERATORS	
mul_start	0
mul_ready	0
mul_a[31:0]	0011111101111000
mul_b[31:0]	0011111101111000
mul_c[31:0]	0011111101110001
div_start	0
div_ready	0
A_temp[31:0]	0011100011000010
div_a[31:0]	0011100011000010
div_b[31:0]	0011110000011101
div_c[31:0]	0011110000011101
sub_start	0
sub_ready	0

Figure A.9: FPGA behavioural simulation timing diagram for the memory multiplexer on the Spartan-3E.

Ultimately the reasons why the FPGA route was not pursued are summarised as follows (listed in decreasing order of the size of the hurdle):

Development Time As identified the development time for writing an HDL implementation of MPC is significantly longer than that of a MCU. This is based not only on the language, but also the constructs required. For example all memory access must be handled manually, at low level, meaning a substantial amount of my time was devoted to efficient memory multiplexing between floating point operators. In addition, parallelization introduced race-conditions which required time consuming verification by analyzing timing plots and ensuring all operations were completed in the correct order. As noted below however, without a timing simulation this was only an estimation, and problems could be found later even if the behavioural simulation showed no errors.

FPGA Resources The Spartan-3E FPGA, even at the time, was a small, low-cost FPGA. This meant the availability of resources such as gates / slices / logic cells as well as block RAM and multipliers was very limited. In addition, the maximum clock rate was limited to around 150MHz for memory access, and 50MHz for floating point calculations. While proposals were put forward for a larger, more powerful FGPA for this work, they never eventuated and we were constrained to a simple, sequential design platform with limited scope for a state-of-the-art implementation.

Compilation Time Compiling an HDL description of even the simple co-processor we had designed began taking an increasing amount of time, where times around 5-10 minutes were becoming standard. This was due to the complex compilation process requiring synthesis, translation, map, place & route and finally bitstream generation, all of which was (and still is) computationally intensive. In addition, given the current architecture design, it was likely that the design would require re-compilation each time the controller was re-tuned, meaning fine-tuning could take hours on a real system.

Verification Simulation For accurate verification of an FPGA design a structural and/or timing simulation must be run. These are both post-synthesis simulations, as opposed to behavioural simulations which are prior to synthesis. Xilinx ISE comes with the free HDL simulation tool ISim, which provides a behavioural simulation tool, but not the more accurate post-synthesis simulations. As mentioned above, given we did not have access to more powerful hardware and we could not verify our algorithm on hypothetical hardware using ModelSim (or a similar post-synthesis verification package), we could not realistically continue development.

Ease of Deployment FGPA Printed Circuit Board (PCB) designs are notoriously complicated, requiring multi-layer (typically 6 or more) boards. In addition, the passive electronics and track layout design require careful consideration of noise and filtering which presents quite a challenge for the hardware designer. In contrast an MCU is comparatively much easier to design for, typically only requiring a dual-layer board and a handful of external passive components. Note this factor was considered due to the intention to deploy the embedded MPC controller on an autonomous vehicle, which meant a custom hardware design would be required.

It is worth noting that the landscape has changed now, with tiny FPGA development boards now available that contain considerable resources at very reasonable prices. However the development obstacles still exist, meaning significant development time is still required to develop an MPC solution in HDL. A possible solution to this is the hybrid MCU/FPGA devices that are now available (such as the Xilinx Zynq), most of which contain an ARM processor and small FPGA, allowing a custom co-processor to be developed in HDL along with an industry standard MCU programmed in C.

A.4.2 Linear Algebra Library

As described in Section 4.4.2, to deploy the MPC and QP algorithms to an embedded microcontroller would require the development of a custom linear algebra library. This task in itself is not novel, but the implementation of such a library requires an efficient, hand-coded and hand-optimized set of routines, given that these functions are the most computationally intensive of both algorithms. To illustrate the efficient, hand-optimized features included in a typical custom routine, consider the `jtmv` routine which performs the following operation

$$\mathbf{y} = a\mathbf{A}^T\mathbf{a} + b\mathbf{y}$$

where a and b are scalars, $\mathbf{a}$ and $\mathbf{y}$ are vectors and $\mathbf{A}$ is a matrix. The code below implements this operation within C:

```c
//JTMV  Matrix (Transposed) * Vector [a*matA^T * vecA + b*y]
void jtmv(const intT rows, const intT cols, const realT a,
           const realT *matA, const realT *vecA, const realT b,
           realT *y)
{
    intT i;
    for(i = 0; i < rows; i++){
           realT x = 0.0;
           intT k = cols;
           while(k--)
               x += a * *matA++ * *vecA++;

           y[i] = x + b*y[i];
           vecA-=cols;
    }
}
```

The hand-coded optimizations in the `jtmv` routine include:

- In-place transpose and scalar factors enable more powerful and succinct function calls than available via MATLAB.
- Pointer arithmetic for indexing through the matrix/vector arrays.
- The dot-product is done in-place rather than calling another function.
- Removal of unnecessary arguments from the original BLAS function (i.e. `dgemv`), including `LDA, INCX, INCY` for simpler implementation.
- Separate routines for transposed arguments to allow faster internal iterations.
- Arguments identified as constant to allow the compiler to generate additional optimizations.

- Contiguous memory utilized where possible when calculating inner loops.
- Experimentation with skipping multiplication by zero during the inner loop was found to actually increase execution time. This is most likely due to a multiplication only requiring a single cycle, versus a comparison and possible code branch. In addition, the problems being solved are predominately dense, so that the benefit was always expected to be minimal.

In other implementations of QP solvers such as CVXGEN [210] and MATLAB's auto-coded implementations [203], matrix-matrix and matrix-vector routines will typically be unrolled. This means all loops (or in some situations, just the inner loop(s)) are written out explicitly, eliminating the need for a `for` loop. The result is a much larger, but typically much faster numerical routine.

The decision to not unroll loops in numerical functions within this work has been made based on the following points:

- Profiling of loop unrolling was not found to noticeably benefit the algorithm when run on the embedded system, and in some cases performance actually deteriorated. It appeared the embedded system compiler was already implementing an efficient set of instructions and thus further optimizations were not advantageous.
- Based on the little to no speed improvement recorded, increasing the memory footprint of the controller by loop unrolling was deemed not worth pursuing for the processors of interest.
- For a standalone controller implementation it is not feasible to recompile on-chip the algorithm, thus when tuning the controller the algorithm must be able to cope with larger or smaller systems without fundamental code changes. In addition, generating unrolled implementations can be extremely time consuming, such as the case for CVXGEN. For a system with 16 decision variables and 104 constraints, generating the solver took over 6 minutes to generate, download and compile. It comparison it only takes seconds for the `quad_wright` or `quad_mehrotra` algorithm to compile, it will always be significantly smaller, it will work on any problem size, and it is generally faster for the problems of interest.

A.4.2.1 `formLinSys` Function

Within the linear algebra library, the most computationally intensive operation was forming the left hand side of the set of linear equations within each quadratic program iteration, as identified in Section 3.4.5.6. This step has been targeted with a purpose-written routine, `formLinSys`, to exploit special characteristics of within the matrices involved in this expression.

The function `formLinSys` performs an in-place calculation of the linear left hand side (Equation 3.60 in Section 3.4.5.6) on top of $\mathbf{H}$, and is on average 50% faster than if Equation 3.60 were implemented using a banded matrix multiplier, and 300%-500% faster than if we were using a general matrix multiplier.

```
void formLinSys(const realT *A, const realT *lamt, const intT ndec,
                const intT mc, realT *RQP)
{
    uintT i, j;
    realT *lt = (realT*)lamt;
    realT *Ai = (realT*)A;
    for(i = 0; i < ndec; i++) {
        realT *Aj = Ai;
        realT *Rj = &RQP[i+i*ndec];
        for(j = i; j < ndec; j++) {
            realT sum = *Rj;
            intT k = mc;
            while(k--)
                sum += *Ai++ * *Aj++ * *lt++;

            *Rj = sum;
            Rj += ndec;
            Ai -= mc;
            lt -= mc;
        }
        Ai += mc;
    }
}
```

Note we also experimented with precalculating some (or all) of the terms resulting from $\mathbf{M}^T\mathbf{M}$ however it was determined the substantial increase in memory required did not offset the gain in performance. In addition, numerical conditioning of the resulting matrix became an issue when implemented in single precision.

A.4.3 TI C28343 Linker Command File

The linker file for implementing embedded MPC controllers on the TI C28343 is listed below. The file is designed to allow larger controllers to be implemented within the high-speed, 0-wait memory available on the IC, as described in Section 4.4.3.2.

```
MEMORY
{
PAGE 0 :
   /* BEGIN is used for the "boot to SARAM" bootloader mode      */
   /* Boot to M0 will go here */
   BEGIN        : origin = 0x000000, length = 0x000002
   RAMM0        : origin = 0x000052, length = 0x0003AE
   // Switch tables, ram funcs, hardly used
   RAML0        : origin = 0x008000, length = 0x000200
   // Program code (8KB is safe)
   RAML1        : origin = 0x008200, length = 0x002000
   /* XINTF zone 7 - program space */
   ZONE7A       : origin = 0x200000, length = 0x00FC00
   /* Reserved - for compatibility to legacy C28x designs. */
   CSM_RSVD     : origin = 0x33FF80, length = 0x000076
    /* 128-bit password locations */
   CSM_PWL_PROG : origin = 0x33FFF8, length = 0x000008
   IQTABLES     : origin = 0x3FE000, length = 0x000b50
   IQTABLES2    : origin = 0x3FEB50, length = 0x00008c
   FPUTABLES    : origin = 0x3FEBDC, length = 0x0006A0
   BOOTROM      : origin = 0x3FF27C, length = 0x000D44
   RESET        : origin = 0x3FFFC0, length = 0x000002

PAGE 1 :
   /* Part of M0, BOOT rom will use this for stack */
   BOOT_RSVD    : origin = 0x000002, length = 0x000050
   /* on-chip RAM block M1 */
   RAMM1        : origin = 0x000400, length = 0x000400
   // Global vars
   RAML2        : origin = 0x00A200, length = 0x003000
   // Constants (1 wait hit for 14000 and above)
   RAML3        : origin = 0x00D200, length = 0x00AE00
   // Initial values of global vars
   RAMH0        : origin = 0x300000, length = 0x010000
   /* XINTF zone 7 - data space */
   ZONE7B       : origin = 0x20FC00, length = 0x000400
}


SECTIONS
{
   codestart        : > BEGIN,     PAGE = 0
   ramfuncs         : > RAML0,     PAGE = 0
   .text            : > RAML1,     PAGE = 0
   .pinit           : > RAML0,     PAGE = 0
   .switch          : > RAML0,     PAGE = 0
   .stack           : > RAMM1,     PAGE = 1
   .cinit           : > RAMH0,     PAGE = 1
   .ebss            : > RAML2,     PAGE = 1
   .econst          : > RAML3,     PAGE = 1
   .esysmem         : > RAMM1,     PAGE = 1
   IQmath           : > RAML1,     PAGE = 0
   IQmathTables     : > IQTABLES,  PAGE = 0, TYPE = NOLOAD
   FPUmathTables    : > FPUTABLES, PAGE = 0, TYPE = NOLOAD

   ZONE7DATA        : > ZONE7B,         PAGE = 1
   .reset           : > RESET,          PAGE = 0, TYPE = DSECT /* not used */
   csm_rsvd         : > CSM_RSVD        PAGE = 0, TYPE = DSECT
   csmpasswds       : > CSM_PWL_PROG    PAGE = 0, TYPE = DSECT
}
```

A.4.4 Processor-In-The-Loop Embedded MPC with the jMPC Toolbox

One of the powerful validation features of the jMPC Toolbox and its auto-code generation framework is the ability to run a PIL implementation, as described in Section 4.5. This allows the generated controller to be automatically deployed to the target hardware, and then run in a simulation loop with the development PC acting as the plant to control. This provides a quick means to check the controller has compiled correctly on the target hardware, as well as the performance achievable on the target. The code example below shows the process for generating, deploying and validating an MPC controller using the jMPC Toolbox and PIL implementation.

To begin, the jMPC controller is defined in MATLAB, as per a normal controller, however rather than simulating it, the framework generates a C version of the controller.

```
% Dynamic System to Control
Gs = tf(2,[0.7 0.2 1]);
Gd = c2d(Gs,0.1);
% Create jSS Objects (Model Format)
Plant = jSS(Gd);
Model = Plant;  %No Model/Plant Mismatch

% Tuning
Np = 8;                 %Prediction Horizon
Nc = [4 2 2];           %Blocking Moves
uwt = 0.5;              %DeltaU Weights
ywt = 0.5;              %Y Weights
% Constraints
con = [];
con.u = [-inf inf 0.2]; %In1  [umin umax delumax]
con.y = [-inf 2];       %Out1 [ymin ymax]
% Estimator Gain
Kest = dlqe(Model);     %Discrete Observer with W,V = eye()

% Simulation Setup
T = 200;                %Length of Simulation
setp = ones(T,1);       %Setpoint
setp(75:150) = 0.5;

% MPC Controller Options
opts = jMPCset('QPSolver','Mehrotra','Single',1);

% Build MPC & Simulation
MPC1 = jMPC(Model,Np,Nc,uwt,ywt,con,Kest,opts);
simopts = jSIM(MPC1,Plant,T,setp);
% Assign Serial Device
simopts.opts.serialdevice = serial('COM6','BaudRate',1250000);
% Generated Embedded MPC Controller
eopts = jMPCeset('arch','c2000'); %Set for TI C28343
embed(MPC1,simopts,eopts);
```

At this point the `jMPC` controller has been built and code has been generated to implement it. Other than a setup routine (to initialize the serial port, timers, etc), and providing the four utility routines as described in Section 4.3.1.1, the target's main function can be as simple as shown in the code snippet below:

```
void main(void)
{
    setup();  //Initialize GPIO, Serial Port, uS Timer
    PILSim(); //Enter PIL MPC Simulation
}
```

All other required functions have been automatically generated, and corresponding communication routines on the development PC are included with the jMPC Toolbox. Once the code has been compiled and set to run on the target, the PIL verification can be initiated with the following MATLAB command:

```
simpil = sim(MPC1,simopts,'PIL')
```

Once the PIL run has completed the entire simulation specified within the `jSIM` object, the results are automatically returned via the results object `simpil`. To verify these results against the same controller, but run instead as a MEX implementation on the same computer, the following commands can be used:

```
simdev = sim(MPC1,simopts,'MEX');
compare(MPC1,simdev,simpil)
```

The `jMPC` method `compare` will automatically generate the comparative plot shown in Figure A.10. It will also optionally output the calculated numerical deviations which can be used to quickly determine a pass or fail.

Using the auto-code generation and PIL framework included as part of the jMPC Toolbox on the above example, the code generation process took 1.2 seconds. This includes automatic verification of the QP solver on the development computer (which can be optionally disabled). Compiling and downloading the controller onto the target took approximately 5 seconds, and running the PIL verification run took just over 4 seconds to simulate 200 samples. While this example is admittedly very small, it shows this process is fast and simple enough to allow rapid development and verification using the target itself.

For the reader's reference, the controller implemented in this example required 9KB of Flash memory (obtained via the compiler `.map` file), 1.3KB of RAM and achieved a maximum sampling rate of 2.7kHz. The bottleneck in the PIL simulation

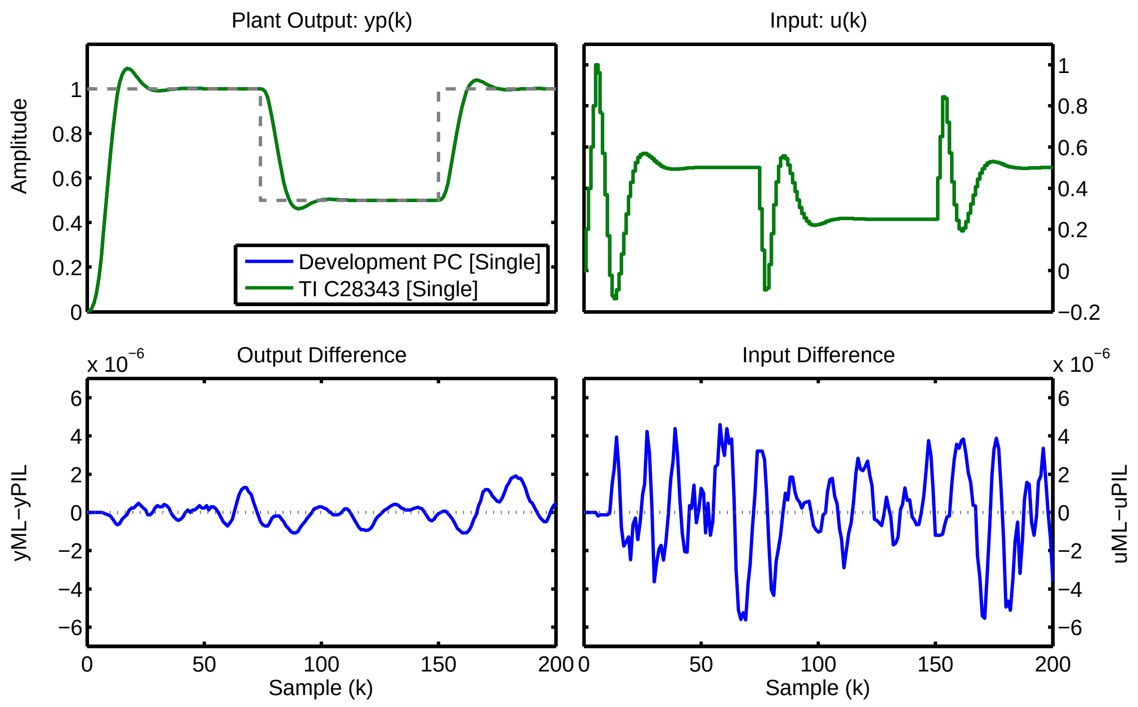


Figure A.10: Example PIL comparison plot: MEX vs TI C28343. Note that in single precision that ϵ is approximately 1×10^{-7}.

is the communication channel between the development computer and the target, where overhead on the PC side causes noticeable delays.

Appendix B

Utility Modelling Thermodynamics and Models

This appendix chapter provides additional detail to the thermodynamics and models developed in Chapter 5. Implementation details are provided, together with examples of the performance of the models developed.

B.1 JSteam

JSteam is the name given to a steam utility modelling framework developed within this work. It includes thermodynamic functions for water and steam, as well as fuel gas and combusted product streams. In addition, it provides a suite of simple models for estimating the performance of key utility system unit operations. The library together with the associated JSteam MATLAB Toolbox is available at `http://www.inverseproblem.co.nz/Software/JSteamDLL.html`, as well as supplied on the Appendix DVD.

B.1.1 JSteam C++ Implementation Example

The code example below illustrates the implementation of a typical IAPWS thermodynamic routine in C++. Note several hand-coded optimizations based on substantial profiling have been utilized to further increase the speed of the calculation. These include loop-unrolling, precalculation of common terms (for example $n \times J$), and utilizing the integer power routines where applicable. The function implements Equation 5.2 in Section 5.3.1.1.

```
//Specific Enthalpy in Region 1 as a Function of P, T
double H = 0;
double tau = 1386.0/t;
double tau_g = tau-1.222;
double pi_g = 7.1-p/16.53;
int i = r1f_no;
while(i)
{
    --i; H += r1f_nJ[i]*pow(pi_g,r1f_I[i])*pow(tau_g,r1f_J[i]-1);
    if(i>3){
        --i; H += r1f_nJ[i]*pow(pi_g,r1f_I[i])*pow(tau_g,r1f_J[i]-1);
        --i; H += r1f_nJ[i]*pow(pi_g,r1f_I[i])*pow(tau_g,r1f_J[i]-1);
        --i; H += r1f_nJ[i]*pow(pi_g,r1f_I[i])*pow(tau_g,r1f_J[i]-1);
        --i; H += r1f_nJ[i]*pow(pi_g,r1f_I[i])*pow(tau_g,r1f_J[i]-1);}
    if(i>2){
        --i; H += r1f_nJ[i]*pow(pi_g,r1f_I[i])*pow(tau_g,r1f_J[i]-1);
        --i; H += r1f_nJ[i]*pow(pi_g,r1f_I[i])*pow(tau_g,r1f_J[i]-1);
        --i; H += r1f_nJ[i]*pow(pi_g,r1f_I[i])*pow(tau_g,r1f_J[i]-1);}
    if(i>1){
        --i; H += r1f_nJ[i]*pow(pi_g,r1f_I[i])*pow(tau_g,r1f_J[i]-1);
        --i; H += r1f_nJ[i]*pow(pi_g,r1f_I[i])*pow(tau_g,r1f_J[i]-1);}
    if(i>0){
        --i; H += r1f_nJ[i]*pow(pi_g,r1f_I[i])*pow(tau_g,r1f_J[i]-1);}
}
H *= tau*0.461526*t;
break;
```

B.1.2 JSteam Water and Steam Functions

The complete JSteam water and steam thermodynamic package includes the routines listed in Table B.1, where the function name format is `Output Input1 Output2`. For example `HPT` calculates enthalpy (`H`), as a function of pressure (`P`) and temperature (`T`). All of the functions listed are derived from regressions within the respective IAPWS formulations.

Table B.1: JSteam water & steam thermodynamic functions. Note units are customizable as per the implementation.

Property	Symbol	Functions
Isobaric Heat Capacity	C_p	`CpPH, CpPS, CpPT, CpPX, CpTX`
Isochroic Heat Capacity	C_v	`CvPH, CvPS, CvPT, CvPX, CvTX`
Specific Enthalpy	H	`HPT, HPS, HTS, HPX, HTX`
Specific Entropy	S	`SPH, SPT, SPX, STX`
Pressure	P	`PHS, PTS, PsatT`
Quality	X	`XPH, XPS, XTS`
Temperature	T	`THS, TPH, TPS, TSX, TsatP`
Thermal Conductivity	K	`KPT`
Dynamic Viscosity	μ	`UPT`
Volume	V	`VPH, VPS, VPT, VPX, VTX`

B.1.3 JSteam Composition Functions

Table B.2 lists the thermodynamic methods available within the Composition object, including where they are derived from. Table B.3 lists the combustion methods included, allowing the object to predict the heat released during combustion of a fuel gas stream, as well as the resulting combustion products.

Table B.2: JSteam composition class thermodynamic methods.

Function	Unit	Property	Thermo Package
`HcT`	kJ/kmol	Enthalpy	NASA Glenn [229]
`ScT`	kJ/(kmol K)	Entropy	NASA Glenn [229]
`CpcT`	kJ/(kmol K)	Heat Capacity	NASA Glenn [229]
`TcH`	°C	Temperature (Iterated)	NASA Glenn [229]
`HcPT`	kJ/kmol	Enthalpy	Peng-Robinson [245]
`ScPT`	kJ/(kmol K)	Entropy	Peng-Robinson [245]
`TcPH`	°C	Temperature (Iterated)	Peng-Robinson [245]
`TcPS`	°C	Temperature (Iterated)	Peng-Robinson [245]
`KcT`	W/(m K)	Thermal Conductivity	Carl Yaw [56]
`UcT`	μPa s	Viscosity	Carl Yaw [56]
`NHV`	kJ/kmol	Net (Lower) Heating Value	VMGThermo [319]
`GHV`	kJ/kmol	Gross (Higher) Heating Value	VMGThermo [319]
`MW`	g/mol	Molecular Weight	NASA Glenn [229]

Table B.3: JSteam composition class combustion methods.

Function	Description
`StoichCombust`	Stoichiometrically Combust Current Object (Used for two methods below)
`Combust`	Combust Object Given Inlet Enthalpy, Inlet Mole Flow and Outlet Temperature and Calculate Duty
`AdCombust`	Adiabatically Combust Object Given Inlet Enthalpy, Inlet Mole Flow and Calculate Outlet Enthalpy
`InExcessO2`	Calculate Inlet (Before Combustion) Excess O_2 Fraction (Wet Basis)
`StackO2`	Calculate Stack (After Combustion) O_2 Mole Fraction (Wet Basis)
`SolveAirMassF`	Solve Mass Flow of Air Required to Match Specified Inlet Excess O_2 or Stack O_2 Fraction
`SolveAirMoleF`	Solve Mole Flow of Air Required to Match Specified Inlet Excess O_2 or Stack O_2 Fraction
`SolveFuleMoleF`	Solve Mole Flow of Fuel Required to Match Specified Stack O_2 Fraction

B.1.4 JSteam Composition Class Example

The code snippet below illustrates the use of the JSteam `Comp` object, including thermodynamic and combustion methods. The example below is common to the internal calculation procedure used within the fuel gas unit operations, as described in Section 5.3.5.

```
//Create JSteam Object
JSteam *JStm = new JSteam();
//Create Composition Objects
Comp *Fuel = new Comp();
Comp *Air = new Comp();

//Fill In Fraction Details
vector<double> FuelFrac = Fuel->CreateFracVec();
FuelFrac[(int)JSteam::Methane] = 0.8;
FuelFrac[(int)JSteam::Ethane] = 0.15;
FuelFrac[(int)JSteam::Propane] = 0.05;
Fuel->SetMoleFrac(FuelFrac);
Air->SetAsAir(); //N2 0.79, O2 0.21

//Determine Air Mole Flow for combustion with 10% excess O2
double fuelM = 2, airM, duty;
airM = Fuel->SolveAirMoleF(Air,0.1,0,fuelM);

//Mix Compositions into Combustible Stream
Comp *FuelAir = Fuel->MixMole(Air,fuelM,airM);
double inH = FuelAir->HcT(30.0); //Fuel Temp of 30C

//Combust Composition and Calculate Duty
Comp *CompProd = FuelAir->Combust(inH,200,fuelM+airM,&duty);
```

B.1.5 Fuel Gas Unit Operation Examples

The following examples are intended to demonstrate that the models developed in Section 5.3.5 provide realistic operating results.

B.1.5.1 Furnace

A typical output of the Furnace unit operation model is shown in Table B.4 for an input fuel gas composition of 100% CH_4.

B.1.5.2 Fired Boiler

A typical output of the Fired Boiler unit operation model is shown in Table B.5 for an input fuel gas composition of 100% CH_4.

Table B.4: Example furnace model results.

Input	Value	Output	Value
Q_{Furnace}	30 MW	M_{Fuel}	0.65 kg/s
T_{Air}	30°C	M_{Air}	11.69 kg/s
T_{Fuel}	30°C	η_{Furnace}	92.69%
T_{Stack}	200°C	$M_{\text{Stack CO}_2}$	1.12 kg/s
$F_{\text{Stack O}_2}$	1% (Mole)		

Table B.5: Example fired boiler model results.

Input	Value	Output	Value
H_{BFW}	250 kJ/kg	H_{Steam}	3214.4 kJ/kg
M_{Steam}	10 kg/s	H_{BD}	1087.4 kJ/kg
T_{Steam}	400°C	M_{BFW}	10.1 kg/s
P_{Steam}	40 bar	M_{Air}	11.59 kg/s
T_{Air}	30°C	M_{Fuel}	0.64 kg/s
T_{Fuel}	30°C	Q_{Boiler}	32.07 MW
T_{Stack}	200°C	η_{Boiler}	92.69%
F_{BD}	1%	$M_{\text{Stack CO}_2}$	1.11 kg/s
$F_{\text{Stack O}_2}$	1% (Mole)		

B.1.5.3 Gas Turbine

A typical output of the Gas Turbine unit operation model is shown in Table B.6 for an input fuel gas composition of 100% CH_4.

Table B.6: Example gas turbine model results.

Input	Value	Output	Value
W_{GTG}	15 MW	M_{Fuel}	0.86 kg/s
η_{GTG}	35%	M_{Air}	53.71 kg/s
T_{Air}	30°C	M_{Exhaust}	54.57 kg/s
T_{Fuel}	30°C	$M_{\text{Stack CO}_2}$	1.52 kg/s
T_{Exhaust}	500°C		
$P_{\text{Compressor}}$	20 bar		

B.1.5.4 Heat Recovery Steam Generator

An example of the base load HRSG is shown in Table B.7. The unit is connected to a 15MW 35% efficient GTG with an exhaust temperature of 500°C running on 100% CH_4.

When operated in maximum firing mode, the example listed in Table B.7 pro-

Table B.7: Example base load HRSG model results.

Input	Value	Output	Value
H_{BFW}	250 kJ/kg	M_{Steam}	6.07 kg/s
T_{Steam}	400°C	T_{Stack}	200°C
T_{MinStack}	200°C	$\Delta T_{\text{Economiser}}$	37.14°C
T_{MinDT}	30°C	$\Delta T_{\text{Superheater}}$	100°C
P_{Steam}	40 bar	$M_{\text{Stack CO}_2}$	1.52 kg/s
F_{BD}	1%		

duces a maximum steam mass flow of 41.3 kg/s which results in approach temperatures of 452.8°C and 1400°C for the economiser and superheater respectively, as well as a mass flow of 4.94 kg/s of CO_2 exiting via the stack. In order to generate this extra steam, 2.12 kg/s of Methane is required for supplementary firing, in addition to the 0.86 kg/s required by the gas turbine.

B.1.6 JSteam Excel Add-In

The key design focus of the package is to utilize Excel is an existing front-end that many engineers are familiar with, and can drive sufficiently well. In this way a dedicated Graphical User Interface (GUI) did not have to be built, and the JSteam package could slot in as a standard Excel Add-In, as shown in Figure B.1.

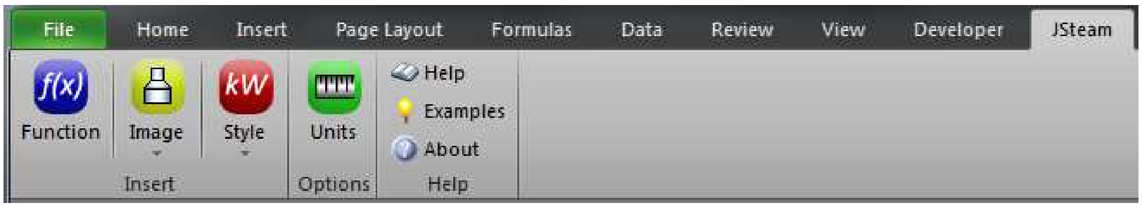


Figure B.1: JSteam Excel ribbon interface.

Using graphics supplied with the add-in, together with standard Excel drawing functionality, a user can create a utility system Process Flow Diagram (PFD) in a matter of minutes, as shown in Figure B.2.

To enter the underlying unit operation models, a function assistant was created which provides a graphical method to insert complex functions automatically. This interface is shown in Figure B.3. Alternatively the user can enter the JSteam functions in the same fashion as standard Excel functions, as shown in Figure B.4.

Following the JSteam modelling guide, a user completes the system mass and energy balance by entering known plant measurements and connecting the unit operations using standard Excel cell references. To converge the model, the built-in

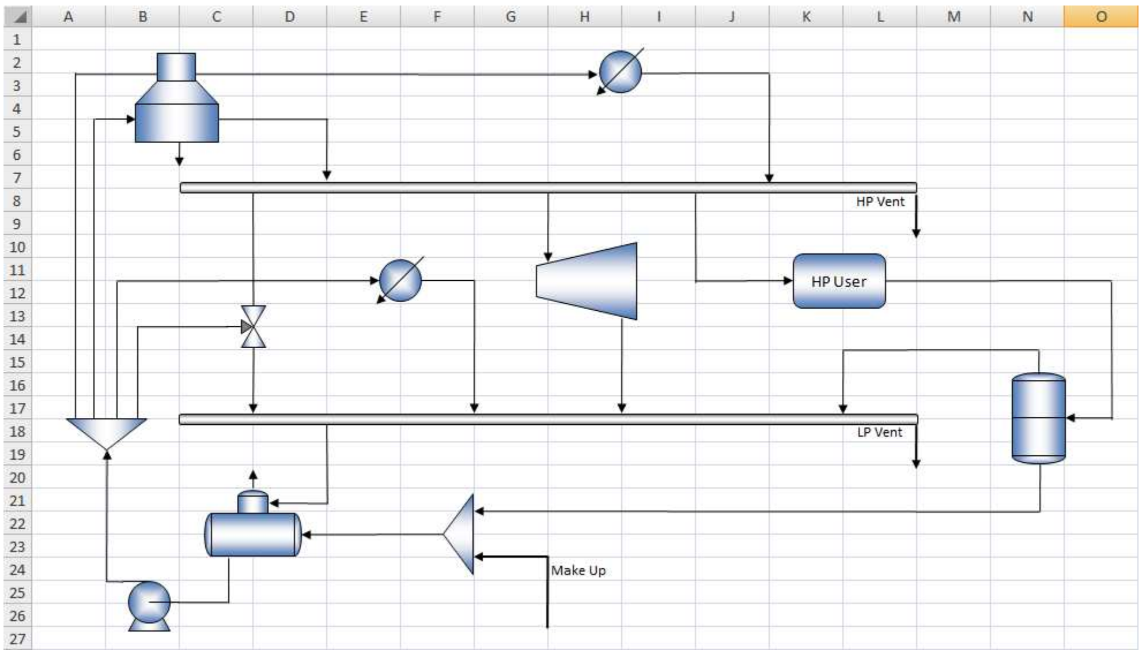
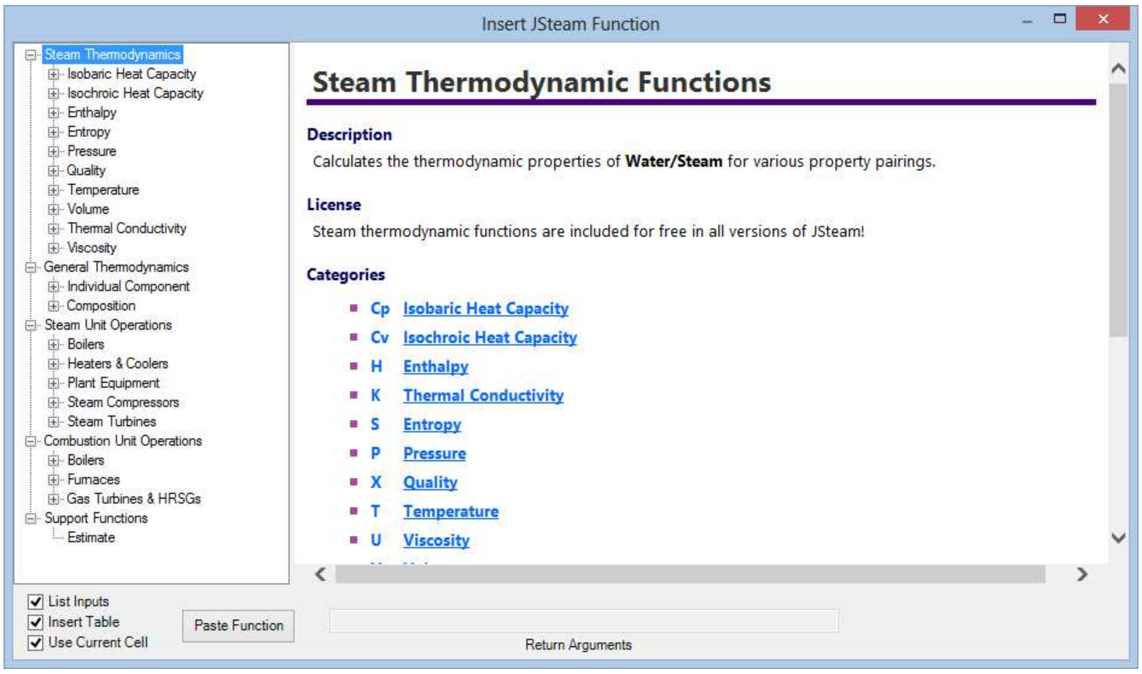


Figure B.2: JSteam Tutorial #1 PFD [62].



Steam Thermodynamic Functions

Description

Calculates the thermodynamic properties of **Water/Steam** for various property pairings.

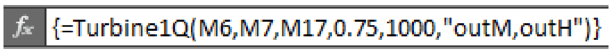
License

Steam thermodynamic functions are included for free in all versions of JSteam!

Categories

- Cp Isobaric Heat Capacity
- Cv Isochroic Heat Capacity
- H Enthalpy
- K Thermal Conductivity
- S Entropy
- P Pressure
- X Quality
- T Temperature
- U Viscosity

Figure B.3: JSteam Excel Function Assistant.



```
{=Turbine1Q(M6,M7,M17,0.75,1000,"outM,outH")}
```

Figure B.4: JSteam Excel function call.

Excel nonlinear solver iterates to solve for a valid operating point, given typical degrees of freedom such as boiler feed water mass flow and deaerator steam supply. A completed PFD is shown in Figure B.5.

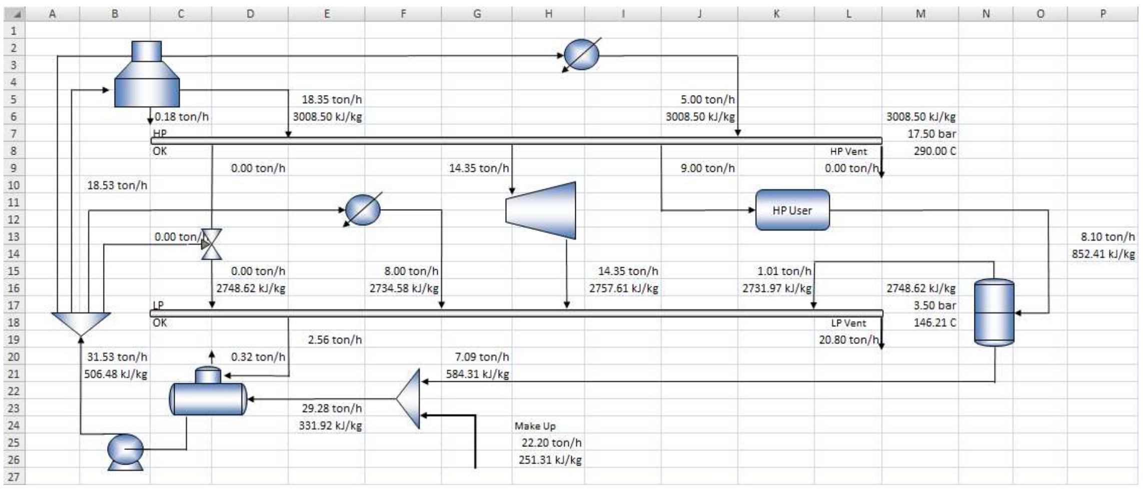


Figure B.5: JSteam Tutorial 1 model.

B.2 Detailed Part-Load Model Examples

The following subsections illustrate the use of the detailed part-load utility models developed, including typical regressed parameters and part-load performance.

B.2.1 Steam Turbine

An example model output for a 1500kW back pressure steam turbine is shown in Figure B.6. The turbine has a inlet of 400°C, 50 bar steam, and an outlet of 10 bar. As expected, the output shaftwork is proportional to the mass flow of steam through the turbine, while the output enthalpy is approximately inversely proportional to the isentropic efficiency. Fitted parameters for Equation 5.37 are shown in Table B.8.

Table B.8: Turbine model example fitted parameters.

α_s	202.23
β_s	1.3666
α_h	-601.9
β_h	0.3105

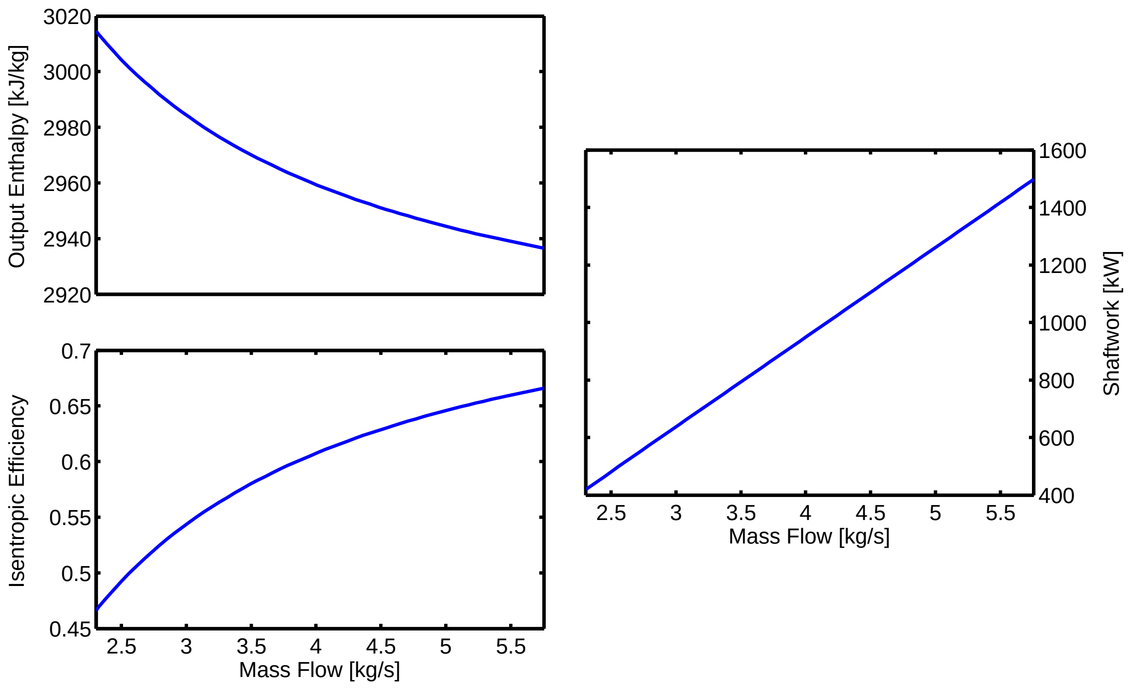


Figure B.6: Example 1500kW turbine results.

B.2.2 Steam Boiler

The example from Figure 5.34 and Table 5.18 in Section 5.5.1 is shown in Figure B.7. The curves presented follow the same trends as shown in the Aguilar et al paper [4], and show a small increase in fuel consumption over the previous ideal JSteam model, as expected. Table B.9 lists the coefficients for each operating case.

Table B.9: JSteam Fired Boiler with variable efficiency coefficients.

Case	α_J	β_J
Base	0.0044	0.0758
80°C air	0.0033	0.0559
250°C stack	0.0059	0.1009

B.2.3 Gas Turbine

To illustrate the effect of part-load performance described by this model, Figure B.8 shows a range of gas turbines and their respective efficiency curves, while Figure B.9 compares the model against published small GTG data in [4]. Note for the model comparison, the loss coefficient (L) has been regressed from operating data, aiding the model fit. However for a model that only requires the rated gas turbine output and a single regressed parameter, the curves match remarkably well, even to the point that they look regressed purely against the data (which is not the case). A

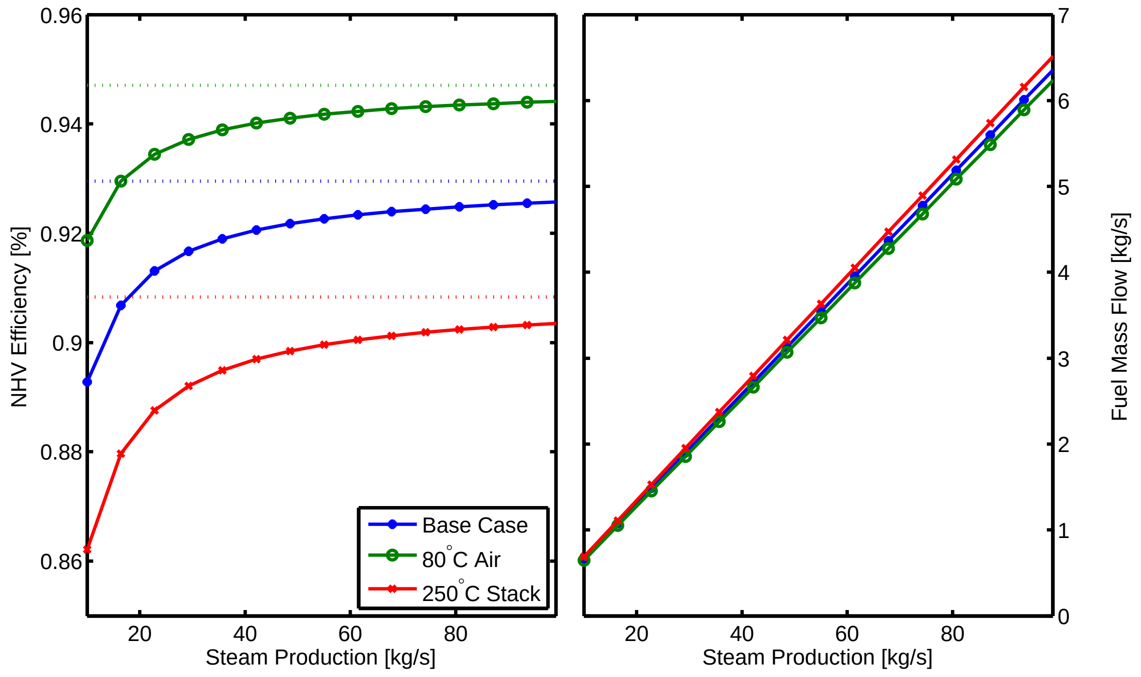


Figure B.7: Boiler efficiency as a function of steam production, including variable efficiency for a 100kg/s methane fired boiler. Dashed lines on the efficiency curve indicate maximum theoretical efficiency.

reason why the Tornado model does not fit as well could be due to the maximum efficiency of the turbine exceeding the fitted model (Equation 5.58), which can be viewed in the original data. While the maximum efficiency could be read off the data and used instead of the complete fit, this exercise is simply to show the model functions reasonably correctly using this correlation.

Table B.10: Small gas turbine regressed parameters.

GTG	α_{gt}	β_{gt}	L
Tempest 7.7MW	2417.59	20102.7	0.315
Tornado 6.5MW	2374.66	20706.0	0.366
Taurus 5.4MW	1962.32	20475.7	0.364
Typhoon 4.6MW	2014.42	21414.7	0.439
Typhoon 4.2MW	1347.52	19572.3	0.322

B.2.4 Heat Recovery Steam Generator (HRSG)

B.2.4.1 Unfired HRSG

Using the unfired HRSG model developed in Section 5.7.1, Figure B.10 shows the predicted steam and power generation potential for a 50MW GTG connected to a single-pressure unfired HRSG with no reheat. Table B.11 shows the parameters

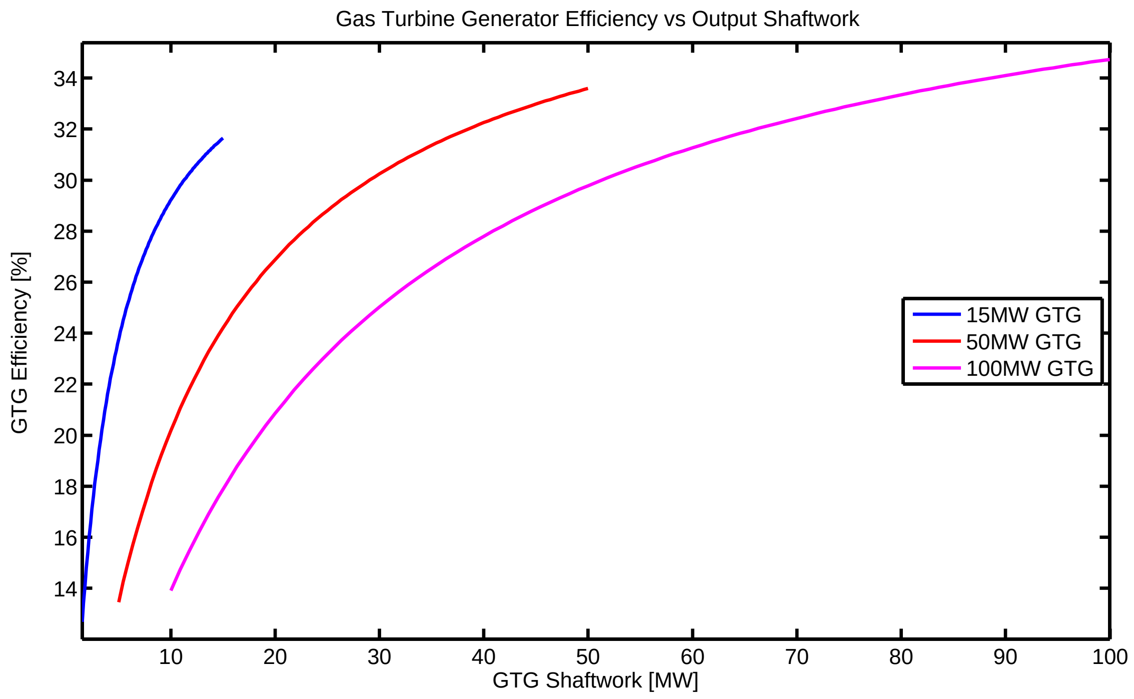


Figure B.8: Model generated GTG efficiency curves for a range of gas turbines.

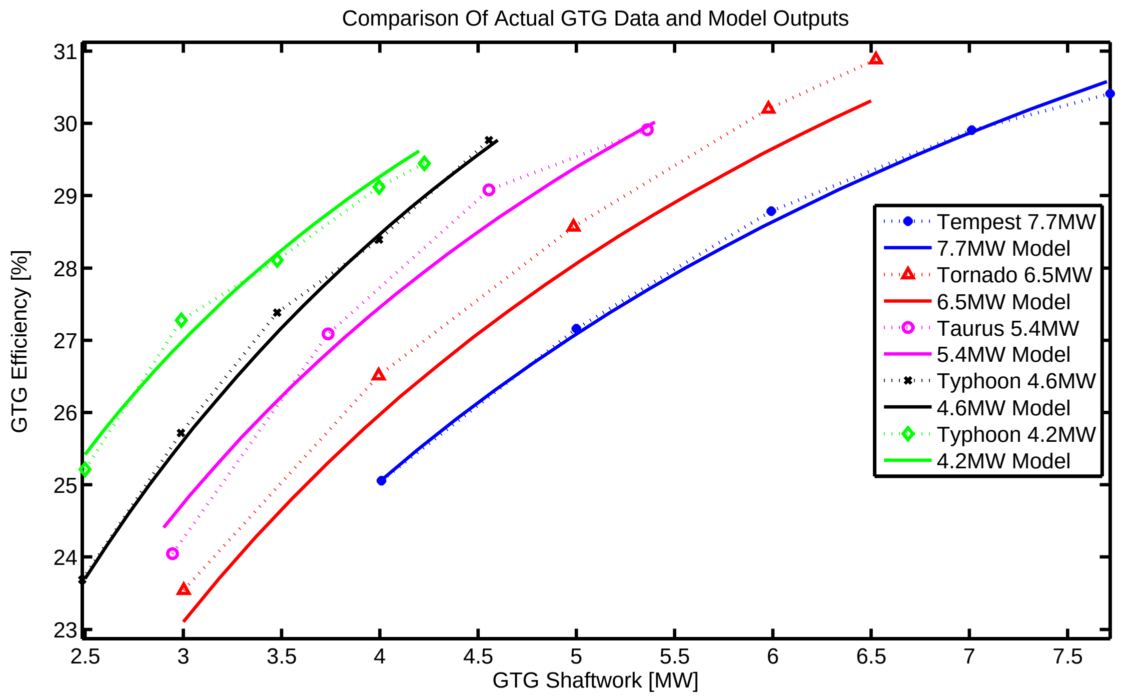


Figure B.9: Model comparison against small gas turbine efficiencies as published in [4].

used to generate this example, while the efficiency is calculated as

$$\text{Cycle Efficiency} = \eta_{\text{thermal}} = \frac{W_{st} + W_{gt}}{\text{NHV}_{\text{fuel}} M_{\text{fuel}}} \tag{B.1}$$

where W_{st} is steam turbine shaftwork and W_{gt} is the gas turbine shaftwork. As is common for simple CCGTs, (i.e. single pressure, single stage) the predicted maximum efficiency is approximately 50%, which closely matches that found in [269] (noting this reference was used for the majority of the specifications).

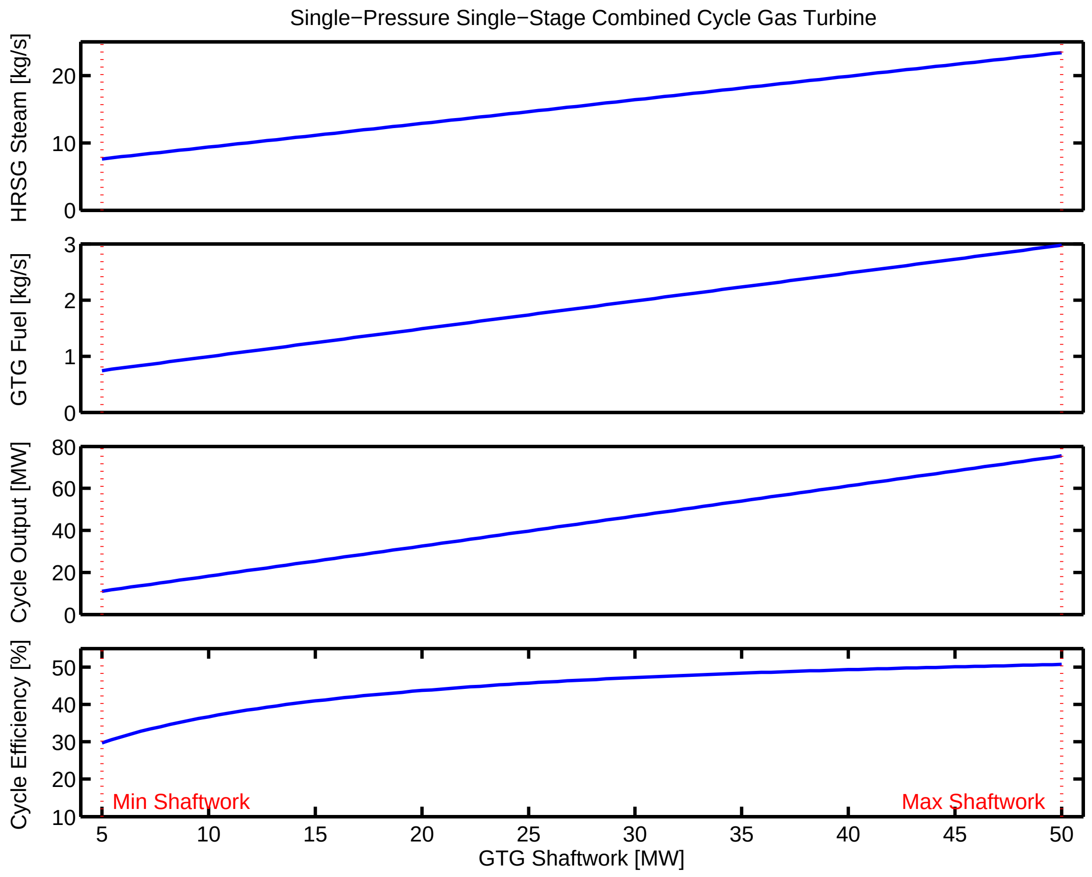


Figure B.10: Steam production and fuel consumption for a 50MW GTG and 535°C, 120 bar unfired HRSG.

B.2.4.2 Secondary Fired HRSG

For a secondary fired HRSG, Figure B.11 shows the model fuel usage surface, including the upper and lower constraints as ‘walls’ within the figure. The system modelled is identical to that from the base load example, with the exception of the addition of a lower limit of O_2 in the stack (0.01 mole fraction), and the secondary fuel being the same as the GTG fuel. Regressed coefficients for this example are listed in Table B.12.

Table B.11: Unfired HRSG example parameters.

Steam Turbine Rated Output	16MW
GTG Rated Output	50MW
GTG Exhaust Temperature	635°C
GTG Fuel Composition	100% Methane
Air Temperature	15°C
Fuel Temperature	15°C
HRSG Minimum Stack Temperature	200°C
Steam Temperature	535°C
Boiler Feed Water Temperature	100°C
Steam Pressure	120 bar
Condenser Pressure	0.07 bar
Blowdown Ratio	1%

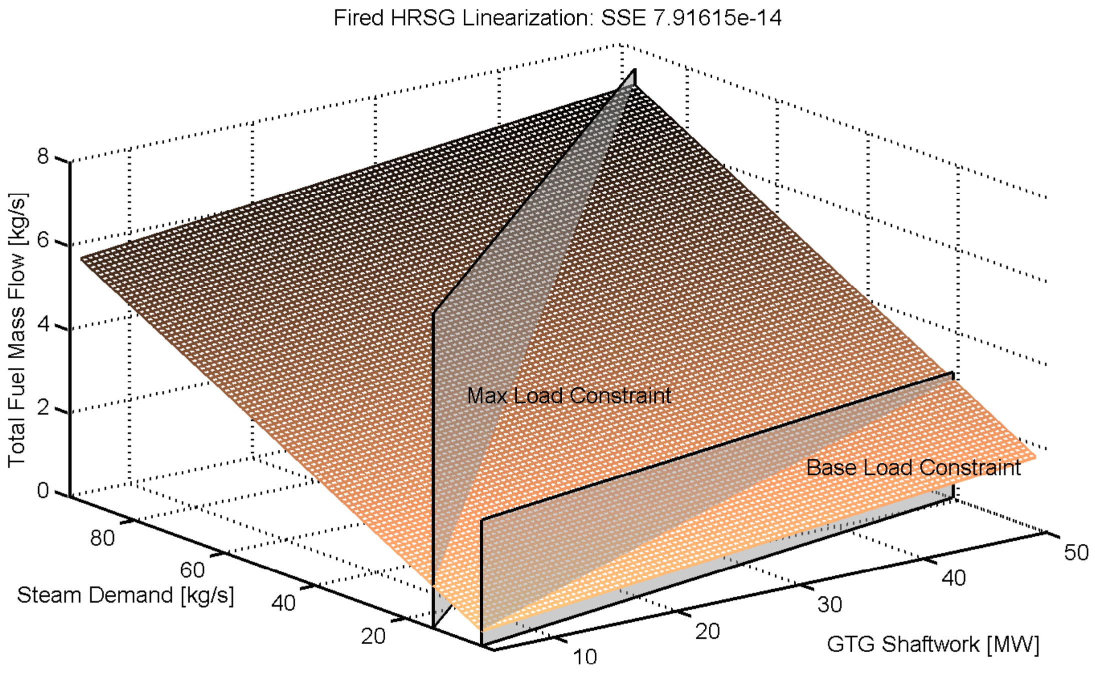


Figure B.11: Fuel flow surface with operational constraints.

Table B.12: Secondary fired HRSG example regressed coefficients.

α_{hrsg}	Total Fuel Intercept	0.02326
β_{hrsg}	Total Fuel GTG Gradient	2.9525×10^{-5}
γ_{hrsg}	Total Fuel Steam Gradient	0.061113
α_{bs}	Base Load Constraint Intercept	0.98887
β_{bs}	Base Load Constraint Gradient	0.000409
α_{ms}	Max Load Constraint Intercept	5.128
β_{ms}	Max Load Constraint Gradient	0.0017338

Appendix C

Optimization Framework Software and Models

This appendix chapter provides additional detail to the optimization framework developed in Chapters 6 and 7, including details of the OPTI Toolbox, the complete steam utility optimization models and validation of the global solver interfaces.

C.1 OPTI Toolbox

As described in Section 6.2, the OPTI Toolbox is a free, open-source MATLAB toolbox that collects together a suite of solvers and provides a common calling structure to all of them. The following sections detail the solvers available within the toolbox, together with extra functionality such as mathematical utilities and viewing SymBuilder results. The OPTI Toolbox is available at `http://www.i2c2.aut.ac.nz/Wiki/OPTI/`, as well as supplied on the Appendix DVD.

C.1.1 Optimization Solvers

Each of the solvers are briefly summarized in each of the subsections below, grouped by problem solving type. Note many of these solvers will also solve problems from other groups, but they are listed within their intended problem group.

C.1.1.1 Linear Programming

Linear Programming (LP) solves problems of the form

$$\begin{aligned} \min_{\mathbf{x}} \; & \mathbf{f}^T\mathbf{x} \\ \text{subject to: } & \mathbf{A}\mathbf{x} \leq \mathbf{b} \\ & \mathbf{A}_{\text{eq}}\mathbf{x} = \mathbf{b}_{\text{eq}} \\ & \mathbf{l}_{\text{b}} \leq \mathbf{x} \leq \mathbf{u}_{\text{b}} \end{aligned}$$

and optionally mixed integer variants (MILP) with the following additional constraints on the decision variable

$$x_i \in \Re, \quad x_j \in \mathbb{Z}, \quad x_k \in \{0,1\}, \quad i \neq j \neq k$$

OPTI includes a number of LP and MILP solvers, detailed below.

Coin-OR Linear Programming (CLP) [100] As detailed in Section 3.3.2.1 CLP was written by John Forrest, and contains both primal and dual simplex solvers, as well as a barrier solver. In addition to solving LPs, CLP can only solve QPs via the dual simplex or barrier solvers. The code is written in C++ and utilizes a MATLAB developed within this work.

Coin-OR Branch and Cut (CBC) [99] Also developed by John Forrest, CBC utilizes CLP as the relaxed problem solver, and Coin-OR Cut Generation Library (CGL) to solve large-scale mixed integer linear problems. Also written in C++, it utilizes a MATLAB interface developed within this work. CBC is currently the second fastest open source MILP solver, falling just behind SCIP (described below under nonlinear solvers), as reported by Hans Mittelmann in [220].

GNU Linear Programming Kit (GLPK) [194] Developed by Andrew Makhorin of the Moscow Aviation Institute, GLPK is a standard go-to MILP solver that contains both simplex and primal-dual interior point solvers, as well as utilizing Gomory cuts together with a branch and bound solver for solving mixed-integer problems. Written in C, GLPK uses the GLPKMEX MATLAB interface with minimal changes to work with OPTI.

LP_Solve [33] LP_Solve is another popular MILP solver originally developed by Michel Berkelaar. Since version 1.5, it has been developed by a number of other authors, and contains interfaces to a suite of high-level languages. The solver utilizes the revised simplex algorithm with branch and bound a branch and bound solver for mixed integer problems. Written in C, LP_Solve utilizes the MATLAB interface supplied with the solver, with minor modifications to work with OPTI.

QSOPT [16] Developed by several authors from IBM, the University of Waterloo, plus AT&T Labs, QSOPT is currently the fastest LP solver for small problems. While officially a closed source project, binaries are distributed freely by the authors, who were kind enough to supply the source for OPTI. The project is written in C, and we developed the MATLAB interface to call it.

C.1.1.2 Quadratic Programming

Quadratic Programming (QP) solves problems of the form

$$\begin{aligned} \min_{\mathbf{x}} \; & \frac{1}{2}\mathbf{x}^T\mathbf{H}\mathbf{x} + \mathbf{f}^T\mathbf{x} \\ \text{subject to: } & \mathbf{A}\mathbf{x} \leq \mathbf{b} \\ & \mathbf{A}_{\text{eq}}\mathbf{x} = \mathbf{b}_{\text{eq}} \\ & \mathbf{l}_{\text{b}} \leq \mathbf{x} \leq \mathbf{u}_{\text{b}} \end{aligned}$$

where $\mathbf{H}$ is typically assumed to be symmetric positive-definite, which results in a convex optimization problem. Optionally, quadratic constraints can be added to form a Quadratically Constrained, Quadratic Program (QCQP)

$$\mathbf{x}^T\mathbf{Q}\mathbf{x} + \mathbf{l}^T\mathbf{x} \leq \mathbf{r}$$

and/or integer or binary constraints to form a MIQP or MIQCQP

$$x_i \in \Re, \quad x_j \in \mathbb{Z}, \quad x_k \in \{0,1\}, \quad i \neq j \neq k$$

OPTI supplies a single dedicated QP solver, however a number of other solvers also solve QPs quite efficiently. For solving QCQPs, MIQPs or MIQCQPs, SCIP is typically used, or alternatively OPTI can convert the problem into an (MI)NLP, and solve it using IPOPT or BONMIN. It is worth pointing out that utility system models, when written out in terms of a mass and energy balance, can be described as an MIQCQP. This is therefore an important problem type within this work, and a useful addition to the OPTI Toolbox. Furthermore, the most common search term within the OPTI Wiki, as reported by Google Analytics, is “QCQP”, indicating this is a very common and important industrial problem class.

Object Orientated Quadratic Programming (OOQP) [108] As described in Section 3.3.2.6, OOQP provides two primal-dual interior point algorithms, both of which can also solve LPs quite efficiently. OOQP is written in C++ and utilizes a MATLAB interface we developed, although this is based on the original interface developed by Michael Gertz.

C.1.1.3 Semidefinite Programming

While not actually used within this work, semidefinite programming is applied successfully to optimal control problems, as well as solving relaxed problems within many integer and global optimization problems. A Semi-Definite Program (SDP) has the following form

$$\begin{aligned} \min_{\mathbf{x}} \; & \mathbf{f}^T\mathbf{x} \\ \text{subject to:} \; & \mathbf{A}\mathbf{x} \leq \mathbf{b} \\ & \mathbf{l}_{\text{b}} \leq \mathbf{x} \leq \mathbf{u}_{\text{b}} \\ & \mathbf{X} = \sum_{i=1}^{n} x_i \mathbf{F}_i - \mathbf{F}_0 \\ & \mathbf{X} \succeq \mathbf{0} \; \text{[Positive Semidefinite]} \end{aligned}$$

where each $\mathbf{F}$ is a symmetric matrix and the resulting expression describes a Linear Matrix Inequality (LMI). OPTI contains two robust interior-point SDP solvers, as described below.

a C library for SemiDefinite Programming (CSDP) [43] Developed in C by Brian Borchers, CSDP is a predictor-corrector implementation of a SDP algorithm proposed by Helmberg, Rendl, Vanderbei and Wolkowicz. The library leverages sparsity and also includes OpenMP parallelized matrix routines. The solver utilizes a MATLAB interface developed within this work, based on suggestions by Johan Löfberg, developer of YALMIP [189].

Dual-scaling SemiDefinite Programming (DSDP) [32] Written by Steven Benson and Yinyu Ye, DSDP is a dual-scaling interior-point solver for semidefinite problems. As with CSDP, DSDP leverages sparsity but also contains algorithms to exploit low-rank structure. DSDP comes with a detailed MATLAB interface, however its approach to storing semidefinite constraints appeared inefficient and therefore a new interface was written.

C.1.1.4 Nonlinear Equation Solving

Nonlinear equation solving, commonly known as multivariable root solving, is solving a general vector nonlinear problem of the following form

$$\mathbf{F}(\mathbf{x}) = \mathbf{0}$$

and optionally may include linear constraints, such as

$$\begin{aligned}
&\mathbf{Ax} \le \mathbf{b} \\
&\mathbf{A}_{\mathrm{eq}}\mathbf{x} = \mathbf{b}_{\mathrm{eq}} \\
&\mathbf{l}_{\mathrm{b}} \le \mathbf{x} \le \mathbf{u}_{\mathrm{b}}
\end{aligned}$$

however these are not supported by the solver mentioned below (HYBRJ). Typically constrained nonlinear equations are solved using a general nonlinear solver, or a nonlinear least squares solver with the fitting data set to **0**. Nonlinear equation solving is of particular importance within this work as it is required to converge simulated utility system models with multiple recycle loops, where 'stream-tearing' is used to iteratively solve a steady state.

MINPACK Powell Hybrid (HYBRJ) [225] Together with LM_DER, described in the next subsection, HYBRJ is an algorithm from the MINPACK project of the late 1970s to early 1980s. The project was run via Argonne National Laboratory and the code survives today as a series of Fortran routines that are virtually unchanged from 1980. The HBYRJ routine is designed to solve a system of nonlinear equations by using a modification of the Powell-Hybrid method and is remarkably efficient, given its age. A C MATLAB interface was developed to interface to the Fortran routines, including both HYBRJ (analytical Jacobian) and HYBRD (internal finite-difference).

C.1.1.5 Nonlinear Least Squares

An extension of nonlinear equation solving, Nonlinear Least Squares (NLS) (also referred to as nonlinear curve fitting) solves a problem of the form

$$\min_{\mathbf{x}} \|\mathbf{F}(\mathbf{x}) - \mathbf{ydata}\|_2^2$$

which is basically attempting to minimize the sum of squared differences between the fitting function and fitting/experimental data (**ydata**). Note the above norm is never actually formed by the user, only the fitting function ($\mathbf{F}(\mathbf{x})$) and data vector ($\mathbf{ydata}$) are supplied and the solver constructs the above internally. Some solvers are also able to solve both bounded and linearly constrained problems

$$\begin{aligned}
&\mathbf{Ax} \leq \mathbf{b} \\
&\mathbf{A}_{\text{eq}}\mathbf{x} = \mathbf{b}_{\text{eq}} \\
&\mathbf{l}_{\text{b}} \leq \mathbf{x} \leq \mathbf{u}_{\text{b}}
\end{aligned}$$

however generally they are limited to bounded problems only (the only exception is LEVMAR). Each of the solvers supplied with OPTI is described below:

Levenberg Marquardt in C/C++ (LEVMAR) [190] Written by Manolis Lourakis, LEVMAR is an implementation of the Levenberg-Marquardt [184, 201] algorithm with modifications to handle both bounded problems, as well as linearly constrained problems. It is worth noting LEVMAR is the only NLS solver to solve problems with linear constraints within the OPTI framework. It does this using a variable elimination strategy based on a QR decomposition as described in [235], as well as using an alternative algorithm for box-constrained problems. While the solver supplies a MATLAB interface, we have opted to write our own one because the version supplied did not interface naturally with OPTI.

MINPACK Levenberg Marquardt (LM_DER) [224] As with HYRBJ above, LM_DER is a MINPACK Fortran routine developed at Argonne National Laboratory around 1980. It uses a modification of the Levenberg Marquardt algorithm to solve unconstrained nonlinear least squares problems. A C MATLAB interface was also developed for this solver, and includes routines that use an analytical Jacobian (LM_DER) and finite differences (LM_DIF).

Intel MKL Trust Region Nonlinear Least Squares (MKLTRNLS) [148] One of the few commercial solvers collected within OPTI, MKLTRNLS is Intel's implementation of a trust-region solver that is packaged with their Math Kernel Library. As I have personally purchased this software, the royalty free licence allows me to distribute it as part of the OPTI Toolbox. MKLTRNLS solves bounded nonlinear least squares problems, using a MATLAB interface developed within this work.

Adaptive Nonlinear Least Squares (NL2SOL) [78] Possibly the best performing nonlinear least squares solver in the OPTI collection, NL2SOL is another early 1980s Fortran solver that is based on an adaptive local-model algorithm. The latest release of NL2SOL is known as DN2GB, and is part of the PORT library, a collection of Fortran mathematical routines hosted by Netlib. This version expanded NL2SOL to solve bounded nonlinear least squares problems as well. A C MATLAB interface was developed to interface to NL2SOL, and includes NL2SNO for use without the user needing to supply analytical gradient information.

C.1.1.6 Nonlinear Programming

Nonlinear Programming (NLP) solves problems of the form

$$\begin{aligned} \min_{\mathbf{x}} \; & f(\mathbf{x}) \\ \text{subject to: } & \mathbf{A}\mathbf{x} \leq \mathbf{b} \\ & \mathbf{A}_{\text{eq}}\mathbf{x} = \mathbf{b}_{\text{eq}} \\ & \mathbf{l}_{\text{b}} \leq \mathbf{x} \leq \mathbf{u}_{\text{b}} \\ & \mathbf{c}(\mathbf{x}) \leq \mathbf{d} \\ & \mathbf{c}_{\text{eq}}(\mathbf{x}) = \mathbf{d}_{\text{eq}} \end{aligned}$$

where $f(\mathbf{x})$, $\mathbf{c}(\mathbf{x})$ and $\mathbf{ceq}(\mathbf{x})$ are generally expected to be smooth, convex, and twice differentiable (however certain solvers, described below, do not enforce this). Mixed integer problems add the following binary/integer constraints

$$x_i \in \Re, \quad x_j \in \mathbb{Z}, \quad x_k \in \{0,1\}, \quad i \neq j \neq k$$

noting that MINLPs are typically regarded as the most computationally expensive problems to solve. OPTI supplies a large number of NLP solvers, however they vary in what constraints and problem types they can solve. Each solver is described below:

Basic Open source Nonlinear Mixed INteger programming (BONMIN) [41] Developed by a group of leading discrete optimization academics, BONMIN is an experimental framework containing six algorithms for solving large-scale MINLPs via standard branch and bound, as well as outer approximations and other hybrid methods. It uses IPOPT for solving relaxed nonlinear problems, as well as CBC as the branch and cut framework. At the original conception of OPTI, BONMIN was the final implementation goal, given it could solve the MINLP utility optimization problems within this work. The interface to BONMIN is a modified version of the IPOPT interface, described below.

FILTERSD [96] Written in Fortran by Roger Fletcher (of BFGS fame), FilterSD solves smooth nonlinear problems with both bounds and general nonlinear constraints. In addition, it contains solvers for both dense and sparse problems.

It will however only find local solutions. FilterSD is part of the Coin-OR framework, although is not in active development. A C MATLAB interface was developed to interface to both versions of the solver.

Interior Point OPtimizer (IPOPT) [323] Possibly the most well-known open source optimization solver available, IPOPT is a large-scale convex nonlinear optimization solver that supports bounds, linear constraints and nonlinear constraints. Given the design focus on large-scale problems, IPOPT leverages sparsity information for both the first and second derivatives, and also within solving the resulting KKT systems by using solvers such as MUMPS to solve the large sparse system of equations that result. Originally written in Fortran, IPOPT is now written in C++ and contains an advanced MATLAB interface developed by Peter Carbonetto.

Limited-memory BFGS Bounded optimization (L-BFGS-B) [347] Implementing the de-factor limited-memory Broyden-Fletcher-Goldfarb-Shanno Hessian update, L-BFGS-B also adds the ability to solve smooth, bounded general nonlinear optimization problems. The solver is implemented in Fortran, however Peter Carbonetto had also written a C interface to L-BFGS-B which was modified to suit the OPTI toolbox.

M1QN3 [111] Developed by Jean-Charles Gilbert, M1QN3 is a large-scale unconstrained nonlinear solver, reportedly used for weather forecasting optimization in France on problems with up to 10^8 variables. As above, it is written in Fortran and also uses a limited-memory BFGS update together with a globalization line search. A C MATLAB interface was developed to interface with the solver.

NonLinear OPTimization (NLOPT) [157] As detailed earlier in Section 6.2.1 NLOPT follows a similar concept as OPTI, by supplying 20 solvers collected together as a single package. Its focus is solely on nonlinear optimization, however it contains algorithms for local and global optimization, as well as derivative and derivative-free options. In addition, it contains an augmented Lagrangian solver for adding general nonlinear constraints to a solver which would otherwise not support them. The package comes with a basic MATLAB interface, however it has been substantially upgraded in order to allow natural interfacing with the OPTI toolbox.

Nonlinear Optimization using Mesh Adaptive Direct search (NOMAD) [81] One of the most popular solvers within the OPTI framework, NOMAD solves non-differentiable nonlinear problems, including problems with nonlinear inequality constraints and integer constraints. It is written in C++ by an academic and industrial group, primarily from Montreal. The MATLAB interface was developed in collaboration with the authors and two versions now exist, one for their team, and one for OPTI users (modified to better suit the framework).

Particle and pattern Swarm (PSWARM) [316] Another derivative-free solver, PSWARM solves non-differentiable nonlinear problems that may be subject to both bounds and linear inequality constraints. It implements both pattern search and particle swarm algorithms, and has a useful feature where that all evaluation points are always feasible (very unusual among optimization solvers). Written in both C and MATLAB, we developed a C interface to the C version to maximize the speed of the algorithm.

Solving Constraint Integer Problems (SCIP) [2] As described earlier in Section 3.3.2.8, SCIP is arguably the most powerful solver in the OPTI collection. SCIP solves linear, quadratic and nonlinear problems to proven global optimality, including non-convex problems that may include integer constraints. Targeted primarily at mixed integer linear and quadratic problems, it is currently the fastest open-source MILP solver available, as well as the only solver in OPTI that can prove a global optimum for an integer, quadratic or nonlinear problem. It is limited to solving problems with only a subset of MATLAB functions. For more details of the interface developed, see Section 7.2.2.

C.1.2 Mathematical Utilities

In addition to supplying a suite of solvers and routines for creating, solving and plotting optimization problems, OPTI provides a library of utility functions and algorithms.

C.1.2.1 File Input/Output

To aid users of existing optimization software to read their optimization problems into MATLAB/OPTI, several File I/O routines have been built into the object. As of the latest version of OPTI, users can read in MPS, LP, GMPL, AMPL, SPDA and SeDuMi models directly into the object

```
>> OptProb = opti('afiro.mps')
```

which will read the `afiro` MPS model into MATLAB then convert it to an OPTI problem, ready for solving. In addition, a user can write any compatible OPTI problem to a MPS, LP, SDPA, SeDuMi or GAMS model using the `write` method,

```
>> write(OptProb,'myOptProb.mps')
```

Currently nonlinear models can only be written to GAMS files, and only if SCIP is installed (it is only supplied with the academic version of OPTI). However this does allow a user to test their problem using the NEOS server (`www.neos-server.org/neos/`) should they want to attempt to solve their problem using a commercial solver.

C.1.2.2 Dynamic System Parameter Estimation

Written to assist with a consulting project of our research group, the dynamic system parameter estimator implements the standard forward-sensitivity equation [128] to derive the gradient of a dynamic system with respect to its parameters. This enables a nonlinear least squares solver to converge faster and more robustly to the true parameter values. The functionality is built directly into the OPTI object

```
>> OptDynProb = opti('ode',ode,'data',tm,zm,'x0',x0,'z0',z0)
```

where `'ode'` is a function handle to the dynamic system (standard MATLAB Cauchy format), `'data'` is the experimental data to fit to, `'x0'` is the initial parameter guess and `'z0'` is the initial states of the ODE. The types of problems and data that can be entered are quite versatile, and the full functionality of OPTI including all NLS solvers is available. A set of detailed examples is presented on the OPTI Wiki at `http://www.i2c2.aut.ac.nz/Wiki/OPTI/index.php/Dynamic/DNLS`.

C.1.2.3 Multi-Start Solver

For optimization problems where the solution is particularly sensitive to the initial solution guess (`x0`), it can be advantageous to search around other initial points to see if a better solution can be found. To automate this process the OPTI method `multisolve` automatically applies a very simple multi-start algorithm to solving the problem. The algorithm divides the problem into hyper-cubes (thus only works for low-dimensional problems) and automatically samples the objective and constraints. For areas meeting certain selection criteria, the optimizer is run within the bounds of the hyper-cube, in an effort to find the true global optimum.

C.2 SymBuilder

SymBuilder is used to develop algebraic models within MATLAB, including symbolic simplification and automatic generation of symbolic derivatives. The framework is described in Section 6.5, with the following subsections detailing the methods available for viewing the solution of a SymBuilder model, as well as the SymUtility class, a derived class for building algebraic steam utility models.

C.2.1 Solution Inspection

A common issue found as the models increased in size was that inspecting the solution vector was became increasingly difficult to parse. This meant decoding whether the solver had returned a feasible or even a realistic solution was a tedious and error-prone task. Furthermore, often key system parameters such as the power balance were the result of internal model calculations, and were complex to calculate manually. Two tools were therefore written into SymBuilder to allow the easy inspection of an optimization run, and are described below.

C.2.1.1 Text Report

Built into the SymBuilder object is the ability to designate expressions and variables as 'result expressions' which can be automatically compiled into a optimization summary. Following the dual-stage steam turbine example from Section C.2.2.9, a common reported variable might be the total power generated by the unit

$$W_{\text{shaftwork}} = W_{S1} + W_{S2} \tag{C.1}$$

For this example we simply enter the expression we want to calculate, together with a name (`BPT_W`) and group (`D`)

```
U.AddResultExp('D:BPT_W','BPT_W1+BPT_W2');
```

which will add the expression for result reporting purposes only. Note that in this instance the result expression has been assigned to group D. This letter is arbitrary, but the groups are used for separating out results into an easy to read format. To name a group one simply calls the relevant method

```
U.AddResultGroup('D','GTG & Turbo Generators Output [kW]');
```

which is again stored in the object. Once the optimization problem has been completed and solved, calling the following method generates a solution summary

```
Results(U)
```

where a snippet of an example report is shown below

```
A: Fuel Gas Consumption [ton/h]
  - BLR1         = 0
  - BLR2         = 0
  - GTG          = 10.312

B: Electricity Balance [kW]
  - PWR          = 0.00070187     [Buy]

C: Water Consumption [ton/h]
  - Make Up      = 5.3235

D: GTG & Turbo Generators Output [kW]
  - GTG          = 10844          [On]
  - TG1          = 5047.8         [On]
  - BPT_W        = 4651.8
```

Most routines listed in Section 6.5.2 automatically add result expressions as part of their routine. This means the report shown is virtually created automatically from start to finish, with only custom models to be added to the result report. For models with a binary variable associated with them, the report will also show the state of the binary variable, which is useful when sanity checking the results.

C.2.1.2 JSteam Excel Export

When a JSteam Excel model is available in addition to the SymBuilder model, the results can be automatically exported to the JSteam model. This functionality has become highly useful in inspecting an optimized point, both in terms of inspecting temperatures and mass flows around the model, and also in diagnosing where the approximated model is deviating from the JSteam models. Exporting to Excel is a simple one line

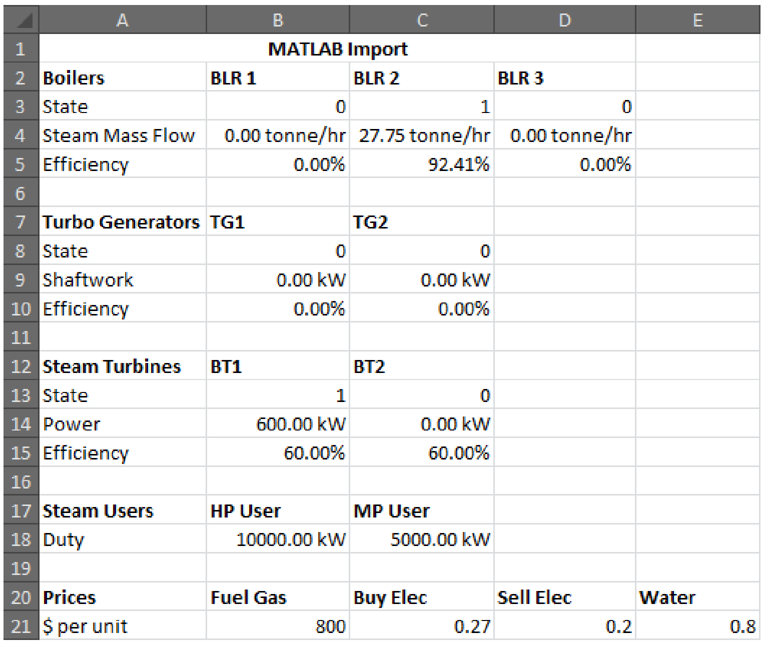
```
ExcelExport(U,'C:\Models\Aguilar3HDR.xlsx',AguilarExport)
```

which automatically propagate user selectable values into the Excel model. As this method requires knowledge of how the user has built the JSteam Excel model, the final argument is a function which defines where the model data is exported to within Excel

```
function ExData = AguilarExport()

ExData = {'[bblr1,bblr2,bblr3]','B3:D3'
          '[m29,m31,m33]','B4:D4'
          '[BLR1_Eff,BLR2_Eff,BLR3_Eff]','B5:D5'
          '[btg1,btg2]','B8:C8'
          '[TG1_W,TG2_W]','B9:C9'
          '[TG1_Eff,TG2_Eff]','B10:C10'
          '[bt1,bt2]','B13:C13'
          '[BT1_W,BT2_W]','B14:C14'
          '[BT1_Eff,BT2_Eff]','B15:C15'
          '[HPU_W,MPU_W]','B18:C18'
          '[Cost_FG,Cost_BElec,Cost_SElec,Cost_Water]','B21:E21'};
```

As shown, the user can specify array arguments (Excel ranges) of variables that exist in the SymBuilder model which are then exported to the corresponding cells in the 'Control' sheet within Excel, as shown in Figure C.1. From the Control



	A	B	C	D	E
1		MATLAB Import			
2	**Boilers**	**BLR 1**	**BLR 2**	**BLR 3**	
3	State	0	1	0	
4	Steam Mass Flow	0.00 tonne/hr	27.75 tonne/hr	0.00 tonne/hr	
5	Efficiency	0.00%	92.41%	0.00%	
6					
7	**Turbo Generators**	**TG1**	**TG2**		
8	State	0	0		
9	Shaftwork	0.00 kW	0.00 kW		
10	Efficiency	0.00%	0.00%		
11					
12	**Steam Turbines**	**BT1**	**BT2**		
13	State	1	0		
14	Power	600.00 kW	0.00 kW		
15	Efficiency	60.00%	60.00%		
16					
17	**Steam Users**	**HP User**	**MP User**		
18	Duty	10000.00 kW	5000.00 kW		
19					
20	**Prices**	**Fuel Gas**	**Buy Elec**	**Sell Elec**	**Water**
21	$ per unit	800	0.27	0.2	0.8

Figure C.1: JSteam Excel model control page with variables imported from MATLAB.

sheet, normal Excel references can then be used to pass these specifications to the JSteam Excel unit operations. The system is flexible and any number of values can be exported to Excel and to any location, including diagnostic expressions if required. The downside to this approach is that the user must exhaustively specify which variables are exported to which cells, but this does allow flexibility within the implementation.

C.2.2 SymUtility

SymUtility derives (in an object-orientated sense) from SymBuilder, thus inheriting functionality from the SymBuilder class. It then further customizes the functionality to include a suite a methods for building steam utility unit operation models and automatically adding them to the optimization problem. The subsections below describe each of the methods developed, including how operational constraints are added formulated within this framework.

Note models built within SymUtility assume duty is in kW, mass flow is in tonne/hr, pressure is in bar and temperature is in °C, based on the common set of units used in the iCON project.

C.2.2.1 Add Steam Boiler (`AddBlrFrd`)

A detailed steam boiler model (Section 5.5) is added to the current `SymUtility` object. The user specifies the operating parameters of the boiler as well as the range of steam production over which the boiler is to be modelled. From this information, the method creates the following model

$$M_{\text{fuel}} = \lambda b_{\text{blr}} + \gamma M_{\text{steam}} \tag{C.2}$$

where λ and γ are regressed parameters and b_{blr} indicates whether the boiler is switched on. Note that this binary parameter is required so that the constant term (λ) does not indicate fuel is required when the boiler is not generating any steam. In addition, as described in Section 6.4.2, two linear constraints are required: One for ensuring the mass flow of steam is zero when the boiler is switched off

$$M_{\text{steam}} - M_{\text{big}} b_{\text{blr}} \leq 0 \tag{C.3}$$

and the second for ensuring the steam mass flow is always above the minimum steam flow (typically set as 10% of the maximum, if not specified) when the boiler is switched on

$$-M_{\text{steam}} + M_{\text{big}} b_{\text{blr}} \leq -M_{\text{steam(min)}} + M_{\text{big}} \tag{C.4}$$

Both constraints use the 'big M' strategy (in this work designated by M_{big}), which for this model is set as

$$M_{\text{big}} = M_{\text{steam(max)}} + 10 \text{ tonne/hr} \tag{C.5}$$

The method also bounds M_{steam} between 0 and $M_{\text{steam(max)}}$, as well as declaring b_{blr} as a binary variable bounded between 0 and 1. Using the operational specifications from the example in Section 5.3.5.2, together with a range of steam production from 5 to 50 tonne/hr, the following model is generated

$$M_{\text{fuel}} = 0.01316 b_{\text{blr}} + 0.063923 M_{\text{steam}}$$

and the constraints are set as

$$\begin{aligned} M_{\text{steam}} - 60 b_{\text{blr}} &\leq 0 \\ -M_{\text{steam}} + 60 b_{\text{blr}} &\leq 55 \\ 0 \leq M_{\text{steam}} &\leq 50 \\ b_{\text{blr}} &\in \{0, 1\} \end{aligned}$$

C.2.2.2 Add Gas Turbine (`AddGTG`)

A detailed open cycle gas turbine model (Section 5.6) is added to the current `SymUtility` object. The user specifies the operating parameters of the gas turbine as well as the range of power production over which the turbine is to be modelled. From this information, the method creates the following model

$$M_{\text{fuel}} = \frac{W_{\text{gtg}} + \alpha_{\text{gtg}} b_{\text{gtg}}}{\beta_{\text{gtg}}} \tag{C.6}$$

where α_{gtg} and β_{gtg} are regressed parameters and b_{gtg} indicates whether the gas turbine is switched on. In addition, two linear constraints are required: One for ensuring power generation is zero when the turbine is switched off

$$W_{\text{gtg}} - M_{\text{big}} b_{\text{gtg}} \leq 0 \tag{C.7}$$

and the second for ensuring the generated power is always above the minimum rated output (typically set as 20% of the maximum, if not specified) when the gas turbine is switched on

$$-W_{\text{gtg}} + M_{\text{big}} b_{\text{gtg}} \leq -W_{\text{gtg(min)}} + M_{\text{big}} \tag{C.8}$$

where the ‘big M’ value is set as

$$M_{\text{big}} = W_{\text{gtg(max)}} + 5000 \text{ kW} \tag{C.9}$$

The method also bounds W_{gtg} between 0 and $W_{\text{gtg(max)}}$, as well as declaring b_{gtg} as a binary variable bounded between 0 and 1. Using the operational specifications from the example in Section 5.3.5.3, together with a range of power production from 5000 to 15000 kW, the following model is generated

$$M_{\text{fuel}} = \frac{W_{\text{gtg}} + 2992.31037 b_{\text{gtg}}}{5275.8795}$$

and the constraints are set as

$$\begin{aligned} W_{\text{gtg}} - 20000 b_{\text{gtg}} &\leq 0 \\ -W_{\text{gtg}} + 20000 b_{\text{gtg}} &\leq 15000 \\ 0 \leq W_{\text{gtg}} &\leq 15000 \\ b_{\text{gtg}} &\in \{0, 1\} \end{aligned}$$

C.2.2.3 Add GTG + Unfired HRSG (`AddGTGHRSG`)

An approximated unfired HRSG model (Section 5.7) and gas turbine model (Section 5.6) are added to the current `SymUtility` object. The user specifies the operating parameters of the HRSG and gas turbine as well as the range of power production over which the turbine is to be modelled. From this information, the method creates the fuel usage model from Section C.2.2.2 and the following steam expression

$$M_{\text{steam}} = \alpha_{\text{steam}} b_{\text{gtg}} + \beta_{\text{steam}} W_{\text{gtg}} \tag{C.10}$$

which describes the production of steam as a function of gas turbine power output. In addition to the gas turbine constraints from the last section, the model adds a further linear constraint using the expression derived above

$$\alpha_{\text{steam}} b_{\text{gtg}} + \beta_{\text{steam}} W_{\text{gtg}} - M_{\text{steam}} = 0 \tag{C.11}$$

which constrains the predicted steam generation to match the mass flow within the respective decision variable. This is required so that the optimizer connects the gas turbine output with the steam production. Using the operational specifications from the example in Section 5.3.5.4, together with a range of power production from 5000 to 15000 kW, the following model is generated

$$M_{\text{fuel}} = \frac{W_{\text{gtg}} + 2992.31037 b_{\text{gtg}}}{5275.8795}$$

and the constraints are set as

$$\begin{aligned}
W_{\text{gtg}} - 20000 b_{\text{gtg}} &\leq 0 \\
-W_{\text{gtg}} + 20000 b_{\text{gtg}} &\leq 15000 \\
6.17605 b_{\text{gtg}} + 0.001283 W_{\text{gtg}} - M_{\text{steam}} &= 0 \\
0 \leq W_{\text{gtg}} &\leq 15000 \\
b_{\text{gtg}} &\in \{0, 1\}
\end{aligned}$$

C.2.2.4 Add GTG + Fired HRSG (`AddGTGHRSG`)

An approximated fired HRSG model (Section 5.7) and gas turbine model (Section 5.6) are added to the current `SymUtility` object. The user specifies the operating parameters of the HRSG and gas turbine as well as the range of power production over which the turbine is to be modelled. From this information, the method creates the following fuel usage model

$$M_{\text{fuel}} = \alpha_{\text{hrsg}} b_{\text{gtg}} + \beta_{\text{hrsg}} W_{\text{gtg}} + \gamma_{\text{hrsg}} M_{\text{steam}} \tag{C.12}$$

where α_{hrsg}, β_{hrsg}, and γ_{hrsg} are regressed parameters. In addition to adding the two gas turbine constraints, the method adds one linear constraint for the minimum production of steam in order to meet the base load production

$$-M_{\text{steam}} + M_{\text{big}} b_{\text{gtg}} \leq -\alpha_{\text{base}} b_{\text{gtg}} - \beta_{\text{base}} W_{\text{gtg}} + M_{\text{big}} \tag{C.13}$$

one constraint for the maximum production of steam

$$M_{\text{steam}} + M_{\text{big}} b_{\text{gtg}} \leq \alpha_{\text{max}} b_{\text{gtg}} + \beta_{\text{max}} W_{\text{gtg}} + M_{\text{big}} \tag{C.14}$$

and one constraint to ensure no steam is produced when unit is switched off

$$M_{\text{steam}} - M_{\text{big}} b_{\text{gtg}} \leq 0 \tag{C.15}$$

where $\alpha_{\text{base}}, \beta_{\text{base}}$ are the regressed base steam coefficients, and $\alpha_{\text{max}}, \beta_{\text{max}}$ are the regressed maximum steam coefficients. The 'big M' value is set as

$$M_{\text{big}} = M_{\text{steam(max)}} + 10 \text{ tonne/hr} \tag{C.16}$$

Using the operational specifications from the example in Section 5.3.5.4, together with a range of power production from 5000 to 15000 kW, the following model is generated

$$M_{\text{fuel}} = 0.19629 b_{\text{gtg}} + 0.0001125 W_{\text{gtg}} + 0.062407 M_{\text{steam}}$$

and the constraints are set as

$$\begin{aligned}
W_{\text{gtg}} - 20000 b_{\text{gtg}} &\leq 0 \\
-W_{\text{gtg}} + 20000 b_{\text{gtg}} &\leq 15000 \\
M_{\text{steam}} + 57.11885 b_{\text{gtg}} &\leq 46.1772 b_{\text{gtg}} + 0.00823 W_{\text{gtg}} + 57.11885 \\
-M_{\text{steam}} + 57.11885 b_{\text{gtg}} &\leq -5.9429 b_{\text{gtg}} - 0.00123 W_{\text{gtg}} + 57.11885 \\
M_{\text{steam}} + 57.11885 b_{\text{gtg}} &\leq 0 \\
0 \leq W_{\text{gtg}} &\leq 15000 \\
b_{\text{gtg}} &\in \{0, 1\}
\end{aligned}$$

C.2.2.5 Add Back Pressure Turbine (`AddBPT`)

A simple steam turbine model (Section 5.3.3.3) with a specified (fixed) efficiency is added to the current `SymUtility` object. The user specifies the input and output pressures, rated output shaft work and specified isentropic efficiency, and the method creates the following model

$$W_{\text{shaftwork}} = \eta M_{\text{steam}} \Delta H_{\text{isen}} \tag{C.17}$$

$$H_{\text{out}} = H_{\text{in}} - \eta \Delta H_{\text{isen}} \tag{C.18}$$

$$\Delta H_{\text{isen}} = \alpha_h + \beta_h H_{\text{in}} \tag{C.19}$$

where α_h and β_h are regressed parameters. This model is tasked with calculating the required mass flow of steam to match a specified output load, where it is expected both the load and the input conditions do not vary widely, so a part-load expression is not required. In order for the optimizer to match the turbine output to the load specification, the following constraint is implemented

$$W_{\text{shaftwork(rated)}} b_{\text{bpt}} - \eta M_{\text{steam}} \Delta H_{\text{isen}} = 0 \tag{C.20}$$

noting the binary variable b_{bpt} has been introduced to model the turbine switched on/off. This bilinear constraint ensures that when the optimizer selects the turbine to switch on, the required mass flow is calculated to produce the specified shaft work.

C.2.2.6 Add Steam Turbo Generator (`AddTurboGen`)

A detailed steam turbine model (Section 5.4) is added to the current `SymUtility` object. The user specifies the operating parameters of the steam turbine as well as the range of output shaft work over which the turbine is to be modelled. From this information, the method creates the following model

$$W_{\text{shaftwork}} = \frac{\alpha - \Delta H_{\text{isen}} M_{\text{steam(max)}}}{5\beta} b_{\text{tgen}} + \frac{6\left(\Delta H_{\text{isen}} - \frac{\alpha}{M_{\text{steam(max)}}}\right)}{5\beta} M_{\text{steam}} \tag{C.21}$$

$$H_{\text{out}} = H_{\text{in}} - \frac{6\Delta H_{\text{isen}}}{5\beta}\left(1 - \frac{\alpha}{\Delta H_{\text{isen}} M_{\text{steam(max)}}}\right)\left(1 - \frac{M_{\text{steam(max)}}}{6\left(M_{\text{steam}} + (1 - b_{\text{tgen}})\right)}\right) \tag{C.22}$$

$$\Delta H_{\text{isen}} = \alpha_h + \beta_h H_{\text{in}} \tag{C.23}$$

where α, β, α_h and β_h are regressed parameters and b_{tgen} indicates whether the steam turbine is switched on. Note this binary term also features in the expression for outlet enthalpy, because without it as M_{steam} approaches zero, H_{out} approaches infinity. This term biases the denominator to ensure that when the steam turbine is not operating, a sensible enthalpy value is still calculated. Moreover, as this enthalpy value is later multiplied by the mass flow in the energy balance, so long as the value is a real number (not NaN or Inf), the model remains sensible. In addition, two linear constraints are required: One for ensuring power generation is zero when the turbine is switched off

$$M_{\text{steam}} - M_{\text{big}} b_{\text{tgen}} \leq 0 \tag{C.24}$$

and the second for ensuring the generated power is always above the minimum rated output (typically set as 40% of the maximum, if not specified) when the steam turbine is operating

$$-M_{\text{steam}} + M_{\text{big}} b_{\text{tgen}} \leq -M_{\text{steam(min)}} + M_{\text{big}} \tag{C.25}$$

where the ‘big M’ value is set as

$$M_{\text{big}} = M_{\text{steam(max)}} + 10 \tag{C.26}$$

The method also bounds M_{steam} between 0 and $M_{\text{steam(max)}}$, as well as declaring b_{tgen} as a binary variable bounded between 0 and 1. Using the operational specifications from the example in Section 5.4.4, together with a maximum output power of 1500kW, the following model is generated

$$\begin{aligned}
W_{\text{shaftwork}} &= 536.656 b_{\text{tgen}} - 155.379 M_{\text{steam}} - 0.26155 b_{\text{tgen}} H_{\text{in}} + 0.0757 H_{\text{in}} M_{\text{steam}} \\
H_{\text{out}} &= H_{\text{in}} - 0.878 \left(0.31 H_{\text{in}} - 601.88\right) \left(\frac{202.23}{1.787 H_{\text{in}} - 3464.68} - 1\right) \\
&\left(\frac{5.76}{1.667 M_{\text{steam}} - 1.667 b_{\text{tgen}} + 1.667} - 1\right)
\end{aligned}$$

and the constraints are set as

$$\begin{aligned}
M_{\text{steam}} - 30.7231 &\leq 0 \\
-M_{\text{steam}} + 30.7231 &\leq 22.4339 \\
W_{\text{shaftwork}} &\leq 1500 \\
b_{\text{tgen}} &\in \{0, 1\}
\end{aligned}$$

C.2.2.7 Add Fixed Efficiency Pump (`AddFixPump`)

A simple steam turbine model (Section 5.3.3.8) with a fixed inlet enthalpy and fixed efficiency is added to the current `SymUtility` object. The user specifies the input and output pressures, inlet enthalpy and efficiency, and the method creates the following model

$$W_{\text{pump}} = M_{\text{steam}} \Delta H_{\text{isen}} \tag{C.27}$$

where ΔH_{isen} is calculated using the JSteam `Pump` function. This model is a simple approximation of a boiler feed water pump that does not contain a part-load model. The function assumes the inlet enthalpy is constant (which is realistic as it is typically very close to saturated water) and the efficiency remains constant, so that the outlet enthalpy and ΔH_{isen} calculated by JSteam remains constant. The result is an estimate of the power required to pump the specified mass flow of steam to the required boiler feed water pressure.

C.2.2.8 Add Steam User (`AddUser`)

A duty-based steam user is added to the current `SymUtility` object, representing the steam demand of a process user. A duty-based steam user is used due to header enthalpies being allowed to 'float' as part of the optimization process, therefore the mass flow of steam required would vary based on the temperature of the header. Given a specified duty and condensate return enthalpy, the method creates the following constraint

$$\frac{Q_{\text{user}}}{H_{\text{in}} - H_{\text{out}}} - M_{\text{steam}} = 0 \tag{C.28}$$

which constrains the mass flow entering the user to match the duty required, given the inlet enthalpy. For users which do not return all input steam as condensate (which is typical), a further mass balance can be implemented

$$M_{\text{steam(out)}} = F_{\text{ret}} M_{\text{steam(in)}} \tag{C.29}$$

where F_{ret} is the condensate return fraction of the user.

C.2.2.9 Modelling Other Unit Operations

In order to model utility operations not included within the library developed so far, SymBuilder contains functionality to enter expressions and constants in order to calculate required optimization parameters. An example unit is a simple constant-efficiency model of a dual-stage steam turbine

$$\begin{aligned}
H_{\text{out}_{S1}} &= H_{\text{in}} - \eta_{S1} \Delta H_{\text{isen}_{S1}} \\
W_{\text{shaftwork}_{S1}} &= \left(M_{\text{steam}_{S1}} + M_{\text{steam}_{S2}}\right)\left(H_{\text{in}} - H_{\text{out}_{S1}}\right) \\
H_{\text{out}_{S2}} &= H_{\text{out}_{S1}} - \eta_{S2} \Delta H_{\text{isen}_{S2}} \\
W_{\text{shaftwork}_{S2}} &= M_{\text{steam}_{S2}}\left(H_{\text{out}_{S1}} - H_{\text{out}_{S2}}\right) \\
W_{\text{shaftwork}} &= W_{\text{shaftwork}_{S1}} + W_{\text{shaftwork}_{S2}}
\end{aligned} \tag{C.30}$$

where the subscript $S1$ indicates mass flow/shaftwork/enthalpy of stage 1, and $S2$ of stage 2. In order to implement the model in SymBuilder, we must be able to generate the required intermediate expressions and constants, in addition to standard constraints. This functionality is provided using two methods, `AddExpression` and `AddConstant`, as shown

```
% Stage 1 Expressions
U.AddExpression('BPT_H1 = h1 - BP1_Eff1*dH_HM');
U.AddExpression('BPT_W1 = (m1+m2)*(1/3.6)*(h1 - BPT_H1)');
% Stage 2 Expressions
U.AddExpression('BPT_H2 = BPT_H1 - BPT_Eff2*dH_ML');
U.AddExpression('BPT_W2 = m2*(1/3.6)*(BPT_H1 - BPT_H2)');

% Operational Constraint
U.AddCon('BPT_W1 + BPT_W2 <= BPTRW');

% Unit Specifications
U.AddConstant('BPTRW',5000,'BPT_Eff1',0.8,'BPT_Eff2',0.75);
```

where `dH_HM` and `dH_ML` are isentropic enthalpy drop expressions for HP to MP and MP to LP, respectively. These can similarly be entered as SymBuilder expressions

```
U.AddExpression(GenIsenHExp(JStm,'dH_HM','h1',HP_P,MP_P,HP_H));
U.AddExpression(GenIsenHExp(JStm,'dH_ML','BPT_H1',MP_P,LP_P,MP_H));
```

Using this workflow arbitrary models can be built up, with the framework automatically substituting expressions and constants into the resulting model objective and constraints, symbolically simplifying it then generating the required MATLAB functions.

C.2.2.10 Power Balance Objective Term

The cost of buying and selling electricity will be different in the majority of electricity markets. This is result of factors such as geographical location, market demands, and transmission system constraints. This means the objective function used in optimizing a utility system must use the correct price depending on whether the system is generating or consuming power. Within this work, it is modeled as

$$C_{\text{power}} = \left(C_{\text{sell}}\left(1 - b_{\text{bal}}\right) + C_{\text{buy}} b_{\text{bal}}\right) W_{\text{total}} \tag{C.31}$$

where C represents cost, b_{bal} is a binary variable indicating whether we are selling or buying power, and W_{total} is the power balance of the site, defined similar to

$$W_{\text{total}} = W_{\text{gtg}} + W_{\text{tgen}} - W_{\text{pump}} - W_{\text{site}} \tag{C.32}$$

noting W_{total} is *positive* when generation exceeds demand (the site has power to sell), and *negative* when the reverse is true and the site must buy power.

In order for the cost function to be correct, b_{bal} must change value depending on the sign of W_{total}. This is modelled as two linear constraints and using the 'big M' strategy, as used in the last subsection. The first constraint ensures the binary variable is off when generating power

$$W_{\text{total}} + M_{\text{big}} b_{\text{bal}} \leq M_{\text{big}} \tag{C.33}$$

while the second ensures the variable is on when buying power

$$-W_{\text{total}} - M_{\text{big}} b_{\text{bal}} \leq 0 \tag{C.34}$$

The 'big M' value can be set to any value larger than the maximum demand and generation potential of the site, typically 10kW larger the maximum absolute value.

C.3 Steam Utility SymBuilder Models

The MATLAB models from Sections 6.6.1 and 6.6.2 are listed in the following sections. To run these models, the reader is referred to the Appendix DVD which contains both the models and case studies included within this work.

C.3.1 Three Header Steam Model

The basic model is shown below, while the full simulation file supplied on the Appendix DVD under SymUtility Models.

```
U = SymUtility;
U.AddSteamGroups();

% Mass Balance Equations
U.AddCon('m1-m2-m3-m4-m5-m6-m7 = 0');        %HP Header
U.AddCon('m10+m3+m5-m11-m12-m13-m14 = 0');  %MP Header
U.AddCon('m9+m16-m18 = 0');                 %User Mixer
U.AddCon('m17+m12+m4+m19-m20-m21 = 0');     %LP Header
U.AddCon('m24-m22-m23 = 0');                %Condensate Mixer
U.AddCon('m26-m15-m8-m27 = 0');             %BFW Splitter
U.AddCon('m27-m28-m30-m32 = 0');            %Boiler Splitter
U.AddCon('m1-m29-m31-m33 = 0');             %Steam Mixer
% Energy Balance Equations
U.AddCon('h1*m1 - h2*(m2+m3+m4+m5+m6+m7) = 0'); %HP Header
U.AddCon('h11*m10 + TG1_H*m3 + BT1_H*m5 - h11*(m11+m12+m13+m14) = 0');
U.AddCon('HPU_H*m9 + MPU_H*m16 - h18*m18 = 0'); %User Mixer
U.AddCon('h20*m17 + BT2_H*m12 + TG2_H*m4 + FLSH_VH*m19 - h20*(m20+m21) = 0');
U.AddCon('FLSH_LH*m22 + MU_H*m23 - h24*m24 = 0'); %Condensate Mixer
% General Bounds
U.AddBound('0 <= m <= 150');
U.AddBound('100 <= h <= 3500');

% Fired Boilers
U.AddBlrFrd(JNet,{'BLR1','m29','m28','h1','bblr1'},Fuel,Air,BLR1_rangeM,
            30,30,200,BFW_H,400,40,0.01,0.01);
U.AddBlrFrd(JNet,{'BLR2','m31','m30','h1','bblr2'},Fuel,Air,BLR2_rangeM,
            30,30,200,BFW_H,400,40,0.01,0.01);
U.AddBlrFrd(JNet,{'BLR3','m33','m32','h1','bblr3'},Fuel,Air,BLR3_rangeM,
            30,30,200,BFW_H,400,40,0.01,0.01);
% Turbo Generators
U.AddTurboGen(JStm,{'TG1','m3','h2','btg1'},HP_P,MP_P,TG1_QMax,HP_H);
U.AddTurboGen(JStm,{'TG2','m4','h2','btg2'},HP_P,LP_P,TG2_QMax,HP_H);
% Back Pressure Turbines
U.AddBPT(JStm,{'BT1','m5','h2','bt1'},HP_P,MP_P,BT1_Q,BT1_Eff,HP_H);
U.AddBPT(JStm,{'BT2','m12','h11','bt2'},MP_P,LP_P,BT2_Q,BT2_Eff,MP_H);
% Desuperheaters
U.AddDesuper({'HPDsp','m2','m8','m10','h2','BFW_H','h11'});
U.AddDesuper({'MPDsp','m11','m15','m17','h11','BFW_H','h20'});
% Deaerator
U.AddDeaerator(JStm,{'DRTR','m20','m24','m25','m26','h20','h24'},DRTR_P,DRTR_VT);
% Steam Users
U.AddUser({'HPU','m6','h2','m9'},HPU_Q,HPU_H,HPU_F);
U.AddUser({'MPU','m13','h11','m16'},MPU_Q,MPU_H,MPU_F);
```

```
% Flash Drum
U.AddFlash(JStm,{'FLSH','m18','m19','m22','h18'},LP_P);
% Pump
U.AddFixPump(JStm,{'PMP','m26','BFW_H',},JStm.HPX(2,0),2,PMP_P,PMP_Eff);
% Make Up Water
U.AddWater({'Make Up','m23','MU_H'},MU_H);
% Headers
U.AddHeader({'HP','h2','m7'},JStm.HPT(HP_P,400));
U.AddHeader({'MP','h11','m14'},JStm.HPX(MP_P,1));
U.AddHeader({'LP','h20','m21'},JStm.HPX(LP_P,1));

% Costs
U.AddConstant('Cost_FG',COST_FG,'Cost_Water',COST_WATER);
U.AddConstant('Cost_BElec',COST_BELEC,'Cost_SElec',COST_SELEC);

% Power Balance
U.AddExpression('PWR = TG1_Q + TG2_Q - BT1RQ*(1-bt1) - BT2RQ*(1-bt2) - PMP_Q');
U.AddPwrBal({'PWR','bp'},60e3);
% Objective
U.AddObj('Cost_FG*(BLR1_FuelM+BLR2_FuelM+BLR3_FuelM) + Cost_Water*m23 -
         (Cost_SElec*(1-bp) + Cost_BElec*bp)*PWR');

Build(U)
```

C.3.2 Four Header Steam Model

The basic model is shown below, while the full simulation file supplied on the Appendix DVD under SymUtility Models.

```
U = SymUtility;
U.AddSteamGroups();

% Mass Balance Equations
U.AddCon('m1+m2-m3-m4-m5-m6-m7-m8-m9-m10 = 0');           %HP Header
U.AddCon('m16+m12+m13+m14+m8-m17-m19-m20-m21-m22 = 0');  %MP Header
U.AddCon('m25+m4+m18+m19+m6-m26-m28-m29 = 0');            %LP Header
U.AddCon('m27+m20+m7-m32 = 0');                   %VLP Header
U.AddCon('m15+m24+m30-m31 = 0');                  %User Header
U.AddCon('m34-m33-m32-m31 = 0');                  %Condensate Mixer
U.AddCon('m36-m23-m37-m38-m11-m39-m40 = 0');      %BFW Splitter
% Energy Balance Equations
U.AddCon('HBLR_H*m1 + HRSG_H*m2 - h3*(m3+m4+m5+m6+m7+m8+m9+m10) = 0'); %HP Header
U.AddCon('h17*m16 + MBLR_H*m12 + WHB_H*m13 + BT1_H1*m14 + BT4_H*m8 -
         h17*(m17+m19+m20+m21+m22) = 0'); %MP Header
U.AddCon('h26*m25 + TG1_H*m4 + BT1_H2*m18 + BT2_H*m6 + BT5_H*m19 -
         h26*(m26+m28+m29) = 0'); %LP Header
U.AddCon('m34*h34 - m33*MU_H - m32*VLP_H - m31*USER_H = 0'); %Condensate Mixer

% General Bounds
U.AddBound('0 <= m <= 350');
U.AddBound('100 <= h <= 3500');

% GTG + HRSG
U.AddGTGHRSG(JNet,{'GTG','GTG_Q','m2','m39','HRSG_H','bgtg1'},Fuel,Air,
             30,30,500,[],GTG_rangeQ,Fuel,180,0.01,BFW_H,HRSG_T,HP_P,0.03);
```

```
% Fired Boilers
U.AddBlrFrd(JNet,{'BLRH','m1','m40','HBLR_H','bblrh'},Fuel,Air,HBLR_rangeM,
            30,30,200,BFW_H,369,HP_P,0.03,0.01);
U.AddBlrFrd(JNet,{'BLRM','m12','m38','MBLR_H','bblrm'},Fuel,Air,MBLR_rangeM,
            30,30,200,BFW_H,264,MP_P,0.03,0.01);
% Waste Heat Boiler
U.AddWHB(JStm,{'WHB','m13','m37','WHB_H'},WHB_Q,BFW_H,WHB_T,MP_P,0.03,WHB_Eff);
% Turbo Generators
U.AddTurboGen(JStm,{'TG1','m4','h3','btg1'},HP_P,LP_P,TG1_QMax,HP_H);
% Back Pressure Turbines
U.AddBPT(JStm,{'BT2','m6','h3','bt2'},HP_P,LP_P,BT2_Q,BT2_Eff,HP_H);
U.AddBPT(JStm,{'BT3','m7','h3','bt3'},HP_P,VLP_P,BT3_Q,BT3_Eff,HP_H);
U.AddBPT(JStm,{'BT4','m8','h3','bt4'},HP_P,MP_P,BT4_Q,BT4_Eff,HP_H);
U.AddBPT(JStm,{'BT5','m19','h17','bt5'},MP_P,LP_P,BT5_Q,BT5_Eff,MP_H);
U.AddBPT(JStm,{'BT6','m20','h17','bt6'},MP_P,VLP_P,BT6_Q,BT6_Eff,MP_H);
U.AddBPT3(JStm,{'BT1','m5','m14','m18','m27','h3','bt1'},HP_P,MP_P,LP_P,
          VLP_P,BT1_Q,BT1_Eff1,BT1_Eff2,BT1_Eff3,HP_H);
% Desuperheaters
U.AddDesuper({'HPDsp','m3','m11','m16','h3','BFW_H','h17'}); %HP Desuperheater
U.AddDesuper({'MPDsp','m17','m23','m25','h17','BFW_H','h26'}); %MP Desuperheater
% Deaerator
U.AddDeaerator(JStm,{'DRTR','m26','m34','m35','m36','h26','h34'},DRTR_P,DRTR_VT);

% Steam Users
U.AddUser({'HPU','m9','h3','m15'},HPU_Q,HPU_H,HPU_F);
U.AddUser({'MPU','m21','h17','m24'},MPU_Q,MPU_H,MPU_F);
U.AddUser({'LPU','m28','h26','m30'},LPU_Q,LPU_H,LPU_F);
% Pumps
U.AddFixPump(JStm,{'PMP','m36','BFW_H'},JStm.HPX(DRTR_P,0),DRTR_P,PMP_P,PMP_Eff);
U.AddFixPump(JStm,{'VACPMP','m32','VLP_H'},JStm.HPX(VLP_P,0),VLP_P,DRTR_P,PMP_Eff);
% Make Up Water
U.AddWater({'Make Up','m33','MU_H'},MU_H);
% Headers
U.AddHeader({'HP','h3','m10'},JStm.HPT(HP_P,350));
U.AddHeader({'MP','h17','m22'},JStm.HPX(MP_P,1));
U.AddHeader({'LP','h26','m29'},JStm.HPX(LP_P,1));

% Costs
U.AddConstant('Cost_FG',COST_FG,'Cost_Water',COST_WATER);
U.AddConstant('Cost_BElec',COST_BELEC,'Cost_SElec',COST_SELEC);
% Constants
U.AddConstant('USER_H',UserH);
U.AddConstant('SITE_Q',SITE_Q);

% Power Balance
U.AddExpression('PWR = GTG_Q + TG1_Q - SITE_Q - BT1RQ*(1-bt1) - BT2RQ*(1-bt2) -
                BT3RQ*(1-bt3) - BT4RQ*(1-bt4) - BT5RQ*(1-bt5) - BT6RQ*(1-bt6) -
                PMP_Q - VACPMP_Q');
U.AddPwrBal({'PWR','bp'},60e3);
% Objective
U.AddObj('Cost_FG*(BLRH_FuelM+BLRM_FuelM+GTG_FuelM) + Cost_Water*m33 -
         (Cost_SElec*(1-bp) + Cost_BElec*bp)*PWR');

% Case Study based Fixed Constraints
switch(cstudy)
    case 1
        U.AddBound('0 <= bt1 <= 0');
        U.AddBound('0 <= bt3 <= 0');
        U.AddBound('0 <= bt5 <= 0');
        U.AddBound('0 <= bt6 <= 0');
        U.AddBound('0 <= bgtg1 <= 0');
    case 11
        U.AddBound('0 <= bt1 <= 0');
        U.AddBound('0 <= bt2 <= 0');
        U.AddBound('0 <= bt3 <= 0');
        U.AddBound('0 <= bt5 <= 0');
        U.AddBound('0 <= bgtg1 <= 0');
    case 2
        U.AddBound('0 <= btg1 <= 0');
        U.AddBound('0 <= bt3 <= 0');
        U.AddBound('0 <= bt4 <= 0');
        U.AddBound('0 <= bt6 <= 0');
    case 3
        U.AddBound('0 <= btg1 <= 0');
        U.AddBound('0 <= bt6 <= 0');

end

Build(U)
```

C.4 Global Optimization Interfaces

Additional details of the MATLAB interfaces developed to BARON and SCIP, both global white-box solvers.

C.4.1 Compatible BARON Interface MATLAB Functions

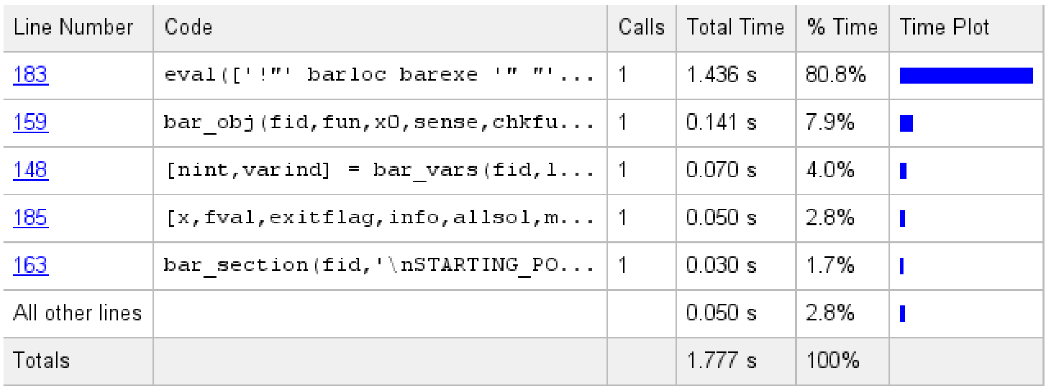
Table C.1 lists the MATLAB functions which have been overloaded with the BARON Interface, thus enabling their use when modelling global optimization problems in MATLAB.

C.4.2 BARON Interface Parsing Performance

As described in Section 7.2.1, the BARON Interface generates a text representation of a MATLAB program suitable for parsing by BARON. Considering the amount of overhead involved in generating and concatenating equation strings, the process is remarkably quick, as shown by the MATLAB profiler in Figure C.2. To convert and solve a 100 variable vectorized Rosenbrock problem into BARON format took just over 140ms (2nd entry), with generating and writing the remainder of the BARON problem taking 100ms (3rd and 5th entries). Solving the problem to global optimality took 1.4s (1st entry), while parsing the results took just 50ms (4th entry).

Table C.1: MATLAB - BARON Interface supported functions.

`abs`	Absolute value
`cumsum`	Cumulative sum
`diag`	Extract diagonal of matrix, or convert vector to diagonal matrix
`diff`	Difference of adjacent elements
`dot`	Dot product
`eval`	Evaluate the object using substituted numerical constants
`exp`	Exponential function
`fliplr`	Flip matrix left/right
`flipud`	Flip matrix up/down
`horzcat`	Horizontal cconcatenation
`isscalar`	Check if object is a scalar
`length`	Return length of the object
`log`	Natural Logarithim
`log10`	Log Base 10
`minus`	Element-wise subtraction
`mpower`	Matrix raised to an integer power, or scalar raised to another scalar
`mrdivide`	Element-wise divide (not matrix divide)
`mtimes`	Matrix multiplication
`norm`	2-norm for vectors, Frobenius norm for matrices
`plus`	Element-wise addition
`power`	Element-wise power
`prod`	Vector product, or matrix row/column product
`rdivide`	Element-wise division
`repmat`	Replicate Matrix
`reshape`	Reshape object
`rot90`	Rotate matrix 90 degrees
`size`	Return object size
`sqrt`	Element-wise square root
`subsasgn`	Assigning elements via an index
`subsref`	Reading elements via an index
`sum`	Vector sum, or matrix row/column sum
`times`	Element-wise Multiplication
`trace`	Trace of a matrix
`transpose`	Transpose object dimensions
`tril`	Extract lower triangular section
`triu`	Extract upper triangular section
`uminus`	Element-wise unary minus
`vertcat`	Vertical concatenation



Line Number	Code	Calls	Total Time	% Time	Time Plot
183	`eval(['!"' barloc barexe '" "'...`	1	1.436 s	80.8%	▬
159	`bar_obj(fid,fun,x0,sense,chkfu...`	1	0.141 s	7.9%	■
148	`[nint,varind] = bar_vars(fid,l...`	1	0.070 s	4.0%	▮
185	`[x,fval,exitflag,info,allsol,m...`	1	0.050 s	2.8%	▮
163	`bar_section(fid,'\nSTARTING_PO...`	1	0.030 s	1.7%	▮
All other lines			0.050 s	2.8%	▮
Totals			1.777 s	100%	

Figure C.2: MATLAB - BARON Interface profile results for 100 variable Rosenbrock NLP.

C.4.3 Compatible SCIP Interface MATLAB Functions

Table C.2 lists the MATLAB functions which have been overloaded with the SCIP Interface, which as per the BARON Interface, allow them to be used to model global optimization problems.

Table C.2: MATLAB - SCIP Interface currently supported functions.

`abs`	Absolute value
`dot`	Dot product
`exp`	Exponential function
`horzcat`	Horizontal cconcatenation
`isscalar`	Check if object is a scalar
`length`	Return length of the object
`log`	Natural Logarithim
`log10`	Log Base 10
`minus`	Element-wise subtraction
`mpower`	Matrix raised to an integer power, or scalar raised to another scalar
`mrdivide`	Element-wise divide (not matrix divide)
`mtimes`	Matrix multiplication
`norm`	2-norm for vectors, Frobenius norm for matrices
`plus`	Element-wise addition
`power`	Element-wise power
`prod`	Vector product, or matrix row/column product
`rdivide`	Element-wise division
`size`	Return object size
`sqrt`	Element-wise square root
`subsasgn`	Assigning elements via an index
`subsref`	Reading elements via an index
`sum`	Vector sum, or matrix row/column sum
`times`	Element-wise Multiplication
`transpose`	Transpose object dimensions
`uminus`	Element-wise unary minus
`vertcat`	Vertical concatenation

C.4.4 Global Optimization Solver Interface Validation

To validate both of the interfaces developed, 16 small test problems have been taken from GLOBALLib [103], a collection of global optimization problems, and 16 from MINLPLib [51], a collection of mixed integer nonlinear problems. For each problem, a reference solution was obtained using the original problem within GAMS and BARON, and then the problem was entered into MATLAB. The interfaces to SCIP and BARON were then used to solve the problem, with the results shown in Table C.3.

As shown, the BARON interface obtains the reference solution in all cases, indicating the interface is correctly functioning. For SCIP, it fails to find a solution for problem `nvs09` within the allowable 10,000 nodes. Note that this problem however occurred in the original GAMS model with SCIP as well, which would indicate this is not an interface issue. For the remainder of the problems, both SCIP and BARON have solved for the correct solution using the MATLAB interfaces developed. This small test validates that both interfaces are functioning correctly, or at least for the problems tested, noting that they are representative of the utility system models, albeit smaller.

Table C.3: Global optimization solver interface validation results.

		SCIP Results			BARON Results		
Name	Reference	Fval	Time[s]	Nodes	Fval	Time[s]	Nodes
`st_e01`	-6.667	-6.667	0.017	1	-6.667	0.173	1
`st_e02`	201.159	201.159	0.018	1	201.159	0.150	1
`st_e03`	-1161.337	-1161.337	0.370	184	-1161.337	0.749	31
`st_e04`	5194.866	5194.866	0.557	17	5194.866	0.635	3
`st_e05`	7049.249	7049.249	0.263	164	7049.249	0.544	25
`st_e06`	0.000	0.000	0.023	1	0.000	0.135	-1
`st_e07`	-400.000	-400.000	0.188	31	-400.000	0.234	1
`st_e08`	0.742	0.742	0.029	1	0.742	0.168	1
`st_e09`	-0.500	-0.500	0.073	15	-0.500	0.206	3
`st_e10`	-16.739	-16.739	0.026	1	-16.739	0.201	3
`st_fp1`	-17.000	-17.000	0.051	5	-17.000	0.243	7
`st_fp2`	-213.000	-213.000	0.020	1	-213.000	0.173	1
`st_fp3`	-15.000	-15.000	0.023	1	-15.000	0.222	1
`st_fp4`	-11.000	-11.000	0.055	3	-11.000	0.205	1
`st_fp5`	-268.015	-268.015	0.055	3	-268.015	0.269	1
`st_fp6`	-39.000	-39.000	0.071	5	-39.000	0.297	3
`nvs01`	12.470	12.470[1]	0.298	25	12.470	0.496	13
`nvs02`	5.964	5.964	0.018	1	5.964	0.360	5
`nvs03`	16.000	16.000	0.014	1	16.000	0.166	1
`nvs04`	0.720	0.720	0.048	3	0.720	0.282	1
`nvs05`	5.471	5.471	0.550	28	5.471	36.327	2610
`nvs06`	1.770	1.770	0.121	13	1.770	0.364	1
`nvs07`	4.000	4.000	0.015	1	4.000	0.154	-1
`nvs08`	23.450	23.450	0.043	1	23.450	1.060	8
`nvs09`	-43.134	-17.517	7.215	10000	-43.134	0.361	1
`nvs10`	-310.800	-310.800	0.022	1	-310.800	0.311	3
`nvs11`	-431.000	-431.000	0.025	3	-431.000	0.536	9
`nvs12`	-481.200	-481.200	0.075	9	-481.200	0.750	12
`nvs13`	-585.200	-585.200	0.090	8	-585.200	0.989	48
`nvs14`	-40358.155	-40358.155	0.019	1	-40358.155	0.340	5
`nvs15`	1.000	1.000	0.018	1	1.000	0.274	4
`nvs16`	0.703	0.703	0.083	4	0.703	0.295	3

[1] Obtained with a tightened tolerance of 1×10^{-9}

Appendix D

Appendix DVD Contents

Table D.1 below lists the software included on the attached DVD. This is provided to support the examples presented in this work.

Table D.1: Software included on the Appendix DVD.

Folder	Description
jMPC Toolbox	The latest jMPC Toolbox version.
OPTI Toolbox	The latest OPTI Toolbox version.
JSteam Toolbox	The latest JSteam Toolbox version.
JSteam Excel Add In	The latest version of the JSteam Excel Add-In.
MPC Models	Dynamic models and associated model predictive controllers used within this work.
SymUtility Models	Both the three and four header hypothetical utility models, ready to run (require JSteam + OPTI Installed). Also included are the Excel versions (requires the Excel Add In).