Carnegie Mellon University

CARNEGIE INSTITUTE OF TECHNOLOGY

THESIS

SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF Doctor of Philosophy

TITLE Applications, Modeling Tools, and Parallel Solution Algorithms for Dynamic Optimization

PRESENTED BY Bethany Nicholson

ACCEPTED BY THE DEPARTMENT OF

Chemical Engineering

LORENZ BIEGLER	4/2816
LORENZ BIEGLER, ADVISOR AND DEPARTMENT HEAD	DATE

APPROVED BY THE COLLEGE COUNCIL

VIJAYAKUMAR BHAGAVATULA	4/28/16
DEAN	DATE

CARNEGIE MELLON UNIVERSITY

Applications, Modeling Tools, and Parallel Solution Algorithms for Dynamic Optimization

A DISSERTATION

SUBMITTED TO THE GRADUATE SCHOOL

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

for the degree of

DOCTOR OF PHILOSOPHY

in

CHEMICAL ENGINEERING

by

BETHANY NICHOLSON

B.S. CHEMICAL ENGINEERING, ILLINOIS INSTITUTE OF TECHNOLOGY

Pittsburgh, Pennsylvania

April, 2016

Acknowledgments

I would first like to thank my advisor Larry Biegler. It has been a true privilege to learn from him during my time at CMU. I greatly appreciate all of the opportunities that Larry has facilitated for me including internships, conferences, and introductions. His patience and dedication towards his students is truly remarkable.

I would also like to thank the members of my committee, Professors Erik Ydstie, Nick Sahinidis, and Marija Ilic for all of their help, guidance, and encouragement during my PhD studies. I'd also like to thank all the members of Prof. Biegler's research group for being wonderful office-mates and for all the interesting conversations about research, in particular Jun, Wei, Xue, and Alex. In addition, I would like to express my sincerest gratitude to John Siirola and Victor Zavala with whom I've had the opportunity to collaborate during internships and who have continued to offer valuable advice and mentoring throughout my PhD. I would like to thank Jean-Paul Watson, Bill Hart, and Carl Laird for their encouragement and support over the years. Lastly, I wish to thank Prof. Don Chmielewski for encouraging me to attend graduate school in the first place.

I would like to gratefully acknowledge funding from the NSF and ExxonMobil Research and Engineering Company which supported this work. Additionally, I would like to thank Sandia National Laboratories, Argonne National Laboratory, and Air Products and Chemicals for their support during my internships.

I would like to thank my friends and family for their unwavering love and support during one of the most challenging chapters of my life. I do not attempt to list every one by name but I would not be where I am today without them. To my amazing friends Sarah, Melissa, Edna, Wayne, and Rob, thank you for making Pittsburgh such a wonderful place to be, supporting me, and being my family away from home. I'd like to thank my mom, my step-dad Clint, my grandma, my aunt Sisi, and my dad for their love and for always believing in me. I would also like to thank my siblings Jeremy and Rebekah for helping to shape me into the person I am today. Finally, I would like to thank Michael for his support, love, and constant encouragement.

Bethany Nicholson
Pittsburgh, PA
April 2016

Abstract

Dynamic optimization problems directly incorporate detailed dynamic models as constraints within an optimization framework. Model predictive control, state estimation, and parameter estimation are all common applications of dynamic optimization which can lead to significant improvements in process efficiency, reliability, safety, and profitability. This dissertation deals with dynamic optimization from three perspectives.

We begin with an application of dynamic optimization. State estimation is a crucial part of the monitoring and/or control of all chemical processes. We make use of a state estimation technique called moving horizon estimation (MHE) which can be formulated as a dynamic optimization problem. However, large-scale MHE formulations may require non-negligible computational time to solve limiting its application for real-time state estimation. An extension of MHE, called Advanced Step Moving Horizon Estimation (asMHE), eliminates this computational delay. Both MHE and asMHE perform well under the assumption of Gaussian measurement noise. We consider the case where this assumption does not hold and measurements are contaminated with large errors. Standard least squares based estimators generate biased estimates even with relatively few gross measurement errors. We therefore extend MHE and asMHE formulations using robust M-Estimators in order to mitigate the bias of these errors on the state estimates. We demonstrate this approach on dynamic models of a CSTR and a distillation column and find that our approach produces fast and accurate state estimates even in the presence of many gross measurement errors.

A well-established method to solve dynamic optimization problems is direct transcription where the differential equations are replaced with algebraic approximations using some numerical method such as a finite-difference or Runge-Kutta scheme. In the second part of this work we present `pyomo.dae`, an open-source modeling framework that enables high-level abstract representations of optimization problems with differential and algebraic equations. A key distinctive feature of `pyomo.dae` is that it does not adhere to standard, predefined formats of optimal control and estimation problems. This enables high modeling flexibility and the consideration of constraints and objective functions in non-standard forms that cannot be easily handled by traditional solution methods and cannot be expressed in other modeling frameworks. `pyomo.dae` also enables the specification of optimization problems with high-order differential equations and partial differential equations on restricted domain types and it provides automatic discretization capabilities to transcribe high-level abstract models into finite-dimensional algebraic problems that can be solved with off-the-shelf optimization solvers.

However, for problems with thousands of state variables and discretization points, direct transcription may result in nonlinear optimization problems that exceed memory and speed limits of most serial computers. In particular, when applying interior point optimization methods, the computational bottleneck and dominant computational cost lies in solving the linear systems resulting from the Newton steps that solve the discretized optimality conditions. To overcome these limits, we exploit the parallelizable structure of the linear system to accelerate the overall interior point algorithm. We investigate two algorithms which take advantage of this property, cyclic reduction and Schur complement decomposition and study the performance of these algorithms when applied to dynamic optimization problems.

Contents

Acknowledgments **i**

Abstract **ii**

Contents **iii**

List of Tables **vi**

List of Figures **vii**

1 Introduction **1**
1.1 Dynamic Optimization . . . 1
1.2 Solution Strategies for Dynamic Optimization . . . 3
1.2.1 Single and Multiple Shooting . . . 3
1.2.2 Direct Transcription . . . 4
1.3 Typical Implementation . . . 5
1.4 Research Problem Statement . . . 8
1.5 Dissertation Overview . . . 9

I Applications **10**

2 State Estimation with Large Measurement Errors **11**
2.1 Literature Review . . . 11
2.2 State Estimation Formulation . . . 13
2.2.1 Moving Horizon Estimation . . . 14
2.2.2 Advanced Step Moving Horizon Estimation . . . 15
2.3 Robust Statistics and M-Estimators . . . 17
2.3.1 Fair Function . . . 19
2.3.2 Redescending Estimator . . . 20
2.3.3 Robust Estimator Tuning and Use . . . 21
2.4 Concluding Remarks . . . 22

3 State Estimation Case Studies **23**
3.1 Implementation . . . 23
3.2 CSTR Case Study . . . 24
3.2.1 Results Case 1: Single Drift . . . 25
3.2.2 Results Case 2: Multiple Drifts . . . 27

3.2.3 Results Case 3: Varying Horizon Length . . . 28
3.3 Distillation Case Study . . . 31
3.3.1 Distillation Model . . . 31
3.3.2 Results Case 1: Drift Errors . . . 32
3.3.3 Results Case 2: Step Errors . . . 33
3.4 Concluding Remarks . . . 36

II Modeling Tools **37**

4 A Modeling and Direct Transcription Framework **38**
4.1 Introduction and Motivation . . . 38
4.1.1 Pyomo . . . 40
4.2 Modeling Components in `pyomo.dae` . . . 42
4.2.1 `ContinuousSet` . . . 42
4.2.2 `DerivativeVar` . . . 47
4.3 Discretization Transformations . . . 51
4.3.1 Finite Difference Transformation . . . 52
4.3.2 Collocation Transformation . . . 54
4.3.3 Applying Multiple Transformations . . . 57
4.4 Package Implementation and Extensibility . . . 58
4.5 Concluding Remarks . . . 62

5 Examples of Modeling Capabilities **64**
5.1 Heat Transfer . . . 64
5.2 Optimal Control . . . 72
5.3 Parameter Estimation for Disease Transmission . . . 75
5.4 Stochastic Optimal Control for Natural Gas Networks . . . 79
5.5 Concluding Remarks . . . 84

III Parallel Solution Algorithms **86**

6 Parallel Algorithms for Dynamic Optimization **87**
6.1 Introduction . . . 87
6.2 Interior-point Method . . . 89
6.3 Exploiting Structure in the KKT System . . . 91
6.4 Cyclic Reduction . . . 94
6.5 Schur Complement Decomposition . . . 97
6.6 Sparsity Analysis . . . 99
6.7 Numerical Conditioning . . . 102
6.8 Concluding Remarks . . . 107

7 Computational Performance of Parallel Solvers **108**
7.1 Schur Complement Decomposition in PIPS-NLP . . . 108
7.1.1 CSTR Case Study . . . 109
7.1.2 Distillation Column Case Study . . . 110
7.2 Cyclic Reduction Prototype . . . 114
7.2.1 QP Structured Without Algebraic Variables . . . 115
7.2.1.1 Larger Blocks . . . 119
7.2.2 QP Structured With Algebraic Variables . . . 122
7.3 Concluding Remarks . . . 127

8 Conclusions **128**
8.1 Summary and Contributions . . . 128
8.2 Recommendations for Future Work . . . 131

Bibliography **134**

Appendices **147**

A Mathematical Models **148**
A.1 General 3 State CSTR . . . 148
A.2 Distillation Column Model . . . 149

List of Tables

3.1 Variance values for noise in CSTR example 25
3.2 Normalized Sum of Squared Errors for CSTR case 1 simulation 27
3.3 Normalized Sum of Squared Errors for CSTR case 2 simulation 28
3.4 Normalized Sum of Squared Errors for CSTR case 3 simulation. LS: Least Squares, FF: Fair Function, RED: Redescending 31
3.5 Variance values for noise in Distillation example 32
3.6 Normalized Sum of Squared Errors for Distillation case 1 simulation 34
3.7 Normalized Sum of Squared Errors for Distillation case 2 simulation 34

4.1 Existing tools for handling optimization problems with differential equations. 39

5.1 Comparison of two implementations of the disease transmission problem . 78
5.2 Effect of spatial discretization resolution on computational performance . . 83

7.1 Time to solve one time step of the CSTR MHE problem with different horizon lengths. Problems are solved using IPOPT and PIPS-NLP. 110
7.2 CSTR NLP problem sizes corresponding to the results in Table 7.1 111
7.3 Time to solve one time step of the Distillation MHE problem with different horizon lengths. Problems are solved using IPOPT and PIPS-NLP. 112
7.4 Distillation NLP problem sizes corresponding to the results in Table 7.3 . . . 113
7.5 Effect of sparsity and the ratio of controls to states on parallel algorithm performance. Runs were done using 12 processors and the Schur complement was factored using cyclic reduction 123

List of Figures

1.1 Diagram of the software platforms, files, and communication required for a typical dynamic optimization application 7

2.1 Comparison of the values of the M-Estimators used in this work 19

3.1 Comparison of the MHE and asMHE formulations with and without M-Estimators for the CSTR case 1 simulation. MHE: MHE with least squares, MHE_{FF}: MHE with Fair Function, MHE_{RED}: MHE with Redescending, similarly for asMHE 26
3.2 Comparison of the MHE and asMHE formulations with and without M-Estimators for the CSTR case 2 simulation. MHE_{RED*}: MHE with Redescending and initialized with Fair Function solution 29
3.3 Comparison of the concentration state estimates for the CSTR case 3 simulation using formulations with horizon lengths of 5, 10, and 20 30
3.4 Comparison of the MHE and asMHE formulations with and without M-Estimators for the Distillation case 1 simulation. MHE: MHE with least squares, MHE_{FF}: MHE with Fair Function, MHE_{RED}: MHE with Redescending, similarly for asMHE. M_{21} is the molar holdup on tray 21, x_{21} is the methanol composition on tray 21 33
3.5 Comparison of the MHE and asMHE formulations with and without M-Estimators for the Distillation case 2 simulation. M_{21} is the molar holdup on tray 21, x_{33} is the methanol composition on tray 33 35

4.1 Summary of Pyomo features 41
4.2 Pyomo workflow including discretization transformations 62

5.1 Solution to the optimal control problem (top) with no restrictions on the control variable and (bottom) restricting the control variable to be piecewise constant. Black dotted line shows the inequality path constraint. 75
5.2 Optimal axial flow profile in gas network for different resolutions. Darker color indicates more discretization points. 83

7.1 (Left) Sparsity pattern for QP case with 20 states and 10 controls. (Right) Sparsity pattern for QP case with 10 states and 20 controls. The tridiagonal blocks have been outlined in red. 117
7.2 Time to factorize a matrix with 2N states and N controls using (Left) cyclic reduction on the full matrix and (Right) cyclic reduction on the Schur complement. 117

7.3 Time to factorize a matrix with 2N states and N controls with several different algorithms run with 12 processors. 118
7.4 Time to factorize a matrix with N states and 2N controls using (Left) cyclic reduction on the full matrix and (Right) cyclic reduction on the Schur complement. 119
7.5 Time to factorize a matrix with N states and 2N controls with several different algorithms run with 12 processors. 120
7.6 Time to factorize a matrix using cyclic reduction on the full matrix for (Left) 2N states and N controls and (Right) N states and 2N controls. 121
7.7 Time to factorize a matrix with N states and 2N controls with the block size fixed to 1500 and varying the number of block rows. 121
7.8 (Left) Sparsity pattern for QP case with 5 states, 5 controls, and 50 algebraics. (Right) Sparsity pattern for QP case with 5 states, 5 controls, and 100 algebraics. 125
7.9 Time to factorize a matrix with N states, N controls, and 10N algebraics using (Left) cyclic reduction on the full matrix and (Right) cyclic reduction on the Schur complement. 126
7.10 Time to factorize a matrix with N states, N controls, and 10N algebraics with several different algorithms run with 12 processors. 126
7.11 Time to factorize a matrix with N states, N controls, and 20N algebraics using (Left) cyclic reduction on the full matrix and (Right) cyclic reduction on the Schur complement. 127

Chapter 1

Introduction

In this chapter we describe the focus and scope of this dissertation. The central theme is dynamic optimization but we approach the topic from a variety of perspectives. We begin this chapter by describing this general class of optimization problems and two strategies for solving them. Next, we review typical implementation strategies for these problems and describe several challenges that arise when formulating and solving them. At the end of the chapter we formally state the research objectives of this dissertation and give an overview of the remaining chapters in the thesis.

1.1 Dynamic Optimization

Dynamic optimization problems are optimization problems constrained by differential and algebraic equations (DAEs) or partial differential and algebraic equations (PDAEs) and are ubiquitous in engineering and science. Application domains include aerospace systems, chemical reactor systems, infrastructure (water, gas, electricity, transportation) networks, buildings, hybrid energy systems, economics and finance, robotics, and environmental engineering. A generic form for this class of problems is shown below

$$
\begin{aligned}
\min \int_{t_0}^{t_F} & \hat{\Phi}(x, y, u) dt + \hat{\Psi}(x(t_F)) \\
\text{s.t.} \quad & \frac{dx}{dt} = \hat{g}(x, y, u) \\
& x(t_0) = \bar{x}_0 \\
& h(x, y, u) = 0 \\
& x^L \leq x \leq x^U \\
& y^L \leq y \leq y^U \\
& u^L \leq u \leq u^U
\end{aligned} \tag{1.1}
$$

where x is a vector of differential variables, y is a vector of algebraic variables, and u is a vector of control, or input, variables. t is an independent continuous domain, such as time, defined over a fixed, closed interval $[t_0, t_f]$. $\hat{g}$ represents the differential equations and h represents the algebraic equations. The initial conditions of the differential states are given by $\bar{x}_0$

In the area of chemical engineering, first-principles based dynamic models are available for most classical unit operations. The differential equations in these models arise from conservation equations such as material and energy balances and the algebraic equations arise from constitutive relationships such as thermodynamic and transport properties. Incorporating these models directly in an optimization framework has been shown to improve process reliability, control, and profits [1]. Common applications of dynamic optimization include state estimation, parameter estimation, and optimal control[2, 3, 4, 5]. However, the computational costs required to solve these problems can limit their on-line applicability and utilization for large-scale processes.

1.2 Solution Strategies for Dynamic Optimization

While dynamic optimization problems are fairly straightforward to formulate given a dynamic process model, the initial challenge in using this optimization approach is picking a solution strategy. We can't solve problem (1.1) directly because optimization solvers can't handle integrals or differential equations. A common solution approach is to reformulate the problem as an algebraic NLP by discretizing the continuous dynamics. There are several ways this can be done including single shooting, multiple shooting, and direct transcription [6]. These methods differ in which parts of the model are discretized and the types of dynamic models they can solve. A brief overview of these techniques is provided below in the context of solving a classical optimal control problem. For this problem the idea is to find a control, or input, trajectory $u(t)$ over the time domain $[t_0, t_f]$ which minimizes some performance metric Φ.

1.2.1 Single and Multiple Shooting

The single shooting method works by discretizing the control trajectory over N stages. Often, the control trajectory is represented as a piece-wise constant function where the control input at each stage k is considered fixed. Once a discretized control trajectory is calculated it is passed to a DAE solver that integrates the DAE model over the entire time domain. This approach removes the DAE model from the optimization problem and replaces it with gradient information from the DAE solver with respect to the controls. The method sequentially iterates between solving an NLP problem to obtain a new guess for the controls and solving the DAE model using the new control trajectory. It has the benefit of incorporating robust DAE solvers which are designed to handle stiff dynamic systems. However, the single shooting approach does not scale well for problems with many degrees of freedom or long time horizons and may be incapable of solving ill-conditioned or unstable systems.

The multiple shooting method expands on the single shooting approach by discretiz-

ing both the control trajectory and the initial conditions for the differential states over N stages. DAE solvers are used to integrate the model over each stage k independently i.e. over individual domains $[t_k, t_{k+1}]$. The NLP problem with this approach solves for both the controls and the initial conditions for the differential states while minimizing the performance metric Φ and simultaneously linking the state profiles at the stage boundaries. This allows multiple shooting to be an effective solution technique for ill-conditioned and unstable systems. The multiple shooting approach performs well on problems with long time horizons but does not scale favorably on problems with many dynamic states.

1.2.2 Direct Transcription

In contrast to the shooting methods described above, the direct transcription approach simultaneously discretizes the control trajectory as well as the entire DAE model, thereby converging the dynamic model directly during the solution of a single large NLP problem. The continuous dynamics in the DAE model are approximated using algebraic equations such as a finite difference or runge-kutta scheme. This leads to a nonlinear programming problem (NLP) that can be solved using an off-the-shelf solver. The fully-discretized version of problem (1.1) is shown below

$$
\begin{aligned}
\min \sum_{k=1}^{N-1} & (\Phi(x_k, y_k, u_k)) + \Psi(x_N) \\
\text{s.t.} \quad & x_{i+1} = g(x_i, y_i, u_i), \quad i = 1, \ldots, N-1 \\
& x_1 = \bar{x}_0 \\
& h(x_i, y_i, u_i) = 0, \quad i = 1, \ldots, N \\
& x^L \leq x_i \leq x^U, \quad i = 1, \ldots, N \\
& y^L \leq y_i \leq y^U, \quad i = 1, \ldots, N \\
& u^L \leq u_i \leq u^U, \quad i = 1, \ldots, N
\end{aligned}
\tag{1.2}
$$

where the problem has been discretized using N discretization points. Algebraic equations used for some common discretization schemes are presented later in Chapter 4.

Direct transcription provides a systematic and convenient way to satisfy the DAE model directly and avoids the computational costs of repeatedly integrating the large-scale DAE model. This method can can be used to solve ill-conditioned and unstable systems and shows favorable computational scaling with respect to the number of differential states and the degrees of freedom. Direct transcription is the solution strategy used throughout this work.

Direct transcription produces a large-scale NLP problem and therefore requires an efficient large-scale NLP solver. However, for industrial-scale dynamic models the NLPs may become intractable in terms of solution time or memory required to formulate the problem. This is a significant challenge for using this technique in practice and is one of the topics addressed by the work in this dissertation.

1.3 Typical Implementation

The solution strategies described above are all well established and widely used methods for solving dynamic optimization problems. However, there are very few tools available for automatically applying one of these solution methods to general dynamic optimization problems. Furthermore, modeling frameworks that do exist for dynamic optimization are often domain specific and difficult to extend. In the case of direct transcription, discretization schemes are often implemented manually on the DAE model under investigation. This makes it time-consuming to try multiple discretizations on a particular model. Moreover, the discretization scheme implemented is restricted based on the user's knowledge of such schemes. High-order discretization schemes, such as orthogonal collocation, are non-trivial to implement from scratch and therefore less likely to be attempted by newcomers to dynamic optimization despite their numerical advantages.

Often times dynamic optimization applications are expressed in specific canonical forms

e.g. optimal control. However, many applications can involve complex and exotic (non-canonical) types of constraints such as multi-point boundary conditions and PDAEs spanning multiple domains. Such types of constraints cannot be easily expressed in existing modeling packages and cannot be easily handled by variational methods. Moreover, the solution of such problems requires sophisticated discretization schemes that might require the combination of different techniques and experimentation. Again, these schemes are usually implemented by hand and are thus tedious, prone to errors, and nontrivial even for experts in dynamic optimization.

Applying a discretization scheme is not the only implementation challenge associated with these problems. For applications such as nonlinear model predictive control or moving horizon estimation a sequence of dynamic optimization problems must be solved and a typical implementation may span several software platforms and many files. A diagram of this type of implementation is shown in Figure 1.1.

This diagram represents the implementation structure for the state estimation work included in this dissertation. The different colors represent different software platforms and the boxes represent different files. The implementation of a MHE state estimator requires 14 files interacting over 3 software platforms. There are many drawbacks associated with this type of implementation. For one, it is often difficult to get different software platforms to communicate and manage data efficiently. It is also difficult to pinpoint bugs and other errors across so many different platforms. Additionally, this type of implementation is difficult to port to other systems and share with collaborators. Lastly, notice that the dynamic optimization technique (i.e. optimal control, state estimation) is completely entwined with the application model being studied which limits the amount of code that can be reused for new dynamic models. This is just one example of how a dynamic optimization technique can be implemented, there is no standardized approach. Every dynamic optimization practitioner will have their own implementation strategy which makes it difficult to fairly compare the performance of different techniques.

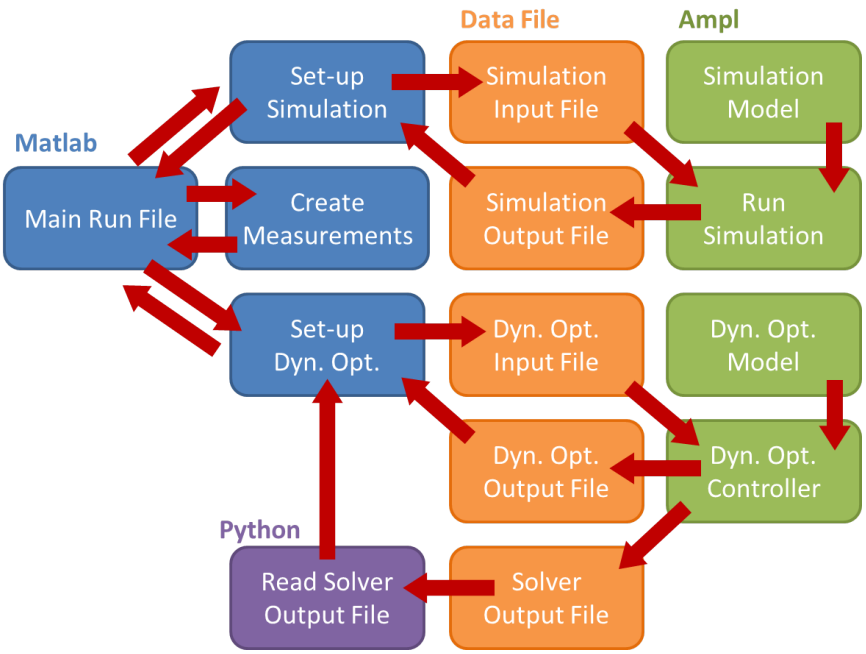


Figure 1.1: Diagram of the software platforms, files, and communication required for a typical dynamic optimization application

Future advances in dynamic optimization will likely require intricate interaction between modeling frameworks and NLP solvers that is not fully supported or easy to implement with current optimization tools. Therefore, tool development for dynamic optimization is an open and active area of research.

1.4 Research Problem Statement

This thesis addresses three significant challenges associated with dynamic optimization. First, we study a useful application of dynamic optimization in the area of state estimation with the objective of obtaining state estimates in real-time in the presence of large measurement errors. We approach this problem using MHE state estimators and reduce the online computational cost using NLP sensitivity. We address large measurement errors using robust M-estimators.

After working with the state estimation techniques described above, we noticed that the implementation of these methods was nontrivial and often quite cumbersome. This posed a significant barrier for developing novel dynamic optimization algorithms as well as using dynamic optimization techniques in real-world applications particularly for newcomers. The second objective of this work is to develop a modeling framework that will simplify the formulation of these problems. The framework we have developed is extremely flexible in the type of problems it can represent and is capable of automatically applying a simultaneous discretization to a dynamic optimization model. In addition, it separates the model abstraction from the discretization scheme allowing the user to experiment and assess the performance of different discretizations in a more systematic manner.

The last challenge addressed in this dissertation is the large-scale NLP problem produced by direct transcription. The objective is to develop parallel solution techniques capable of solving these large NLPs and thereby enable dynamic optimization techniques to be applied to industrial sized dynamic models. We address this objective by exploiting sparsity and structure inherent to these problems and we investigate a promising parallel

algorithm called cyclic reduction.

1.5 Dissertation Overview

This dissertation is separated into three parts. In the first part, we describe an application of dynamic optimization which illustrates the potential of dynamic optimization for improving process operation and control. Chapter 2 describes the state estimation problem and how it can be formulated as a dynamic optimization problem. Chapter 3 applies the state estimators developed in the previous chapter to two classical chemical engineering unit operations. In the second part of this thesis, Chaper 4 describes a new modeling framework for representing dynamic optimization problems and Chapter 5 demonstrates the flexibility of this framework on a variety of dynamic models. The third and final part of this dissertation addresses parallel solution algorithms for dynamic optimization. Chapter 6 explains how structure arises in these problems and describes two algorithms capable of exploiting that structure. Chapter 7 presents computational results for several parallel solvers. Finally, Chapter 8 summarizes the main contributions of this work and suggests avenues for future research.

Part I

Applications

Chapter 2

State Estimation with Large Measurement Errors

In this chapter we introduce one common application of dynamic optimization, state estimation. We begin with a brief introduction and literature review on state estimation and then describe two strategies for formulating the problem using dynamic optimization. Finally, we propose an extension for these strategies for dealing with large measurement errors. This chapter is based on work published in [7].

2.1 Literature Review

Improvement of on-line operation of chemical processes requires accurate knowledge of the current state of the system. This is particularly true for strategies that rely on first-principles models of the chemical plant, such as Model Predictive Control [8] or Real-Time Optimization [9]. Complicating factors that make real time state estimation challenging include the impracticality or infeasibility of measuring every state of a process directly, and delays that arise from measurements that take a significant amount of time to obtain.

Assuming that sufficient plant measurements can be obtained in real-time, one now needs to use these measurements to obtain the states of the system. This can be done by developing a model of the process which includes relationships between the measured and unmeasured state variables along with variables to estimate plant-model mismatch, unknown disturbances, and measurement noise. With this model several state estimation approaches can be applied; these approaches differ by assumptions they make about the

linearity or nonlinearity of the plant model and the probability distribution of the error and noise variables. State estimation and the closely related topic of data reconciliation have been studied under a broad variety of assumptions. For nonlinear systems, the Extended Kalman Filter (EKF)[10] is commonly applied in practice. While EKF is relatively easy to implement, it has been shown to have poor performance for highly nonlinear systems [11, 12]. Related estimation methods that also deal with nonlinear systems include the Unscented Kalman Filter [13], the Ensemble Kalman Filter [14, 15, 16], and the Particle Filter [17, 18]. While each of these methods has pros and cons, one common drawback is their inability to deal with bounds on the states. Ignoring these constraints can lead to an increase in the estimation error or the divergence of the estimator [19]. On the other hand, heuristic strategies, such as clipping, can greatly reduce the performance of the estimator. Other remedies to handle constrained nonlinear state estimation include Nonlinear Recursive Dynamic Data Reconciliation [20], Unscented Recursive Nonlinear Dynamic Data Reconciliation [20, 21], the Constrained Ensemble Kalman Filter [12], and the Constrained Particle Filter [22].

In contrast to the above estimators, the state estimation problem can be formulated directly as a nonlinear programming (NLP) problem. Here we consider Moving Horizon Estimation (MHE) [23, 24, 25, 26], which uses a batch of past measurements to find the optimal state estimates. MHE has been shown to have very desirable asymptotic stability properties [27] and often performs better than EKF [19]. In addition, constraints and bounds on plant states are handled directly by the NLP solver. On the other hand, computational delay is a major drawback for implementing MHE in an industrial setting [28].

A nice overview of methods for state estimation is given in [29], where it is noted that computational complexity still remains a significant challenge. Efficient algorithms for MHE include work done by Ohtsuka and Fujii [30] and Tenny and Rawlings [31]. More recently, an MHE based real-time iteration approach has been investigated in [32]. Additionally, Abrol and Edgar [33] applied an in situ adaptive tabulation based MHE for

on-line state estimation. In this work we make use of a fast MHE strategy based on NLP sensitivity developed by Zavala et al. [34].

Another important factor for state estimation is the robustness of our estimate. Sensors can fail or be contaminated in such a way that their measurements are vastly different from the true plant state. In this work we reduce the influence of bad measurements in state estimation through a data reconciliation and gross error detection framework. A summary of early work done in this field is given in [35]. Recent reviews of the state of the art in outlier detection can be found in [36, 37, 38]. General approaches to gross error detection include Principal Component Analysis [39], Cluster Analysis [40], Artificial Neural Networks [41], and Robust Statistics [42].

Related work has also been done in the process control literature, including investigating sensor faults and fault tolerance. A thorough review of work in this area can be found in [43]. Of particular relevance to this work is the study of Chen and You that considers fault-tolerant sensor system for noisy or drifing sensors [44]. However, it should be noted that in this work we are considering strategies to mitigate the effects of gross error measurements and not to explicitly detect or identify bad measurements.

In this chapter we consider the Moving Horizon Estimation (MHE) formulation [26] as well as its extension called advanced step Moving Horizon Estimation (asMHE) [34, 45]. Both strategies are described in more detail in Section 2.2. Our main contribution in this chapter is to extend these formulations using robust M-Estimators in order to mitigate the effect of gross errors and make our state estimates more robust. A brief overview of robust statistics and M-Estimators is given in Section 2.3.

2.2 State Estimation Formulation

As mentioned in the introduction, there are many ways to approach the state estimation problem. In this chapter we focus on its formulation as a nonlinear optimization problem. The formulations used in this work are briefly described below.

2.2.1 Moving Horizon Estimation

Moving Horizon Estimation (MHE) [26] is a well known strategy for constrained state estimation. The overall idea with MHE is to estimate the current state of the system using only the last N states and measurements directly, where we refer to N as the horizon length. In other words, we split time into two sets $T_1 = \{l|0 \leq l < k - N\}$ and $T_2 = \{l|k - N \leq l \leq k\}$ where k is the current time step. The NLP associated with this formulation is shown below. Notice that the summation terms in the objective function are only over T_2.

$$\begin{aligned} \{\hat{z}_{k-N|k}, ..., \hat{z}_{k|k}\} = \arg \min_{\{z_{k-N}, ..., z_k\}} & \Phi(z_{k-N}) + \frac{1}{2} \sum_{l=k-N}^{l=k} v_l^T R_l^{-1} v_l + \frac{1}{2} \sum_{l=k-N}^{l=k-1} w_l^T Q_l^{-1} w_l \\ \text{s.t.} \quad & z_{l+1} = f(z_l) + w_l \\ & y_l = h(z_l) + v_l \\ & z^{LB} \leq z_l \leq z^{UB}, \quad l \in T_2 \end{aligned} \tag{2.1}$$

where $\{\hat{z}_{k-N|k}, ..., \hat{z}_{k|k}\}$ refers to the optimal state estimates at each time step in the horizon. Also, $\Phi(z_{k-N}) = ||z_{k-N} - \hat{z}_{k-N|k-1}||^2_{\hat{\Pi}^{-1}_{k-N|k-1}}$ is the arrival cost and represents all the information in T_1, which is not included in the horizon. v_l is the vector of measurement noise, R_l is the covariance matrix of the measurement noise, w_l is the vector of unknown disturbances, and Q_l is the covariance matrix of the unknown disturbances. Also, the equality constraints are a state-space model of the plant where $h(z_l)$ is the measurement model and $f(z_l)$ is the system model. Additionally, bounds can be added to the states. A key assumption behind this formulation is that the measurement noise and unmeasured disturbances are zero mean, Gaussian random variables. Of course, when gross measurement errors and outliers are present, this may no longer be a valid assumption.

NLP (2.1) is solved at every time step, k. After finding a solution, the horizon moves forward in time by including the new measurement at the next time step $(k + 1)$ and simultaneously dropping the measurement farthest in the past $(k-N)$. Past measurements that are not within the horizon window are accounted for in the arrival cost. Good state estimates with the MHE formulation require that we have a good approximation of the

arrival cost or a sufficiently long horizon window. Moreover, a tradeoff exists between the two and we can obtain good estimates with a shorter horizon window if we have a better approximation of the arrival cost. The longer the horizon, the larger the NLP being solved and thus the more computational time required to solve it. To estimate the arrival cost we use an approximation derived in [46] where the arrival cost can be computed directly from the Reduced Hessian of NLP (2.1) at the solution. This strategy is particularly attractive if we solve (2.1) using a large-scale solver with exact second derivatives, as this becomes a computationally cheap calculation. The computational results in the next chapter make use of the interior point solver IPOPT [47].

Even with a small horizon, NLP (2.1) still leads to computational delay (i.e. the time between obtaining the measurements and computing the estimated states for the controller), particularly when the state-space model representing the process is large and complex. For on-line state estimation, this formulation could be problematic due to the computational delay of solving (2.1) on-line, which can cause instabilities when the estimates are used for process control purposes. Thus, in the next section we consider an extension of the MHE formulation that is better suited for on-line use.

2.2.2 Advanced Step Moving Horizon Estimation

Advanced Step Moving Horizon Estimation (asMHE) [34, 45] is based on a similar advanced step strategy for Nonlinear Model Predictive Control [48]. It is also comparable to a real-time iteration scheme for SQP methods previously presented by [32, 49]. The basic idea is that a background NLP can be formed with measurements predicted one time step in the future. The NLP (2.1) can then be solved between sampling times using this prediction as the new measurement. Once the actual measurement is obtained our state estimates can be updated using NLP sensitivity. The advantage of this strategy is that the only on-line calculation is the sensitivity update step, which results in significant reduction of the computational delay for the state estimates.

NLP sensitivity analysis [50] provides estimates for the partial derivatives of the optimal solution with respect to system parameters. These sensitivites can then be used to approximate the optimal solution under parameter perturbations. To illustrate how this works in the context of state estimation, we first represent problem (2.1) as a generic NLP

$$\min F(x, p_0), \quad \text{s.t.} \quad c(x, p_0) = 0, \quad x \geq 0 \tag{2.2}$$

where x represents the state variables and p_0 is the nominal value of the problem parameters. For state estimation, the parameters are the process measurements. In the asMHE formulation, the nominal parameter values are the predicted process measurements. The Karush-Kuhn-Tucker (KKT) conditions, or optimality conditions, of (2.2) are given by

$$\begin{aligned} \nabla_x L = \nabla_x f(x; p_0) + \nabla_x c(x; p_0)\lambda - \nu = 0 \\ c(x; p_0) = 0 \\ XVe = 0, \end{aligned} \tag{2.3}$$

where $X = \text{diag}(x)$, $V = \text{diag}(\nu)$, e is a vector of ones, and λ and ν are Lagrange multipliers. We represent the optimal solution to (2.2) as $s^*(p_0)^T = [x^{*T}, \lambda^{*T}, \nu^{*T}]$ and estimate how the optimal solution changes with a perturbation to the nominal parameter values, from p_0 to p_1. In problem (2.1) we denote p as the measurement at time k and define the parameter perturbation from the predicted measurement (p_0) to the actual measurement (p_1). Under mild regularity conditions of the NLP [51], the optimal solution to the perturbed problem can be approximated using a Taylor series expansion

$$\hat{s}^*(p_1) \approx s^*(p_0) + \left.\frac{\partial s^T}{\partial p}\right|_{p_0} (p_1 - p_0) \tag{2.4}$$

To find the partial derivative needed in (2.4) we recognize the optimality conditions (2.3) as implicit functions of the parameters p, represented as $Q(s(p), p) = 0$. We then apply the implicit function theorem to Q around $(s^*(p_0), p_0)$ to get

$$\frac{dQ(s^*(p_0), p_0)}{dp}^T = \frac{\partial Q(s^*(p_0), p_0)}{\partial p}^T + \frac{\partial Q(s^*(p_0), p_0)}{\partial s}^T \left.\frac{\partial s(p)}{\partial p}^T\right|_{p_0} = 0. \tag{2.5}$$

which can be solved for the partial derivative in (2.4). Furthermore, the $\partial Q/\partial s$ term is exactly the KKT matrix used in interior point solvers such as IPOPT [47]. This means that once IPOPT finds the optimal solution to problem (2.2), the factorized form of the KKT matrix is available and the sensitivity required in (2.4) can be calculated from a single backsolve [48]. Estimating the optimal solution in this way is much faster than solving the entire NLP again. To summarize, the advanced step MHE (asMHE) formulation consists of the following steps:

1. At time t_{k-1} generate a prediction of the process measurements $\hat{y}_k$ using the state estimates z_{k-1} and process model equations.
2. Between t_{k-1} and t_k solve the NLP (2.2) in background with measurement prediction at t_k and hold the factorized KKT matrix at the solution.
3. At time t_k obtain actual measurements y_k.
4. Update solution of NLP (2.2) using Equations (2.4), (2.5), and the factorized KKT matrix; performed on-line.
5. Update the expected value $\hat{z}_{k-N+1|k}$ and covariance matrix $\hat{\Pi}_{k-N+1|k}$ of the arrival cost using the Extended Kalman Smoother described in [46].
6. Set $k = k + 1$ and return to Step 1.

Using asMHE for state estimation relies on our predicted measurements being close to the actual measurements obtained. However, if we assume that some of our measurements are contaminated with gross errors we need to assess their influence on the accuracy of the state estimates. For this, we consider ways to formulate our state estimation problem to be robust to gross errors.

2.3 Robust Statistics and M-Estimators

Gross errors and outliers can be caused by sensor failure or miscalibration and can include sensor bias or drifting. For states that need to be recorded manually, gross errors can result

from human error as well. Detecting such errors and eliminating their bias on the state estimates is a crucial step in obtaining estimates that can be considered robust. In this work we employ robust M-Estimators for gross error detection because they can easily be implemented within our NLP formulation.

All maximum-likelihood type estimators, or M-Estimators, try to maximize some associated likelihood function [42]. In the case of the Bayesian statistical framework that led to the MHE formulation (2.1), the likelihood function associated with the observed measurements is the conditional probability density function, $p(y_l|z_l)$. In other words, we would like to find the state estimates which maximize the probability of our observed measurements. When we assume Gaussian measurement noise, the minimum variance estimator associated with this likelihood function is the least squares estimator which is the $v_l^T R_l^{-1} v_l$ term in the objective function of (2.1). Assuming that R_l is diagonal, the least squares estimator can also be written as

$$\rho_j^{LS}(\epsilon_j) = \frac{1}{2}\epsilon_j^2 \tag{2.6}$$

where $\epsilon_j = (y_j - \hat{y}_j)/\sigma_j$ is the studentized error, the difference between our actual and estimated measurement, or the residual, normalized by the standard deviation. If the measurement noise is not Gaussian, the least squares estimator may not give adequate performance, as it is not robust to deviations from the assumed Gaussian distribution. The robustness of an M-Estimator is assessed by the estimator's influence function, which is proportional to the first derivative of the estimator. An estimator is considered robust if its influence function is bounded as the observations go to infinity. The first derivative of the least squares estimator is

$$\frac{d\rho_j^{LS}(\epsilon_j)}{d\epsilon_j} = \epsilon_j \tag{2.7}$$

Therefore, its influence function is linear and unbounded as the observations go to infinity.

When our measurements are contaminated with gross errors, the assumption that the measurement noise follows a zero-mean Gaussian distribution may not hold and it could

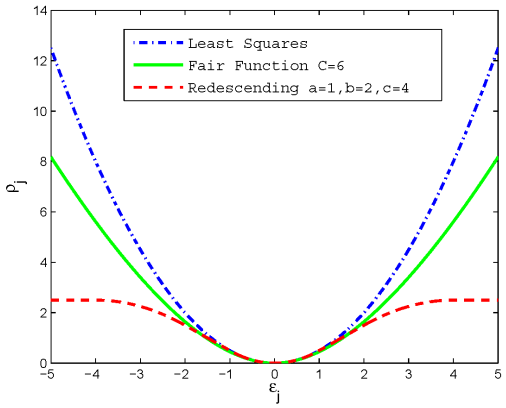


Figure 2.1: Comparison of the values of the M-Estimators used in this work

be difficult to determine the exact noise distribution. So, if we use the least squares estimator, any gross errors could have an undue influence on our state estimates. To avoid this we consider estimators that are robust to deviations from our Gaussian assumption. Two robust M-Estimators are considered in this work, the Fair (Huber-type) and Redescending (Hampel) estimators.

2.3.1 Fair Function

The Fair Function [52] is given by

$$\rho_j^F(\epsilon_j) = C^2 \left[\frac{|\epsilon_j|}{C} - \log\left(1 + \frac{|\epsilon_j|}{C}\right) \right] \tag{2.8}$$

where ρ_j is the estimator associated with the j^{th} measurement, C is a tuning parameter, and ϵ_j is the studentized prediction error. The influence function for the Fair Function is proportional to its first derivative:

$$\frac{d\rho_j^F(\epsilon_j)}{d\epsilon_j} = \frac{\epsilon_j}{1 + \frac{|\epsilon_j|}{C}} \tag{2.9}$$

The Fair Function is plotted in Figure 2.1 and is convex with smooth first and second derivatives. Notice that for small residuals the Fair Function is a good approximation of

the least squares estimator. As the residuals become larger, the Fair Function transitions from increasing quadratically to increasing linearly. Therefore, as the residuals approach infinity, the Fair Function does not increase as fast as the least squares function and so less weight will be put on gross error measurements, i.e. measurements that lead to large residuals. In other words, the influence function of the Fair Function will approach a constant, confirming that the Fair Function is robust to large measurement errors. The shape of the Fair Function can be adjusted using its tuning parameter C. Smaller values of C will cause the Fair Function to become linear at smaller residuals thus causing it to deviate more from the least squares function.

2.3.2 Redescending Estimator

The second M-Estimator used in this work is Hampel's Redescending estimator [53] which is given by the piecewise function shown below

$$\rho_j^R(\epsilon_j) = \begin{cases} \frac{1}{2}\epsilon_j^2, & 0 \leq |\epsilon_j| \leq a \\ a|\epsilon_j| - \frac{a^2}{2}, & a < |\epsilon_j| \leq b \\ ab - \frac{a^2}{2} + \frac{a(c-b)}{2}\left[1 - \left(\frac{c-|\epsilon_j|}{c-b}\right)^2\right], & b < |\epsilon_j| \leq c \\ ab - \frac{a^2}{2} + \frac{a(c-b)}{2}, & |\epsilon_j| > c, \end{cases} \tag{2.10}$$

where ϵ_j is again the studentized prediction error and a, b, and c are tuning parameters which define the 4 regions in the estimator. Each region behaves differently depending on the magnitude of the residuals. The first derivative of the Redescending estimator is

$$\frac{d\rho_j^R(\epsilon_j)}{d\epsilon_j} = \begin{cases} \epsilon_j, & 0 \leq |\epsilon_j| \leq a \\ \pm a, & a < |\epsilon_j| \leq b \\ \frac{\pm a(c-|\epsilon_j|)}{(c-b)}, & b < |\epsilon_j| \leq c \\ 0, & |\epsilon_j| > c, \end{cases} \tag{2.11}$$

and the Redescending estimator is plotted in Figure 2.1. It is nonconvex and has smooth first derivatives but nonsmooth second derivatives. We see that for small residuals in $[-a, a]$ the Redescending estimator is exactly the least squares estimator. In the second region, when $a < |\epsilon_j| \leq b$, the estimator increases linearly with respect to the residuals. As the residuals become even larger, less influence is placed on them until eventually the estimator just becomes a constant value. This translates to an influence function which becomes zero after passing the threshold defined by the tuning parameter c, which can be seen in (2.11). In practice, this has the same effect as removing any gross error measurement that exceeds this threshold. The Redescending estimator depends on 3 tuning parameters rather than just a single one like the Fair Function. Thus, it is inherently more difficult to determine their optimal values.

2.3.3 Robust Estimator Tuning and Use

In practice, robust M-estimators should be tuned to the problem under investigation. The Fair Function is typically tuned by relating its tuning constant C to the asymptotic efficiency [54]. For the Redescending estimator there is less agreement as to the best tuning strategy and numerous methods have been proposed including one based on the Akaike Information Criterion [54] and one based on estimating efficiecy using Monte Carlo simulations [42]. In our experience, good performance can be obtained from the Fair Function or the Redescending estimator for a range of tuning parameter values. Therefore, systematic tuning of the estimators used in the simulations shown in the next chapter was not required and not investigated further.

M-estimators can be incorporated into our NLP formulation by simply replacing the measurement error terms in the objective function with the ρ function of the desired esti-

mator. Our new NLP problem becomes

$$\{\hat{z}_{k-N|k}, ..., \hat{z}_{k|k}\} = \arg \min_{\{z_{k-N},...,z_k\}} \Phi(z_{k-N}) + \frac{1}{2} \sum_{l=k-N}^{l=k} \rho_l^{ME}(v_l) + \frac{1}{2} \sum_{l=k-N}^{l=k-1} w_l^T Q_l^{-1} w_l$$
$$\text{s.t.} \quad z_{l+1} = f(z_l) + w_l \tag{2.12}$$
$$y_l = h(z_l) + v_l$$
$$z^{LB} \leq z_l \leq z^{UB}$$

One danger of using robust M-Estimators is the possibilty of type-1 errors, i.e. incorrectly identifying good measurements as gross errors. While there is no way to completely eliminate type-1 errors there is a trade-off that must be considered between the influence put on large residuals and the number of type-1 errors. This trade-off must be taken into account when choosing the tuning parameters for the robust estimators in order to achieve good state estimation. Another concern with robust M-Estimators in NLP (2.12) is the nonconvexity of the estimator. Even with nonlinear plant models, the likelihood of obtaining local solutions (and type-1 errors) is higher with nonconvex functions. This is especially the case with the Redescending estimator, in contrast to the convex Fair Function.

2.4 Concluding Remarks

In this chapter we introduce the state estimation problem and describe two convenient ways of formulating it as a dynamic optimization problem. Both MHE and asMHE have the advantage of directly incorporating nonlinear process models and bounds on process variables. Additionally, asMHE drastically reduces the on-line computational cost of obtaining state estimates.

Another significant challenge for obtaining good state estimates is dealing with large measurement errors. We propose using robust M-Estimators to overcome this challenge and illustrate how they can be incorporated in the MHE and asMHE formulations. In the next chapter we demonstrate the performance of these state estimators.

Chapter 3

State Estimation Case Studies

In this chapter we demonstrate the state estimators described in the previous chapter on two chemical engineering process models. We begin by describing the implementation of these methods and then compare the performance of various state estimators on dynamic models of a CSTR and a distillation column. This chapter is based on work published in [7].

3.1 Implementation

The dynamic state space models used in the simulations in this chapter are systems of differential algebraic equations (DAE). One way of optimizing DAE systems, known as the simultaneous approach, is to discretize the entire model with respect to time and approximate the differential equations using piecewise polynomials. The resulting problem is an NLP with only algebraic constraints. In this work we make use of Radau collocation to discretize the differential equations in our models [6].

Numerical simulations were implemented using MATLAB [55] and AMPL [56] as modeling platforms and the NLP was solved using IPOPT [47]. IPOPT is an open source NLP solver for solving large problems which utilizes a Newton-based interior point algorithm. To compute the NLP sensitivity and MHE arrival cost information, an extension to the IPOPT solver called sIPOPT [57] was used. Moreover, for the update of the arrival cost we use the sIPOPT-based procedure from [46]. This arrival cost update is well-suited for multirate sampling and missing data, including gross errors eliminated by M-estimators.

Finally, when implementing the Redescending M-estimator in the objective function of

(2.12), we used a smoothed approximation of the piecewise function (2.10) as discussed in [54]. We do this to ensure continuous first and second derivatives, which is a property required by our solver.

3.2 CSTR Case Study

To illustrate and compare the state estimators described in this chapter we first consider a non-isothermal continuously stirred tank reactor (CSTR) with 3 states. We introduce gross errors into these simulations by assuming that our measurements drift away from their true values. We consider 6 state estimators in this chapter that compare the use of MHE and asMHE with the 3 M-estimators described by (2.6), (2.8), and (2.10).

The states are plotted for each case study. However, in order to compare the estimators quantitatively we also calculate the normalized Sum of Squared Errors (SSE). For each estimator this is evaluated as follows

$$SSE = \sum_{l \in T} \sum_{j \in S} (\hat{z}_{j,l} - z_{j,l})^2 \tag{3.1}$$

where $\hat{z}_{j,l}$ is the j-th element of the normalized state estimate vector at time l, $z_{j,l}$ is the j-th element of the normalized true state vector at time l, T is the set of sampling times, and S is the set of state variables. The state vectors are normalized to values between 0 and 1 using their upper and lower bounds.

The dynamic CSTR model used in this case study describes the exothermic reaction between sodium thiosulfate (component A) and hydrogen peroxide (component B). The model equations are given in Appendix A.1. There are 3 dynamic states in this model, the concentration of A C_A, the reactor temperature T_R, and the cooling water temperature T_{cw}.

In our simulations we assume that we measure T_R and T_{cw} and estimate all three states. The covariance values for the measurement noise and the unknown disturbances are given in Table 3.1. As mentioned in the implementation section, the model is discretized using orthogonal collocation with 5 finite elements and 3 collocation points. The number of finite

	Measurement Noise Variance	State Disturbance Variance
C_A	-	10^{-8}
T_R	0.0625	10^{-4}
T_{cw}	0.0625	10^{-4}

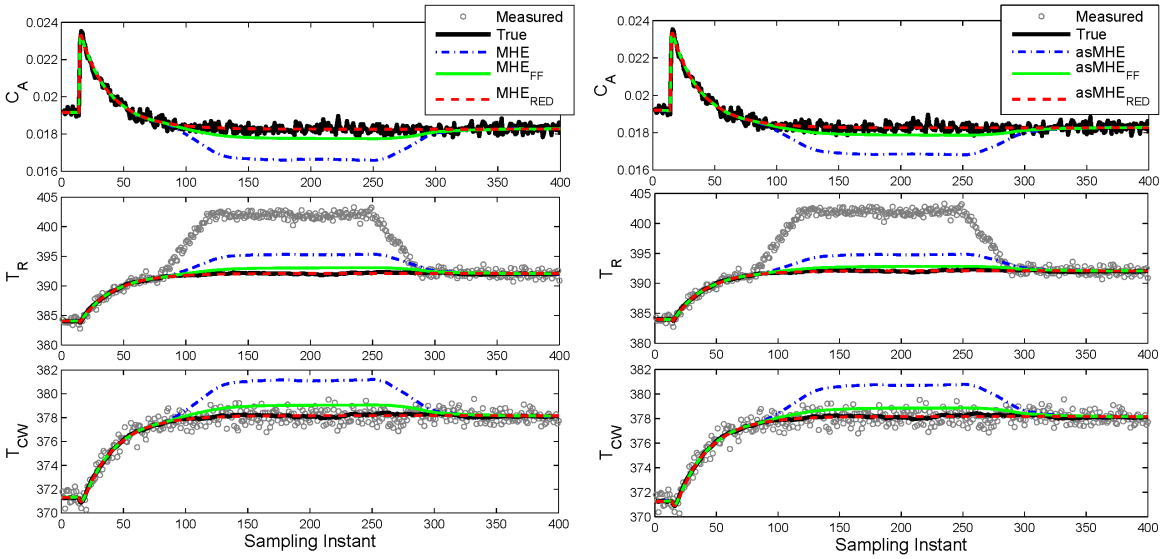
Table 3.1: Variance values for noise in CSTR example

elements is also the horizon length for the estimators. This discretization scheme results in an NLP (2.12) with 184 variables and 169 constraints being solved at each time step. The simulations are run for 400 time steps and each time step is 1.2 seconds. In order to isolate the effect the M-estimators have on the state estimates we consider an ideal scenario where there is no plant-model mismatch, i.e. we know the state-space model, the model parameters, and the noise variances exactly. The same tuning parameters are used for all case studies, for the Fair Function $C = 20$ and for the Redescending estimator $a = 1$, $b = 2$, $c = 6$.

3.2.1 Results Case 1: Single Drift

In this first case we introduce gross errors into our simulation by having one of our temperature measurements drift away from the true state. This temperature drift is introduced in the reactor temperature measurement, T_R. The maximum error is 10 degrees above the true measurement and we increase the gross error by 0.25 degrees per sample time. The drift starts at time step 80, stays at the maximum error level for 130 time steps, and then decreases to zero at a rate of 0.25 degrees per sample time.

The simulation results are shown in Figure 3.1 and Table 3.2. It is clear that the formulations with least squares have the worst performance, as the gross error in T_R causes significant contamination in all three state estimates. The formulations with the Fair Function perform much better than least squares while the Redescending formulations give the best state estimates.



(a) MHE

(b) asMHE

Figure 3.1: Comparison of the MHE and asMHE formulations with and without M-Estimators for the CSTR case 1 simulation. MHE: MHE with least squares, MHE_{FF}: MHE with Fair Function, MHE_{RED}: MHE with Redescending, similarly for asMHE

In addition, we note that the results for MHE and asMHE are nearly identical. This is significant because, as mentioned previously, asMHE relies on NLP sensitivity to approximate the optimal state estimates. As the approximations are very good, the advantage of the asMHE formulation is that most of the computational expense is off-line between sampling times, and state estimates are generated much faster. The average CPU time required to solve the full NLP (2.12) was 0.0108 seconds. The CPU time for completing the on-line sensitivity update for the asMHE formulations is too small to measure in this example. This difference in on-line computational time between MHE and asMHE is not meaningful for such a small scale example. However, for the larger example in Section 3.3 the CPU time reduction is essential for the application.

	MHE	asMHE
Least Squares	0.0597	0.0435
Fair Function	0.0053	0.0031
Redescending	1.712×10^{-4}	1.735×10^{-4}

Table 3.2: Normalized Sum of Squared Errors for CSTR case 1 simulation

3.2.2 Results Case 2: Multiple Drifts

In this case we introduce temperature drift errors into both temperature measurements. The drift in T_R is the same as in case 1. The drift in the cooling water temperature T_{cw} begins at time step 150, increases at a rate of 0.4 degrees per time step, reaches a maximum error of 7.2 degrees, and then decreases to zero at the same rate starting at sample time 300.

The results for this case are shown in Figure 3.2 and Table 3.3. Again we see that the formulations with least squares perform the worst and the ones with the Fair Function do significantly better. Here the interesting finding is the behavior of the nonconvex Redescending MHE formulation. When the MHE formulation with Redescending is run using the same initial conditions as the other cases, it converges to an inferior local solution. This problem of inferior local solutions and the proposed initialization strategy for the Redescending estimator has previously been noted in [42] and [54]. To address this, we initialized the MHE Redescending formulation with the MHE Fair Function solution for each horizon. With this initialization the Redescending MHE formulation solves very quickly and behaves very well, as in the case 1 simulation. The results of this run are shown in the Figure 3.2a. While we were able to get good state estimates with the MHE Redescending estimator using a better intialization, we note that in order to implement this strategy the MHE Fair Function and MHE Redescending formulations have to be solved in series at every time step. This increases the computational time required to get the state estimates.

	MHE	asMHE
Least Squares	0.1157	0.0802
Fair Function	0.0150	0.0079
Redescending	1.641×10^{-4}	1.677×10^{-4}

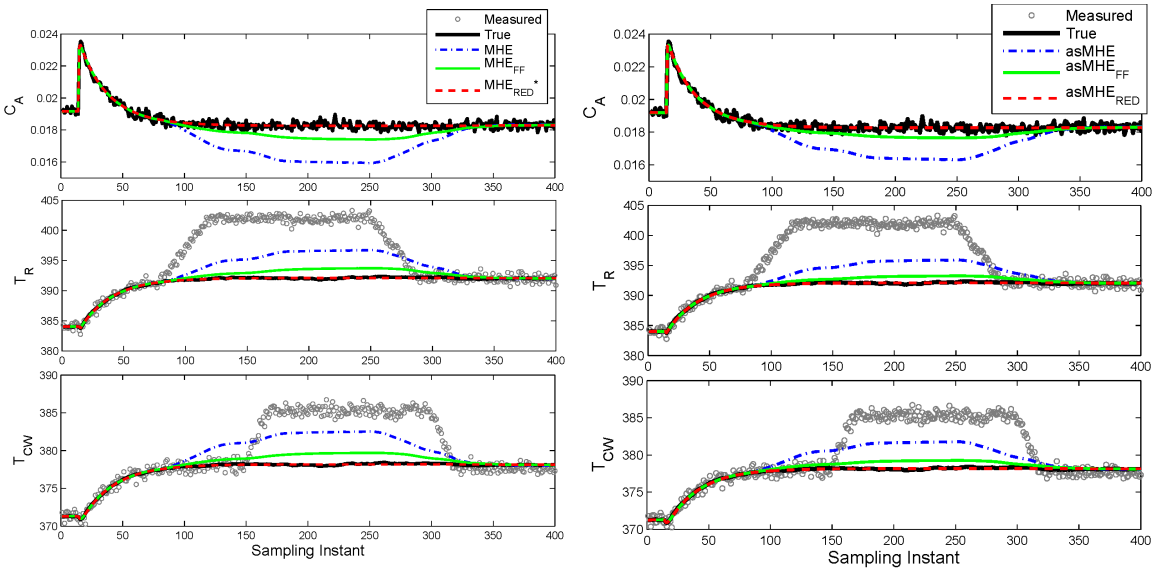
Table 3.3: Normalized Sum of Squared Errors for CSTR case 2 simulation

Finally, we note that these inferior local solutions were encountered only in the Redescending MHE formulation and not in the Redescending asMHE formulation. This is due to asMHE having a better initialization, which comes directly from the model prediction, and not the actual measurement. In the case of gross errors, the MHE initialization is severely biased while the asMHE initialization is not. Again, we have assumed no plant-model mismatch; as long as our previous state estimate is close to the true state, our predicted measurement is going to be close as well. Moreover, it turns out that the sensitivity update does not change the estimate much, which implies that the estimate is not very sensitive to the new (outlier) measurement and depends more on the older measurements in the horizon. Also, we believe that this improved initialization explains why asMHE performs consistently better than MHE; as observed in the SSE values presented for each simulation.

3.2.3 Results Case 3: Varying Horizon Length

In the first two cases a time horizon of 5 was used for all simulations. In this case we tried longer time horizons of 10 and 20 time steps along with the multiple drift errors of the previous case. The results are shown in Figure 3.3 and Table 3.4. Only plots of concentration are included because the trends for the other states were the same.

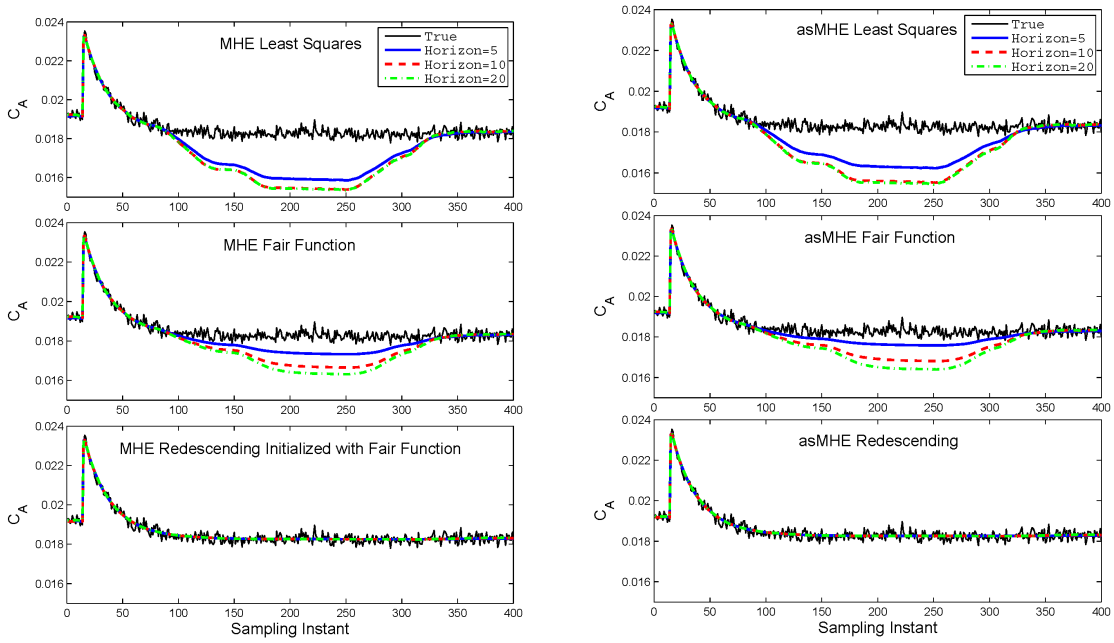
For state estimation it is expected that longer horizons will result in better state estimates because they put less weight on any single measurement. However, with gross errors and longer horizons we found that ithe least squares and Fair Function formulations lead to



(a) MHE

(b) asMHE

Figure 3.2: Comparison of the MHE and asMHE formulations with and without M-Estimators for the CSTR case 2 simulation. $MHE_{RED}*$: MHE with Redescending and initialized with Fair Function solution



(a) MHE

(b) asMHE

Figure 3.3: Comparison of the concentration state estimates for the CSTR case 3 simulation using formulations with horizon lengths of 5, 10, and 20

worse state estimates when drifting errors are introduced to the system. This difference is most notable in the plots of the Fair Function for MHE and asMHE, where the estimates approach their true values as the horizon size decreases. This trend also appears for the least squares formulations where the state estimates approach a plateau with increasing horizon size. The trend with longer horizons can be explained because with long lasting drifting errors we end up with many biased measurements in the time series. By increasing N we include more gross errors in the horizon and they have more effect than with smaller N. On the other hand, horizon length does not affect the Redescending cases, because the effect of biased measurements is largely eliminated.

	MHE			asMHE		
Horizon Length	LS	FF	RED	LS	FF	RED
5	0.1157	0.0150	1.6412×10^{-4}	0.0802	0.0079	1.6769×10^{-4}
10	0.1718	0.0452	1.5993×10^{-4}	0.1539	0.0368	1.6040×10^{-4}
20	0.1751	0.0667	1.6319×10^{-4}	0.1630	0.0610	1.6476×10^{-4}

Table 3.4: Normalized Sum of Squared Errors for CSTR case 3 simulation. LS: Least Squares, FF: Fair Function, RED: Redescending

3.3 Distillation Case Study

Now that we know how these state estimators perform on a small example we consider a detailed dynamic model of a binary distillation column [58]. Again we introduce gross drifting errors as with the CSTR example. We also consider gross errors as step changes to simulate complete sensor failure.

3.3.1 Distillation Model

The distillation column modeled in this example separates methanol and n-propanol. The model originates from [58] and consists of an index-2 system of DAEs. The distillation column model from [58] was later reformulated in [59] to index 1. The model equations are shown in Appendix A.2. We consider a column with $N_T = 40$ trays along with a total condenser and a reboiler. The feed stream enters the column at tray $i = 21$.

Assumptions in this model include an ideal vapor phase, constant pressure drop across the trays, and negligible vapor molar holdups. Raoult's law is used to model phase equilibrium with non-ideal behavior addressed using tray efficiencies. Francis' weir equation is used to model the liquid flow rate between trays. In our simulations temperature, T_i, and the liquid volume holdup, n_i^v, on each tray are the measurements and we estimate the methanol composition in the liquid phase, x_i as well as the liquid molar holdup, M_i, on

	Measurement Noise Variance	State Disturbance Variance
T_i	6.25×10^{-2}	-
n_i^v	10^{-8}	-
x_i	-	10^{-5}
M_0	-	10
M_i	-	1
M_{N_T+1}	-	5

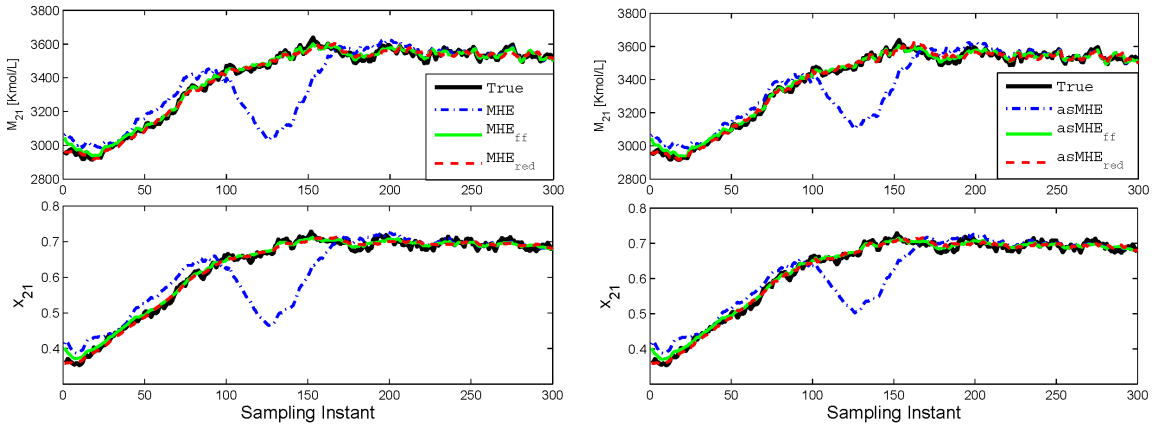
Table 3.5: Variance values for noise in Distillation example

each tray.

For our state estimation formulations we use a horizon length of 10 measurements and we discretize the model using orthogonal collocation with 3 collocation points. NLP (2.12) now has 21,642 variables and 20,718 constraints. The simulations are run for 300 time steps and each sampling time is 60 seconds. Again we assume that there is no plant-model mismatch in order to isolate the effects of the gross errors on each formulation. The covariance values for the measurement noise and unknown disturbances are given in Table 3.5. The M-Estimator tuning parameters are as follows for all cases: Fair Function ($C = 6$), Redescending ($a = 1$, $b = 2$, $c = 4$).

3.3.2 Results Case 1: Drift Errors

In this first case we simulate drifts in the temperature measurements on 10 of the trays. The drifts were generated by randomly selecting the tray on which the drift would occur, the start time and duration of the drift, the maximum magnitude of the drift, and the rate at which the drift reached that maximum. Additionally, they were formulated so that the drifts would disappear before the end of the simulation so that the state estimators could recover once the gross errors were removed. The results are shown in Figure 3.4 and Table 3.6.



(a) MHE

(b) asMHE

Figure 3.4: Comparison of the MHE and asMHE formulations with and without M-Estimators for the Distillation case 1 simulation. MHE: MHE with least squares, MHE_{FF}: MHE with Fair Function, MHE_{RED}: MHE with Redescending, similarly for asMHE. M_{21} is the molar holdup on tray 21, x_{21} is the methanol composition on tray 21

Notice that significant shifts can be seen in the estimates of both the MHE and asMHE formulations without M-estimators. Both the Fair Function and Redescending formulations are able to mitigate the affects of the gross error. Again we see that the MHE and asMHE estimates are very similar but the average on-line time required for the asMHE formulation is only 0.022% of the time required to solve the MHE problem. The MHE formulation with least squares took on average 45.48 CPU seconds to solve. The on-line component of the asMHE formulation with least squares took on average 0.01 CPU seconds. Thus, using asMHE along with the robust M-estimators we are able to generate fast and accurate state estimates.

3.3.3 Results Case 2: Step Errors

In this case we simulate gross errors generated by sudden step changes in the measurements. We also introduce even more errors into our measurements in an effort to further

	MHE	asMHE
Least Squares	23.0092	16.2973
Fair Function	1.0122	0.9680
Redescending	0.9191	0.7955

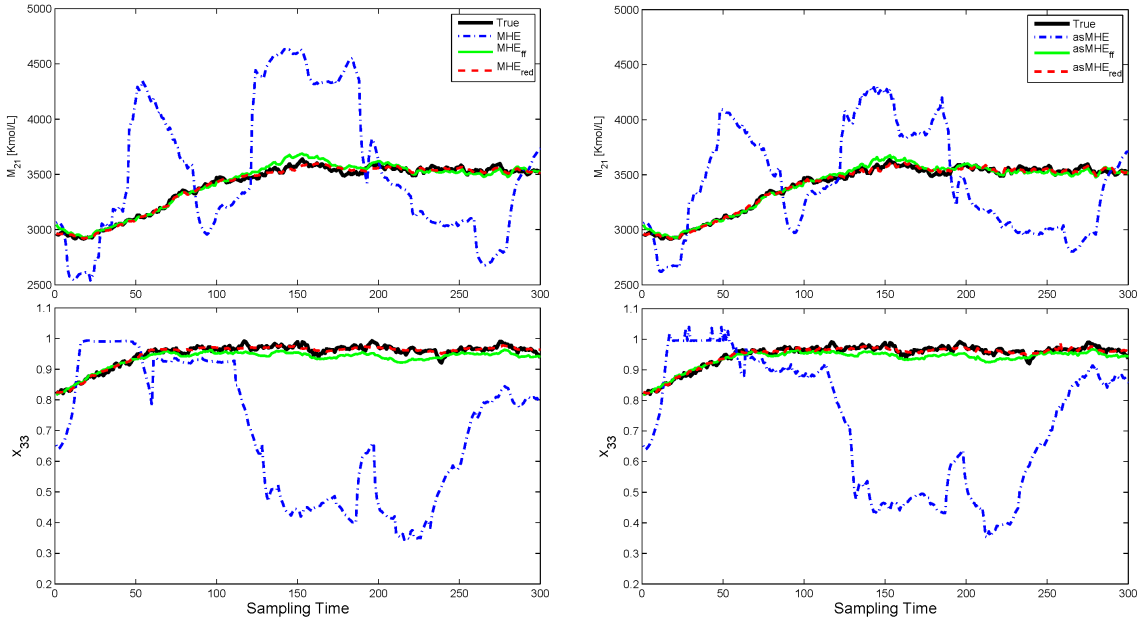
Table 3.6: Normalized Sum of Squared Errors for Distillation case 1 simulation

	MHE	asMHE
Least Squares	864.4267	594.6171
Fair Function	2.2703	1.8590
Redescending	0.9618	0.8180

Table 3.7: Normalized Sum of Squared Errors for Distillation case 2 simulation

challenge the formulations. We introduce step errors into the temperature measurements on 20 of the trays as well as the volumetric holdup measurements on 20 of the trays. The step errors were generated randomly using a similar approach as that described in the previous case. Additionally, we no longer assume that the gross errors disappear before the end of the simulations. This results in a case where about 15% of all measurements contain gross errors. The results are shown in Figure 3.5 and Table 3.7.

We see that while both the MHE and asMHE formulations with the least squares function are unable to track the true state, the Redescending formulations are still able to predict accurate state estimates. An interesting feature of the asMHE least squares results for the methanol composition is the appearance of tiny jumps in the estimate, which slightly violate the upper bound on the state. This is most likely caused by a change in the active set of constraints for NLP (2.12) when the predicted measurements are replaced with the actual plant measurements containing gross errors in the asMHE algorithm.



(a) MHE

(b) asMHE

Figure 3.5: Comparison of the MHE and asMHE formulations with and without M-Estimators for the Distillation case 2 simulation. M_{21} is the molar holdup on tray 21, x_{33} is the methanol composition on tray 33

3.4 Concluding Remarks

In these case studies we found that the Redescending estimator formulations are able to follow the true state of the system even when a significant portion of the measurements were contaminated with gross errors. However with too many gross errors the MHE formulation with the Redescending estimator may get stuck at local solutions due to its non-convexity. This problem could be overcome by first solving the NLP with the Fair Function estimator, a convex objective. The asMHE formulation with the Redescending estimator does not encounter the problem with the non-convexity and is much faster than the MHE version. Overall, this chapter has shown that using robust M-Estimators can be an effective strategy for state estimation when our measurements are contaminated with gross errors. Additionally, robust M-Estimators can easily be incorporated into on-line state estimation formulations such as asMHE resulting in fast and robust estimators.

Part II

Modeling Tools

Chapter 4

A Modeling and Direct Transcription Framework

In the second part of this thesis we focus on developing modeling tools for dynamic optimization problems with the goal of simplifying the typical implementation for advanced dynamic optimization techniques, such as state estimation, and making dynamic optimization more accessible to non-expert users.

4.1 Introduction and Motivation

There are many modeling frameworks available to users interested in incorporating differential equations in optimization formulations. An alternative is to apply direct transcription and solve the resulting finite dimensional representation in an algebraic modeling language. Algebraic modeling languages typically allow the user to represent optimization problems using a concise and natural syntax, perform model checking and automatic differentiation, and provide interfaces for communicating with optimization solvers. Well-known algebraic modeling languages include GAMS [60] and AMPL [56]. A limitation of these tools is that they define their own proprietary syntax for representing optimization problems and are not open-source. Consequently, they have limited extensibility.

Modeling tools that are embedded in high-level programming languages such as Python, Matlab, and Julia provide more flexibility to incorporate new syntax and model processing capabilities and to re-use legacy models. Examples of such tools include FlopC++ [61], PuLP [62], Pyomo [63], and more recently JuMP [64]. While these languages allow the user

Tool	Types	Non-Canonical Forms	Open Source	Open Language	Solution Methods
ACADO [65]	ODE, DAE		X	C++	Shooting Methods, Collocation
APMonitor [66]	ODE, DAE	X			Collocation
GPOPSII [67]	ODE, DAE				Collocation
gPROMS [68]	ODE, DAE, PDAE	X			Shooting Methods, Collocation
Optimica and JModelica.org [69]	ODE, DAE		X		Shooting, Collocation
SOCS [70] and SOS [71]	ODE,DAE			Fortran	Finite Difference, Collocation
TACO [72]	ODE, DAE		X		Shooting
Tomlab PROPT [73]	ODE, DAE	X			Collocation
`pyomo.dae`	ODE, DAE, PDAE	X	X	Python	Finite Difference, Collocation

Table 4.1: Existing tools for handling optimization problems with differential equations.

to formulate problems in a high level programming language, they are currently restricted in the classes of optimization problems they can represent. Specifically, only a small subset of these tools provide syntax extensibility and processing (discretization) capabilities for differential equations. An overview of such tools and their capabilities is provided in Table 4.1. The last row compares the new package proposed in this work, `pyomo.dae`, with several existing software tools. The table summarizes the types of differential equations each tool can handle. The table also indicates whether the tool is open-source and whether the tool represents models using a high-level programming language (as opposed to a proprietary syntax). The last column in the table lists the solution methods each tool provides to the user.

Most of the existing tools that can handle differential equations require the user to formulate their model using a proprietary language. Notable exceptions are ACADO and SOCS/SOS which can be used directly from C++ and Fortran, respectively. ACADO is open source and provides both shooting and direct collocation methods for solving the problems. However, ACADO is not able to handle large scale problems and cannot handle optimal control problems in non-canonical forms. While these tools provide tailored solution algorithms for these problems they are unable to represent models which deviate even slightly from some required structure. APMonitor offers some additional flexibility over ACADO in handling differential equations in non-canonical forms and high-index

DAE systems and it is free to use via a web-server but it is not open-source. Moreover, APMonitor also uses proprietary syntax.

Of the tools surveyed here, gPROMS is the only one that provides complete flexibility in the types of differential equations that can be modeled, including support for PDAEs and non-canonical forms. gPROMS, however, is a commercial software. Tools such as Optimica, TACO, and Tomlab's PROPT are extensions for existing modeling languages in Modelica, AMPL, and Matlab; respectively. None of these tools include support for differential equations is non-canonical forms and are extensions of proprietary syntax.

In contrast to the tools described above, `pyomo.dae` seeks to be a flexible and extensible tool for representing systems of ODEs, DAEs, and PDAEs on certain domains and automatically applying numerical discretization techniques for converting differential equations to algebraic equations. Using `pyomo.dae` a user may represent ordinary and partial differential equations of arbitrary order, of arbitrary dimension, and in any form. This includes complex mixed partial derivatives. `pyomo.dae` is also capable of solving optimal control problems in non-canonical forms.

`pyomo.dae` has been developed as an extension to the algebraic modeling language Pyomo, which is based on the high-level programming language Python. In Section 4.1.1 we will give a brief overview of Pyomo and then in Section 4.2 we introduce the key modeling constructs in `pyomo.dae` which lead to a new level of dynamic optimization model abstraction. Section 4.3 reviews the automatic discretization schemes that have been implemented to transform dynamic models to algebraic approximations. Finally, we discuss how `pyomo.dae` can be extended to custom discretization schemes in Section 4.4.

4.1.1 Pyomo

Pyomo is an open-source algebraic modeling language written in Python [63, 74]. Since its introduction, Pyomo has undergone major restructuring and extensions resulting in a stable and flexible modeling language. An overview of the current features of Pyomo is

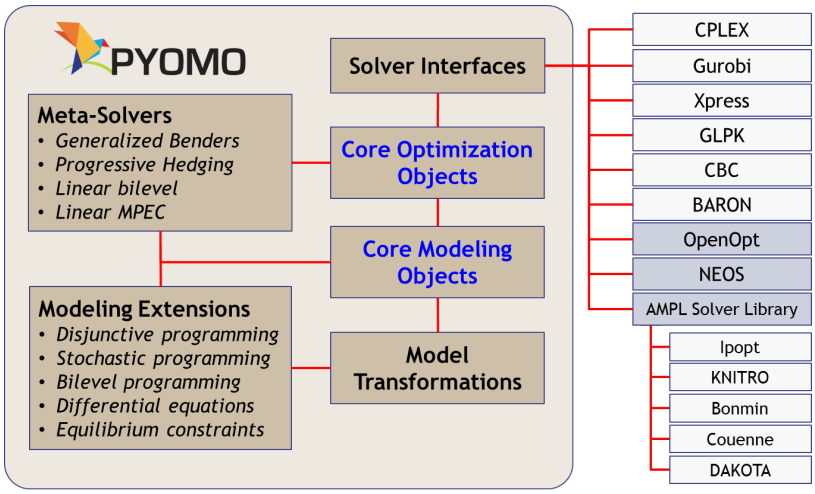


Figure 4.1: Summary of Pyomo features

shown in Figure 4.1. Pyomo supports a wide range of problem types including Linear Programming (LP), Mixed-Integer Programming (MIP), Nonlinear Programming (NLP), and Mixed-Integer Nonlinear Programming (MINLP). Pyomo also includes interfaces for a variety of optimization solvers and provides automatic differentiation (AD) for NLP problems using the open source AMPL Solver Library (ASL).

Pyomo's main advantage over other algebraic modeling languages is that it is written in a high level programming language. This means that a user does not have to learn a specialized modeling language in order to formulate and solve optimization problems; a basic understanding of Python is all that is required. Models are represented using Python objects and can be formulated and manipulated in sophisticated ways using simple scripts. Furthermore, Pyomo users have access to a large collection of other Python packages which include tools for plotting, numerical and statistical analysis, and input/output. This opens the door for the development of novel algorithms, complicated model formulations, and general model transformations. All of these features make Pyomo a promising platform for implementing extensions for higher problem classes such as dynamic optimization.

4.2 Modeling Components in pyomo.dae

One of the main constructs of any standard algebraic modeling language is the notion of an indexing set. This is what allows a user to quickly and compactly specify a set of related variables or a set of constraints of the same form. In standard algebraic modeling languages, however, indexing sets are assumed to be discrete sets. This makes it challenging to represent optimization problems with continuous domains and dynamics without first converting the model to a discretized approximation manually. Notably, pyomo.dae is able to represent continuous domains and arbitrary derivatives by introducing two new modeling components into Pyomo. In this section we will introduce these components and give a brief overview of how they can be used to express rich sets of models with differential equations. More details about using these components can be found in the online documentation at http://www.pyomo.org.

4.2.1 ContinuousSet

Analogous to the notion of a discrete set already available in modeling languages, we introduce the notion of a *continuous set* which is a key construct that enables high-level model abstractions. The ContinuousSet construct allows users to represent continuous bounded domains. Such domains can be interpreted as 'spatial' or 'time' domains but such interpretations are not necessary. In other words, a continuous domain can represent any domain of a function (e.g., a parameter domain). We highlight that we separate the notation of the model syntax from that of a model solution. In other words, the abstraction seeks to provide syntax and not to prescribe a solution method. In designing the syntax, however, we have made sure to support communication of relevant information down to solution methods.

The ContinuousSet is similar in structure to that of Pyomo's Set component and can be used to index variables and constraints. Variables and constraints can be defined over

an arbitrary number of continuous domains. For instance, a variable of the form,

$$w(\ell, x, t),\ \ell \in \mathcal{L} := \{0, 1, ..., L\},\ t \in \mathcal{T} := [0, T],\ x \in \mathcal{X} := [0, X], \tag{4.1}$$

defines a variable with indices ℓ spanning the discrete set $\mathcal{L}$, where each variable $w(\ell, \cdot, \cdot)$ is defined over the continuous rectangular domain $\mathcal{T} \times \mathcal{X} = [0, T] \times [0, X]$. By default, a `ContinuousSet` will be treated as a closed set. The code to declare variable (4.1) using `pyomo.dae` is shown below (assuming L, T, and X are previously defined constants).

Listing 4.1: Declaring a variable over discrete and continuous domains

```
1 m = ConcreteModel()
2 m.l = RangeSet(0,L)
3 m.x = ContinuousSet(bounds=(0,X))
4 m.t = ContinuousSet(bounds=(0,T))
5 m.w = Var(m.l,m.x,m.t)
```

A `ContinuousSet` must be initialized with two numeric values representing the upper and lower bounds of the continuous domain. A user may also specify additional points in the domain that will be communicated to the solver. Such data can be used, for instance, to ensure that a discretization mesh matches a subset of points, which in turn might be needed to specify path constraints or match points with experimental data. The following code snippet provides an example of how to do this:

Listing 4.2: Declaring continuous domains using `ContinuousSet` components

```
1 # declare by providing bounds
2 model.t = ContinuousSet(bounds=(0,5))
3 # declare specific points in domain
4 model.x = ContinuousSet(initialize=[0,1,2.5,3.3,5])
```

A user may also declare a `ContinuousSet` on an abstract Pyomo model and initialize it using a separate data file. In fact, most valid ways to declare and initialize a Pyomo `Set` can be used to declare and initialize a `ContinuousSet`.

When a model is discretized manually, the indexing set for a continuous variable is typically an arbitrary set of integers and the user is responsible for converting each index to the desired time or length scale as a separate step. This is one of the most common places where modeling errors occur, particularly when a more complex discretization scheme is used such as collocation on finite elements. It can be particularly challenging to do this in the case of unequally spaced finite elements. Furthermore, this is a necessary step for plotting and analyzing the results. One of the main design strengths of the `ContinuousSet` in `pyomo.dae` is that this step has been eliminated and domain scaling is done directly in the set. In other words, every discrete point in a `ContinuousSet` corresponds to an actual point in the bounded continuous domain being represented.

The `ContinuousSet` can also be used to receive information from the solution layer. For instance, if the set is discretized using a collocation method over finite elements, one can obtain a list of the discretization points used so that the user can interrogate and manipulate a `ContinuousSet` component and implement customized initializations or constraints over a `ContinuousSet`.

The notion of a continuous set provides high flexibility to the user to impose constraints and objectives in non-standard forms. In particular, we do not impose restrictions on the number of independent dimensions of a variable and constraint. For instance, we enable the specification of a variable of the form,

$$w(\ell, x, y, z, t),\ \ell \in \mathcal{L},\ t \in \mathcal{T},\ x \in \mathcal{X},\ y \in \mathcal{Y},\ z \in \mathcal{Z}. \tag{4.2}$$

Analogous to the definition of a subset of a discrete set $\bar{\mathcal{L}} \subset \mathcal{L}$ is the definition of a subset of a continuous set $\bar{\mathcal{T}} \subset \mathcal{T}$. Here, the subset $\bar{\mathcal{T}}$ can be both discrete (with elements in $[0, T]$) or continuous. Constraints involving variables over a continuous set must be defined over a subset of a continuous set. For instance, if we define $\mathcal{X} = [0, 2], \mathcal{T} = [1, 10]$, all of the

following constraints are valid:

$$h(w(\ell, x, t)) = 0,\ x \in [0, 1],\ t \in [5, 10]$$
$$h(w(\ell, x, t)) = 0,\ x \in \{0, 0.1, 0.5, 1\},\ t \in \mathcal{T}$$
$$h(w(\ell, x, t)) = 0,\ x \in \{0, 0.1, 0.5, 1\},\ t \in \{2, 3, 4\}.$$

The current implementation of a continuous set in `pyomo.dae` does not allow a `ContinuousSet` component to be defined as a subset of another `ContinuousSet`. This extension is on-going work. Constraints over continuous subsets are currently implemented using constraint skipping, as follows:

Listing 4.3: Declaring constraints over a subset of a continuous set

```
m.xsubset = Set(initialize=[0,0.1,0.5,1])
m.tsubset = Set(initialize=[2,3,4])
m.x = ContinuousSet(bounds=(0,2),initialize=m.xsubset)
m.t = ContinuousSet(bounds=(1,10),initialize=m.tsubset)

def _con1(m,lidx,xidx,tidx):
        if xidx <= 1 and tidx >= 5:
                return m.h[lidx,xidx,tidx] == 0
        else:
                return Constraint.Skip
m.con1 = Constraint(m.l,m.x,m.t,rule=_con1)

def _con2(m,lidx,xidx,tidx):
        return m.h[lidx,xidx,tidx] == 0
m.con2 = Constraint(m.l,m.xsubset,m.t,rule=_con2)

def _con3(m,lidx,xidx,tidx):
        return m.h[lidx,xidx,tidx] == 0
m.con3 = Constraint(m.l,m.xsubset,m.tsubset,rule=_con3)
```

The ability to impose constraints at discrete points allows the user to impose path and

point constraints. For instance, we can consider:

$$h(w_1(x,0)) = f(w_1(x,T)), \quad x \in \mathcal{X}$$
$$h(w_1(x,T)) = f(w_1(x,T-\delta)), \quad x \in \mathcal{X}$$
$$h(w_1(X,0)) = f(w_1(X/2,T/2)).$$

where $h(\cdot)$ and $f(\cdot)$ are arbitrary functions. This construct allows imposition of multi-point boundary conditions which include periodicity and delay constraints. Recirculation (cyclic) constraints are typical in chemical reactors, periodicity constraints are typical in model predictive control, and delay constraints are typical in signal and transportation networks. The constraints expressed above can be implemented as follows:

Listing 4.4: Declaring multi-point boundary conditions

```
1  m.x = ContinuousSet(bounds=(0,X),initialize=[X/2])
2  m.t = ContinuousSet(bounds=(0,T),initialize=[T/2,T-delta])
3
4  def _con1(m,xidx):
5          return m.h[xidx,0] == m.f[xidx,T]
6  m.con1 = Constraint(m.x,rule=_con1)
7  def _con2(m,xidx):
8          return m.h[xidx,T] == m.f[xidx,T-delta]
9  m.con2 = Constraint(m.xsubset,rule=_con2)
10 def _con3(m):
11         return m.h[X,0] == m.f[X/2,T/2]
12 m.con3 = Constraint(rule=_con3)
```

We also allow the user to impose constraints between variables and across domains. For instance, the following constraint is valid:

$$h(w(\ell,x,T)) = f(w(\ell+1,x,0)), \quad x \in \mathcal{X}.$$

Listing 4.5: Declaring constraints across domains

```
m.l = RangeSet(1,10)
m.x = ContinuousSet(bounds=(0,X))
m.t = ContinuousSet(bounds=(0,T))

def _con(m,lidx,xidx):
        if lidx == 10:
                return Constraint.Skip
        else:
                return m.h[lidx,xidx,T] == m.f[lidx+1,xidx,0]
m.con = Constraint(m.l,m.x,rule=_con)
```

This feature enables the user to couple physical elements described by different sets of differential equations. For instance, multiple domains arise in lithium-ion batteries, fuel cells, buildings, and gas networks. Coupling across domains is also common in multi-stage optimal control where the horizon is lifted into stages.

Combined with the modular design of Pyomo, `pyomo.dae` leads to a higher level of model abstraction which allows for an intuitive structure for formulating and linking dynamic models over several domains. However, it should be noted that we do not make any claims about being able to *solve* arbitrary models over continuous domains. Our tool simply allows the modeler to formulate their problem in a more straightforward manner.

4.2.2 `DerivativeVar`

The `DerivativeVar` component is used to declare a derivative of a Pyomo `Var`. A variable may only be differentiated with respect to a `ContinuousSet` that it is indexed by and the indexing sets of a `DerivativeVar` will be identical to those of the `Var` it is differentiating. We also allow for derivatives of arbitrary order to be expressed. In the case of

variable $w(\ell, x, t)$ defined in (4.1) the derivatives

$$\frac{\partial^\alpha w(\ell, x, t)}{\partial t^\alpha} \tag{4.3a}$$

$$\frac{\partial^\beta w(\ell, x, t)}{\partial x^\beta} \tag{4.3b}$$

$$\frac{\partial^\alpha \partial^\beta w(\ell, x, t)}{\partial t^\alpha \partial x^\beta}, \tag{4.3c}$$

can be imposed for $\alpha, \beta = 1, 2, ...$ The following snippet illustrates the declaration of a first, second-order, and mixed derivative. The variable being differentiated is supplied as the only positional argument and the type of derivative is specified using the 'wrt' keyword argument (or the more verbose 'withrespectto').

Listing 4.6: Declaring derivatives

```
1 m.dwdt = DerivativeVar(m.w, wrt=m.t)
2 m.dwdx2 = DerivativeVar(m.w, wrt=(m.x,m.x))
3 m.dwdxt = DerivativeVar(m.w, wrt=(m.x,m.t))
```

A variable defined over a continuous set is assumed (from a modeling standpoint) to be sufficiently smooth and therefore its derivatives exist. It is the responsibility of the user to express a model fulfilling these needs. Again, the syntax is agnostic to the solution method used (and of its requirements).

Derivatives can be expressed in the body of any constraint. This is an important feature because we do not impose any predefined structure to differential models. This enables the expression of DAEs with complex mass matrices or complex boundary conditions. For instance, the following constraints are valid expressions,

$$\frac{\partial w(\ell, x, t)}{\partial x} = f(w(\ell, x, t)),\ \ell \in \mathcal{L},\ x \in \mathcal{X},\ t \in \mathcal{T}$$

$$0 = f\left(\frac{\partial w(\ell, x, t)}{\partial x}, w(\ell, x, t)\right),\ \ell \in \mathcal{L},\ x \in \mathcal{X},\ t \in \mathcal{T}.$$

The code implementing such constraints has the form:

Listing 4.7: Declaring constraints with derivatives

```
def _con1(m,lidx,xidx,tidx):
        return m.dwdx[lidx,xidx,tidx] == m.f[lidx,xidx,tidx]
m.con1 = Constraint(m.l,m.x,m.t,rule=_con1)

def _con2(m,lidx,xidx,tidx):
        return 0 == m.dwdx[lidx,xidx,tidx] * m.f[lidx,xidx,tidx]
m.con2 = Constraint(m.l,m.x,m.t,rule=_con2)
```

The code snippet below provides a more explicit description of how to express `DerivativeVar` components.

Listing 4.8: Declaring derivatives using `DerivativeVar` components

```
model = ConcreteModel()
model.s = Set(initialize=['a','b'])
model.t = ContinuousSet(bounds=(0,5))
model.l = ContinuousSet(bounds=(-10,10))
model.x = Var(model.t)
model.y = Var(model.s,model.t)
model.z = Var(model.t,model.l)

# Declare first derivative of model.x with respect to model.t
model.dxdt = DerivativeVar(model.x, withrespectto=model.t)

# Declare second derivative of model.y with respect to model.t
# Note that this DerivativeVar will be indexed by both model.s and model.t
model.dydt2 = DerivativeVar(model.y, wrt=(model.t,model.t))

# Declare partial derivative of model.z with respect to model.l
# Note that this DerivativeVar will be indexed by both model.t and model.l
model.dzdl = DerivativeVar(model.z, wrt=(model.l), initialize=0)

# Declare mixed second order partial derivative of model.z with respect
# to model.t and model.l and set bounds
```

```
22 model.dz2 = DerivativeVar(model.z, wrt=(model.t, model.l), bounds=(-10,10))
```

The design of the `DerivativeVar` component diverges from the core Pyomo components in its use of positional arguments. Typically, positional arguments are used to specify indexing sets of a particular component. However, for `DerivativeVar` components, a variable is supplied as a positional argument. This design choice was meant to embed the notion of a derivative being an operation on a variable rather than a separate object by itself.

We also note that the 'initialize' keyword argument shown in the above code will initialize the value of a derivative and is not the same as specifying an initial condition or boundary constraint.

After the derivatives in a model have been declared using `DerivativeVar` components, differential equations are declared as standard Pyomo constraints and are not required to have any particular form. The following code snippet shows how one might declare an ordinary or partial differential equation using one or more of the derivatives defined in the previous code sample.

Listing 4.9: Declaring differential equations

```
1  # An ordinary differential equation
2  def _ode_rule(m, tidx):
3      if tidx == 0:
4          return Constraint.Skip
5      return m.dxdt[tidx] == m.x[tidx]**2
6  model.ode = Constraint(model.t,rule=_ode_rule)
7
8  # A partial differential equation
9  def _pde_rule(m, tidx, lidx):
10     if tidx == 0 or lidx == -10 or lidx == 10:
11         return Constraint.Skip
12     return m.dzdl[tidx,lidx] == m.dz2[tidx,lidx]
13 model.pde = Constraint(model.t,model.l,rule=_pde_rule)
```

A modeler might not want to apply a differential equation at one or both boundaries of a continuous domain. This can be addressed explicitly in the `Constraint` declaration using 'Constraint.Skip' as shown above. By default, a constraint declared over a `ContinuousSet` will be applied at every discretization point contained in the set. One can also think of this as the distinction between applying a constraint over an open or closed set.

In the case of boundary conditions for PDAEs, we highlight that the user is responsible for providing consistent expressions and `pyomo.dae` provides syntax to communicate such constraints (e.g., derivative objects and access to specific points).

4.3 Discretization Transformations

Before a Pyomo model with `ContinuousSet` and `DerivativeVar` components can be sent to a finite dimensional solver it must first be sent through a discretization transformation. This transformation converts the dynamic optimization model to a purely algebraic model using a simultaneous discretization approach. In other words, the continuous domains in the model are discretized and any derivatives are approximated using algebraic equations defined at the discretization points. Two families of discretization schemes are currently implemented in `pyomo.dae`: finite differences and collocation.

By separating the model from the discretization scheme we give the user the opportunity to experiment wtih several discretization schemes to find the one that works best with their particular problem. This also provides the freedom to combine schemes in non-standard forms such as using collocation to discretize a spatial domain and finite difference method to discretize in time or viceversa.

One of the key differences between `pyomo.dae` and similar tools is that the discretized model is returned to the user after a transformation is applied. This allows the user to interrogate the discretized model and see the discretization equations that have been added to the model. Furthermore, it allows the user to modify the model after it has been dis-

cretized.

Applying one of the discretization schemes in `pyomo.dae` to a differential equation is analogous to approximating the solution of that differential equation using a numerical method. Numerical methods differ in terms of accuracy and the type of problems they can be applied to. While we will provide a brief overview of the methods implemented in `pyomo.dae`, a detailed review of these methods is outside the scope of this work. We refer the reader to a more comprehensive text on numerical methods for more information on these techniques and details of their applicability, for instance [75, 76, 77, 78].

4.3.1 Finite Difference Transformation

Finite difference methods are the simplest numerical methods to apply manually. They approximate the derivative at a particular point using a difference equation. The `dae.finite_difference` transformation in `pyomo.dae` includes implementations of several finite difference schemes. The most commonly used finite difference method is the backward difference method (also called Implicit or Backward Euler). In our implementation a discretization is applied to a particular continuous domain and propagated to each derivative and constraint over that domain. For instance, after applying the backward difference method to the domain t the derivative and constraint

$$g\left(\frac{dx(t)}{dt}, f(x(t), u(t))\right) = 0, \quad t \in [0, T] \tag{4.4}$$

become,

$$\left.\frac{dx}{dt}\right|_{t_{k+1}} = \frac{x_{k+1} - x_k}{h}, \quad k = 0, ..., N-1 \tag{4.5}$$

$$g\left(\left.\frac{dx}{dt}\right|_{t_{k+1}}, f(x_{k+1}, u_{k+1})\right) = 0, \quad k = 0, ..., N-1 \tag{4.6}$$

where $x_k = x(t_k)$, $t_k = kh$, and h is the step size between discretization points or the size of each finite element. We highlight that the discretization scheme is applied to all constraints and variables of the model in a given continuous domain. Higher order derivative terms

are approximated using recursive schemes. When a `dae.finite_difference` transformation is applied to a Pyomo model, equations such as (4.5) are automatically generated and added to the discretized Pyomo model as equality constraints. The following Python script applies the backward difference method to a Pyomo model. The code also shows how to add a constraint to a discretized model.

Listing 4.10: Applying a finite difference discretization to a Pyomo model

```
from pyomo.environ import *
from pyomo.dae import *

# Import concrete Pyomo model
from pyomoExample import model

# Discretize model using Backward Difference method
discretizer = TransformationFactory('dae.finite_difference')
discretizer.apply_to(model,nfe=20,wrt=model.time,scheme='BACKWARD')

# Add another constraint to discretized model
def _sum_limit(m):
    return sum(m.x1[i] for i in m.time) <= 50
model.con_sum_limit = Constraint(rule=_sum_limit)

# Solve discretized model
solver = SolverFactory('ipopt')
results = solver.solve(model)
```

There are several discretization options available to a `dae.finite_difference` transformation which can be specified as keyword arguments to the 'apply' function of the transformation object.

These keywords are:

'nfe': The desired number of finite element points to be included in the discretization. The default value is 10.

'wrt': Indicates which `ContinuousSet` the transformation should be applied to. If this keyword argument is not specified then the same scheme will be applied to all continuous sets.

'scheme': Indicates which finite difference method to apply. Options are 'BACKWARD', 'CENTRAL', or 'FORWARD'. The default scheme is the backward difference method.

If the existing number of finite element points in a `ContinuousSet` is less than the desired number, new discretization points will be added to the set automatically. If a user specifies a number of finite element points which is less than the number of points already included in the `ContinuousSet` then the transformation will ignore the specified number and proceed with the larger set of points. Discretization points will never be removed from a `ContinousSet` during the discretization transformation.

4.3.2 Collocation Transformation

Orthogonal collocation over finite elements is a popular numerical method for solving differential equations. This technique works by first breaking a continuous domain into $N-1$ finite elements. Then over each finite element i the profile of the differential variable $x(t)$ is approximated using a polynomial of order $K+1$. This polynomial is defined using K collocation points which are additional discretization points within each finite element i. Continuity is enforced at the finite element boundaries for the differential variables x.

The discretization equations for a constraint of the form (4.4) are given by

$$\left.\frac{dx}{dt}\right|_{t_{ij}} = \frac{1}{h_i}\sum_{j=0}^{K} x_{ij}\frac{d\ell_j(\tau_k)}{d\tau}, \quad k = 1,\ldots,K,\ i = 1,\ldots,N-1 \tag{4.7a}$$

$$0 = g\left(\left.\frac{dx}{dt}\right|_{t_{ij}}, f(x_{ik},u_{ik})\right), \quad k = 1,\ldots,K,\ i = 1,\ldots,N-1 \tag{4.7b}$$

$$x_{i+1,0} = \sum_{j=0}^{K} \ell_j(1)x_{ij}, \quad i = 1,\ldots,N-1, \tag{4.7c}$$

where $t_{ij} = t_{i-1} + \tau_j h_i$, $x(t_{ij}) = x_{ij}$, and where we we note that the solution $x(t)$ is interpolated as:

$$x(t) = \sum_{j=0}^{K} \ell_j(\tau)x_{ij}, \quad t \in [t_{i-1}, t_i], \quad \tau \in [0,1] \tag{4.8a}$$

$$\ell_j(\tau) = \prod_{k=0,\neq j}^{K} \frac{(\tau-\tau_k)}{(\tau_j-\tau_k)}. \tag{4.8b}$$

The benefit to using a collocation method over a finite difference method is that it results in significantly more accurate algebraic approximations. The drawback is that this scheme is much harder to implement manually and debug. Furthermore, implementing the collocation discretization equations on high-order derivatives, partial derivatives, or differential equations that are not in a standard form is nontrivial.

There are many variations of collocation methods; these mainly differ in the functional representation of the state profile over each finite element and how the collocation points are defined. The current version of `pyomo.dae` includes two types of collocation methods. They both use Lagrange polynomials to represent the state profiles but differ in the choice of collocation points τ_k. One uses shifted Gauss-Radau roots and the other one uses shifted Gauss-Legendre roots. By shifted we mean that the collocation points τ are defined in the domain $\tau \in [0,1]$ rather than $\tau \in [-1,1]$. For more information on orthogonal collocation we refer the reader to Chapter 10 of [6].

The following Python script applies the collocation method with Lagrange polynomials and Gauss-Radau roots to a Pyomo model. The code also shows how to add an objective

function to a discretized model.

Listing 4.11: Applying a collocation discretization to a Pyomo model

```
from pyomo.environ import *
from pyomo.dae import *

# Import concrete Pyomo model
from pyomoExample2 import model

# Discretize model using Radau Collocation
discretizer = TransformationFactory('dae.collocation')
discretizer.apply_to(model,nfe=20,ncp=6,scheme='LAGRANGE-RADAU')

# Add objective function after model has been discretized
def obj_rule(m):
    return sum((m.x[i]-m.x_ref)**2 for i in m.time)
model.obj = Objective(rule=obj_rule)

# Solve discretized model
solver = SolverFactory('ipopt')
results = solver.solve(model)
```

Notice that this transformation allows the user to specify the number of finite elements, 'nfe', independently of the number of collocation points per finite element, 'ncp'. The discretization options available to a `dae.collocation` transformation are the same as those described above for the `dae.finite_difference` transformation with the modifications shown below:

'scheme': The desired collocation scheme, either 'LAGRANGE-RADAU' or 'LAGRANGE-LEGENDRE'. The default is 'LAGRANGE-RADAU'.

'ncp': The number of collocation points within each finite element. The default value is 3.

It should be noted that any points that exist in a `ContinuousSet` before a transforma-

tion is applied will be used as finite element boundaries and not as collocation points. The locations of the collocation points cannot be specified by the user, they must be generated by the transformation.

4.3.3 Applying Multiple Transformations

Discretizations can be applied independently to each `ContinuousSet` in a model. This provides the user with great flexibility. For example, the same scheme can be applied with different resolutions in different domains:

Listing 4.12: Applying different discretization resolutions to different continuous domains

```
1 discretizer = TransformationFactory('dae.finite_difference')
2 discretize.apply_to(model,wrt=model.t1,nfe=10)
3 discretize.apply_to(model,wrt=model.t2,nfe=100)
```

This feature can be used, for instance, when discretizing different pipelines in a network that exhibit different dynamic behavior or discretizing different stages in an optimal control model with different resolutions.

Different schemes can also be applied to different domains. For example, we can apply a forward difference method to one `ContinuousSet` and the central finite difference method to another `ContinuousSet`:

Listing 4.13: Applying different finite difference transformations to different continuous domains

```
1 discretizer = TransformationFactory('dae.finite_difference')
2 discretizer.apply_to(model,wrt=model.t1,scheme='FORWARD')
3 discretizer.apply_to(model,wrt=model.t2,scheme='CENTRAL')
```

In addition, the user may combine finite difference and collocation discretizations. For example:

Listing 4.14: Combining finite difference and collocation schemes

```
discretizer_fe = TransformationFactory('dae.finite_difference')
discretizer_fe.apply_to(model,wrt=model.t1,nfe=10)
discretizer_col = TransformationFactory('dae.collocation')
discretizer_col.apply_to(model,wrt=model.t2,nfe=10,ncp=5)
```

If the user would like to apply the same discretization to all `ContinuousSet` components in a model, this can be done by specifying a single discretization without the 'wrt' keyword argument. This will apply the same scheme to all `ContinuousSet` components in the model that have not been already discretized.

4.4 Package Implementation and Extensibility

One of the main implementation goals for automatic discretization in `pyomo.dae` is extensibility as there are a plethora of discretization schemes documented in the literature and some specialized schemes might be required in certain applications. As part of our implementation we have developed a general transformation framework along with certain utility functions so that advanced users may easily implement their own custom discretization schemes while reusing the syntax of the modeling language. The transformation framework consists of the following steps:

1. Specify Discretization Options
2. Discretize the Continuous Sets
3. Update Model Components
4. Add Discretization Equations
5. Return Discretized Model

If a user would like to create a custom finite difference scheme then they only have to re-implement step (4) in the framework. The discretization equations for a particular scheme have been isolated from of the rest of the code for implementing the transformation. For example, the specific function for the forward finite difference method is:

Listing 4.15: `pyomo.dae` implementation of the forward finite difference scheme

```
1 def _forward_transform(v,s):
2     """
3     Applies the Forward Difference formula of order O(h) for first derivatives
4     """
5     def _fwd_fun(i):
6         tmp = sorted(s)
7         idx = tmp.index(i)
8         return 1/(tmp[idx+1]-tmp[idx])*(v(tmp[idx+1])-v(tmp[idx]))
9     return _fwd_fun
```

In this function, 'v' represents the continuous variable or function that the method is being applied to while 's' represents the set of discrete points in the continuous domain. In order to implement a custom finite difference method, a user would have to duplicate the above function and just replace the equation next to the first return statement with their method. After implementing a custom finite difference method using the above function template, the only other change that must be made is to add the custom method to the 'all_schemes' dictionary in the Finite_Difference_Transformation class.

In the case of a custom collocation method, changes will be made in steps (2) and (4) of the transformation framework. The code below shows the function specific to a collocation

scheme with Lagrange interpolation polynomials and Radau collocation points.

Listing 4.16: `pyomo.dae` implementation of collocation using Lagrange polynomials and Radau roots

```
def _lagrange_radau_transform(v,s):
    ncp = s.get_discretization_info()['ncp']
    adot = s.get_discretization_info()['adot']
    def _fun(i):
        tmp = sorted(s)
        idx = tmp.index(i)
        if idx == 0: # Don't apply this equation at initial point
            raise IndexError("list index out of range")
        low = s.get_lower_element_boundary(i)
        lowidx = tmp.index(low)
        return sum(v(tmp[lowidx+j])*adot[j][idx-lowidx] \
                *(1.0/(tmp[lowidx+ncp]-tmp[lowidx])) for j in range(ncp+1))
    return _fun
```

In addition to implementing the discretization equations, the user would have to ensure that the desired collocation points are added to the `ContinuousSet` being discretized. In `pyomo.dae`, Radau or Legendre collocation points are calculated following chapter 10 in [6] if the user has the Python package Numpy installed otherwise they are extracted from a stored Python dictionary.

So far we have shown that `pyomo.dae` is extensible in terms of implementing custom discretization schemes but it is also extensible in a more general sense. The modeling abstraction introduced here allows for general implementations of any model transformation or operation typically applied to optimal control problems. For instance, the control profiles can be constrained to follow predefined functions. To illustrate this, we extended the `dae.collocation` transformation with a function that forces a control variable to have a certain profile; for example, piecewise constant or piecewise linear. This is useful, for instance, in optimal control problems where the control variable is often restricted to be

constant over each finite element while the other state variables are not. This function can be implemented by reducing the number of free collocation points for a particular variable. The `reduce_collocation_points()` function is specified using the following keywords:

'var': The variable being restricted to fewer collocation points

'contset': The continuous set indexing the specified 'var' which has been previously discretized.

'ncp': The new number of collocation points. Must be at least 1 and less than the number of collocation points used to discretize the 'contset'.

The function may only be applied to a model after a collocation discretization transformation has been called to discretize the specified `ContinousSet`. The function works by adding constraints to the discretized model which force any extra, undesired collocation points to be interpolated from the others. These constraints have the following form over finite element i:

$$u_i k = \sum_{j=1}^{\bar{K}} \gamma_{kj} u_i j, \; k = \bar{K}+1, ..., K \tag{4.9}$$

where K is the original number of collocation points and $\bar{K}$ is the reduced number of points. A sample implementation is shown below.

Listing 4.17: Enforce path constraints on continuous variables

```
1 # Discretize model using Radau Collocation
2 discretizer = TransformationFactory('dae.collocation')
3 discretizer.apply_to(model, wrt=model.time, nfe=20, ncp=6)
4
5 # Control variable u made constant over each finite element
6 discretizer.reduce_collocation_points(var=model.u,contset=model.time,ncp=1)
```

We also note that `pyomo.dae` does not distinguish between state and control variables. This means that the `reduce_collocation_points()` function can be applied just as

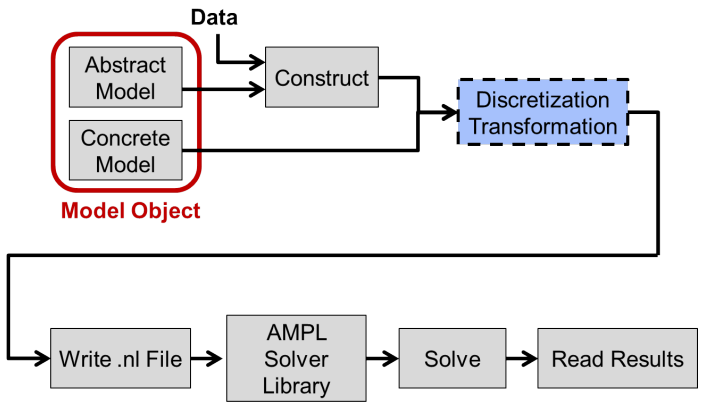


Figure 4.2: Pyomo workflow including discretization transformations

easily to impose a path constraint on a continuous state variable.

The modular nature of Pyomo also allows for extensions in terms of solvers for dynamic optimization. For example, a typical workflow with the current implementation of `pyomo.dae` is shown in Figure 4.2. A straightforward extension would be to replace the 'Write .nl File' and 'AMPL Solver Library' blocks with another tool for automatic differentiation such as CasADi [79]. Or the workflow could be extended to output the dynamic model, before applying a discretization scheme, and then solving the problem using a shooting method. Finally, there are several parallel solvers available for dynamic optimization problems that rely on the discretized model being separated at the finite element boundaries. A general implementation of this step could easily be added to `pyomo.dae`.

4.5 Concluding Remarks

In developing `pyomo.dae` our emphasis was on flexibility of the syntax. This certainly does not mean that every problem can be solved using this tool. Also, because we are not enforcing a structure on the differential equations or a restriction on the order of the derivatives, we do not check for modeling errors or consistency. The responsibility falls on the

user not to implement an inconsistent model. A similar observation also applies to general algebraic modeling languages in which the user is responsible for not implementing models with incompatible constraints or empty feasible regions, for instance. However, now that the `pyomo.dae` syntax is available, we can create model templates for creating models with canonical structures and with that prevent the user from specifying incompatible models. This would allow us to implement solution algorithms tailored for a known structure, check for modeling errors, and more easily interface with existing integrators. The next chapter demonstrates the flexibilty of `pyomo.dae` by applying it to a wide variety of dynamic optimization problems. We also examine the computational costs associated with automatic discretization.

Chapter 5

Examples of Modeling Capabilities

In this chapter we provide examples that illustrate the modeling flexibility provided by `pyomo.dae`. We begin with a simple heat transfer problem and walk step-by-step through the code. We then include a small example of an optimal control problem which is a common application for dynamic optimization. Next, we show a medium-scale parameter estimation problem for a dynamic disease transmission model that demonstrates how to embed time-dependent data into a model. Our final example is a stochastic optimal control model for natural gas networks which shows how to combine all of the modeling elements. This model also demonstrates the scalability of `pyomo.dae`. All examples use Pyomo version 4.3, Python version 2.7.6, and IPOPT version 3.10.1 on a desktop computer running Ubuntu 14.04. The models can be accessed at `https://software.sandia.gov/svn/public/pyomo/pyomo/trunk/examples/dae/`.

5.1 Heat Transfer

This example demonstrates the solution of a one-dimensional heat equation. The equation, initial condition, and boundary conditions are shown below [80].

$$\pi^2 \frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} \tag{5.1a}$$

$$u(x,0) = \sin(\pi x) \tag{5.1b}$$

$$u(0,t) = 0 \tag{5.1c}$$

$$\pi e^{-t} + \frac{\partial u}{\partial x}(1,t) = 0 \tag{5.1d}$$

$$x \in [0,1],\ t \in [0,T]. \tag{5.1e}$$

This problem has a number of interesting features including first and second order partial derivatives, boundary conditions involving both the variable and its partial derivative, and nonlinearity from a trigonometric function. In Listing 5.1 we first present a Pyomo model that implements a manual discretization (without `pyomo.dae`). The model was discretized in space using the backward difference method and was discretized in time using orthogonal collocation with Radau points. The first line in the model imports the needed Pyomo package. Lines 4-6 declare the number of discretization points and lines 9-11 declare sets containing all the discretization points. Lines 14-20 define the collocation matrix for 4th order Lagrange polynomials with Radau roots. Computing this matrix is often one of the more tedious parts of implementing a model. Lines 22 and 23 define the spatial and time scaling for each finite element and lines 25-30 define the variables in the model including the differential variables and derivatives in the model.

Models that are discretized manually typically use discretization points without any physical meaning i.e. time/distance scaling is handled separately from the discretization points themselves. This can often lead to modeling errors and confusion especially with more complicated discretization schemes such as collocation. The model below tries to overcome this difficulty by introducing an additional differential variable 'm.t' which is used to calculate the properly scaled time points. Lines 32-54 define constraints representing the heat transfer model (5.1a). The discretization schemes are applied in lines 56-93. Finally a dummy objective function is declared in line 95 and the model is solved using IPOPT. The entire implementation requires 98 lines of code and while changing the number of finite element points is straight-forward, changing the number of collocation points or either of the discretization schemes will require significant changes to the implementation.

Listing 5.1: Pyomo heat tranfer model discretized manually

```
1 from pyomo.environ import *
2
3 m = ConcreteModel()
```

```
4  m.nfet = Param(initialize=20)    # Number of finite elements in time
5  m.ncp = Param(initialize=4)      # Number of collocation points per finite element
6  m.nfex = Param(initialize=25)    # Number of finite elements in space
7  m.pi = Param(initialize=3.1416) # Pi
8
9  m.tfe = RangeSet(0,value(m.nfet)-1) # Set of finite element points in time
10 m.tcp = RangeSet(1,value(m.ncp))     # Set of collocation points in time
11 m.x = RangeSet(0,value(m.nfex))      # Set of finite element points in space
12
13 # Collocation Matrix
14 radau_adot={(1,1):-9.0000000000000,(1,2):-4.1393876913398,(1,3):1.7393876913398,\
15          (1,4):-2.9999999999999,(2,1):10.0488093998274,(2,2):3.2247448713915,\
16          (2,3):-3.5678400846904,(2,4):5.5319726474218,(3,1):-1.3821427331607,\
17          (3,2):1.1678400846904,(3,3):0.7752551286084,(3,4):-7.5319726474218,\
18          (4,1):0.3333333333333,(4,2):-0.2531972647421,(4,3):1.0531972647421,\
19          (4,4):5.0000000000000}
20 m.adot = Param(m.tcp,m.tcp,initialize=radau_adot)
21
22 m.ht = Param(initialize=2.0/m.nfet) # Length of time finite element
23 m.hx = Param(initialize=1.0/m.nfex) # Length of spatial finite element
24
25 m.u = Var(m.x,m.tfe,m.tcp,initialize=0.2)
26 m.t = Var(m.tfe,m.tcp)
27
28 m.dudx = Var(m.x,m.tfe,m.tcp)
29 m.dudx2 = Var(m.x,m.tfe,m.tcp)
30 m.dudt = Var(m.x,m.tfe,m.tcp)
31
32 def _pde(m,i,j,k):
33     if i == 0 or i == value(m.nfex) or k == 0:
34         return Constraint.Skip
35     return m.pi**2*m.dudt[i,j,k] == m.dudx2[i,j,k]
36 m.pde = Constraint(m.x,m.tfe,m.tcp,rule=_pde)
37
```

```
38 def _initcon(m,i):
39     if i == 0 or i == value(m.nfex):
40         return Constraint.Skip
41     return m.u[i,0,1] == sin(m.pi*i*m.hx)
42 m.initcon = Constraint(m.x,rule=_initcon)
43
44 def _lowerbound(m,j,k):
45     return m.u[0,j,k] == 0
46 m.lowerbound = Constraint(m.tfe,m.tcp,rule=_lowerbound)
47
48 def _upperbound(m,j,k):
49     return m.pi*exp(-m.t[j,k])+m.dudx[value(m.nfex),j,k] == 0
50 m.upperbound = Constraint(m.tfe,m.tcp,rule=_upperbound)
51
52 def _init_t(m):
53     return m.t[0,1] == 0
54 m._init_t = Constraint(rule=_init_t)
55
56 # Apply finite difference discretization equations
57 def _dudx_backwardDifference(m,i,j,k):
58     if i == 0:
59         return Constraint.Skip
60     return (m.u[i,j,k]-m.u[i-1,j,k])/m.hx == m.dudx[i,j,k]
61 m.dudx_backwardDiff = Constraint(m.x,m.tfe,m.tcp,rule=_dudx_backwardDifference)
62
63 def _dudx2_backwardDifference(m,i,j,k):
64     if i == 0 or i == 1:
65         return Constraint.Skip
66     return (m.u[i-2,j,k]-2*m.u[i-1,j,k]+m.u[i,j,k])/m.hx**2 == m.dudx2[i,j,k]
67 m.dudx2_backwardDiff = Constraint(m.x,m.tfe,m.tcp,rule=_dudx2_backwardDifference)
68
69 # Apply collocation discretization equations
70 def _dudt_collocation(m,i,j,k):
71     if k == 1:
```

```
72          return Constraint.Skip
73      return sum(m.u[i,j,s]*m.adot[s,k] for s in m.tcp) == m.ht*m.dudt[i,j,k]
74  m.dudt_collocation = Constraint(m.x,m.tfe,m.tcp,rule=_dudt_collocation)
75
76  def _dt_collocation(m,j,k):
77      if k == 1:
78          return Constraint.Skip
79      return sum(m.t[j,s]*m.adot[s,k] for s in m.tcp) == m.ht
80  m.dt_collocation = Constraint(m.tfe,m.tcp,rule=_dt_collocation)
81
82  # Apply collocation continuity equations
83  def _u_continuity(m,i,j):
84      if j == value(m.nfet)-1:
85          return Constraint.Skip
86      return m.u[i,j+1,1] == m.u[i,j,4]
87  m.u_continuity = Constraint(m.x,m.tfe,rule=_u_continuity)
88
89  def _t_continuity(m,j):
90      if j == value(m.nfet)-1:
91          return Constraint.Skip
92      return m.t[j+1,1] == m.t[j,4]
93  m.t_continuity = Constraint(m.tfe,rule=_t_continuity)
94
95  m.obj = Objective(expr=1)
96
97  solver = SolverFactory('ipopt')
98  results = solver.solve(m,tee=True)
```

We now contrast the Pyomo model with manual discretization to one that has been discretized using `pyomo.dae`. The code is shown below. This implementation requires one additional package imports in order to gain access to the desired modeling components. Lines 4-8 define the modeling sets and variables. Note that the time and spatial scaling is handled explicitly in the ContinuousSet components. This means that after applying a dis-

cretization scheme, the components 'm.t' and 'm.x' will contain actual points in time and space respectively. This also eliminates the need for an additional differential variable to represent time as we had in the manually discretized model. Lines 10-13 define the derivatives that appear in the model, lines 15-34 define the equations in the heat transfer model, and line 36 defines the dummy objective function. These sections are nearly identical to the previous implementation. Lines 38-42 apply discretization schemes to the model. Line 41 applies the Backward Difference method to the spatial ContinuousSet, 'm.x', and line 42 applies Radau collocation to the time ContinuousSet, 'm.t'. Again, the model is solved using IPOPT.

The implementation using `pyomo.dae` requires only half the number of lines of the manually discretized model, leading to a much more concise and easy to follow model. The discretization schemes have been completely separated from the model itself. Notice that the second implementation doesn't have any of the discretization-specific parameters used in the first. Furthermore, changing the number of finite elements, the number of collocation points, or the entire discretization scheme is equally simple by modifying one of the 4 lines of code defining the discretization. To illustrate this point, we show some alternate ways to discretize the heat transfer model below.

Listing 5.2: Pyomo heat tranfer model discretized using `pyomo.dae`

```
from pyomo.environ import *
from pyomo.dae import *

m = ConcreteModel()
m.pi = Param(initialize=3.1416)
m.t = ContinuousSet(bounds=(0,2))
m.x = ContinuousSet(bounds=(0,1))
m.u = Var(m.x,m.t)

# Declare derivatives in the model
m.dudx = DerivativeVar(m.u,wrt=m.x)
m.dudx2 = DerivativeVar(m.u,wrt=(m.x,m.x))
m.dudt = DerivativeVar(m.u,wrt=m.t)

# Declare PDE
def _pde(m,i,j):
    if i == 0 or i == 1 or j == 0:
        return Constraint.Skip
    return m.pi**2*m.dudt[i,j] == m.dudx2[i,j]
m.pde = Constraint(m.x,m.t,rule=_pde)

def _initcon(m,i):
    if i == 0 or i == 1:
        return Constraint.Skip
    return m.u[i,0] == sin(m.pi*i)
m.initcon = Constraint(m.x,rule=_initcon)

def _lowerbound(m,j):
    return m.u[0,j] == 0
m.lowerbound = Constraint(m.t,rule=_lowerbound)

def _upperbound(m,j):
```

```
33      return m.pi*exp(-j)+m.dudx[1,j] == 0
34  m.upperbound = Constraint(m.t, rule=_upperbound)
35
36  m.obj = Objective(expr=1)
37
38  # Discretize using Finite Difference and Collocation
39  discretizer = TransformationFactory('dae.finite_difference')
40  discretizer2 = TransformationFactory('dae.collocation')
41  discretizer.apply_to(m, nfe=25, wrt=m.x, scheme='BACKWARD')
42  discretizer2.apply_to(m, nfe=20, ncp=3, wrt=m.t)
43
44  solver = SolverFactory('ipopt')
45  results = solver.solve(m, tee=True)
```

Listing 5.3: Alternate discretizations for heat transfer model using `pyomo.dae`

```
# Discretize entire model using Finite Difference Method
discretizer = TransformationFactory('dae.finite_difference')
discretizer.apply_to(m,nfe=25,wrt=m.x,scheme='BACKWARD')
discretizer.apply_to(m,nfe=20,wrt=m.t,scheme='FORWARD')

# Discretize entire model using Collocation
discretizer = TransformationFactory('dae.collocation')
discretizer.apply_to(m,nfe=10,ncp=3,wrt=m.x,scheme='LAGRANGE-LEGENDRE')
discretizer.apply_to(m,nfe=20,ncp=3,wrt=m.t,scheme='LAGRANGE-RADAU')
```

Experimenting with the type of discretization scheme along with the resolution of the desired scheme becomes systematic. However, we are careful to note that not every discretization scheme is appropriate for every model. Different boundary/initial conditions may need to be applied depending on the model and the discretization scheme used. `pyomo.dae` does not perform any model checking to ensure the applied discretization is appropriate for the model it is applied to. This is currently the responsibility of the user. The `pyomo.dae` framework, however, provides the necessary constructs to enable the user to tailor the specification of boundary conditions.

5.2 Optimal Control

The purpose of optimal control problems is to find an optimal sequence of inputs to a dynamic system in order to meet some objective on that system. For example, a typical objective might be minimizing or maximizing the final value of some state variable within

a fixed time horizon. A small example was obtained from [81]:

$$\min \quad x_3(t_f) \tag{5.2a}$$

$$\text{s.t.} \quad \dot{x}_1 = x_2 \tag{5.2b}$$

$$\dot{x}_2 = -x_2 + u \tag{5.2c}$$

$$\dot{x}_3 = x_1^2 + x_2^2 + 0.005 \cdot u^2 \tag{5.2d}$$

$$x_2 - 8 \cdot (t - 0.5)^2 + 0.5 \leq 0 \tag{5.2e}$$

$$x_1(0) = 0,\, x_2(0) = -1,\; x_3(0) = 0, t_f = 1 \tag{5.2f}$$

This problem has three state variables x_1, x_2, x_3 and one control u. The following code snippet shows how to implement some of the interesting features in this model including the objective function (5.2a) and a complex path constraint (5.2e). Discretizing problem (5.2) using orthogonal collocation with 7 finite elements and 6 collocation points per element will result in the optimal profiles shown in the top plots of Figure 5.1. Note that in this solution the control variable is allowed to change at every time point. In many applications this sort of continuous change in the control variable is undesirable or impossible to implement. Therefore, in practice, the control variable is often restricted to be piecewise constant over a certain time interval. This sort of restriction can easily be implemented in `pyomo.dae` using the `reduce_collocation_points()` function. An example of how to use this function is shown in the code snippet presented below. The last line restricts the control variable u to only 1 free collocation point per finite element rendering it piecewise constant. If we instead wanted the control variable to be piecewise linear we would only have to modify the 'ncp' keyword argument in the `reduce_collocation_points()` function to 'ncp=2' which would give the control variable two free collocation points, or degrees of freedom, per finite element.

Listing 5.4: Partial implementation of problem (5.2)

```
1 # Setting the objective function
2 m.obj = Objective(expr=m.x3[1])
```

```
3
4 # Declaring the path constraint
5 def _con(m, tidx):
6         return m.x2[tidx]-8*(tidx-0.5)**2+0.5 <= 0
7 m.con = Constraint(m.t, rule=_con)
```

Listing 5.5: Code for discretizing the optimal control model and reducing the degrees of freedom

```
1 # Discretize model using radau collocation
2 discretizer = TransformationFactory('dae.collocation')
3 discretizer.apply_to(model, wrt=model.t, nfe=7, ncp=6)
4
5 # Restrict control to be piecewise constant
6 discretizer.reduce_collocation_points(model, var=model.u, ncp=1, contset=model.t)
```

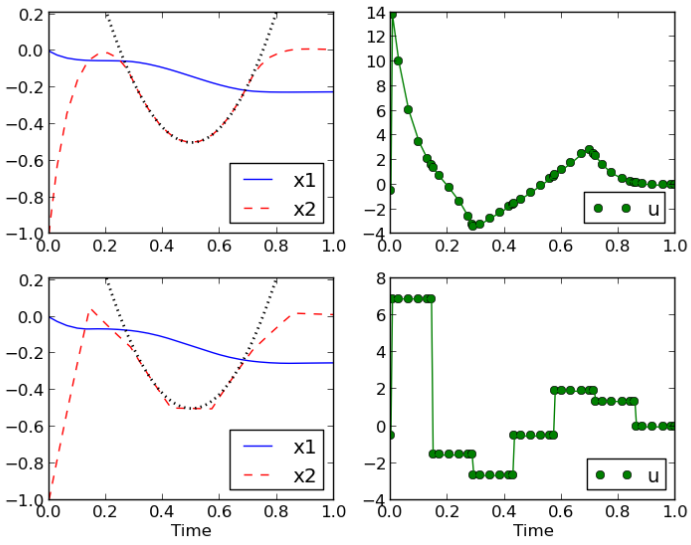


Figure 5.1: Solution to the optimal control problem (top) with no restrictions on the control variable and (bottom) restricting the control variable to be piecewise constant. Black dotted line shows the inequality path constraint.

5.3 Parameter Estimation for Disease Transmission

The next example is a parameter estimation problem taken from [82]. This problem estimates parameters for a childhood infectious disease transmission model from disease case

data. The mathematical model is:

$$\min \quad \omega_I \sum_{i \in \mathcal{F}} (\epsilon_{I_i})^2 + \omega_\phi \sum_{k \in \mathcal{T}} (\epsilon_{\phi_k})^2 \tag{5.3a}$$

$$\text{s.t.} \quad \frac{dS}{dt} = \frac{-\beta(y(t))S(t)I(t)}{N} - \epsilon_I(t) + B(t) \tag{5.3b}$$

$$\frac{dI}{dt} = \frac{\beta(y(t))S(t)I(t)}{N} + \epsilon_I(t) - \gamma I(t) \tag{5.3c}$$

$$\frac{d\phi}{dt} = \frac{\beta(y(t))S(t)I(t)}{N} + \epsilon_I(t) \tag{5.3d}$$

$$R_k^\star = \eta_k(\phi_{i,k} - \phi_{i,k-1}) + \epsilon_{\phi_k}, \ k \in \mathcal{T} \tag{5.3e}$$

$$\beta_k = \beta^{\text{mag}} \cdot \beta_k^{\text{patt}}, \ k \in \mathcal{T} \tag{5.3f}$$

$$1.0 = \frac{\sum_{k \in \mathcal{T}} \beta_k^{\text{patt}}}{\text{len}(\mathcal{T})} \tag{5.3g}$$

$$\bar{S} = \frac{\sum_{i \in \mathcal{F}} S_i}{\text{len}(\mathcal{F})}, \quad \bar{\beta} = \frac{\sum_{k \in \mathcal{T}} \beta_k}{\text{len}(\mathcal{T})} \tag{5.3h}$$

$$0 \leq I(t), \quad S(t) \leq N, \quad 0 \leq \beta(y(t)), \quad 0 \leq \phi(t) \tag{5.3i}$$

where S represents the number of people susceptible to the disease, I is the number of people with the disease who are infectious, and ϕ is the cumulative incidence of the disease. $R^\star$ is the reported incidence of the disease and is a known input at a discrete set of reporting times. For a more detailed description of the variables and parameters in the model we refer the reader to [82].

The purpose of this model is to estimate the time-varying transmission parameter β under the assumption that β varies seasonally but is the same from year to year. This introduces an interesting feature of multiple time sets; $\mathcal{T}$, the set of reporting times over a single year and $\mathcal{F}$, the set of all reporting times over a 20 year period. The set $\mathcal{F}$ is represented by the `pyomo.dae` continuous set `model.TIME`. The code snippet below shows how equation (5.3d) is implemented accounting for these multiple time sets.

Listing 5.6: Code for implementing the differential equation (5.3d) in the gas network model

```
1 def _phidot_eq(model,i):
```

```
2     if i == 0:
3         return Constraint.Skip
4     fe = model.TIME.get_upper_element_boundary(i)
5     j = model.TIME._fe.index(fe)
6     return model.phidot[i] == model.eps_I[j] \
7       + (model.beta[model.P_BETA_NDX[j]]*model.I[i]*model.S[i])/model.P_POP
8 model.phidot_eq = Constraint(model.TIME, rule=_phidot_eq)
```

Lines 4 and 5 take any time point $i \in \mathcal{F}$ and determine the corresponding upper finite element boundary `fe`. The parameter `model.P_BETA_NDX` then maps this finite element value to the correct index in $\mathcal{T}$ for `model.beta`.

The disease infection model has three differential equations and was discretized with 520 finite elements and 3 collocation points per finite element using orthogonal collocation with Lagrange polynomials and Radau roots. This leads to an NLP with 10,458 variables and 9,910 equality constraints. The model was discretized such that all finite element points correspond to a reporting time for $R^\star$. Similar to the previous example, the model was implemented twice, once by doing the discretization manually and once by using `pyomo.dae`. Table 5.1 compares the timing and solver results for the two models. The creation time refers to the amount of time it takes to read the python script and build the optimization model in memory. For the implementation using `pyomo.dae`, the creation time also includes the time for automatic discretization. If we compare the creation time for the two implementations we see that they don't differ significantly showing that there is little overhead associated with applying the discretization automatically.

If we compare the solver results for the two implementations, we see a similar result. In particular, the solve time for the two models is practically the same and it takes the solver the same number of iterations to find the optimal solution. Even though the timing results reported in the table show little difference between the performance of the two implementations, there is a crucial time which is not reported here and that is the implementation time. The first time a modeler implements a discretization scheme like collocation manu-

	Manual Discretization	Using `pyomo.dae`
Creation Time (sec)	1.38	2.20
Solve Time (CPU sec)	2.65	2.57
IPOPT Iterations	27	27
Objective ($\times 10^-5$)	1.4716	1.4716

Table 5.1: Comparison of two implementations of the disease transmission problem

ally, it could easily take over an hour to apply and debug a model with a single differential equation. In contrast, incorporating a differential equation using `pyomo.dae` is as simple as writing a regular model constraint and selecting which discretization scheme to apply which can be implemented in a fraction of the time.

Model initialization is an important step in being able to solve problems with differential equations. The following code snippet shows how the variable S is initialized for this particular example.

Listing 5.7: Code for initializing S in the disease transmission model

```
def _init_S(model,i):
    if i==0:
        return model.init_S_init
    fe = model.TIME.get_upper_element_boundary(i)
    j = model.TIME._fe.index(fe)
    return model.init_S[j]
def _people_bounds(model):
    return (0.0, model.P_POP)
model.S = Var(model.TIME, initialize=_init_S, bounds=_people_bounds)
```

We highlight that this is an illustration of how a `pyomo.dae` model can potentially be initialized but many alternatives are possible. In particular, the approach presented here initializes the finite element points using a profile defined by the time-varying parameter `model.init_S` and it initializes the intermediate collocation points by setting them

equal to the value at the upper finite element boundary. We choose this initialization approach because it is flexible enough to handle discretizations with any number of collocation points. pyomo.dae does not currently provide any automatic frameworks for model initialization. However, using existing Pyomo and Python constructs, a user can create initializations.

5.4 Stochastic Optimal Control for Natural Gas Networks

The last example seeks to optimize gas network inventories while accounting for uncertainties in the system. The idea is to satisfy uncertain gas demands in the network by building up inventory in the pipelines in a way that minimizes the required compression

power. The model was proposed in [83] and is shown below.

$$\min \sum_{t\in\mathcal{T}}\sum_{i\in\mathcal{S}_n} c_s s_{i,t}\Delta\tau + \sum_{t\in\mathcal{T}}\sum_{\ell\in\mathcal{L}} c_e P_{\ell,t}\Delta\tau + \sum_{t\in\mathcal{T}}\sum_{j\in\mathcal{D}} c_d(d_{j,t}-\bar{d}_{j,t})^2 \tag{5.4a}$$

$$+ \sum_{k\in\mathcal{X}}\sum_{\ell\in\mathcal{L}} c_T(p_{\ell,T,k}-p_{\ell,0,k})^2 + \sum_{k\in\mathcal{X}}\sum_{\ell\in\mathcal{L}} c_T(f_{\ell,T,k}-f_{\ell,0,k})^2 \tag{5.4b}$$

$$\text{s.t. } \frac{\partial p_\ell(x,\tau)}{\partial\tau} = -c_{1,\ell}\frac{\partial f_\ell(x,\tau)}{\partial x}, \quad \ell\in\mathcal{L},\ x\in[0,L_\ell],\ \tau\in[0,T] \tag{5.4c}$$

$$\frac{\partial f_\ell(x,\tau)}{\partial t} = -c_{2,\ell}\frac{\partial p_\ell(x,\tau)}{\partial x} - c_{3,\ell}\frac{f_\ell(x,\tau)|f_\ell(x,\tau)|}{p_\ell(x,\tau)}, \quad \ell\in\mathcal{L},\ x\in[0,L_\ell],\ \tau\in[0,T] \tag{5.4d}$$

$$p_\ell(L_\ell,\tau) = \theta_{rec(\ell)}(\tau), \ell\in\mathcal{L},\ \tau\in[0,T] \tag{5.4e}$$

$$p_\ell(0,\tau) = \theta_{snd(\ell)}(\tau),\ \ell\in\mathcal{L}_p,\ \tau\in[0,T] \tag{5.4f}$$

$$p_\ell(0,\tau) = \theta_{snd(\ell)}(\tau) + \Delta\theta_\ell(\tau),\ \ell\in\mathcal{L}_a,\ \tau\in[0,T] \tag{5.4g}$$

$$\sum_{\ell\in\mathcal{L}_n^{rec}} f_\ell(L_\ell,\tau) - \sum_{\ell\in\mathcal{L}_n^{snd}} f_\ell(0,\tau) + \sum_{i\in\mathcal{S}_n} s_i(\tau) - \sum_{j\in\mathcal{D}_n} d_j(\tau) = 0,\ n\in\mathcal{N}\ \tau\in[0,T] \tag{5.4h}$$

$$P_\ell(\tau) = c_p\cdot T\cdot f_\ell(0,\tau)\left(\left(\frac{\theta_{snd(\ell)}(\tau)+\Delta\theta_\ell(\tau)}{\theta_{snd(\ell)}(\tau)}\right)^{\frac{\gamma-1}{\gamma}} - 1\right),\ \ell\in\mathcal{L},\ \tau\in[0,T] \tag{5.4i}$$

$$\theta_{sup(i),\tau} = \bar{\theta}_i^{sup},\ i\in\mathcal{S},\ \tau\in[0,T] \tag{5.4j}$$

$$0 = -c_{1,\ell}\frac{\partial f_\ell(x,0)}{\partial x},\ \ell\in\mathcal{L},\ x\in[0,L_\ell] \tag{5.4k}$$

$$0 = -c_{2,\ell}\frac{\partial p_\ell(x,0)}{\partial x} - c_{3,\ell}\frac{f_\ell(x,0)|f_\ell(x,0)|}{p_\ell(x,0)},\ \ell\in\mathcal{L},\ x\in[0,L_\ell] \tag{5.4l}$$

$$P_\ell^L \le P_\ell(\tau) \le P_\ell^U,\ \ell\in\mathcal{L}_a,\ \tau\in[0,T] \tag{5.4m}$$

$$\theta_\ell^{suc,L} \le \theta_{snd(\ell)}(\tau) \le \theta_\ell^{suc,U},\ \ell\in\mathcal{L}_a,\ \tau\in[0,T] \tag{5.4n}$$

$$\theta_\ell^{dis,L} \le \theta_{snd(\ell)}(\tau) + \Delta\theta_\ell(\tau) \le \theta_\ell^{dis,U},\ \ell\in\mathcal{L}_a,\ \tau\in[0,T] \tag{5.4o}$$

This model includes detailed network dynamics given by the PDEs (5.4c) and (5.4d). Note that the PDEs are indexed by the number of links $\mathcal{L}$ in the network which means that even moderately sized networks will require the simultaneous solution of many PDEs. Uncertainty is included in this formulation by considering multiple scenarios for the gas demand and formulating the problem as a stochastic optimization problem. In this case,

the PDEs would have to be replicated for each scenario. Overall this leads to an optimization model that changes dramatically in size with changes in the number of network links, scenarios, and discretization points. The implementations for the partial differential equations and the pressure boundary conditions are shown in the code snippets below.

Listing 5.8: Code for implementing the PDE equations (5.4c) and (5.4d)

```
# First PDE for gas network model
def flow_rule(m,j,i,t,k):
    if t == m.TIME.first() or k == m.DIS.last():
        return Constraint.Skip # Skip at initial time or final location
    return m.dpxdt[j,i,t,k]/3600 + m.c1[i]/m.llength[i]*m.dfxdx[j,i,t,k] == 0
model.flow = Constraint(model.SCEN,model.LINK,model.TIME,
                        model.DIS,rule=flow_rule)

# Second PDE for gas network model
def press_rule(m,j,i,t,k):
    if t == m.TIME.first() or k == m.DIS.last():
        return Constraint.Skip # Skip at initial time or final location
    return m.dfxdt[j,i,t,k]/3600 ==
        -m.c2[i]/m.llength[i]*m.dpxdx[j,i,t,k] - m.slack[j,i,t,k]
model.press = Constraint(model.SCEN,model.LINK,model.TIME,
                        model.DIS,rule=press_rule)

def slackeq_rule(m,j,i,t,k):
    if t == m.TIME.last():
        return Constraint.Skip
    return m.slack[j,i,t,k]*m.px[j,i,t,k] == m.c3[i]*m.fx[j,i,t,k]*m.fx[j,i,t,k]
model.slackeq = Constraint(model.SCEN,model.LINK,model.TIME,
                        model.DIS,rule=slackeq_rule)
```

Listing 5.9: Code for implementing the pressure boundary conditions (5.4e), (5.4f), and (5.4g)

```
# boundary conditions pressure, passive links
```

```
2  def presspas_start_rule(m,j,i,t):
3      return m.px[j,i,t,m.DIS.first()] == m.p[j,m.lstartloc[i],t]
4  model.presspas_start = Constraint(model.SCEN,model.LINK_P,
5                                    model.TIME,rule=presspas_start_rule)
6
7  def presspas_end_rule(m,j,i,t):
8      return m.px[j,i,t,m.DIS.last()] == m.p[j,m.lendloc[i],t]
9  model.presspas_end = Constraint(model.SCEN,model.LINK_P,
10                                   model.TIME,rule=presspas_end_rule)
11
12 # boundary conditions pressure, active links
13 def pressact_start_rule(m,j,i,t):
14     return m.px[j,i,t,m.DIS.first()] == m.p[j,m.lstartloc[i],t]+m.dp[j,i,t]
15 model.pressact_start = Constraint(model.SCEN,model.LINK_A,
16                                   model.TIME,rule=pressact_start_rule)
17
18 def pressact_end_rule(m,j,i,t):
19     return m.px[j,i,t,m.DIS.last()] == m.p[j,m.lendloc[i],t]
20 model.pressact_end = Constraint(model.SCEN,model.LINK_A,
21                                   model.TIME,rule=pressact_end_rule)
```

There are two continuous domains in this model, time and one dimension in space. The model was discretized in time using a backward finite difference scheme and was discretized in space using a forward finite difference scheme. The model was discretized with 47 time intervals and multiple numbers of spatial points. Table 5.2 shows the computational effort required for different spatial discretization resolutions and Figure 5.2 shows how the flow profile converges as the number of discretization points increases. The largest problem solved had over 400,000 variables and constraints. Note that the creation time for the model is only a small fraction of the overall solution time. Again, the creation time includes the time to apply a discretization scheme automatically. We also see that the creation time does not increase as dramatically as the solve time when the number of discretization points increases. This illustrates that the automatic discretization routines are scalable.

Table 5.2: Effect of spatial discretization resolution on computational performance

Scenarios	N_x	Variables	Constraints	Iterations	Creation (sec)	Solve (sec)
1	2	9686	9144	35	0.64	2.27
1	6	25718	25128	45	1.71	11.85
1	10	41750	41112	43	2.92	19.26
1	20	81830	81072	50	5.87	40.84
1	60	242150	240912	56	17.72	147.72
1	100	402470	400752	67	31.54	412.18
3	2	29056	27872	37	1.84	17.50
3	10	125248	123776	54	8.96	124.44
3	20	245488	243656	64	18.07	276.18

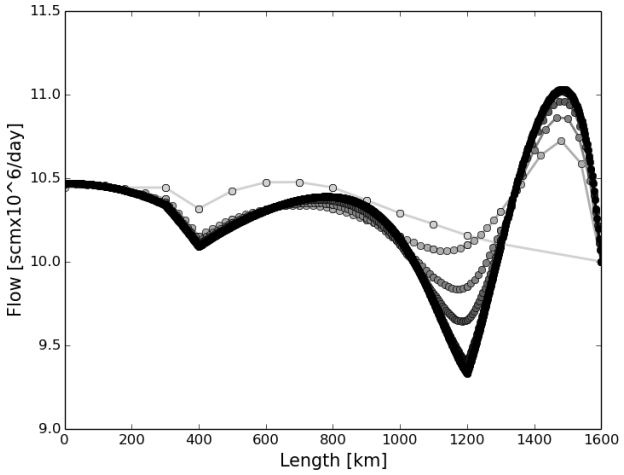


Figure 5.2: Optimal axial flow profile in gas network for different resolutions. Darker color indicates more discretization points.

It should also be mentioned that the model reported in [83] was first implemented in Pyomo using a manual discretization scheme. During the process of converting the manu-

ally discretized model to `pyomo.dae` we actually discovered an "off-by-one" error in the manual discretization. Such errors are easy to make and often difficult to find, given the complexity of the models. In this case `pyomo.dae` becomes handy because it provides a concise and systematic way of discretizing the variables and constraints in the model.

We again highlight that `pyomo.dae` does not perform any model checking. Therefore it is up to the user to write a well-posed model and pick an appropriate discretization scheme for their model. This is worth noting for this example because it is up to the user to skip the enforcement of the PDE at certain domain boundaries as shown in Listing 5.8 in order to ensure consistency between the specified boundary conditions and the applied discretization scheme.

5.5 Concluding Remarks

The level of model abstraction introduced by `pyomo.dae` opens the door for novel model formulations and the implementation of generalized frameworks. Common applications of dynamic optimization, such as model predictive control (MPC), which require solving a sequence of optimization problems can be implemented as general frameworks and separated from the system model they are applied to. In fact, a general implementation of MPC has already been implemented using this framework.

Furthermore, there is also potential for `pyomo.dae` to be used in multigrid/multiscale applications. By separating the model from the discretization scheme, we enable general implementations of algorithms requiring the sequential solution of problems with different discretization resolutions or to communicate discretization information to specialized solvers.

In addition to the core modeling components found in every algebraic modeling language, Pyomo also has other specialized packages for handling generalized disjunctive programming (GDP) and stochastic programming (SP) problems. These extensions follow the modularized design of Pyomo and can be combined with `pyomo.dae`. For example,

this allows a user to easily represent GDP problems with differential equations or SP problems incorporating a dynamic model in each scenario. Providing the user a straightforward way to represent these complex models enables the development of new algorithms for solving them.

Part III

Parallel Solution Algorithms

Chapter 6
Parallel Algorithms for Dynamic Optimization

The modeling framework in the previous section provides an intuitive way for formulating dynamic optimization problems. In this section, we now shift the focus to efficient solution strategies for these problems. As mentioned previously, one of the main challenges of using a direct transcription approach to solve dynamic optimization problems is solving the large-scale nonlinear optimization problem resulting from the discretization. This challenge is exacerbated when considering industrial-scale dynamic models. Parallel decomposition strategies such as Schur complement decomposition have been shown to significantly reduce the computational costs of solving large-scale dynamic optimization problems. In this chapter we review this approach and also propose another parallel algorithm called cyclic reduction (CR) for these problems.

6.1 Introduction

Several parallel decomposition strategies for dynamic optimization have been proposed in the literature. Parallel techniques have been developed for both sequential and simultaneous dynamic optimization solution methods[84, 85, 86, 87, 88]. Our work makes use of a simultaneous discretization, or direct transcription, approach and exploits the structure of the system imposed by the discretization. Similar approaches have exploited a bordered block diagonal structure that arises in these systems and solved them using different versions of Schur complement decomposition[89, 90, 91, 92, 93].

Our work differs in that it takes advantage of a block tridiagonal structure. We make use of an algorithm called cyclic reduction which is a parallel algorithm for solving tridiagonal or block tridiagonal linear systems. It was originally proposed by Golub and Hockney in the 1960's [94] and has historically been applied in the context of solving partial differential equations numerically [95, 96, 97, 98]. A nice review of the history of CR and its applications can be found in [99].

The original formulation of CR proposed by Hockney was found to be numerically unstable. A stable variant of the algorithm was later proposed by Buneman[95]. Since then, the stability and numerical conditioning of CR has been studied in [100, 101, 102]. It has been shown that under the condition of diagonal dominance or block diagonal dominance CR is well-defined and stable.

CR has previously been applied to solve optimal control problems in parallel however it was done within the context of a multiple shooting approach[103, 104]. More recently [105] applied CR to solve quadratic programming problems (QPs) arising from model predictive control with the Mehrotra Predictor-Corrector interior point method. The KKT matrix was written in a dense normal form and cyclic reduction was applied to the block tridiagonal structure with dense blocks.

In this work we consider a more general class of nonlinear dynamic optimization problems and apply CR in the context of a simultaneous discretization approach solved using an interior point method. We work with the symmetric indefinite augmented form of the KKT matrix which has a block diagonal structure with sparse blocks. Following the analysis in [105] we also investigate the numerical conditioning and stability of CR applied to dynamic optimization problems.

Several high-performance implementations of CR have been developed over the years. Well-known implementations include BLKTRI[106], BCYCLIC[107], and the GPU implementation of Zhang et al.[108]. However none of these implementations is ideally suited to handle the sparse, structured, nonconstant blocks arising from dynamic optimization

problems. BLKTRI is specialized to handle separable elliptic partial differential equations, BCYCLIC is designed to handle dense blocks, and the GPU implementation solves tridiagonal matrices rather than block tridiagonal matrices. The main contribution of this work is to show that CR is a promising approach for dynamic optimization and motivate the future development of a high-performance implementation of CR tailored for these problems.

In order to demonstrate how structure arises in the context of dynamic optimization, the next section gives a brief overview of one common approach for solving general nonlinear optimization problems, the interior-point method. The next section also shows how linear algebra becomes important for solving nonlinear optimization problems. We then illustrate how we can exploit structure in the linear systems arising from dynamic optimization problems in Section 6.3. Sections 6.4 and 6.5 describe the two algorithms used in this work and Sections 6.6 and 6.7 provide a more detailed analysis of the fill-in produced by the algorithms as well as the conditioning of the algorithms.

6.2 Interior-point Method

The interior-point algorithm described here is based on the one described in [47]. This algorithm solves problems with the following form

$$\min_{x \in \mathbb{R}} \quad f(x) \tag{6.1a}$$

$$\text{s.t.} \quad c(x) = 0 \tag{6.1b}$$

$$x \geq 0 \tag{6.1c}$$

where $f : \mathbb{R}^n \to \mathbb{R}$ and $c : \mathbb{R}^n \to \mathbb{R}^m$ are nonlinear functions of $x \in \mathbb{R}^n$. The first step of the algorithm is to move the inequality constraints into the objective function, multiplied by a barrier parameter μ. This forms the following barrier problem

$$\min_{x \in \mathbb{R}} \quad f(x) - \mu \sum_{i=1}^{n} \ln(x_i) \tag{6.2a}$$

$$\text{s.t.} \quad c(x) = 0 \tag{6.2b}$$

The overall idea of an interior-point algorithm is to solve a sequence of barrier problems (6.2) with decreasing values of μ. As μ approaches zero, the solution of (6.2) approaches the solution of our original problem (6.1). In order to solve (6.2), we must satisfy the Karush-Kuhn-Tucker (KKT) conditions, or first order optimality conditions, given by

$$\nabla_x \mathcal{L}(x, \lambda, \nu) = \nabla_x f(x) + \nabla_x c(x)\lambda - \nu = 0 \tag{6.3a}$$

$$c(x) = 0 \tag{6.3b}$$

$$XVe - \mu e = 0 \tag{6.3c}$$

where $X = diag(x)$ and $V = diag(\nu)$. $\lambda \in \mathbb{R}^m$ is the vector of Lagrange multipliers for the equality constraints and $\nu \in \mathbb{R}^n$ represents the multipliers for the bound constraints. The KKT conditions (6.3) form a system of nonlinear equations which can be solved using a modified version of Newton's method. For a fixed value of μ, the following linear system must be solved at each iteration k of Newton's method

$$\begin{bmatrix} W_k & A_k^T & -I \\ A_k & 0 & 0 \\ V_k & 0 & X_k \end{bmatrix} \begin{bmatrix} \Delta x_k \\ \Delta \lambda_k \\ \Delta \nu_k \end{bmatrix} = - \begin{bmatrix} \nabla_x f(x_k) + A_k^T \lambda_k - \nu_k \\ c(x_k) \\ X_k V_k e - \mu e \end{bmatrix} \tag{6.4}$$

where $W_k := \nabla_{xx} \mathcal{L}(x_k, \lambda_k, \nu_k)$ and $A_k^T := \nabla_x c(x_k)$. However, because Newton's method will require a number of iterations, the nonsymmetric linear system (6.4) is typically reduced to a symmetric form which allows for more efficient solution using sparse, symmetric linear algebra solvers. This is done by eliminating the last block row resulting in the augmented form of the KKT system shown below.

$$\begin{bmatrix} W_k + \Sigma_k & A_k^T \\ A_k & 0 \end{bmatrix} \begin{bmatrix} \Delta x_k \\ \Delta \lambda_k \end{bmatrix} = - \begin{bmatrix} r_k \\ c(x_k) \end{bmatrix} \tag{6.5}$$

where $\Sigma_k = X_k^{-1} V_k$ and $r_k = \nabla_x f(x_k) + A_k^T \lambda_k + \mu X_k^{-1} e$. Throughout the rest of this chapter we refer to system (6.5) as the 'KKT system' and the left most term as the 'KKT matrix'. We also refer to the vector multiplied by the KKT matrix as the 'variable vector'. For large-scale NLP problems, solving the KKT system (6.5) becomes the dominant time-consuming step in the overall interior point algorithm. Furthermore, the system may become prohibitively large to store the full KKT matrix in memory. For this reason, this work investigates ways to speed up the solution of the KKT system using several parallel linear algebra techniques. These techniques exploit certain structures in the KKT matrix that arise from applying a discretization scheme to solve a dynamic optimization problem. The next section describes how these structures appear.

6.3 Exploiting Structure in the KKT System

Consider the simplified version of problem (1.2) below.

$$\begin{aligned} \min & \sum_{k=1}^{N-1} (\Phi(x_k, y_k, u_k)) + \Psi(x_N) \\ \text{s.t.} \quad & x_{i+1} = g(x_i, y_i, u_i), \quad i = 1, \ldots, N-1 \\ & x_1 = constant \end{aligned} \tag{6.6}$$

The KKT matrix for this problem can be written as

$$\begin{bmatrix} W & f_w^T \\ f_w & 0 \end{bmatrix} \begin{bmatrix} w \\ \lambda \end{bmatrix} = - \begin{bmatrix} r \\ c(\alpha) \end{bmatrix} \tag{6.7}$$

where $w^T = [x^T, y^T, u^T]$ represents the primal variables. For notation simplicity we have dropped the subscript k referring to the Newton method iteration and the Δ in the variable

vector. Recall that our model has been discretized using N discretization points, therefore if we expand the variable vector it has the form

$$z^T = [x_2^T, \ldots, x_N^T, y_1^T, \ldots, y_{N-1}^T, u_1^T, \ldots, u_{N-1}^T, \lambda_1^T, \ldots, \lambda_{N-1}^T] = [w^T, \lambda^T] \tag{6.8}$$

If we rearrange the variable vector by grouping together primal and dual variables corresponding to the same discretization point we get

$$z^T = [y_1^T, u_1^T, \lambda_1^T, x_2^T, y_2^T, u_2^T, \lambda_2^T, \ldots, x_{N-1}^T, y_{N-1}^T, u_{N-1}^T, \lambda_{N-1}^T, x_N^T] \tag{6.9}$$

The KKT matrix corresponding with this rearranged variable vector has the following block tridiagonal structure

$$\begin{bmatrix}
W_{y_1y_1} & W_{y_1u_1} & f_{y_1}^T & & & & & & & & & \\
W_{u_1y_1} & W_{u_1u_1} & f_{u_1}^T & & & & & & & & & \\
f_{y_1} & f_{u_1} & 0 & -I & & & & & & & & \\
 & & -I & W_{x_2x_2} & W_{x_2y_2} & W_{x_2u_2} & f_{x_2}^T & & & & & \\
 & & & W_{y_2x_2} & W_{y_2y_2} & W_{y_2u_2} & f_{y_2}^T & & & & & \\
 & & & W_{u_2x_2} & W_{u_2y_2} & W_{u_2u_2} & f_{u_2}^T & & & & & \\
 & & & f_{x_2} & f_{y_2} & f_{u_2} & 0 & -I & & & & \\
 & & & & & & -I & W_{x_3x_3} & W_{x_3y_3} & W_{x_3u_3} & f_{x_3}^T & \\
 & & & & & & & W_{y_3x_3} & W_{y_3y_3} & W_{y_3u_3} & f_{y_3}^T & \\
 & & & & & & & W_{u_3x_3} & W_{u_3y_3} & W_{u_3u_3} & f_{u_3}^T & \\
 & & & & & & & f_{x_3} & f_{y_3} & f_{u_2} & 0 & \ddots \\
 & & & & & & & & & & & -I \\
 & & & & & & & & & & -I & W_{x_Nx_N}
\end{bmatrix} \tag{6.10}$$

where the blocks have been outlined for clarity. This is the first of two structures that will be exploited in this work. The second structure arises by noticing that the matrix (6.10)

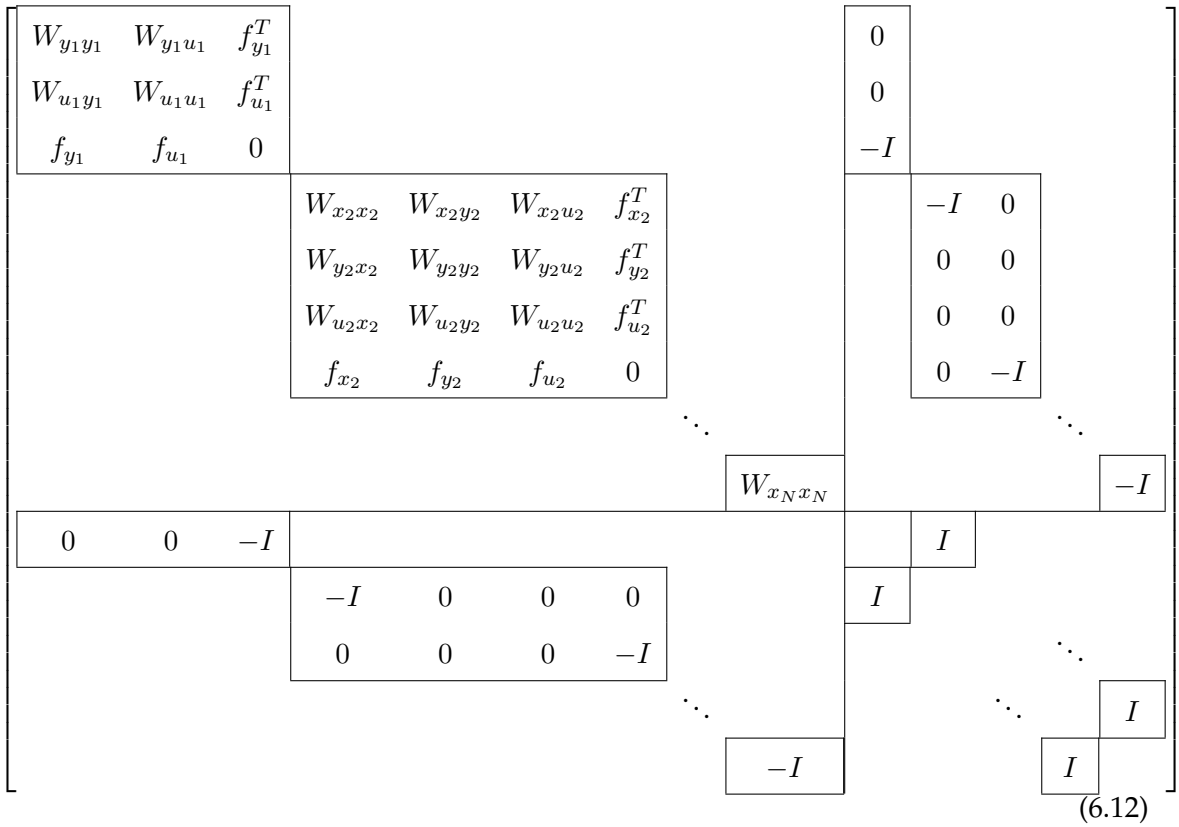
is almost block diagonal except for the off-diagonal identity terms. These identities arise from the coupling in the dynamic model from one time step to the next. We can move these coupling terms to the borders of the matrix by introducing decoupling variables v and γ and the following equations to the KKT system

$$v_i = x_{i+1},\ i = 1, \ldots, N-1$$
$$\gamma_i = \lambda_{i-1},\ i = 2, \ldots, N$$

The decoupling variables are added to the end of the variable vector and are again grouped by discretization point such that

$$z^T = [y_1^T, u_1^T, \lambda_1^T, x_2^T, y_2^T, u_2^T, \lambda_2^T, \ldots, x_{N-1}^T, y_{N-1}^T, u_{N-1}^T, \lambda_{N-1}^T, x_N^T, v_1^T, \gamma_2^T, v_2^T, \ldots, v_{N-1}^T, \gamma_N^T] \tag{6.11}$$

Our KKT matrix now has the following bordered block diagonal structure



(6.12)

Notice that the diagonal blocks from our original KKT matrix are now decoupled and block diagonal. Additionally, the blocks outlined in the right and bottom borders of the matrix are also block diagonal. These are both features that we can exploit.

To take advantage of the block tridiagonal structure in (6.10) we apply an algorithm called cyclic reduction. For the bordered block diagonal structure in (6.12) we apply Schur complement decomposition. Both of these techniques are parallelizable. These algorithms are described in the next two sections.

6.4 Cyclic Reduction

Cyclic reduction is a parallel algorithm for solving tridiagonal or block tridiagonal linear systems where half of the unknowns are eliminated at each pass of the algorithm. To

illustrate this consider the linear system shown below.

$$\begin{bmatrix} A_1 & C_1 & & & & \\ B_2 & A_2 & C_2 & & & \\ & B_3 & A_3 & C_3 & & \\ & & \ddots & \ddots & \ddots & \\ & & & B_N & A_N \end{bmatrix} \begin{bmatrix} z_1 \\ z_2 \\ z_3 \\ \vdots \\ z_N \end{bmatrix} = \begin{bmatrix} r_1 \\ r_2 \\ r_3 \\ \vdots \\ r_N \end{bmatrix} \tag{6.13}$$

We assume that A_j, B_j, and C_j are matrices rather than scalar values. We refer to each row in the tridiagonal matrix as a block row. To derive the CR operations for each pass, consider the first three rows of pass i:

$$\begin{aligned} A_1^{(i)} z_1^{(i)} + C_1^{(i)} z_2^{(i)} &= r_1^{(i)} \\ B_2^{(i)} z_1^{(i)} + A_2^{(i)} z_2^{(i)} + C_2^{(i)} z_3^{(i)} &= r_2^{(i)} \\ B_3^{(i)} z_2^{(i)} + A_3^{(i)} z_3^{(i)} + C_3^{(i)} z_4^{(i)} &= r_3^{(i)}. \end{aligned}$$

We use these equations to write $z_1^{(i)}$ and $z_3^{(i)}$ in terms of $z_2^{(i)}$ and $z_4^{(i)}$.

$$\begin{aligned} z_1^{(i)} &= (A_1^{(i)})^{-1}(r_1^{(i)} - C_1^{(i)} z_2^{(i)}) \\ z_3^{(i)} &= (A_3^{(i)})^{-1}(r_3^{(i)} - B_3^{(i)} z_2^{(i)} - C_3^{(i)} z_4^{(i)}) \end{aligned}$$

and substitute to get:

$$[A_2^{(i)} - B_2^{(i)}(A_1^{(i)})^{-1}C_1^{(i)} - C_2^{(i)}(A_3^{(i)})^{-1}B_3^{(i)}]z_2^{(i)} - C_2^{(i)}(A_3^{(i)})^{-1}C_3^{(i)} z_4^{(i)} = r_2^{(i)} - B_2^{(i)}(A_1^{(i)})^{-1} r_1^{(i)} - C_2^{(i)}(A_3^{(i)})^{-1} r_3^{(i)}$$

which is rewritten for the next pass as:

$$A_1^{(i+1)} z_1^{(i+1)} + C_1^{(i+1)} z_2^{(i+1)} = r_1^{(i+1)}$$

Similarly, we can write z_{2j-1} and z_{2j+1} in terms of z_{2j}, z_{2j-2}, and z_{2j+2} as follows:

$$\begin{aligned} B_{2j-1}^{(i)} z_{2j-2}^{(i)} + A_{2j-1}^{(i)} z_{2j-1}^{(i)} + C_{2j-1}^{(i)} z_{2j}^{(i)} &= r_{2j-1}^{(i)} \\ B_{2j}^{(i)} z_{2j-1}^{(i)} + A_{2j}^{(i)} z_{2j}^{(i)} + C_{2j+1}^{(i)} z_{2j+1}^{(i)} &= r_{2j}^{(i)} \\ B_{2j+1}^{(i)} z_{2j}^{(i)} + A_{2j+1}^{(i)} z_{2j+1}^{(i)} + C_{2j+1}^{(i)} z_{2j+2}^{(i)} &= r_{2j+1}^{(i)}. \end{aligned}$$

which leads to:

$$z_{2j-1}^{(i)} = (A_{2j-1}^{(i)})^{-1}(r_{2j-1}^{(i)} - B_{2j-1}^{(i)} z_{2j-2}^{(i)} - C_{2j-1}^{(i)} z_{2j}^{(i)}) \quad (6.14)$$

$$z_{2j+1}^{(i)} = (A_{2j+1}^{(i)})^{-1}(r_{2j+1}^{(i)} - B_{2j+1}^{(i)} z_{2j}^{(i)} - C_{2j+1}^{(i)} z_{2j+2}^{(i)}) \quad (6.15)$$

and we substitute to get:

$$-B_{2j}^{(i)}(A_{2j-1}^{(i)})^{-1}B_{2j-1}^{(i)} z_{2j-2}^{(i)} + [A_{2j}^{(i)} - B_{2j}^{(i)}(A_{2j-1}^{(i)})^{-1}C_{2j-1}^{(i)} - C_{2j}^{(i)}(A_{2j+1}^{(i)})^{-1}B_{2j+1}^{(i)}] z_{2j}^{(i)}$$
$$-C_{2j}^{(i)}(A_{2j+1}^{(i)})^{-1}C_{2j+1}^{(i)} z_{2j+2}^{(i)} = r_{2j}^{(i)} - B_{2j}^{(i)}(A_{2j-1}^{(i)})^{-1} r_{2j-1}^{(i)} - C_{2j}^{(i)}(A_{2j+1}^{(i)})^{-1} r_{2j+1}^{(i)}$$

which is rewritten for the next pass $i+1$ as:

$$B_j^{(i+1)} z_{j-1}^{(i+1)} + A_j^{(i+1)} z_j^{(i+1)} + C_j^{(i+1)} z_{j+1}^{(i+1)} = r_j^{(i+1)} \quad (6.16)$$

where

$$\begin{aligned} B_j^{(i+1)} &= B_{2j}^{(i)}(A_{2j-1}^{(i)})^{-1}B_{2j-1}^{(i)} \\ A_j^{(i+1)} &= [A_{2j}^{(i)} - B_{2j}^{(i)}(A_{2j-1}^{(i)})^{-1}C_{2j-1}^{(i)} - C_{2j}^{(i)}(A_{2j+1}^{(i)})^{-1}B_{2j+1}^{(i)}] \\ C_j^{(i+1)} &= C_{2j}^{(i)}(A_{2j+1}^{(i)})^{-1}C_{2j+1}^{(i)} \end{aligned} \quad (6.17)$$

As a result this introduces zeroes into the tridiagonal structure leading to the matrix:

$$\begin{bmatrix} A_1^{(i)} & C_1^{(i)} & & & & & & & \\ & A_1^{(i+1)} & 0 & C_1^{(i+1)} & & & & & \\ & B_3^{(i)} & A_3^{(i)} & C_3^{(i)} & & & & & \\ & B_2^{(i+1)} & 0 & A_2^{(i+1)} & 0 & C_2^{(i+1)} & & & \\ & & & B_5^{(i)} & A_5^{(i)} & C_5^{(i)} & & & \\ & & & B_3^{(i+1)} & 0 & A_3^{(i+1)} & 0 & C_3^{(i+1)} & \\ & & & & & \ddots & \ddots & \ddots & \\ & & & & & & B_{N/2}^{(i+1)} & 0 & A_{N/2}^{(i+1)} \end{bmatrix} \quad (6.18)$$

if N is even. Notice that the variables of pass $i+1$ correspond to the even variables of pass i, or $z_j^{(i+1)} = z_{2j}^{(i)}$.

With a permutation of the block rows and columns in (6.18) we can then solve independently for the even variables from the even rows and back-subtitute to recover the odd variables from the odd equations. The system at pass $i+1$ to solve for the even variables of pass i will again have a block tridiagonal structure as shown below

$$\begin{bmatrix} A_1^{(i+1)} & C_1^{(i+1)} & & & & \\ B_2^{(i+1)} & A_2^{(i+1)} & C_2^{(i+1)} & & & \\ & B_3^{(i+1)} & A_3^{(i+1)} & C_3^{(i+1)} & & \\ & & \ddots & \ddots & \ddots & \\ & & & B_{N/2}^{(i+1)} & A_{N/2}^{(i+1)} \end{bmatrix} \begin{bmatrix} z_1^{(i+1)} \\ z_2^{(i+1)} \\ z_3^{(i+1)} \\ \vdots \\ z_{N/2}^{(i+1)} \end{bmatrix} = \begin{bmatrix} r_1^{(i+1)} \\ r_2^{(i+1)} \\ r_3^{(i+1)} \\ \vdots \\ r_{N/2}^{(i+1)} \end{bmatrix}$$

but this system has only half as many block rows as the previous pass. We can continue to apply CR passes until only a single block row is left or until the system is small enough to solve directly.

During each pass the operations on the odd or even rows can be done independently of the other odd or even rows respectively. For example, equations (6.14) and (6.15) require the odd $A^{(i)}$ blocks to be factorized and each of these block factorizations can be done independently, and therefore in parallel. Similarly, the calculation of the $A_j^{(i+1)}$, $B_j^{(i+1)}$, and $C_j^{(i+1)}$ blocks in equation (6.16) can be done independently for each j.

6.5 Schur Complement Decomposition

The second algorithm we investigate is blocked Cholesky decomposition or Schur complement decomposition. This algorithm can be used to solve bordered block diagonal linear systems of the form shown below

$$\left[\begin{array}{cccc|c} A_1 & & & & G_1 \\ & A_2 & & & G_2 \\ & & \ddots & & \vdots \\ & & & A_N & G_N \\ \hline G_1^T & G_2^T & \dots & G_N^T & J \end{array}\right] \left[\begin{array}{c} z_1 \\ z_2 \\ \vdots \\ z_N \\ \hline \eta \end{array}\right] = \left[\begin{array}{c} r_1 \\ r_2 \\ \vdots \\ r_N \\ \hline b \end{array}\right] \tag{6.19}$$

The first step in the algorithm is to perform a block elimination on the blocks A_1 through A_N and substitute them into the bottom equation.

$$(J - \sum_{i=1}^{N} G_i^T A_i^{-1} G_i)\eta = b - \sum_{i=1}^{N} G_i^T A_i^{-1} r_i \tag{6.20}$$

This forms the Schur complement matrix S.

$$S = J - \sum_{i=1}^{N} G_i^T A_i^{-1} G_i \tag{6.21}$$

The next step is to factorize S and solve for the variables η.

$$\eta = S^{-1}(b - \sum_{i=1}^{N} G_i^T A_i^{-1} r_i) \tag{6.22}$$

Once η is known the remaining unknowns can be calculated from

$$z_i = A_i^{-1}(r_i - G_i\eta),\ i = 1, \dots, N \tag{6.23}$$

Each step of this algorithm can be done in parallel. In the first step, each of the A_i blocks can be factored independently of the others. The Schur complement can be pieced together in parallel because the border G blocks are block diagonal. The final step of solving for the z_i variables is trivially parallelizable. It turns out that factorizing the Schur complement can also be done in parallel because the Schur complement has a block tridiagonal structure as shown below.

$$S = \begin{bmatrix} S_1 & I & & & \\ I & S_2 & & & \\ & & \ddots & I & \\ & & I & S_N \end{bmatrix} \tag{6.24}$$

This means that cyclic reduction can be used to factorize the Schur complement. Previous work applying the Schur complement approach to dynamic optimization has worked with S as a whole and applied a serial, sparse symmetric linear algebra solver to factor S [85, 92, 109]. Our approach factors S in parallel and exploits additional structure compared to what has been done in the literature.

6.6 Sparsity Analysis

Maintaining sparsity is an important strategy for solving large-scale linear systems. In this section we prove that the fill-in produced by applying CR to a sparse KKT system will be limited to certain parts of the matrix and that much of the sparsity of the original system will be preserved through the CR recursions.

Recall the block tridiagonal structure of the KKT matrix (6.10) and define $E = [I\ 0\ 0]$, $F_k = [f_{x_k}\ f_{y_k}\ f_{u_k}]$, $z_k^0 = [x_k^T\ y_k^T\ u_k^T\ \lambda_k^T]^T$, and

$$W_k = \begin{bmatrix} W_{x_k x_k} & W_{x_k y_k} & W_{x_k u_k} \\ W_{y_k x_k} & W_{y_k y_k} & W_{y_k u_k} \\ W_{u_k x_k} & W_{u_k y_k} & W_{u_k u_k} \end{bmatrix} \tag{6.25}$$

then the tridiagonal blocks can be rewritten in the following form

$$A_k^0 = \begin{bmatrix} W_k & (F_k)^T \\ F_k & 0 \end{bmatrix}, \quad B_k^0 = \begin{bmatrix} 0 & -E^T \\ 0 & 0 \end{bmatrix}, \quad C_k^0 = \begin{bmatrix} 0 & 0 \\ -E & 0 \end{bmatrix} = (B_k^0)^T \tag{6.26}$$

where the superscript '0' refers to the initial form of the blocks before any CR recursions. Also recall, $x_k \in \mathbb{R}^{n_x}, y_k \in \mathbb{R}^{n_y}, u_k \in \mathbb{R}^{n_u}, \lambda_k \in \mathbb{R}^m, F \in \mathbb{R}^{m\times(n_x+n_y+n_u)}$, and $A_k \in \mathbb{R}^{(n_x+n_y+n_u+m)\times(n_x+n_y+n_u+m)}$.

Now let $E \in \mathbb{R}^{n_x\times(n_x+n_y+n_u)}$ take the following form $E = [D\ 0]$. Assume $D \in \mathbb{R}^{n_x\times n_x}$ is nonsingular and let

$$A^{-1} = \begin{bmatrix} A_{11} & A_{12}^T & A_{13}^T \\ A_{12} & A_{22} & A_{23}^T \\ A_{13} & A_{23} & A_{33} \end{bmatrix}$$

First, let us investigate the fill-in for the off-diagonal blocks from one CR pass to the next. For the off-diagonal block $B_k^{(i+1)}$ we have:

$$\begin{aligned} B_k^{(i+1)} &= -B_{2k}^{(i)}(A_{2k-1}^{(i)})^{-1}B_{2k-1}^{(i)} \\ &= -\begin{bmatrix} -D^{(i)T}A_{13} & -D^{(i)T}A_{23} & -D^{(i)T}A_{33} \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}\begin{bmatrix} 0 & 0 & -D^{(i)T} \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \\ &= -\begin{bmatrix} 0 & 0 & D^{(i)T}A_{13}D^{(i)T} \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \end{aligned}$$

similarly for the off-diagonal block $C_k^{(i+1)}$ we have:

$$
\begin{aligned}
C_k^{(i+1)} &= -C_{2k}^{(i)}(A_{2k+1}^{(i)})^{-1}C_{2k+1}^{(i)} \\
&= -\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ -D^{(i)}A_{11} & -D^{(i)}A_{12}^T & -D^{(i)}A_{13}^T \end{bmatrix}\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ -D^{(i)} & 0 & 0 \end{bmatrix} \\
&= -\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ D^{(i)}A_{13}^T D^{(i)} & 0 & 0 \end{bmatrix}
\end{aligned}
$$

Notice that $C_k^{(i+1)} = B_k^{(i+1)T}$ and $D^{(i+1)} = D^{(i)}A_{13}^T D^{(i)}$ and thus any fill-in that occurs in the off-diagonal blocks during the CR recursions is limited to the $(n_x \times n_x)$ sized D blocks.

If we look at the fill-in for the diagonal block $A_k^{(i+1)}$ we have:

$$
\begin{aligned}
A_k^{(i+1)} &= A_{2k}^{(i)} - B_{2k}^{(i)}(A_{2k-1}^{(i)})^{-1}C_{2k-1}^{(i)} - C_{2k}^{(i)}(A_{2k+1}^{(i)})^{-1}B_{2k+1}^{(i)} \\
&= A_{2k}^{(i)} - \begin{bmatrix} 0 & 0 & -D^{(i)T} \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}\begin{bmatrix} A_{11} & A_{12}^T & A_{13}^T \\ A_{12} & A_{22} & A_{23}^T \\ A_{13} & A_{23} & A_{33} \end{bmatrix}\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ -D^{(i)} & 0 & 0 \end{bmatrix} \\
&\quad - \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ -D^{(i)} & 0 & 0 \end{bmatrix}\begin{bmatrix} A_{11} & A_{12}^T & A_{13}^T \\ A_{12} & A_{22} & A_{23}^T \\ A_{13} & A_{23} & A_{33} \end{bmatrix}\begin{bmatrix} 0 & 0 & -D^{(i)T} \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \\
&= A_{2k}^{(i)} - \begin{bmatrix} D^{(i)T}A_{33}D^{(i)} & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} - \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & D^{(i)}A_{11}D^{(i)T} \end{bmatrix}
\end{aligned} \tag{6.27}
$$

Thus, regardless of the original sparsity of $A_{2k}^{(i)}$, the fill-in for $A_k^{(i+1)}$ is restricted to the two $(n_x \times n_x)$ blocks in the top-left and bottom-right corners. This is significant because for dynamic optimization problems the diagonal blocks of the KKT matrix in symmetric indefinite augmented form are typically very sparse and $n_x \ll n_u + n_y + m$.

In contrast to the analysis provided above, we note that cyclic reduction applied to factorize the Schur complement will not see any benefit from sparsity preservation because the diagonal blocks of the Schur complement are dense.

6.7 Numerical Conditioning

In order to use CR in practice we have to ensure that the diagonal blocks, $A_k^{(i)}$, remain nonsingular and well-conditioned enough to perform a stable reduction in each pass. In [105], a key point of using the dense normal form of the KKT matrix was that the diagonal blocks are positive definite and they remain positive definite for every pass of CR. This provides the necessary condition that the elimination using the diagonal block is well-defined for each CR pass. Despite the positive definiteness of the diagonal blocks, the author still observed some conditioning problems but the source of those difficulties was not clear.

Our approach works on the symmetric indefinite augmented form of the KKT matrix, instead of the normal form. This allows sparse factorization routines to be used on much larger problems. Following the conditioning proof from [105] and describing the CR algorithm in terms of Schur complements as was done in [110], we can prove the following theorem.

Theorem 6.7.1. *Assume that K^0 is nonsingular. Then the block tridiagonal matrix $K^{(i)}$ remains nonsingular for every CR pass i as long as the diagonal blocks $A_k^{(i)}$ are nonsingular.*

Proof. With K^0 nonsingular, we begin the induction by representing our block tridiagonal matrix at pass i as:

$$K^{(i)} = \begin{bmatrix} A_1 & C_1 & & & \\ B_2 & A_2 & C_2 & & \\ & B_3 & A_3 & C_3 & \\ & & \ddots & \ddots & \ddots \\ & & & B_{N_i} & A_{N_i} \end{bmatrix} \tag{6.28}$$

At pass i we assume that $K^{(i)}$ is nonsingular and $\det(K^{(i)}) \neq 0$. By permuting the rows and columns of $K^{(i)}$ to group the odd block rows and the even block rows we get the following structure

$$\tilde{K}^{(i)} = \left[\begin{array}{ccccc|ccccc} A_1 & & & & & C_1 & & & & \\ & A_3 & & & & B_3 & C_3 & & & \\ & & \ddots & & & & B_5 & \ddots & & \\ & & & A_{N_i-2} & & & & \ddots & C_{N_i-2} & \\ & & & & A_{N_i} & & & & & B_{N_i} \\ \hline B_2 & C_2 & & & & A_2 & & & & \\ & B_4 & C_4 & & & & A_4 & & & \\ & & \ddots & \ddots & & & & \ddots & & \\ & & & B_{N_i-1} & C_{N_i-1} & & & & A_{N_i-1} & \end{array}\right] \tag{6.29}$$

and $\det(\tilde{K}^{(i)}) = \pm\det(K^{(i)})$. Since we assume the odd A_k blocks are nonsingular we perform block Gaussian elimination to eliminate the blocks in the bottom left corner of $\tilde{K}^{(i)}$ resulting in the matrix $\hat{K}^{(i)}$. This forms the Schur complement in the bottom right corner.

$$\hat{K}^{(i)} = \left[\begin{array}{ccc|cccc} A_1 & & & C_1 & & & \\ & A_3 & & B_3 & C_3 & & \\ & & \ddots & & B_5 & \ddots & \\ & & & & & \ddots & C_{N_i-2} \\ & & A_{N_i} & & & & B_{N_i} \\ \hline & & & A_1^{(i+1)} & C_1^{(i+1)} & & \\ & & & B_2^{(i+1)} & A_2^{(i+1)} & C_2^{(i+1)} & \\ & 0 & & & \ddots & \ddots & \ddots \\ & & & & & B_{N_{i+1}}^{(i+1)} & A_{N_{i+1}}^{(i+1)} \end{array}\right] \tag{6.30}$$

Because $A_k^{(i+1)}, B_k^{(i+1)},$ and $C_k^{(i+1)}$ are given in (6.17) the Schur complement is identical to $K^{(i+i)}$. Because of the block diagonal structure of the odd A_k blocks, the Gaussian elimination steps, performed by adding multiples of one row to another, leaves the determinant unchanged.

$$\det\left(\tilde{K}^{(i)}\right) = \det\left(\hat{K}^{(i)}\right) \neq 0 \tag{6.31}$$

Furthermore, $\hat{K}^{(i)}$ can be recognized as an upper triangular matrix for which the determinant can be written as a product of the diagonal blocks.

$$\det\left(\hat{K}^{(i)}\right) = \left(\prod_{k\in\text{odd}} \det\left(A_k^{(i)}\right)\right) \det\left(K^{(i+1)}\right) \neq 0 \tag{6.32}$$

From this expression we know that $K^{(i+1)}$ is nonsingular. We note that we can repeat the same Schur complement operation for $K^{(i+1)}$ as long as the diagonal $A_k^{(i+1)}$ blocks are nonsingular. □

We know from the analysis in the previous section that the structure of $A_k^{(i+1)}$ will be the same as $A_{2k}^{(i)}$ with fill-in in the top left and bottom right corners

$$
\begin{aligned}
A_k^{(i+1)} &= \begin{bmatrix} Q_k^{(i)} & (F_k)^T \\ F_k & \Delta^{(i)} \end{bmatrix} - \begin{bmatrix} D^{(i)T} A_{33}^{(i)} D^{(i)} & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & D^{(i)} A_{11}^{(i)} D^{(i)T} \end{bmatrix} \\
&= \begin{bmatrix} Q_k^{(i)} & (F_k)^T \\ F_k & \Delta^{(i)} \end{bmatrix} - \begin{bmatrix} \phi^{(i)} & 0 \\ 0 & \tilde{\Delta}^{(i)} \end{bmatrix} = \begin{bmatrix} Q_k^{(i+1)} & (F_k)^T \\ F_k & \Delta_k^{(i+1)} \end{bmatrix}
\end{aligned} \tag{6.33}
$$

where ϕ and Δ are general representations of the fill-in at any CR pass and $A_k^{(0)}$ is defined as in (6.26). Now let

$$
A = \begin{bmatrix} Q & (F)^T \\ F & \Delta \end{bmatrix} \tag{6.34}
$$

where we have dropped the k subscript and (i) superscript to simplify the notation. We now state sufficient conditions that allow A to be nonsingular.

Theorem 6.7.2. *The matrix A is nonsingular when F is full rank and Q is sufficiently positive definite (i.e., $Q - \tilde{F}_1^T \Lambda_+^{-1} \tilde{F}_1$ is positive definite).*

Proof. An eigenvalue decomposition on Δ leads to:

$$
\Delta = X \Lambda X^T \tag{6.35}
$$

where $X^T = X^{-1}$. Furthermore, we can find a permutation matrix P such that

$$
P X^T \Delta X P^T = \begin{bmatrix} \Lambda_+ & & \\ & \Lambda_- & \\ & & 0 \end{bmatrix} \tag{6.36}
$$

where Λ_+ and Λ_- are diagonal matrices with the positive and negative eigenvalues of Δ respectively. Defining

$$
H = \begin{bmatrix} I & \\ & PX^T \end{bmatrix} \tag{6.37}
$$

we perform a congruence transformation on A using H and obtain the following matrix

$$\begin{aligned} HAH^T &= \begin{bmatrix} I & \\ & PX^T \end{bmatrix} \begin{bmatrix} Q & (F)^T \\ F & \Delta \end{bmatrix} \begin{bmatrix} I & \\ & XP^T \end{bmatrix} \\ &= \begin{bmatrix} Q & \tilde{F}_1 & \tilde{F}_2 & \tilde{F}_3 \\ \tilde{F}_1^T & \Lambda_+ & & \\ \tilde{F}_2^T & & \Lambda_- & \\ \tilde{F}_3^T & & & 0 \end{bmatrix} \end{aligned} \tag{6.38}$$

where $\tilde{F} = FXP^T = \begin{bmatrix} \tilde{F}_1 & \tilde{F}_2 & \tilde{F}_3 \end{bmatrix}$ and $\tilde{F}$ is also full rank. From this it follows that $\tilde{F}_1, \tilde{F}_2,$ and $\tilde{F}_3$ are also full rank. From Sylvester's law of inertia, if HAH^T is nonsingular then A is also nonsingular. To determine conditions for nonsingular HAH^T we define the submatrices:

$$K_b = \begin{bmatrix} Q & \tilde{F}_1 \\ \tilde{F}_1^T & \Lambda_+ \end{bmatrix} \quad \text{and} \quad K_a = \begin{bmatrix} K_b & \tilde{F}_2 \\ \tilde{F}_2^T & \Lambda_- \end{bmatrix}$$

We note that K_b is positive definite because both Λ_+ and the Schur complement $(Q - \tilde{F}_1^T \Lambda_+^{-1} \tilde{F}_1)$ are positive definite. Next we see that K_a is nonsingular because $\tilde{F}_2$ is full rank and the Schur complement $(\Lambda_- - \tilde{F}_2^T K_b^{-1} \tilde{F}_2)$ is negative definite. The latter follows because Λ_- is negative definite, $\tilde{F}_2$ is full rank and K_b is positive definite.

Finally we see that HAH^T and hence A are nonsingular because $\tilde{F}_3$ is full rank, K_a is nonsingular and hence the Schur complement $-\tilde{F}_3^T K_a^{-1} \tilde{F}_3$ is nonsingular. □

Because we cannot guarantee that Q will be sufficiently positive definite, it may be necessary to regularize Q by adding a positive term along its diagonal such that $(Q + \delta I - \tilde{F}_1^T \Lambda_+^{-1} \tilde{F}_1)$ is positive definite. This regularization step may be required to complete any of the CR passes. However, in practice we noticed that for our ill-conditioned test problems, regularizing Q during the first CR pass was sufficient to keep the subsequent CR passes nonsingular and well-conditioned.

6.8 Concluding Remarks

This chapter has demonstrated how structure arises in dynamic optimization problems and presents two linear algebra algorithms for exploiting this structure. While the Schur complement approach has been widely studied in the context of dynamic optimization, the cyclic reduction algorithm has not been. To the best of our knowledge, CR has never been applied to solve dynamic optimization problems in the way proposed here, by operating on the symmetric indefinite augmented form of the KKT matrix. Compared to the Schur complement approach, CR does not require the addition of decoupling variables and constraints and therefore operates on smaller linear systems.

This chapter also presented a careful analysis of CR in the context of dynamic optimization with respect to sparsity preservation and numerical conditioning. We have shown that the fill-in caused by CR passes is limited to certain regions of the tridiagonal blocks preserving much of the sparsity of the original system. In addition we derived sufficient conditions for non-singular and well-conditioned CR passes. The computational performance of both algorithms is studied in the next chapter.

Chapter 7

Computational Performance of Parallel Solvers

In this chapter we investigate the performance of the algorithms presented in Chapter 6. We first demonstrate the Schur complement algorithm on the dynamic models studied in Chapter 3. These results illustrate several challenges that arise when parallel algorithms are applied to dynamic optimization problems. The second half of this chapter presents results for both the cyclic reduction algorithm and Schur complement decomposition applied to structured test matrices.

7.1 Schur Complement Decomposition in PIPS-NLP

For this study we make use of a parallel interior point solver called PIPS-NLP (Parallel Interior Point for NLP)[93]. PIPS-NLP is an open-source solver developed by researchers at Argonne National Laboratory which has been designed to handle structured nonlinear optimization problems. It is implemented in C and C++ and uses MPI for parallelization. PIPS-NLP includes an implementation of the Schur complement decomposition approach for solving the linear systems arising in the interior point algorithm with a bordered block-diagonal, or arrowhead, structure. However, it does not exploit any additional structure in the Schur complement, rather it solves it as a whole.

Structure is communicated to PIPS-NLP by dividing optimization problems into several pieces and using suffixes to define the coupling between the pieces. Suffixes are components of an algebraic modeling language for communicating additional problem informa-

tion to a solver. Each piece of the optimization problem is written to a separate NL file. In the case of dynamic optimization, the problem is divided at finite element boundaries.

In the following case studies we compare the performance of PIPS-NLP against the serial interior point solver IPOPT. For both solvers MA57 was used for matrix factorizations. Solver options for the interior point algorithms were set to equivalent values where possible. The tests were run on a desktop computer running Linux with 8 processors (4 cores with hyper-threading). The serial version of PIPS-NLP, which does not exploit any problem structre, is also included in the comparison. The same initial guess is used for both solvers.

7.1.1 CSTR Case Study

The first case study was performed on the 3-state CSTR model introduced in Chapter 3. Again, the model equations can be found in Appendix A.1. For this case study we solve a single instance of the MHE state estimation problem i.e. a single snapshot in time. We examine the parallel performance of PIPS-NLP as the size of the problem and the number of processors are varied. The size of the problem is determined by the discretization scheme applied, in this case orthogonal collocation over finite elements with 3 collocation points. We vary the problem size by varying the number of finite elements (NFE) used in the discretization. Table 7.1 shows the solution times and Table 7.2 shows the problem sizes and solver iteration counts.

For the parallel runs, the model was divided such that the number of NL files was equal to the number of processors being used to solve the problem. This means that for most cases each NL file contained a piece of the model over several finite elements. This is significant because additional coupling variables and constraints had to be added to the model depending on the number of pieces. Thus, as shown in Table 7.2 the size of the problem increases as the number of processors increases.

The solution time required for this model is extremely small, under a second for the

Table 7.1: Time to solve one time step of the CSTR MHE problem with different horizon lengths. Problems are solved using IPOPT and PIPS-NLP.

Solver	Processors	Solve Time (sec)			
		NFE = 5	NFE = 10	NFE = 20	NFE = 50
IPOPT	1	0.043	0.082	0.137	0.262
PIPS-NLP	1	0.066	0.144	0.268	0.534
	2	0.053	0.127	0.177	0.345
	4	0.04	0.083	0.102	0.215
	8	-	0.207	0.381	0.320

largest model considered. Therefore, there isn't much speed-up in using a parallel solver. We see slightly faster solve times for PIPS-NLP as the number of processors is increased from 1 to 4 however we see a degradation in performance when we increase to 8 processors due to both increased communication costs between processors and increased problem size. PIPS-NLP is able to match the performance of IPOPT when run with 4 processors and for the larger problems it has the fastest solve times.

7.1.2 Distillation Column Case Study

The second case study considers the binary distillation column model introduced in Chapter 3. The model equations are given in Appendix A.2. Again we consider a single instance of the MHE problem and we vary the problem size by changing the number of finite elements used in the discretization. The NLPs in this case are significantly larger and more challenging to solve than in the CSTR case. Careful model initialization was required to promote solver convergence. In addition, several solver options had to be tuned in order for the solvers to converge reliably for these cases. Regularization was required for every problem instance during one or more of the interior point iterations. Solve times for these

Table 7.2: CSTR NLP problem sizes corresponding to the results in Table 7.1

		NFE = 5	NFE = 10	NFE = 20	NFE = 50
IPOPT	# Variables	184	404	844	2164
	# Constraints	169	374	784	2014
	# Iterations	25	29	25	21
PIPS-NLP	# Variables	184	404	844	2164
1 processor	# Constraints	169	374	784	2014
	# Iterations	25	29	25	21
PIPS-NLP	# Coupling Vars.	3	3	3	3
2 processors	# Variables	190	410	850	2170
	# Constraints	175	380	790	2020
	# Iterations	22	31	22	21
PIPS-NLP	# Coupling Vars.	9	9	9	9
4 processors	# Variables	206	426	866	2186
	# Constraints	191	396	806	2036
	# Iterations	18	24	21	19
PIPS-NLP	# Coupling Vars.	-	21	21	21
8 processors	# Variables	-	458	898	2218
	# Constraints	-	428	838	2068
	# Iterations	-	30	32	19

cases are presented in Table 7.3 and problem sizes are reported in Table 7.4.

Table 7.3: Time to solve one time step of the Distillation MHE problem with different horizon lengths. Problems are solved using IPOPT and PIPS-NLP.

Solver	Processors	Solve Time (sec)			
		NFE = 5	NFE = 10	NFE = 20	NFE = 50
IPOPT	1	12.86	36.85	33.797	163.509
PIPS-NLP	1	22.87	65.054	62.925	292.316
	2	18.157	61.887	51.182	293.635
	4	12.95	68.444	78.536	270.681
	8	-	141.883	102.09	1094.012

For this case we see no benefit in using a parallel solver. IPOPT has significantly faster solve times for every test case. In addition, we see some inconsistency in PIPS-NLP in terms of the best number of processors to use. The poor performance of PIPS-NLP can be attributed to the computational cost of the Schur complement decomposition approach when there are a large number of coupling variables. The bottleneck of this algorithm is the explicit formation and factorization of the Schur complement. The size of the Schur complement is directly related to the amount of coupling in the problem which is determined by the number of dynamic states in the model and the number of processors used to solve the problem.

Previous studies applying the Schur complement approach to dynamic optimization considered models with only a couple dozen state variables and noted performance deterioration as the number of processors increased [92, 109]. Furthermore, this performance degradation is also noted in studies applying the Schur complement approach to general structured nonlinear optimization problems [91, 111].

In our distillation column model we have 84 differential states corresponding to the

Table 7.4: Distillation NLP problem sizes corresponding to the results in Table 7.3

		NFE = 5	NFE = 10	NFE = 20	NFE = 50
IPOPT	# Variables	8724	19419	40809	104979
	# Constraints	8304	18579	39129	100779
	# Iterations	53	71	27	35
PIPS-NLP	# Variables	8724	19419	40809	104979
	# Constraints	8304	18579	39129	100779
	# Iterations	53	68	27	35
PIPS-NLP	# Coupling Vars.	84	84	84	84
2 processors	# Variables	8892	19587	40977	105147
	# Constraints	8472	18747	39297	100947
	# Iterations	51	64	23	39
PIPS-NLP	# Coupling Vars.	252	252	252	252
4 processors	# Variables	9228	19923	41313	105483
	# Constraints	8808	19083	39633	101283
	# Iterations	47	60	23	41
PIPS-NLP	# Coupling Vars.	-	588	588	588
8 processors	# Variables	-	20595	41985	106155
	# Constraints	-	19755	40305	101955
	# Iterations	-	55	22	55

methanol composition and the liquid molar holdup on each of the 42 stages in the column. This is already a significant amount of coupling and Table 7.4 shows how the number of coupling variables multiplies even further as the number of processors increases. Clearly, the increased computational costs associated with forming and factorizing a Schur complement with this much coupling far exceeds any parallel speed-up produced by the rest of the algorithm. The performance is further reduced by the regularization required for this model. As noted in [112], whenever the regularization term is adjusted in PIPS-NLP, the Schur complement must be re-built and factorized.

These results show that in order for a parallel algorithm to be effective in solving dynamic optimization problems, it has to maintain parallel efficiency as the number of coupling terms increases. The next section shows how the cyclic reduction algorithm does just that.

7.2 Cyclic Reduction Prototype

PIPS-NLP does not contain an implementation of the cyclic reduction algorithm. Therefore we implemented prototypes of both the cyclic reduction and Schur complement algorithms in order to compare their performance. We tested the algorithms on two versions of a simple quadratic programming (QP) problem. The two versions differ in their objective function, sparsity, and inclusion of algebraic constraints and variables. We chose to study an equality constrained QP test problem because solving it would require the solution of a single KKT system. As will be shown below, incorporating or not incorporating algebraic variables as well as changing the ratio between the number of state, control, and algebraic variables will impact the structure and sparsity of the the KKT matrix. This can have a significant impact on the performance of the two algorithms considered in this work.

The algorithms were prototyped in Matlab R2012a [80] and tested on a Dell PowerEdge T420 server with 12 cores and 64 GB of RAM. The Matlab parallel computing toolbox was used for parallelization. In our implementations, all linear algebra on the individual

blocks is done using the HSL solver MA57 [113] which was called using the MEX interface. The test matrices all had 100 block rows which corresponds to a discretization with 100 time steps. The first algorithm we tested applied cyclic reduction to the full KKT matrix in block tridiagonal form. The second algorithm introduced decoupling variables to the KKT system and followed the Schur complement decomposition approach, factorizing the Schur complement using cyclic reduction.

We compare our algorithms against Matlab's built-in linear solver which is invoked by a backslash. We will refer to this solver as 'Matlab Backslash'. When Matlab Backslash is applied to a sparse matrix, it checks several properties of that matrix such as bandwidth, band density, and symmetry before deciding which algorithm is most appropriate for that system [114]. For instance, Matlab Backslash includes embedded versions of the high-performance, state-of-the-art, serial solvers UMFPACK[115], MA57[113], and LAPACK's banded solver[116]. Because Matlab Backslash has been fine-tuned for performance over many years, and because of its adaptive algorithm selection, we feel it was the best benchmark to compare our parallel algorithm prototypes against. However, it is important to keep in mind that Matlab Backslash is a general linear solver designed to work well on a wide variety of problems while our algorithms are tailored to handle a very specific structure and exploit a priori knowledge of that structure. The Matlab Backslash results illustrate the problem size boundary at which it becomes beneficial to use our structure-tailored parallel algorithms over a general serial algorithm. For all the test matrices described in the next two sections Matlab Backslash used MA57 to solve the full KKT matrix.

7.2.1 QP Structured Without Algebraic Variables

The QP problem for this test case represents a simple optimal control problem where the dynamics in the model have been approximated using a linear algebraic equation. We only include state and control variables and don't impose any algebraic constraints besides the dynamics. The problem is shown below

$$\min \frac{1}{2}\sum_{k=1}^{N} x_k^T Q x_k + \frac{1}{2}\sum_{k=0}^{N-1} u_k^T R u_k \tag{7.1a}$$

$$\text{s.t. } x_{k+1} = Ax_k + Bu_k,\ k = 0, \ldots, (N-1) \tag{7.1b}$$

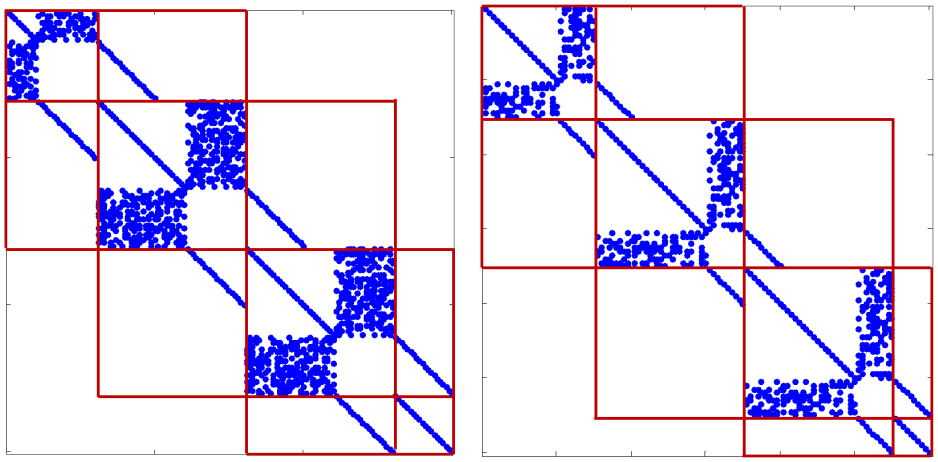
$$x_0 = x(0) \tag{7.1c}$$

where $x_k \in \mathbb{R}^{n_x}$ are vectors of state variables and $u_k \in \mathbb{R}^{n_u}$ are vectors of control variables for each time point k. The matrices A and B are randomly generated sparse matrices each with a density of 50%. They were generated such that the matrix $[A\ B]$ has full row rank. Q and R are randomly generated diagonal matrices with all positive entries. We did not encounter any problems with ill-conditioning for this test case.

The results presented here solve a single instance of the KKT system that would arise from problem (7.1). We form and solve the KKT system under two conditions, one with twice as many state variables as control variables and one with twice as many control variables as state variables. Figure 7.1 shows the sparsity patterns for these ratios of state to control variables. The case with more states than controls results in denser diagonal blocks in the KKT matrix.

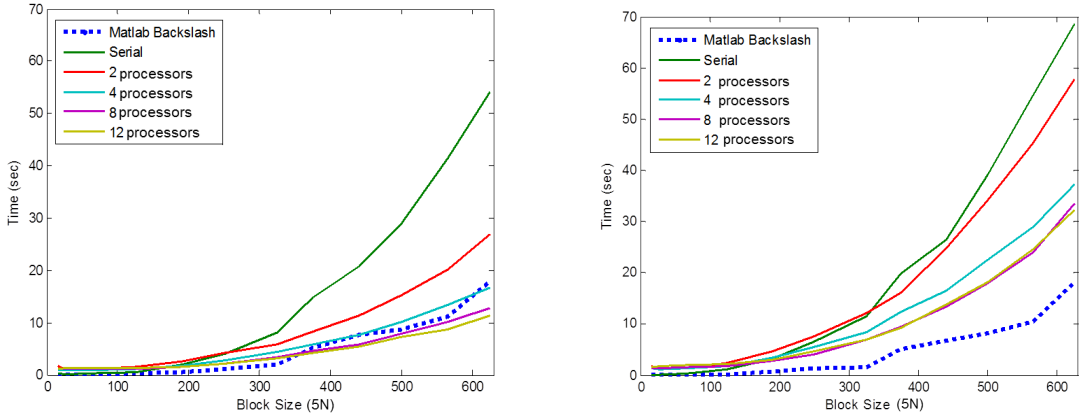
In order to understand the computational scaling of our algorithms we generated and solved KKT systems with varying block sizes while keeping the ratio of states to controls constant. Recall that each test system has 100 block rows so a block size of 400 corresponds to an overall linear system with 40,000 variables and equations. In the first case we had $2N$ state variables and N control variables leading to a diagonal block size of $5N$. The timing results are shown in Figure 7.2.

We see that Matlab Backslash is able to solve the smaller systems, with diagonal block sizes less than 300, faster than our parallel algorithms. The Schur complement approach has the slowest times for all the block sizes tested. In contrast, cyclic reduction used to factorize the full matrix is competitive with Matlab Backslash for the smaller cases and faster for the larger cases when run with 8 or more processors. For both parallel algorithms



(a) (b)

Figure 7.1: (Left) Sparsity pattern for QP case with 20 states and 10 controls. (Right) Sparsity pattern for QP case with 10 states and 20 controls. The tridiagonal blocks have been outlined in red.



(a) (b)

Figure 7.2: Time to factorize a matrix with 2N states and N controls using (Left) cyclic reduction on the full matrix and (Right) cyclic reduction on the Schur complement.

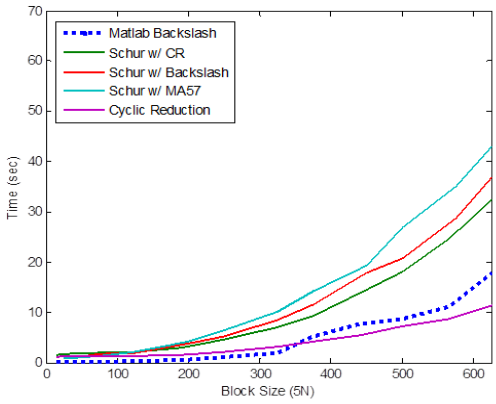
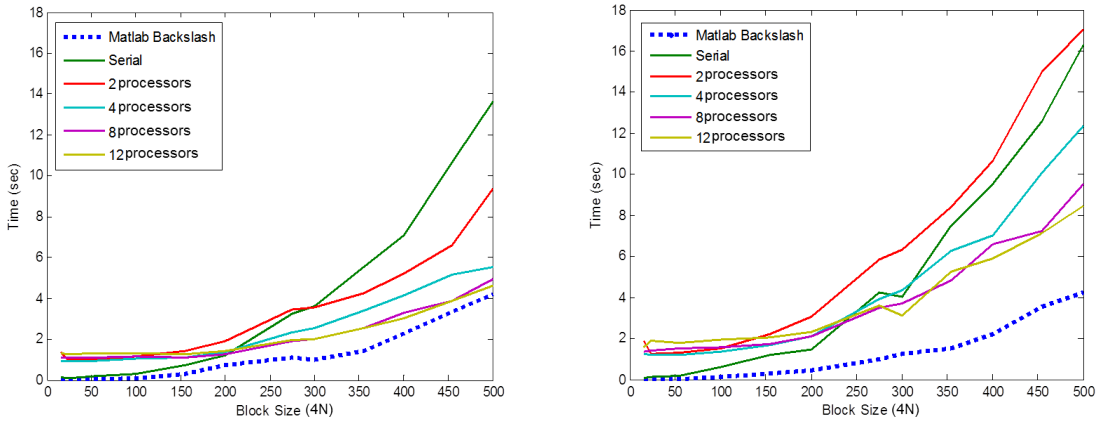


Figure 7.3: Time to factorize a matrix with 2N states and N controls with several different algorithms run with 12 processors.

we see some speed-up with an increasing number of processors. However, the results suggest that using more than 8 processors will not improve performance significantly.

We also tested other versions of the Schur complement decomposition approach where the Schur complement was factorized using the HSL solver MA57 or using Matlab Backslash. A comparison of these variations run with 12 processors is shown in Figure 7.3. The implementation using cyclic reduction to factorize the Schur complement is the fastest. However, none of the Schur complement implementations come close to matching the performance of Matlab Backslash on the larger test systems.

The second case we studied had N state variables and $2N$ control variables leading to a diagonal block size of $4N$. Recall that this case results in sparser diagonal blocks than in the first case. The timing results are shown in Figure 7.4. Again we see that Matlab Backslash performs well over the entire range of block sizes and the Schur complement implementation is noticeably slower even when run with 12 processors. The cyclic reduction approach comes close to matching the Matlab Backslash times for the larger block sizes when run with 8 or 12 processors but it doesn't surpass the performance of Matlab Backslash as in the previous case.



(a) (b)

Figure 7.4: Time to factorize a matrix with N states and 2N controls using (Left) cyclic reduction on the full matrix and (Right) cyclic reduction on the Schur complement.

If we compare the performance of several versions of the Schur complement approach, shown in Figure 7.5, we see more variability in which version is the fastest for different block sizes but for the larger systems again we see that the Schur complement approach with cyclic reduction has the best performance. However, once again the Schur complement approach is significantly slower than Matlab Backslash.

7.2.1.1 Larger Blocks

Given the results on block sizes up to 500, we test the cyclic reduction algorithm on even larger block sizes. Figure 7.6 shows the timing results for block sizes up to 3500 and 4500 for the cases with more states and more controls respectively. The overall linear systems being solved have hundreds of thousands of variables and constraints. The truncated results for the 2 and 4 processor cases was caused by a 2GB transmission limit for each processor imposed by Matlab's Parallel Computing Toolbox. This limit has been removed in more recent versions of Matlab.

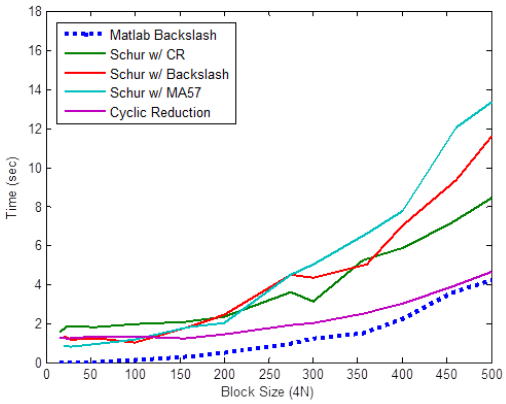
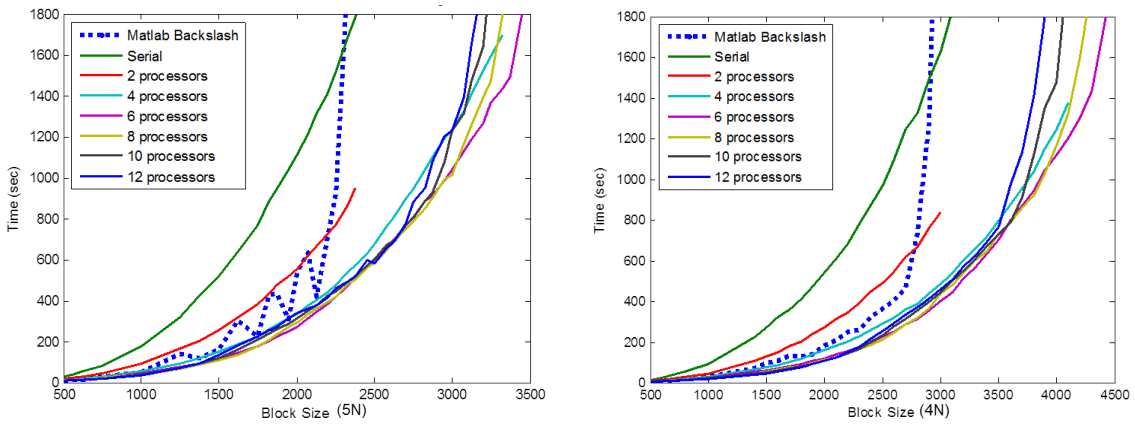


Figure 7.5: Time to factorize a matrix with N states and 2N controls with several different algorithms run with 12 processors.

The figures show a dramatic increase in the solution time for Matlab Backslash with block sizes greater than 2100 in the first case and 2600 in the second case. These explosions in the solution time correspond to the server running out of RAM space and starting to use swap space. Notice that the runs using CR also hit this upswing in time but only with significantly larger block sizes. This seems to indicate that our algorithm implementation is more memory efficient in solving these systems than what is built into Matlab. Using CR, we are able to solve large block tridiagonal systems faster than using a general linear algebra solver and we are able to solve larger systems on the same computer hardware.

Figure 7.6 also shows that for the larger block sizes eventually the case using 12 processors becomes slower than the case with 10, which becomes slower than the case with 8, etc. This slow-down illustrates the additional memory and communication overhead required when more processors are used.

We also note that we see very little speed-up moving from 8 to 12 processors and therefore we would not expect significant speed-up with the addition of more processors. The CR algorithm is not perfectly parallelizable and will run into the challenge of idle processors when the number of even block rows in the tridiagonal matrix is less than the number



(a) (b)

Figure 7.6: Time to factorize a matrix using cyclic reduction on the full matrix for (Left) 2N states and N controls and (Right) N states and 2N controls.

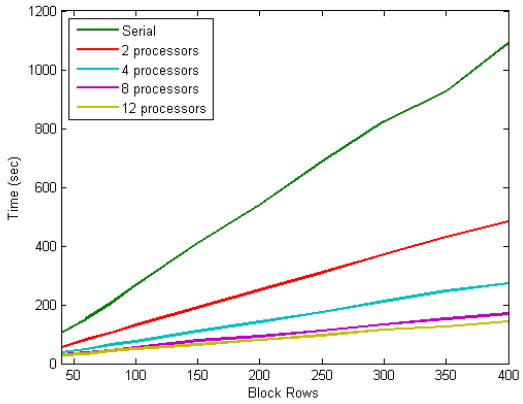


Figure 7.7: Time to factorize a matrix with N states and 2N controls with the block size fixed to 1500 and varying the number of block rows.

of available processors. This point is reached faster when more processors are used. Figure 7.7 shows that as we increase the number of block rows in our original system, we see more of a performance difference between the 8 and 12 processor cases. With a greater number of block rows, more CR recursions can be completed before we start having idle processors.

Comparing the results for the two test cases shows that sparsity has a significant effect on algorithm performance. In order to better understand this effect we took a closer look at the two factors controlling the sparsity of our KKT systems, the ratio of control to state variables and the density of the matrices A and B. Table 7.5 shows the solve times for the two algorithms run with 12 processors on KKT systems with 100 block rows and the block size fixed to 1000. The Schur complement times use cyclic reduction to factorize the Schur complement. As the ratio of controls to states increases, the system becomes more sparse. Similarly, reducing the density of the random sparse matrices A and B from 50% to 5% results in a significantly sparser system.

The results show that as the systems become more sparse, the two algorithms perform similarly in terms of solve time. On the other hand, as the systems become more dense, the performance of the Schur complement approach degrades more than that of the CR approach. The results for the larger ratios of controls to state variables motivates the need to consider algebraic variables and constraints in our QP structured problem, in order to get a more comprehensive idea of how these algorithms will perform on real dynamic optimization problems. These test cases are presented in the next section.

7.2.2 QP Structured With Algebraic Variables

For these test cases algebraic variables and linear constraints are added to the QP problem. We also consider a more general form for the objective function. The problem formulation is shown below.

Table 7.5: Effect of sparsity and the ratio of controls to states on parallel algorithm performance. Runs were done using 12 processors and the Schur complement was factored using cyclic reduction

A,B density is 50%			A,B density is 5%		
Ratio $\frac{\#\text{ Controls}}{\#\text{ States}}$	Cyclic Reduction Time (sec)	Schur Complement Time (sec)	Ratio $\frac{\#\text{ Controls}}{\#\text{ States}}$	Cyclic Reduction Time (sec)	Schur Complement Time (sec)
0.5	37.8	115.7	0.5	23.4	37.7
2	20.9	57.7	2	12.6	15.6
4	11.6	31.6	4	7.3	8.2
8	7.7	16.0	8	4.5	4.5
16	4.6	8.4	16	3.2	3.0
32	2.8	5.0	32	2.2	2.7
64	2.2	3.3	64	1.6	2.9

$$\min \quad \frac{1}{2}\sum_{k=0}^{N} z_k^T W z \tag{7.2a}$$

$$\text{s.t. } x_{k+1} = Ax_k + Bu_k + Cy_k,\ k = 0, \ldots, (N-1) \tag{7.2b}$$

$$0 = Dx_k + Eu_k + Fy_k,\ k = 0, \ldots, (N-1) \tag{7.2c}$$

$$x_0 = x(0) \tag{7.2d}$$

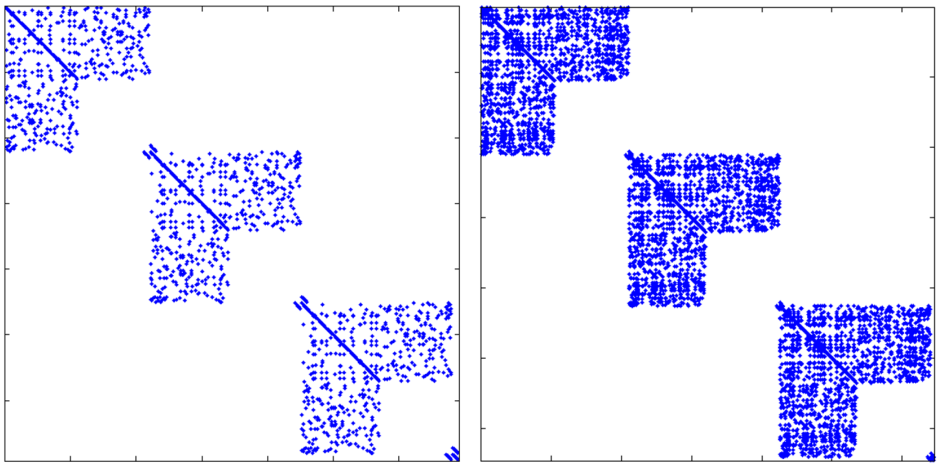
where $y_k \in \mathbb{R}^{n_y}$ are vectors of algebraic variables and $z_k^T = [x_k^T, u_k^T, y_k^T]$. The matrices A, B, C, D, E, and F are randomly generated, full-rank sparse matrices with a density of 5%. W is the Hessian of the objective function. W is randomly generated as a sparse symmetric matrix with a density of 5%. Notice that the randomly generated matrices are

an order of magnitude more sparse than in the previous test cases.

For this test case we did encounter some ill-conditioning and numerical error propagation. We experimented with several strategies for handling this ill-conditioning by adding a positive diagonal matrix to W. Regularization was done only once before the first CR pass. The regularization technique used for the results presented here was to make W positive definite by adding a diagonal term that dominates the largest negative eigenvalue in W. Calculating the eigenvalues is straightforward in Matlab for our test cases. However, for a high-performance implementation of CR it is not a practical approach. We also tested a regularization approach similar to the inertia correction step in IPOPT and found that it was also effective in stabilizing the CR recursions. However proving that this approach will always be sufficient remains an open question.

We considered two cases of problem (7.2). The first had the same number of controls as states and 10 times the number of algebraics as states. The second had the same number of controls as states and 20 times the number of algebraics as states. The sparsity patterns for the KKT matrices for these cases are shown in Figure 7.8. Notice that the off-diagonal blocks are almost entirely zeros because the ratio of states to algebraics is very small. This also means that the fill-in from CR for these systems will be negligible compared to the overall size and sparsity of the blocks.

Timing results for the first case are shown in Figure 7.9. Compared with the previous QP problem we are able to solve systems with larger block sizes due to the sparser random matrices used in this problem. The results show that CR applied to the full matrix performs very well, beating Matlab's backslash for almost all the block sizes with only 4 processors. We also see improved performance for the Schur complement approach with 8 or 12 processors which is competitive with Matlab's Backslash over the entire range of block sizes. For the 2 and 4 processors cases, the Schur complement approach quickly reaches the 2GB transmission limit imposed by Matlab indicating that the Schur complement algorithm is significantly more memory intensive than CR. On the other hand, if we compare CR with

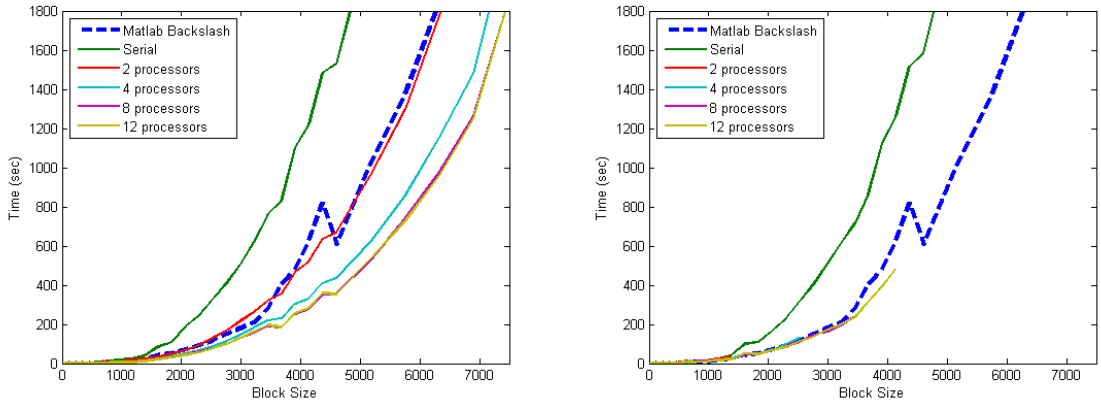


(a) (b)

Figure 7.8: (Left) Sparsity pattern for QP case with 5 states, 5 controls, and 50 algebraics. (Right) Sparsity pattern for QP case with 5 states, 5 controls, and 100 algebraics.

several versions of the Schur complement approach, as shown in Figure 7.10, it becomes even more apparent that CR significantly outperforms the other algorithms.

For the second case with 20 times the number of algebraics as states, we see an even more dramatic difference in solve times between CR and Matlab Backslash. These results are shown in Figure 7.11. CR is able to solve systems with a block size of 6500 in about half of the time it takes Matlab Backslash. This corresponds to a system with 650,000 variables and constraints. In addition, CR is faster than backslash for most of the test matrices using only 2 processors. The Schur complement approach shows promising timing performance on these systems but hits the memory transmission limits very quickly. These results very clearly show the benefit of using a structure-exploiting linear solver over a general linear solver for these large, sparse systems.



(a) (b)

Figure 7.9: Time to factorize a matrix with N states, N controls, and 10N algebraics using (Left) cyclic reduction on the full matrix and (Right) cyclic reduction on the Schur complement.

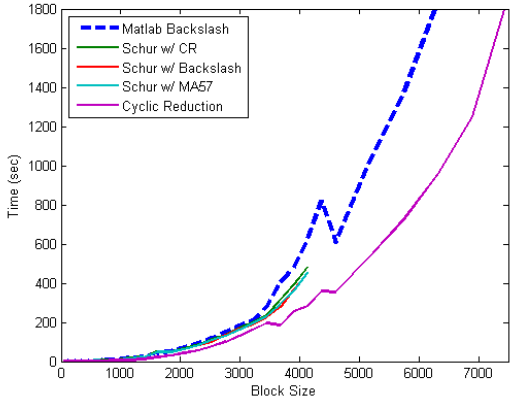
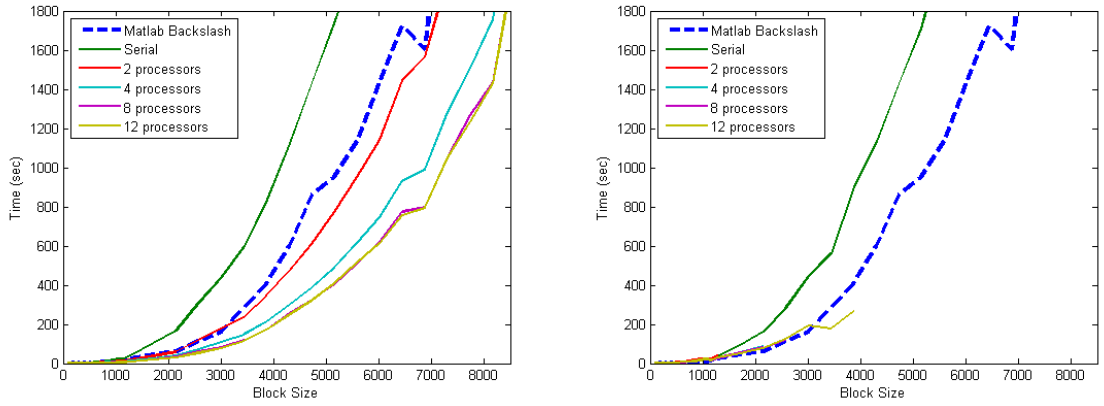


Figure 7.10: Time to factorize a matrix with N states, N controls, and 10N algebraics with several different algorithms run with 12 processors.



(a) (b)

Figure 7.11: Time to factorize a matrix with N states, N controls, and 20N algebraics using (Left) cyclic reduction on the full matrix and (Right) cyclic reduction on the Schur complement.

7.3 Concluding Remarks

The results in this chapter show that cyclic reduction is a promising parallel linear algebra approach for dynamic optimization problems. We have demonstrated the performance of this algorithm on a number of test matrices and shown that CR outperforms state-of-the-art, serial linear algebra solvers. In addition, the performance of the cyclic reduction algorithm does not deteriorate as significantly as the Schur complement decomposition approach when there are many dynamic states. This is due in part to the fact that CR operates on smaller linear systems compared to the Schur complement approach and CR also preserves sparsity more effectively than the Schur complement approach. As mentioned before, the CR implementation used in this work is a prototype. It remains to be seen whether or not a high-performance implementation of CR within an interior point framework will show similar performance advantages.

Chapter 8

Conclusions

In this chapter we summarize our contributions and offer suggestions for future work.

8.1 Summary and Contributions

Chapter 1 presents a brief introduction to dynamic optimization. We review three common solution techniques including direct transcription which reformulates the problem as a single large NLP. We then describe current implementation practices and challenges for dynamic optimization techniques including the manual implementation of discretization schemes and implementations over multiple software platforms. Finally, Chapter 1 describes the research objectives of this dissertation.

Applications

Chapter 2 introduces the state estimation problem. We review previous work in state estimation and describe two methods for formulating the problem using dynamic optimization, MHE and asMHE. MHE is a useful technique for constrained state estimation and produces accurate state estimates using a finite set of past measurements. asMHE has all the benefits of MHE with the added advantage of reduced on-line computational cost. Chapter 2 also reviews the concept of robust M-estimators as a strategy for dealing with large measurement errors and illustrates how they can be incorporated in an MHE framework. Chapter 3 describes several case studies applying the state estimators in Chapter 2 to models of a CSTR and a distillation column. We describe the implementation and demonstrate how the addition of robust M-estimators reduces the influence of measure-

ment errors.

The major contributions of this section are listed as follows:

- Extension of MHE and asMHE using robust M-Estimators to mitigate the effect of large measurement errors on state estimates.
- Application of novel, robust state estimators to classical chemical engineering models of a CSTR and a distillation column.
- Analysis of the performance of novel, robust state estimators under a variety of prolonged measurement error scenarios simulating sensor drift and failure.
- Development of a state estimator that produces fast and accurate state estimates even in the presence of many large measurement errors by combining the computational efficiency of asMHE with the robustness of Hampel's Redescending estimator.

Modeling Tools

Chapter 4 describes available algebraic modeling frameworks for dynamic optimization and their limitations, motivating the development of `pyomo.dae`. We develop the framework in Pyomo because it is an open-source, full-featured algebraic modeling language developed in the high-level programming language Python. We explain how the two modeling components introduced by `pyomo.dae` are flexible enough to represent arbitrary differential equations over bounded rectangular domains. Chapter 4 also presents the automatic discretization features included in `pyomo.dae` and several discretization schemes included in the package including orthogonal collocation over finite elements. In addition, we briefly describe the implementation and demonstrate the extensibility of `pyomo.dae`. Chapter 5 tests `pyomo.dae` on a wide variety of dynamic optimization problems including the numerical solution of a PDE boundary value problem, optimal control, and parameter estimation. Furthermore, we show that the computational cost of using automatic discretization scales well with problem size and the number of discretization points.

The major contributions of this section are listed as follows:

- Development of a novel model abstraction for general dynamic optimization problems that separates the dynamic model from the solution technique.
- Design and development of modeling components for concisely and intuitively representing arbitrary differential equations over bounded rectangular domains.
- Design and development of a flexible and extensible direct transcription framework for automatically applying a discretization scheme to DAE and PDAE models.

Parallel Solution Algorithms

Chapter 6 reviews a common NLP solution approach, the interior point method, and describes the linear systems resulting from the Newton steps that solve the discretized optimality conditions. We then illustrate how exploitable structure is imposed on these linear systems when a direct transcription approach is applied to a dynamic optimization problem. Chapter 6 also includes descriptions of two parallel algorithms capable of exploiting this structure. The Schur complement decomposition approach exploits a bordered block-diagonal structure and the cyclic reduction algorithm exploits a block tridiagonal structure. We propose a novel way to apply cyclic reduction to the symmetric indefinite form of the KKT matrix and carefully analyze the sparsity and numerical conditioning properties of applying cyclic reduction to KKT systems arising from dynamic optimization. Chapter 7 studies the computational performance of the parallel algorithms presented in Chapter 6. We apply an existing parallel interior point solver to dynamic models of a CSTR and a distillation column and demonstrate the poor scaling of the Schur complement approach when a model has many differential states. We also demonstrate a prototype of the cyclic reduction algorithm and show that it may be able to overcome some of the limitations of the Schur complement approach.

The major contributions of this section are listed as follows:

- Proposal to use the cyclic reduction algorithm to solve the symmetric indefinite KKT

system arising from discretized dynamic optimization problems.

- Analysis of the effect of cyclic reduction recursions on the sparsity and structure of the KKT matrix.
- Development of two theorems describing sufficient conditions for cyclic reduction recursions to remain non-singular and well-conditioned when applied to KKT systems.
- Analysis of the performance of cyclic reduction when applied to structured test systems representing KKT matrices arising from dynamic optimization.

8.2 Recommendations for Future Work

State Estimation

Further work is required to explore the limits of the asMHE formulation with the Redescending estimator in terms of what percentage of gross error measurements it can overcome. A careful sensitivity analysis of the different tuning parameters for the robust M-Estimators would also be interesting. In practice these estimators would be tuned to the particular application rather than set to reasonable, general values as in this study.

It would also be worthwhile to continue exploring the performance of these formulations by considering additional case studies with different types of errors. Our work focused mainly on gross errors that might be caused by a sensor drifing away from the true value or failing entirely. It would be interesting to see how the formulations perform with random outliers to see what effect the type of gross error has on the performance of the state estimators. Moreover, structured outliers as found in fault detection need to be studied.

Additionally, one of the major assumptions with the case studies presented in this thesis was that there was no plant-model mismatch. Preliminary results relaxing this assumption show that the amount of plant-model mismatch can have a significant effect on the perfor-

mance of the estimators presented here. Most notably, it causes the MHE and asMHE estimates to differ significantly. More work is needed to systematically investigate this observation.

Finally, state estimation is the first step in strategies for real-time optimization. It would be interesting to combine on-line, robust state estimators with nonlinear model predictive control (NMPC) in an effort to create an overall strategy for on-line plant optimization that is more robust.

pyomo.dae Modeling Framework

Currently `pyomo.dae` can only handle bounded rectangular continuous domains. In the future we plan on extending our tool to represent unbounded domains. This would make it easier to represent control problems over infinite horizons. We also plan on adding a modeling component to represent integrals.

`pyomo.dae` includes a variety of discretization transformations as well as a framework for users to implement their own custom discretization. However, simultaneous discretization is just one of several approaches for solving this class of problem. Single and multiple shooting methods are also common solution strategies [117, 118, 119]. We plan to eventually link `pyomo.dae` to existing integrators, such as Sundials [120], and then add implementations of shooting methods. An integrator would also allow us to simulate dynamic models in order to initialize our discretized optimization problem or develop hybrid full-discretization/shooting algorithms.

Another useful extension would be more sophisticated frameworks for model initialization and specification of time dependent data. In the current implementation a user can provide a careful initialization for a model after it has been discretized. However, this is not entirely straightforward and the initialization is tied to the discretization applied. We would also like to implement several data interpolation schemes in order to make this initialization process easier and consistent across any choice of discretization. An interpo-

lation scheme would also be useful for estimating differential variable values at any point in the continuous domain after the model has been solved, especially in the case of a collocation discretization where a high order functional form of the state profile already exists. An automatic, element-by-element initialization would be another possible extension.

Parallel Solvers

In the future we plan on implementing a high performance version of cyclic reduction and embedding it within the interior point solver PIPS-NLP. This will allow us to test cyclic reduction on a variety of large-scale dynamic optimization problems and more thoroughly compare its performance against the Schur complement approach.

We also plan on investigating a variant of CR which is better suited for parallel implementation and overcomes the challenge of idle processors seen in the classical implementation. This variant was proposed in [121] and is essentially the standard block cyclic reduction algorithm applied to every block row and without the back substitution step. We expect to see more significant speed-up as we add processors using this approach.

Finally, we plan on taking a deeper look at the more technical details of CR applied to KKT systems related to dynamic optimization. We observed some very favorable stability and conditioning properties and hope to gain a better theoretical understanding of our observations.

Bibliography

[1] L. T. Biegler, A. M. Cervantes, and A. Wächter, "Advances in simultaneous strategies for dynamic process optimization," *Chemical Engineering Science*, vol. 57, no. 4, pp. 575–593, 2002.

[2] S. J. Qin and T. A. Badgwell, "An overview of nonlinear model predictive control applications," in *Nonlinear model predictive control*, pp. 369–392, Springer, 2000.

[3] M. Diehl, H. G. Bock, J. P. Schlöder, R. Findeisen, Z. Nagy, and F. Allgöwer, "Real-time optimization and nonlinear model predictive control of processes governed by differential-algebraic equations," *Journal of Process Control*, vol. 12, no. 4, pp. 577–585, 2002.

[4] L. T. Biegler, "An overview of simultaneous strategies for dynamic optimization," *Chemical Engineering and Processing: Process Intensification*, vol. 46, no. 11, pp. 1043–1053, 2007.

[5] T. Binder, L. Blank, H. G. Bock, R. Bulirsch, W. Dahmen, M. Diehl, T. Kronseder, W. Marquardt, J. P. Schlöder, and O. von Stryk, "Introduction to model based optimization of chemical processes on moving horizons," in *Online optimization of large scale systems*, pp. 295–339, Springer, 2001.

[6] L. T. Biegler, *Nonlinear Programming: Concepts, Algorithms, and Applications to Chemical Processes*. SIAM, 2010.

[7] B. Nicholson, R. López-Negrete, and L. T. Biegler, "On-line state estimation of nonlinear dynamic systems with gross errors," *Computers & Chemical Engineering*, vol. 70, pp. 149–159, 2014.

[8] J. B. Rawlings and D. Q. Mayne, *Model Predictive Control: Theory and Design*. Nob Hill Publishing, 2009.

[9] J. Forbes and T. Marlin, "Design cost: a systematic approach to technology selection for model-based real-time optimization systems," *Computers & Chemical Engineering*, vol. 20, no. 67, pp. 717 – 734, 1996.

[10] A. E. Bryson and Y. C. Ho, *Applied Optimal Control: Optimization, Estimation, and Control*. New York, USA: Taylor and Francis, 1975.

[11] F. Daum, "Nonlinear filters: beyond the kalman filter," *Aerospace and Electronic Systems Magazine, IEEE*, vol. 20, no. 8, pp. 57–69, 2005.

[12] J. Prakash, S. C. Patwardhan, and S. L. Shah, "Constrained nonlinear state estimation using ensemble kalman filters," *Industrial & Engineering Chemistry Research*, vol. 49, no. 5, pp. 2242–2253, 2010.

[13] S. Julier, J. Uhlmann, and H. Durrant-Whyte, "A new method for the nonlinear transformation of means and covariances in filters and estimators," *Automatic Control, IEEE Transactions on*, vol. 45, no. 3, pp. 477–482, 2000.

[14] G. Evensen, "Sequential data assimilation with a nonlinear quasi-geostrophic model using monte carlo methods to forecast error statistics," *Journal of Geophisical Research*, vol. 99, no. C5, pp. 143–162, 1994.

[15] G. Burgers, P. J. van Leeuwen, and G. Evensen, "Analysis scheme in the ensemble kalman filter," *Monthly Weather Review*, vol. 126, no. 6, pp. 1719–1724, 1998.

[16] P. Houtekamer and H. Mitchell, "Data assimilation using an ensemble kalman filter technique," *Monthly Weather Review*, vol. 126, pp. 796–811, 1998.

[17] Z. Chen, "Bayesian filtering: From kalman filters to particle filters, and beyond,"

Technical report, Adaptive Syst. Lab., McMaster University, Hamilton, ON, Canada, 2003.

[18] S. Arulampalam, S. Maskell, N. Gordon, and T. Clapp, "A tutorial on particle filters for on-line non-linear/non-gaussian bayesian tracking," *IEEE Transactions on Signal Processing*, vol. 50, pp. 174–188, 2002.

[19] E. L. Haseltine and J. B. Rawlings, "Critical evaluation of extended kalman filtering and moving-horizon estimation," *Ind. Eng. Chem. Res.*, vol. 44, pp. 2451–2460, 2005.

[20] P. Vachhani, R. Rengaswamy, V. Gangwal, and S. Narasimhan, "Recursive estimation in constrained nonlinear dynamical systems," *AIChE Journal*, vol. 51, no. 3, pp. 946–959, 2004.

[21] P. Vachhani, S. Narasimhan, and R. Rengaswamy, "Robust and Reliable Estimation via Unscented Recursive Nonlinear Dynamic Data Reconciliation," *J. Process Control*, vol. 16, pp. 1075–1086, 2006.

[22] J. Prakash, S. Shah, and S. Patwardhan, "Constrained state estimation using particle filters," in *Proceedings of the 17th IFAC World Congress* (M. J. Chung and P. Misra, eds.), vol. 17, pp. 6472–6477, 2008.

[23] K. Muske and J. Rawlings, "Receding horizon recursive estimation," in *Proceedings of the American Control Conference, June, San Fransisco, CA*, 1993.

[24] H. Michalska and D. Mayne, "Moving horizon observers and observer-based control," *Automatic Control, IEEE Transactions on*, vol. 40, no. 6, pp. 995–1006, 1995.

[25] D. G. Robertson, J. H. Lee, and J. B. Rawlings, "A moving horizon-based approach for least-squares estimation," *AIChE Journal*, vol. 42, no. 8, pp. 2209–2224, 1996.

[26] C. V. Rao, *Moving-Horizon Strategies for the Constrained Monitoring and Control of Nonlinear Discrete-Time Systems*. PhD thesis, University of Wisconsin-Madison, 2000.

[27] C. V. Rao, J. B. Rawlings, and D. Q. Mayne, "Constrained state estimation for nonlinear discrete-time systems: Stability and moving horizon approximations," *IEEE Trans. Auto. Cont.*, vol. 48, no. 2, pp. 246–258, 2003.

[28] J. Ramlal, K. Allsford, and J. Hedengren, "Moving horizon estimation for an industrial gas phase polymerization reactor," in *Proceedings of IFAC Symposium on Nonlinear Control Systems Design (NOLCOS), Pretoria, South Africa*, 2007.

[29] J. B. Rawlings and B. R. Bakshi, "Particle filtering and moving horizon estimation," *Comput. Chem. Eng.*, vol. 30, pp. 1529–1541, 2006.

[30] T. Ohtsuka and H. Fujii, "Nonlinear receding-horizon state estimation by real-time optimixation technique," *Journal of Guidance, Control, and Dynamics*, vol. 19, no. 4, 1996.

[31] M. Tenny and J. Rawlings, "Efficient moving horizon estimation and nonlinear model predictive control," in *Proceedings of the American control conference, Anchorage, AK.*, 2002.

[32] P. Kühl, M. Diehl, T. Kraus, J. P. Schlöder, and H. G. Bock, "A real-time algorithm for moving horizon state and parameter estimation," *Computers & Chemical Engineering*, vol. 35, no. 1, pp. 71 – 83, 2011.

[33] S. Abrol and T. F. Edgar, "A fast and versatile technique for constrained state estimation," *Journal of Process Control*, vol. 21, no. 3, pp. 343 – 350, 2011.

[34] V. M. Zavala, C. D. Laird, and L. T. Biegler, "A fast moving horizon estimation algorithm based on nonlinear programming sensitivity," *Journal of Process Control*, vol. 18, no. 9, pp. 876–884, 2008.

[35] C. M. Crowe, "Data reconciliation progress and challenges," *Journal of Process Control*, vol. 6, no. 23, pp. 89 – 98, 1996.

[36] V. Hodge and J. Austin, "A survey of outlier detection methodologies," *Artificial Intelligence Review*, vol. 22, no. 2, pp. 85–126, 2004.

[37] P. Kadlec, B. Gabrys, and S. Strandt, "Data-driven soft sensors in the process industry," *Computers & Chemical Engineering*, vol. 33, no. 4, pp. 795 – 814, 2009.

[38] V. Chandola, A. Banerjee, and V. Kumar, "Anomaly detection: A survey," *ACM Comput. Surv.*, vol. 41, pp. 15:1–15:58, July 2009.

[39] H. Tong and C. M. Crowe, "Detection of gross erros in data reconciliation by principal component analysis," *AIChE Journal*, vol. 41, no. 7, pp. 1712–1722, 1995.

[40] J. Chen and J. Romagnoli, "A strategy for simultaneous dynamic data reconciliation and outlier detection," *Computers & Chemical Engineering*, vol. 22, no. 45, pp. 559 – 562, 1998.

[41] P. Vachhani, R. Rengaswamy, and V. Venkatasubramanian, "A framework for integrating diagnostic knowledge with nonlinear optimization for data reconciliation and parameter estimation in dynamic systems," *Chemical Engineering Science*, vol. 56, no. 6, pp. 2133 – 2148, 2001.

[42] D. B. Özyurt and R. W. Pike, "Theory and practice of simultaneous data reconciliation and gross error detection for chemical processes," *Computers & Chemical Engineering*, vol. 28, no. 3, pp. 381 – 402, 2004.

[43] Y. Zhang and J. Jiang, "Bibliographical review on reconfigurable fault-tolerant control systems," *Annual Reviews in Control*, vol. 32, no. 2, pp. 229 – 252, 2008.

[44] T. Chen and R. You, "A novel fault-tolerant sensor system for sensor drift compensation," *Sensors and Actuators A: Physical*, vol. 147, no. 2, pp. 623 – 632, 2008.

[45] R. López Negrete, *Nonlinear Programming Sensitivity Based Methods for Constrained State Estimation*. PhD thesis, Carnegie Mellon University, 2011.

[46] R. López-Negrete and L. T. Biegler, "A moving horizon estimator for processes with multi-rate measurements: A nonlinear programming sensitivity approach," *Journal of Process Control*, vol. 22, no. 4, pp. 677 – 688, 2012.

[47] A. Wächter and L. T. Biegler, "On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming," *Mathematical Programming*, vol. 106, pp. 25–57, 2006.

[48] V. M. Zavala, C. D. Laird, and L. T. Biegler, "Fast implementations and rigorous models: Can both be accommodated in nmpc?," *International Journal of Robust and Nonlinear Control*, vol. 18, no. 8, pp. 800–815, 2008.

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

[50] A. V. Fiacco and Y. Ishizuka, "Sensitivity and stability analysis for nonlinear programming," *Annals of Operations Research*, vol. 27, pp. 215–235, 1990.

[51] A. V. Fiacco, *Introduction to sensitivity and stability analysis in nonlinear programming*. Mathematics in Science and Engineering, Elsevier Science, 1983.

[52] P. J. Huber, *Robust Statistics*. New York: John Wiley and Sons, 1981.

[53] F. R. Hampel, "The influence curve and its role in robust estimation," *Journal of the American Statistical Association*, vol. 69, no. 346, pp. 383–393, 1974.

[54] N. Arora and L. T. Biegler, "Redescending estimators for data reconciliation and parameter estimation," *Computers & Chemical Engineering*, vol. 25, no. 1112, pp. 1585 – 1599, 2001.

[55] MATLAB, *version 7.12.0 (R2011a)*. Natick, Massachusetts: The MathWorks Inc., 2011.

[56] R. Fourer, D. Gay, and B. Kernighan, *AMPL: A Modeling Language for Mathematical Programming*. Scientific Press, 1993.

[57] H. Pirnay, R. López-Negrete, and L. T. Biegler, "Optimal sensitivity based on ipopt," *Mathematical Programming Computation*, vol. 4, no. 4, pp. 307–331, 2012.

[58] M. Diehl, *Real-Time Optimization for Large Scale Nonlineat Processes*. PhD thesis, Universität Heidelberg, 2011.

[59] R. López-Negrete, F. J. DAmato, L. T. Biegler, and A. Kumar, "Fast nonlinear model predictive control: Formulation and industrial process applications," *Computers & Chemical Engineering*, vol. 51, no. 0, pp. 55 – 64, 2013.

[60] G. D. Corporation, "General Algebraic Modeling System (GAMS) Release 24.2.1." Washington, DC, USA, 2013.

[61] T. H. Hultberg, "Flopc++ an algebraic modeling language embedded in c," in *in Operations Research Proceedings 2006, ser. Operations Research Proceedings, K.-H. Waldmann and*, pp. 187–190, Springer, 2006.

[62] S. Mitchell, M. OSullivan, and I. Dunning, "Pulp: a linear programming toolkit for python," 2011.

[63] W. E. Hart, J.-P. Watson, and D. L. Woodruff, "Pyomo: modeling and solving mathematical programs in python," *Mathematical Programming Computation*, vol. 3, no. 3, pp. 219–260, 2011.

[64] I. Dunning, J. Huchette, and M. Lubin, "JuMP: A modeling language for mathematical optimization," *arXiv:1508.01982 [math.OC]*, 2015.

[65] B. Houska, H. Ferreau, and M. Diehl, "ACADO Toolkit – An Open Source Framework for Automatic Control and Dynamic Optimization," *Optimal Control Applications and Methods*, vol. 32, no. 3, pp. 298–312, 2011.

[66] J. Hendengren, "Apmonitor modeling language," 2014.

[67] M. A. Patterson and A. V. Rao, "Gpops-ii: A matlab software for solving multiple-phase optimal control problems using hp-adaptive gaussian quadrature collocation methods and sparse nonlinear programming," *ACM Trans. Math. Softw.*, vol. 41, pp. 1:1–1:37, Oct. 2014.

[68] Process Systems Enterprise, "gPROMS," 1997-2014.

[69] J. Åkesson, K.-E. Årzén, M. Gäfvert, T. Bergdahl, and H. Tummescheit, "Modeling and optimization with optimica and jmodelica.orglanguages and tools for solving large-scale dynamic optimization problems," *Computers & Chemical Engineering*, vol. 34, no. 11, pp. 1737–1749, 2010.

[70] J. T. Betts and W. P. Huffman, "Sparse optimal control software socs," *Mathematics and Engineering Analysis Technical Document MEA-LR-085, Boeing Information and Support Services, The Boeing Company, PO Box*, vol. 3707, pp. 98124–2207, 1997.

[71] J. T. Betts, "Sparse optimization suite (sos)," *Applied Mathematical Analysis, LLC*, 2013.

[72] S. Leyffer and C. Kirches, "Taco - a toolkit for ampl control optimization," *Mathematical Programming Computation*, pp. 1–39, 2013.

[73] P. E. Rutquist and M. M. Edvall, "Propt-matlab optimal control software," *Tomlab Optimization Inc*, 2010.

[74] W. E. Hart, C. Laird, J.-P. Watson, and D. L. Woodruff, *Pyomo–optimization modeling in python*, vol. 67. Springer Science & Business Media, 2012.

[75] U. M. Ascher, R. M. Mattheij, and R. D. Russell, *Numerical solution of boundary value problems for ordinary differential equations*, vol. 13. Siam, 1994.

[76] U. M. Ascher and L. R. Petzold, *Computer methods for ordinary differential equations and differential-algebraic equations*, vol. 61. Siam, 1998.

[77] K. E. Brenan, S. L. Campbell, and L. R. Petzold, *Numerical solution of initial-value problems in differential-algebraic equations*, vol. 14. Siam, 1996.

[78] J. Butcher, *Numerical Methods for Ordinary Differential Equations*. Wiley, 2003.

[79] J. Andersson, *A General-Purpose Software Framework for Dynamic Optimization*. PhD thesis, Arenberg Doctoral School, KU Leuven, Department of Electrical Engineering (ESAT/SCD) and Optimization in Engineering Center, Kasteelpark Arenberg 10, 3001-Heverlee, Belgium, October 2013.

[80] MATLAB, *version 8.3.0 (R2012a)*. Natick, Massachusetts: The MathWorks Inc., 2012.

[81] D. Jacobson and M. Lele, "A transformation technique for optimal control problems with a state variable inequality constraint," *Automatic Control, IEEE Transactions on*, vol. 14, pp. 457–464, Oct 1969.

[82] D. P. Word, *Nonlinear programming approaches for efficient large-scale parameter estimation with applications in epidemiology*. PhD thesis, Texas A&M University, Department of Chemical Engineering, College Station, TX, 2013.

[83] V. M. Zavala, "Stochastic optimal control model for natural gas networks," *Computers & Chemical Engineering*, vol. 64, pp. 103 – 113, 2014.

[84] A. Hartwich, K. Stockmann, C. Terboven, S. Feuerriegel, and W. Marquardt, "Parallel sensitivity analysis for efficient large-scale dynamic optimization," *Optimization and Engineering*, vol. 12, no. 4, pp. 489–508, 2011.

[85] V. M. Zavala, C. D. Laird, and L. T. Biegler, "Interior-point decomposition approaches for parallel solution of large-scale nonlinear parameter estimation problems," *Chemical Engineering Science*, vol. 63, no. 19, pp. 4834–4845, 2008.

[86] D. B. Leineweber, I. Bauer, H. G. Bock, and J. P. Schlöder, "An efficient multiple shooting based reduced sqp strategy for large-scale dynamic process optimization.

part 1: theoretical aspects," *Computers & Chemical Engineering*, vol. 27, no. 2, pp. 157–166, 2003.

[87] D. B. Leineweber, A. Schäfer, H. G. Bock, and J. P. Schlöder, "An efficient multiple shooting based reduced sqp strategy for large-scale dynamic process optimization: Part ii: Software aspects and applications," *Computers & chemical engineering*, vol. 27, no. 2, pp. 167–174, 2003.

[88] J. Frasch, *Parallel Algorithms for Optimization of Dynamic Systems in Real-Time*. PhD thesis, Otto-von-Guericke University Magdeburg, 2014.

[89] X. Zhang, R. H. Byrd, and R. B. Schnabel, "Parallel methods for solving nonlinear block bordered systems of equations," *SIAM journal on scientific and statistical computing*, vol. 13, no. 4, pp. 841–859, 1992.

[90] D. Feng and R. B. Schnabel, "Globally convergent parallel algorithms for solving block bordered systems of nonlinear equations," *Optimization Methods and Software*, vol. 2, no. 3-4, pp. 269–295, 1993.

[91] J. Kang, Y. Cao, D. P. Word, and C. D. Laird, "An interior-point method for efficient solution of block-structured nlp problems using an implicit schur-complement decomposition," *Computers & Chemical Engineering*, vol. 71, pp. 563–573, 2014.

[92] D. P. Word, J. Kang, J. Akesson, and C. D. Laird, "Efficient parallel solution of large-scale nonlinear dynamic optimization problems," *Computational Optimization and Applications*, vol. 59, no. 3, pp. 667–688, 2014.

[93] N. Chiang, C. G. Petra, and V. M. Zavala, "Structured nonconvex optimization of large-scale energy systems using pips-nlp," in *Power Systems Computation Conference (PSCC), 2014*, pp. 1–7, Aug 2014.

[94] R. W. Hockney, "A fast direct solution of poisson's equation using fourier analysis," *J. ACM*, vol. 12, pp. 95–113, Jan. 1965.

[95] O. Buneman, "Compact non-iterative poisson solver.," tech. rep., Stanford Univ., Calif. Inst. for Plasma Research, 1969.

[96] M. A. Diamond and D. L. Ferreira, "On a cyclic reduction method for the solution of poisson's equations," *SIAM Journal on Numerical Analysis*, vol. 13, no. 1, pp. 54–70, 1976.

[97] T. E. Rosmond and F. D. Faulkner, "Direct solution of elliptic equations by block cyclic reduction and factorization," *Monthly Weather Review*, vol. 104, no. 5, pp. 641–649, 1976.

[98] P. N. Swarztrauber, "A direct method for the discrete solution of separable elliptic equations," *SIAM Journal on Numerical Analysis*, vol. 11, no. 6, pp. 1136–1150, 1974.

[99] D. A. Bini and B. Meini, "The cyclic reduction algorithm: from poisson equation to stochastic processes and beyond," *Numerical Algorithms*, vol. 51, no. 1, pp. 23–60, 2009.

[100] D. Heller, "Some aspects of the cyclic reduction algorithm for block tridiagonal linear systems," *SIAM Journal on Numerical Analysis*, vol. 13, no. 4, pp. 484–496, 1976.

[101] P. Amodio and F. Mazzia, "Backward error analysis of cyclic reduction for the solution of tridiagonal systems," *mathematics of computation*, vol. 62, no. 206, pp. 601–617, 1994.

[102] P. Yalamov and V. Pavlov, "Stability of the block cyclic reduction," *Linear Algebra and its applications*, vol. 249, no. 1, pp. 341–358, 1996.

[103] S.-C. Chang, T.-S. Chang, and P. B. Luh, "A hierarchical decomposition for large-scale optimal control problems with parallel processing structure," *Automatica*, vol. 25, no. 1, pp. 77–86, 1989.

[104] S. J. Wright, "Partitioned dynamic programming for optimal control," *SIAM Journal on optimization*, vol. 1, no. 4, pp. 620–642, 1991.

[105] D. D. Le, "Parallelisierung von Innere-Punkte-Verfahren mittels Cyclic Reduction." Bachelor's thesis, OvGU Magdeburg, 2014. Bachelorthesis.

[106] P. N. Swarztrauber and R. A. Sweet, "Algorithm 541: Efficient fortran subprograms for the solution of separable elliptic partial differential equations [d3]," *ACM Trans. Math. Softw.*, vol. 5, pp. 352–364, Sept. 1979.

[107] S. P. Hirshman, K. S. Perumalla, V. E. Lynch, and R. Sánchez, "Bcyclic: A parallel block tridiagonal matrix cyclic solver," *Journal of Computational Physics*, vol. 229, no. 18, pp. 6392–6404, 2010.

[108] Y. Zhang, J. Cohen, and J. D. Owens, "Fast tridiagonal solvers on the gpu," *ACM Sigplan Notices*, vol. 45, no. 5, pp. 127–136, 2010.

[109] C. D. Laird, A. V. Wong, and J. Åkesson, "Parallel solution of large-scale dynamic optimization problems," in *21st European Symposium on Computer-Aided Process Engineering*, 2011.

[110] W. Gander and G. H. Golub, "Cyclic reductionhistory and applications," *Scientific computing (Hong Kong, 1997)*, pp. 73–85, 1997.

[111] J. Kang, N. Chiang, C. D. Laird, and V. M. Zavala, "Nonlinear programming strategies on high-performance computers," in *Proc. of the IEEE Conference on Decision and Control, Osaka, Japan*, 2015.

[112] N.-Y. Chiang and V. M. Zavala, "An inertia-free filter line-search algorithm for large-scale nonlinear programming," *Computational Optimization and Applications*, pp. 1–28, 2016.

[113] I. S. Duff, "Ma57—a code for the solution of sparse symmetric definite and indefinite systems," *ACM Trans. Math. Softw.*, vol. 30, pp. 118–144, June 2004.

[114] MATLAB, *MATLAB function reference*, ch. mldivide, pp. 5702–5709. The MathWorks Incorporated, 2015.

[115] T. A. Davis, "Algorithm 832: Umfpack v4.3—an unsymmetric-pattern multifrontal method," *ACM Trans. Math. Softw.*, vol. 30, pp. 196–199, June 2004.

[116] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK Users' Guide*. Philadelphia, PA: Society for Industrial and Applied Mathematics, third ed., 1999.

[117] H. G. Bock and K.-J. Plitt, "A multiple shooting algorithm for direct solution of optimal control problems," in *PROCEEDINGS OF THE IFAC WORLD CONGRESS*, 1984.

[118] R. Sargent and G. Sullivan, "The development of an efficient optimal control package," in *Optimization Techniques*, pp. 158–168, Springer, 1978.

[119] D. Kraft, "On converting optimal control problems into nonlinear programming problems," in *Computational mathematical programming*, pp. 261–280, Springer, 1985.

[120] A. C. Hindmarsh, P. N. Brown, K. E. Grant, S. L. Lee, R. Serban, D. E. Shumaker, and C. S. Woodward, "Sundials: Suite of nonlinear and differential/algebraic equation solvers," *ACM Trans. Math. Softw.*, vol. 31, pp. 363–396, Sept. 2005.

[121] R. Hockney and C. Jesshope, *Parallel computers: architecture, programming and algorithms*. Adam Hilger, 1981.

[122] C. Qu and J. Hahn, "Computation of arrival cost for moving horizon estimation via unscented kalman filtering," *Journal of Process Control*, vol. 19, pp. 358–363, 2009.

[123] S. Rajaraman, J. Hahn, and M. Mannan, "A methodology for fault detection, isolation, and identification for nonlinear processes with parametric uncertainties," *Ind. Eng. Chem. Res.*, vol. 43, no. 21, pp. 6774–6786, 2004.

Appendix A

Mathematical Models

This appendix contains mathematical models used in this thesis. The first model is for a general 3 state CSTR and the second model is for a binary distillation column. Both models are index-1 DAE systems.

A.1 General 3 State CSTR

The dynamic CSTR model shown below describes the exothermic reaction between sodium thiosulfate(component A) and hydrogen peroxide(component B). The model comes from Qu and Hahn[122] and values for the model parameters were taken from Rajaraman *et al.*[123].

$$\frac{dC_A}{dt} = \frac{F}{V}(C_A^{in} - C_A) - 2k(T_R)C_A^2 \tag{A.1a}$$

$$\frac{dT_R}{dt} = \frac{F}{V}(T_R^{in} - T_R) + \frac{2(-\Delta H_R)k(T_R)C_A^2}{\rho C_P} - \frac{UA}{V\rho C_P}(T_R - T_{cw}) \tag{A.1b}$$

$$\frac{dT_{cw}}{dt} = \frac{F_{cw}}{V_{cw}}(T_{cw}^{in} - T_{cw}) + \frac{UA}{V_{cw}\rho_{cw}C_{Pcw}}(T_R - T_{cw}) \tag{A.1c}$$

$$k(T_R) = k_0 \exp\left[\frac{-E_a}{RT_R}\right] \tag{A.1d}$$

$$C_A \in [0,1], T_R \in [200,420], T_{cw} \in [200,420] \tag{A.1e}$$

The 3 states in this model, C_A, T_R, and T_{cw}, represent the concentration of A, the reactor temperature, and the cooling water temperature, respectively. The known inputs are the feed and cooling water flow rates, F and F_{cw}. The inlet concentration of A is given by C_A^{in}, the temperature of the inlet stream is T_R^{in}, and the temperature of the inlet cooling water

is T_{cw}^{in}. The other parameters in the model are as follows: reactor volume (V), reactor area (A), reaction heat (ΔH_R), inlet feed stream density (ρ), inlet feed heat capacity (C_P), global heat transference coefficient (U). Similar parameters for the cooling water are represented with the subscript cw.

A.2 Distillation Column Model

The distillation column model presented here separates methanol and n-propanol. The model is based off an index-2 DAE system reported in [58]. This model was later reformulated to be index-1 by Lopez-Negrete *et al.*[59]. The index-1 model equations are shown below. We consider a column with $N_T = 40$ trays along with a total condenser and a reboiler. Trays are numbered from the bottom up with the reboiler being $i = 0$ and the condenser being $i = N_T + 1$. The feed stream enters the column at tray $i = 21$. Variable and coefficient definitions are listed in the nomenclature section at the end of this section. Assumptions in this model include an ideal vapor phase, constant pressure drop across the trays, and negligible vapor molar holdups.

First, the following mass balances are enforced for each stage i.

$$\dot{M}_i = V_{i-1} - V_i + L_{i+1} - L_i + F_i, \quad i = 1, ..., N_T \tag{A.2a}$$

$$\dot{M}_{N_T+1} = V_{N_T} - D - L_{N_T+1} \tag{A.2b}$$

$$\dot{M}_0 = L_1 - V_0 - B \tag{A.2c}$$

$$R = \frac{L_{N_T+1}}{D} \tag{A.2d}$$

$$M_i \dot{x_i} = V_{i-1}(y_{i-1} - x_i) - V_i(y_i - x_i) + L_{i+1}(x_{i+1} - x_i) + F_i(z_{f,i} - x_i) \tag{A.2e}$$

$$M_0 \dot{x_0} = L_1(x_1 - x_0) - V_0(y_0 - x_0) \tag{A.2f}$$

$$M_{N_T+1} \dot{x}_{N_T+1} = V_{N_T}(y_{N_T} - x_{N_T+1}) \tag{A.2g}$$

Next, the model includes the following equilibrium and summation equations. Raoult's law is used to model phase equilibrium with non-ideal behavior addressed using tray

efficiencies.

$$P_{i-1} = P_i + \Delta P_i, \quad i = 1, ..., N_{T+1} \tag{A.3a}$$

$$P_i = P^s_{i,1}(T_i)x_i + (1 - x_i)P^s_{i,2}(T_i) \tag{A.3b}$$

$$P^s_{i,j} = \exp\left(A_j - \frac{B_j}{T_i + C_j}\right) \tag{A.3c}$$

$$\dot{T}_i = -\frac{(P^s_{i,1} - P^s_{i,2})\dot{x}_i}{(\partial P^s_{i,1}/\partial T_i)x_i + (\partial P^s_{i,1}/\partial T_i)(1 - x_i)} \tag{A.3d}$$

$$y_i = \alpha_i x_i \frac{P^s_{i,1}}{P_i} + (1 - \alpha_i)y_{i-1} \tag{A.3e}$$

$$y_0 = x_0 \frac{P^s_{0,1}}{P_0} \tag{A.3f}$$

The model also includes energy balances.

$$h^L_i(x_i, T_i) = x_i \bar{h}^L_{i,1}(T_i) + (1 - x_i)\bar{h}^L_{i,2}(T_i) \tag{A.4a}$$

$$h^V_i(y_i, T_i, P_i) = y_i \bar{h}^V_{i,1}(T_i, P_i) + (1 - y_i)\bar{h}^V_{i,2}(T_i, P_i) \tag{A.4b}$$

$$\dot{M}_i h^L_i + M_i \left(\dot{x}_i \frac{\partial h^L_i}{\partial x_i} + \dot{T}_i \frac{\partial h^L_i}{\partial T_i}\right) = V_{i-1}h^V_{i-1} - V_i h^V_i - V_i h^V_i +$$
$$L_{i+1}h^L_{i+1} - L_i h^L_i + F_i h^L(z_{f,i}, T_{f,i}, P_{f,i}) \tag{A.4c}$$

$$\dot{M}_0 h^L_0 + M_0 \left(\frac{\partial h^L_0}{\partial x_0}\dot{x}_0 + \frac{\partial h^L_0}{\partial T_0}\dot{T}_0\right) = Q_R - Q_{\text{loss}} - V_0 h^V_0 + L_1 h^L_1 - B h^L_0 \tag{A.4d}$$

$$\dot{M}_{N_T+1} h^L_{N_T+1} + M_{N_T+1}\left(\frac{\partial h^L_{N_T+1}}{\partial x_{N_T+1}}\dot{x}_{N_T+1} + \frac{\partial h^L_{N_T+1}}{\partial T_{N_T+1}}\dot{T}_{N_T+1}\right) = V_{N_T}\left(h^V_{N_T} - h^L_{N_T+1}\right) - Q_C \tag{A.4e}$$

and finally, equations for the hydrodynamics using the Francis weir formula.

$$n^v_i = M_i V^m_i(x_i, T_i) \tag{A.5a}$$

$$V^m_i(x_i, T_i) = x_i \bar{V}^m_{i,1}(T_i) + (1 - x_i)\bar{V}^m_{i,2}(T_i) \tag{A.5b}$$

$$L_i V^m_i(x_i, T_i) = W_i \left(n^v_i - n^{v,\text{ref}}_i\right)^{\frac{3}{2}} \tag{A.5c}$$

Nomenclature

$\bar{h}_{i,j}^{L}$ Liquid enthalpy for pure component j on tray i

A_j, B_j, C_j Antoine's equation constants for component j

Q_R Reboiler heat duty

V_i Vapor flow rate leaving tray i

α_i Tray i efficiency

$\bar{h}_{i,j}^{V}$ Vapor enthalpy for pure component j on tray i

$\bar{V}_{i,j}^{m}$ Molar volume of pure component j on tray i

B Bottoms flow rate from reboiler

D Distillate flow rate from condenser

F_i Feed flow rate entering tray i

h_i^L Liquid enthalpy on tray i

h_i^V Vapor enthalpy on tray i

L_i Liquid flow rate leaving tray i

M_i Molar hold-up on tray i

n_i^v Liquid volume holdup on tray i

$n_i^{v,\mathrm{ref}}$ Volume of tray i

P_i Total pressure on tray i

$P_{i,j}^{s}$ Vapor pressure of component j on tray i

Q_C Condenser heat duty

Q_{loss} Reboiler heat loss

R Reflux ratio

T_i Temperature on tray i

V_i^m Molar volume of liquid on tray i

W_i Weir constant for tray i

x_i Mole fraction of methanol in liquid on tray i

y_i Mole fraction of methanol in vapor on tray i

$z_{f,i}$ Mole fraction of methanol in feed entering tray i