Streamlining of the state-dependent Riccati equation controller algorithm for an embedded implementation

Sergey Katsev

Follow this and additional works at: http://scholarworks.rit.edu/theses

Recommended Citation

This Thesis is brought to you for free and open access by the Thesis/Dissertation Collections at RIT Scholar Works. It has been accepted for inclusion in Theses by an authorized administrator of RIT Scholar Works. For more information, please contact ritscholarworks@rit.edu.
Streamlining of the State-Dependent Riccati Equation Controller Algorithm for an Embedded Implementation

by

Sergey Katsev

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of Master of Science in Computer Engineering

Supervised by

Associate Professor of Computer Engineering Dr. Juan Cockburn
Department of Computer Engineering
Kate Gleason College of Engineering
Rochester Institute of Technology
Rochester, New York
November 2006

Approved By:

Dr. Juan Cockburn
Associate Professor of Computer Engineering
Primary Adviser

Dr. Marcin Lukowiak
Assistant Professor of Computer Engineering

Dr. Roy S. Czernikowski
Professor of Computer Engineering
Thesis Release Permission Form

Rochester Institute of Technology
Kate Gleason College of Engineering

Title: Streamlining of State-Dependent Riccati Equation Controllers for an Embedded Implementation

I, Sergey Katsev, hereby grant permission to the Wallace Memorial Library reproduce my thesis in whole or part.

Sergey Katsev

Date
Dedication

To my grandparents and my mom, who gave up what they had in the Ukraine so that I could make a life for myself in the United States.
First, I would like to thank my advisor, Dr. Juan C. Cockburn for being patient and mentoring me through this work. Also I would like to thank my committee, Dr. Roy S. Czernikowski and Dr. Marcin Lukowiak. Many thanks go to the other people and organizations that helped me with technical aspects, in particular David Bishop and other members of the IEEE VHDL-200X Fast Track group and the authors of the LAPACK and SLICOT libraries. I would also like to thank Eric Nelson and John Schnyder for general advice and proofreading. Finally, I would like to thank Dr. Andreas Savakis for giving me the final push I needed.
Abstract

In many practical control problems the dynamics of the plant to be controlled are nonlinear. However, in most cases the controller design is based on a linear approximation of the dynamics. One of the reasons for this is that, in general, nonlinear control design methods are difficult to apply to practical problems. The State Dependent Riccati Equation (SDRE) control approach is a relatively new practical approach to nonlinear control that has the simplicity of the classical Linear Quadratic control method. This approach has been recently applied to control experimental autonomous air vehicles with relative success. To make the SDRE approach practical in applications where the computational resources are limited and where the dynamic models are more complex it would be necessary to re-examine and streamline this control algorithm. The main objective of this work is to identify improvements that can be made to the implementation of the SDRE algorithm to improve its performance. This is accomplished by analyzing the structure of the algorithm and the underlying functions used to implement it.

At the core of the SDRE algorithm is the solution, in real time, of an Algebraic Riccati Equation. The impact of the selection of a suitable algorithm to solve the Riccati Equation is analyzed. Three different algorithms were studied. Experimental results indicate that the Kleinman algorithm performs better than two other algorithms based on Newton’s method. This work also demonstrates that appropriately setting a maximum number of iterations for the Kleinman approach can improve the overall system performance without degrading accuracy significantly.

Finally, a software implementation of the SDRE algorithm was developed and benchmarked to study the potential performance improvements of a hardware implementation. The test plant was an inverted pendulum simulation based on experimental hardware. Bottlenecks in the software implementation were identified and a possible hardware design to remove one such bottleneck was developed.
# Contents

Dedication ........................................................................................................ iii

Acknowledgments ............................................................................................ iv

Abstract ............................................................................................................ v

Glossary .............................................................................................................. xi

1 Introduction .................................................................................................... 1
   1.1 Thesis outline ........................................................................................... 5

2 Supporting Work ............................................................................................. 7
   2.1 Background ............................................................................................... 7
   2.2 SDRE methodology .................................................................................. 8
      2.2.1 Linear Quadratic Regulator [1] ......................................................... 9
      2.2.2 SDRE controller design ................................................................. 9
      2.2.3 SDC parametrization ....................................................................... 12
      2.2.4 Selection of $Q$ and $R$ matrices ................................................... 13
   2.3 The Continuous-Time Algebraic Riccati Equation .................................... 14
   2.4 Riccati Equation solution algorithms ..................................................... 16
      2.4.1 Newton algorithm ......................................................................... 17
      2.4.2 Newton’s method with Exact Line Search [7] ............................... 18
      2.4.3 Kleinman iteration [25, 26] ......................................................... 19
      2.4.4 Schur Decomposition .................................................................... 20
   2.5 Previous SDRE implementations ............................................................ 21

3 Iterative Method Comparison ...................................................................... 24
   3.1 Accuracy analysis .................................................................................... 25
   3.2 Speed analysis ........................................................................................ 27
   3.3 Kleinman maximum iterations ................................................................ 35
List of Figures

2.1 SDRE Algorithm Diagram ........................................ 12
2.3 Pseudo-Code for Exact Line Search Algorithm ................. 18
2.4 Pseudo-Code for Kleinman Algorithm .......................... 19
2.5 Pseudo-code for Schur Algorithm [27] .......................... 23

3.1 Iterative method accuracy ........................................ 26
3.2 Matlab code for one Kleinman Algorithm iteration ............. 28
3.3 Matlab code for one Newton Algorithm iteration ............... 28
3.4 Average iterations to converge to a solution .................... 30
3.5 Total Newton and Kleinman iterations .......................... 31
3.6 Kleinman and Newton Speed Results with Trend Line ........... 34
3.7 Effect of Kleinman iterations on number of total algorithm invocations ... 36
3.8 Effect of Kleinman iterations on result accuracy ................ 37
3.9 Kleinman Max Iterations vs. Total Time ......................... 39
3.10 Sample Application Maximum Kleinman Iteration Complexity ... 41

4.1 Pendulum Parameters Block Diagram ............................ 43
4.2 Quanser Consulting Inc. Linear Inverted Pendulum [33] ........ 44
4.3 Inverted Pendulum diagram, including sensor and actuator locations ... 45
4.4 Inverted Pendulum Simulink Model ............................. 46
4.5 Inverted Pendulum Models sub-block ............................ 47
4.6 Simulation: Pendulum Angle ..................................... 49
4.7 Simulation: Cart Position ......................................... 50
4.8 Simulation: Pendulum Control .................................... 51
4.9 Simulation: Pendulum angle at steady state with pulse disturbance .... 52
4.10 Simulation: Cart position at steady state with pulse disturbance .... 53
4.11 Simulation: Pendulum control action at steady state with pulse disturbance ... 54
4.12 Q & R matrix analysis: Angle under-penalization ................ 54
4.13 Q & R matrix analysis: Angle over-penalization – Pendulum Angle .... 55
List of Tables

2.1 CARE solution algorithms summary ........................................... 21
2.2 Execution Time Comparisons for Schur and Kleinman Methods ........ 22
2.3 Previous Work Summary .......................................................... 23

3.1 Effect of Step Size ($t$) on Exact Line Search Convergence .............. 27
3.2 Iterative Method Accuracy Results ............................................ 29
3.3 Iterative Method Timing Results (seconds) .................................. 32
3.4 Kleinman iterations timing test .................................................. 38
3.5 Maximum Iterations Computation using Equation 3.2 ...................... 40

4.1 Quanser Inverted Pendulum parameters ..................................... 46

6.1 Function Execution Time .......................................................... 66
6.2 Inverted Pendulum Maximum Value .......................................... 77
6.3 Multiplication Unit Fraction Precision Results .............................. 80
6.4 Matrix Multiplication Synthesis Results ..................................... 81
A

**adaptive control**  A control technique which involves modifying the control law used to compensate for the fact that the system parameters may be varying or uncertain [17].

**affine**  This term refers to a function with a constant slope. A function is affine if it is equal to a constant plus a linear function of variables [31].

C

**causality**  The outputs of a causal system depend only on the current and past inputs, not on future inputs [18].

**controllable**  A controllable system is one where each state can be steered to the origin using a control within a finite amount of time. For the formal definition, see Appendix A.

D

**detectable**  A detectable system is one in which the unobservable states are stable. For the formal definition, see Appendix A.
**F**

**feedback linearization**  A linearization technique which develops a control input that makes the closed-loop system from inputs to outputs linear [32].

**full-state feedback**  Refers to the property of certain control laws that are functions of all the states of the system.

**G**

**gain margin**  A measure of stability for a feedback system (along with phase margin). Gain margin is the difference between a unity gain and \(|A_{OL}\beta(\omega_{180})|\) where \(\omega_{180}\) is the frequency at which the open loop phase is \(-180^\circ\). The gain margin can be viewed as the change in open loop gain required to make the closed loop system unstable. Systems with larger gain margins can withstand larger changes to their parameters without becoming unstable in a closed loop [30].

**gain scheduling**  A non-linear control technique which attempts to control a system by applying different linear controllers at different operating points.

**H**

**Hamilton-Jacobi equation (HJE)**  A first-order, non-linear partial differential equation that arises in the solution of optimal control problems such as the Linear Quadratic Regulator. In general, this equation is very difficult to solve.

**Hamiltonian matrix**  Any matrix \(A\) which satisfies the condition \(KA\) is symmetric, where \(K = \begin{pmatrix} 0 & I_n \\ -I_n & 0 \end{pmatrix}\) is a signature matrix. This matrix has several useful properties: i) \(A^T\) is a Hamiltonian matrix, ii) \(trace(A) = 0\), and iii) Given \(A\) and \(B\) Hamiltonian matrices, \(AB\) is a Hamiltonian matrix if and only if \(A\) and \(B\) commute [1].
**Hurwitz matrix**  A square matrix is called a Hurwitz matrix if all of its eigenvalues have real parts less than zero.

**I**

**input-affine**  A nonlinear system which is affine in the inputs, but not necessarily in the states.

**J**

**Jacobian linearization**  A linearization technique that approximates a nonlinear differential equation by retaining the first term of its Taylor Series expansion near an operating point $p$.

**L**

**LAPACK**  (L)inear (A)lgebra (PACK)age. This is a library of routines written in Fortran 77 for solving various linear algebra problems [2].

**linear system**  A system which satisfies the properties of superposition and scaling. i.e. Given two inputs $x_1(t)$ and $x_2(t)$ and their outputs $y_1(t) = H(x_1(t))$ and $y_2(t) = H(x_2(t))$, a linear system satisfies $\alpha y_1(t) + \beta y_2(t) = H(\alpha x_1(t) + \beta x_2(t))$ for any scalar $\alpha$ and $\beta$ [18].

**linearization**  The process of finding a linear approximation to a function. This approximation can be valid at only a given point (local linearization), or it can be valid at every point in the function (global linearization). Global linearization is generally impossible for complex functions, as it requires an infinite sum of linear functions.

**loop shaping**  A frequency domain control design method based on modifying the Bode of the open loop system to achieve desired performance specifications.
**M**

**minimum-phase property** A system is minimum-phase if both it and its inverse are causal and stable. Equivalently, it contains no poles or zeros in the open right half plane [32].

**N**

**negative-definite** A symmetric matrix \( M \in \mathbb{R}^{n \times n} \) is negative-definite if \( x^T M x < 0 \) for all non-zero \( x \in \mathbb{R}^n \), or equivalently, all of its eigenvalues are negative.

**negative-semidefinite** A symmetric matrix \( M \in \mathbb{R}^{n \times n} \) is negative-semidefinite if \( x^T M x \leq 0 \) for all non-zero \( x \in \mathbb{R}^n \), or equivalently, all of its eigenvalues are less than or equal to zero.

**O**

**observable** An observable system is one in which every state causes has an effect on the output. For a formal definition, see Appendix A.

**observer** A state observer or *estimator* is a system that estimates the states of another system by “observing” its inputs and outputs.

**optimal control** A field of control system theory which is concerned with finding a control law which minimizes a certain performance function [1].

**P**

**phase margin** A measure of stability for a feedback system (along with *gain margin*). The phase margin can be viewed as the change in open loop phase shift required to make a closed loop system unstable. The phase margin also measures the system’s tolerance to time delay [30].
**positive-definite** A symmetric matrix $M \in \mathbb{R}^{n \times n}$ is positive-definite if $x^T M x > 0$ for all non-zero $x \in \mathbb{R}^n$, or equivalently, all of its eigenvalues are positive.

**positive-semidefinite** A symmetric matrix $M \in \mathbb{R}^{n \times n}$ is positive-semidefinite if $x^T M x \geq 0$ for all non-zero $x \in \mathbb{R}^n$, or equivalently, all of its eigenvalues are greater than or equal to zero.

**R**

**real-time system** A system which is subject to some kind of specific timing constraint [36]. This period of time for the constraint can vary, but for control systems is related to the bandwidth of the system being controlled by the Nyquist sampling theorem.

**root locus** A graphical representation of the locus of the closed loop poles of a system as the values of a gain in the characteristic polynomial change, usually from 0 to $\infty$.

**S**

**sampling period** The time between samples taken from a continuous signal to make a discrete signal.

**stabilizable** A stabilizable system is one where the uncontrollable states are stable. For the formal definition, see Appendix A.

**state-dependent coefficient parametrization (SDC parametrization)** A representation of the state equations of a nonlinear system of the form $A(x)x + B(x)u$, hence the name state-dependent. A given SDC parametrization of a nonlinear system is not unique [14].
**State Dependent Riccati Equation (SDRE)**  A Continuous-Time Algebraic Riccati Equation which contains coefficient matrices whose values depend on the states of the system being controlled.

**state space model**  A mathematical model of a physical system consisting of a set of first order differential equations, each describing the evolution of one state, and an output equation.
Chapter 1

Introduction

The theory of control for linear systems has been in development for over a century. Today, many different linear control approaches exist, including root locus, loop shaping, state-space design [18], and optimal control [1]. The first three methods are examples of classical control. Although they are able to stabilize a system and satisfy constraints imposed by the designer, they are not guaranteed to produce a system with the best possible state trajectory as it is regulated. The last item, optimal control, is a modern control technique which does just that: the resulting system performs optimally [1]. This means that there is no other controller which can regulate the system to the reference point by spending less energy. There are many fields, including physics, economics, ecology, medicine, etc. that rely on optimal control [28].

One of the most important contributions to optimal control theory is the theory of Linear Quadratic Regulators (LQR) [1]. The method gets its name because in it, the control law is the solution to an optimization problem where the cost function is a quadratic equation of the states and inputs and the system has linear dynamics. The Linear Quadratic Regulator assumes the availability of all the states and thus results in a full state feedback law. It has a guaranteed gain margin ($k_g$) of $\infty$ and phase margin ($\gamma$) of at least 60 degrees. While these margins satisfy the practical guidelines for control system design, they are based on the assumption that the model of a plant is very close to the actual dynamics. Many practical applications result in nonlinear dynamic models and therefore the LQR approach cannot be directly applied. In principle, one should use a nonlinear controller
design approach in these cases. However, most nonlinear control techniques, like adaptive control, require extensive calculations. Because of this, they either require powerful computers, or can not be used for real-time applications. Therefore, it is of practical importance to develop a computationally-simpler control algorithm that would be able to obtain similar performance on more complex systems.

The State Dependent Riccati Equation (SDRE) is one approach that presents a compromise between computationally-intensive non-linear control algorithms and simple but inadequate linear control algorithms. The SDRE approach takes advantage of Semi-global linearization (Section 2.1) to produce a model of the system on which linear control techniques (LQR) can be used. The method uses a state-dependent coefficient parametrization (Section 2.2) to produce a constant state-space model of the system at each sampling period, thus effectively creating a linear system out of a non-linear one. However, the overall system is still nonlinear and time-varying. By minimizing a quadratic performance function at each sampling time, a full-state feedback control law similar to the LQR controller is obtained.

The SDRE method has been extensively studied in recent years [4, 12, 14, 17, 21, 29, 38], with several important aspects of the algorithm being established. Cloutier and Beeler [4, 12] present the SDRE technique and review the existing control theory associated with it, demonstrating the applicability and performance of the method. Hammett et al. [21] analyzed the stability of the method, while Hull [22] demonstrated some of the problems with the SDRE method when proper offline-parameters are not selected and proposed solutions to those problems, proving the ruggedness of the method. Cockburn [13] showed how to obtain an SDRE feedback law when soft constraints are imposed on the states of the system.

Other researchers showed the applicability of the SDRE approach to real-time systems. Erdem [17] uses SDRE to control a double inverted pendulum in real time using Mat- lab. Menon et al. [29] concentrate on producing a real-time software package for solving SDRE. One of the motivations for their work is to advance the acceptance of the SDRE
method by disproving the notion that it requires advanced numerical algorithms and computational resources. The software package presented in [29] is able to regulate a six-state missile flight control system in real-time with a sampling rate of 2 kHz. Similarly, Bogdanov [10] implemented an SDRE controller for real-time control of a helicopter based on a twelve-state model. The results in [10] show that a Geode GX1 processor is able to calculate 71 control actions per second. (However, for this work, “quantitative measures of the control were difficult to obtain from the recorded data, as the effect of the pilot feedback in ‘correcting’ an observed trajectory cannot be accounted for”).

The common goal in all of this work is to make the SDRE method applicable for a wider variety of practical control problems. However, although [29] and [10] were able to implement stand-alone real-time systems controlled by the SDRE approach, it is noticeable that the performance of the controller dropped significantly between a six and a twelve state system. Clearly, if the SDRE approach is to be used in real-time with more complex systems, the algorithm must be streamlined to improve its performance.

To describe the problem that will be addressed in this work, a quick description of the SDRE method will be given (for more details see Section 2.2). First, the designer determines an appropriate State-Dependent Coefficient (SDC) parametrization for the plant to be controlled. Then, off-line parameters, like the $Q$ and $R$ weighing matrices that represent the performance specifications of the system are found. As states are measured, the state-dependent state equations of the dynamic model are evaluated, the associated Algebraic Riccati Equation (Section 2.3) is formed and solved to calculate the required control action in that sampling period. There are a number of algorithms to solve Algebraic Riccati Equations. These algorithms belong to two groups: numerically robust algorithms, which calculate the Riccati Equation solution from its Hamiltonian Matrix, and those which iteratively determine the solution. The proposed SDRE implementations [29] use one method from each group: a numerically robust method to compute the solution when a previous solution doesn’t exist (e.g., to initialize or reset the algorithm), and an iterative technique to refine an existing solution when the system dynamics have not changed significantly since
it was calculated. Since in some instances the iterative approach may fail to converge, so the numerically robust approach is used also as a back-up.

The numerically robust algorithm investigated in this work is the Schur method (Section 2.4.4), which is reliable but slow. The iterative algorithm used for refinement is Kleinman’s Algorithm (Section 2.4.3). In our experiment we observed that with a good choice of SDC parametrization and off-line parameters, the Schur method is used very infrequently (0.8% of SDRE solutions for the example presented in this work). Because of this, to improve the performance (average speed of a Riccati Equation solution) of the SDRE approach, the focus should be on improving the iterative method.

The aim of this work is to improve the performance of SDRE by finding bottlenecks that can be removed. We analyze several aspects of the SDRE algorithm as it is described above and in Section 2.2. To accomplish this, the SDRE algorithm is simulated using Matlab as it controls a test application: an inverted pendulum. The effect of the proper selection of off-line parameters on performance of the control system using the Matlab simulation is analyzed. We then examine the design of the SDRE implementation by comparing several iterative algorithms used in the Riccati Equation solution. Particularly, the performance of the Kleinman algorithm is compared with two other iterative algorithms: Newton’s Method and Newton’s Method with Exact Line Search. The results show that Kleinman’s method, which is the iterative method used in all of the implementations mentioned above is, in fact, the best performing of the three. The tests proposed in this work can be used to compare other iterative algorithms as well.

After selecting the best iterative method to use for the SDRE approach, the full SDRE algorithm is implemented in C and benchmarked, allowing for analysis of the bottlenecks within the algorithm. Finally, a hardware-software co-design is presented for one of the bottleneck elements. It is demonstrated that this is an efficient way to gain performance for a portion of the algorithm that performs poorly in a software implementation.

The contributions of this work are to several portions of the SDRE approach. We show
the effect of off-line parameter selection on the final performance of the method. In particular, the effect of $Q$ and $R$ matrix selection is demonstrated. We compare three refinement algorithms, selecting the best one for the SDRE method. We also show that limiting the maximum number of iterations for the selected refinement algorithm can improve the overall performance of the system. Finally, find bottlenecks present in the SDRE approach and design a hardware component to remove one of these bottlenecks.

1.1 Thesis outline

We start the background discussion in Section 2.1 with a discussion of problems with existing control algorithms. The SDRE method is presented in Section 2.2 because it has characteristics which avoid these problems.

We go on to establish the SDRE method in detail, particularly the selection of off-line parameters like SDC Parametrization and $Q$ and $R$ matrix selection. Before we can discuss the solution of the SDRE, the Riccati Equation is presented in Section 2.3. The various algorithms for solving this equation are then presented in Section 2.4. With all of the theory for SDRE established, Chapter 2 ends with a discussion of previous SDRE implementations.

Because a picture (or in this case an example) is worth a thousand words, an application for use in the rest of the work is presented in Chapter 4. The inverted pendulum is used because it is inherently non-linear but still simple enough to demonstrate the main issues involved in the design and implementation of SDRE controllers. We begin our own work in Chapter 3 with an analysis of the iterative algorithms used for SDRE calculation. In Chapter 5 we analyze the maximum number of iterations to use for the Kleinman algorithm, and the effect of changing this maximum number of iterations on the performance of the SDRE. That chapter also contains the analysis of the SDRE software implementation’s bottlenecks (a full description of the SDRE software package can be found in Appendix B). Of these bottlenecks, the LAPACK function $dhseqr$ is selected for improvement because it has the
highest overall execution time. A hardware implementation of this function is presented in Chapter 6. Finally, conclusions and suggestions for future work are presented in Chapter 7.
Chapter 2

Supporting Work

This chapter goes into the details of what is presented in the introduction. We start by justifying the usefulness of the SDRE approach in Section 2.1. The SDRE method is described in detail in Section 2.2. We also describe the process of SDC Parametrization and the selection of $Q$ and $R$ matrices because these choices determine whether the SDRE is a well-conditioned equation. Before presenting algorithms that can be used to solve the State Dependent Riccati Equation in Section 2.4, the continuous-time Algebraic Riccati Equation is explained in Section 2.3. The chapter ends with an overview of previous SDRE method implementations.

2.1 Background

Linear system control is an area of control theory that is well developed. Part of the reason for this is the fact that this subject has been studied for over a century. Another reason is that until recently, linear control design techniques were able to be satisfactorily applied to most systems. However, because performance requirements of control systems keep increasing, the mathematical models of those systems (which are used to design the control) are constantly getting more complex. Models which are able to represent a system over its entire operating range are often nonlinear [17]. Because of this, it became necessary to develop and apply non-linear control techniques.
Several non-linear control techniques such as Jacobian Linearization, Feedback Linearization, Adaptive Control, and Gain Scheduling exist. Techniques like Jacobian Linearization and Feedback Linearization are only effective for a small subset of nonlinear systems, since they rely on the system remaining in the linearized region where the model is accurate. Adaptive Control is a great technique for linear control, but it has not yet been sufficiently developed for non-linear control. In addition, the computation cost of Adaptive Control is high. Gain Scheduling does not have a rigorous-enough method to reliably produce valid controllers [17].

The main problems with existing nonlinear control algorithms can be listed as: high computational cost, lack of structure, and poor applicability. One method that avoids these problems is the State Dependent Riccati Equation approach. This method uses a simple, systematic procedure. Furthermore, it is applicable to a wide range of nonlinear systems.

### 2.2 SDRE methodology

The non-linear control methods with low ranges of applicability all use Local Linearization techniques to form a model of a plant. Even though this results in a simple model, the techniques must re-linearize the model very frequently or rely on the fact that the system will not leave the very small linearized region during operation. More advanced techniques, like Adaptive Control and Gain Scheduling use much more complex globally linearized models of the plant. The SDRE approach uses a semi-global linearization – a compromise between applicability and complexity.

These two issues (applicability and complexity) with non-linear control algorithms are addressed by the SDRE approach because of its relationship to the Linear Quadratic Regulator (LQR). SDRE essentially treats non-linear (input-affine) systems as linear systems. Furthermore, simulation results show that the SDRE approach has robustness characteristics similar to the LQR [17].
2.2.1 Linear Quadratic Regulator [1]

The standard Linear Quadratic Regulator technique is summarized since it is the basis for the SDRE approach. The LQR can be applied to linear systems of the form:

\[ \dot{x}(t) = A(t)x(t) + B(t)u(t) \quad (2.1) \]
\[ y(t) = C(t)x(t) \quad (2.2) \]

Here, \( A(t) \), \( B(t) \) and \( C(t) \) are, in general, matrix functions of time, and \( x(t) \) represents the system state. The performance specifications are captured by the quadratic performance index function which represents the trade-off between the size of the states and the control action and is given by:

\[ J = \frac{1}{2} \int_0^\infty x^T Q x + u^T R u \, dt \quad (2.3) \]

The LQR approach finds a control law \( (u) \) that minimizes this performance index. A unique property of this optimal controller is that the control law has state-feedback form, that is, \( u \) depends on the states only.

The SDRE approach can be regarded as an extension of the LQR. It produces a sub-optimal non-linear controller. This is because of the approximations required in parametrization of the non-linear system, as well as using a Riccati Equation to approximate the solution to the optimal control problem rather than solving the corresponding Hamilton-Jacobi equation [14].

2.2.2 SDRE controller design

The first two advantages of the SDRE method have been presented above. Now, the third advantage: a systematic controller-generation process is presented. Note the similarity between this approach and the LQR.

Instead of using a linear model, the SDRE starts with an input-affine nonlinear model of the form [12]:

\[ \dot{x} = f(x) + g(x)u \quad (2.4) \]
with associated performance index:

$$J = \frac{1}{2} \int_0^\infty x^T Q(x)x + u^T R(x)u \, dt$$  \hspace{1cm} (2.5)$$

where $$x \in \mathbb{R}^n$$, $$u \in \mathbb{R}^m$$, $$f(x) \in \mathcal{C}^k$$, $$g(x) \in \mathcal{C}^k$$, $$Q(x) \in \mathcal{C}^k$$, $$R(x) \in \mathcal{C}^k$$, $$k \geq 1$$, $$Q(x) \geq 0$$, and $$R(x) > 0$$ for all $$x$$. It is assumed that $$f(0) = 0$$ and $$g(x) \neq 0$$ for all $$x$$.

The problem can now be formulated as a minimization problem associated with the performance index in Equation 2.5:

$$\min_u \frac{1}{2} \int_0^\infty x^T Q(x)x + u^T R(x)u \, dt$$

subject to

$$\dot{x} = f(x) + g(x)u$$

$$x(0) = x_0$$

The solution of this problem is equivalent to solving an associated Hamilton-Jacobi equation (HJE) [1]. However, because solving the Hamilton-Jacobi equation is very difficult, the HJE is approximated using a State Dependent Riccati Equation (Equation 2.8). This makes the problem feasible, although it leads to a suboptimal controller.

The control law in this problem, like the LQR, is also a state feedback law which depends on the solution to the State Dependent Riccati Equation. This can be seen by re-writing the system in Equation 2.4 as:

$$\dot{x} = A(x)x + B(x)u$$  \hspace{1cm} (2.7)$$

where $$f(x) = A(x)x$$ and $$B(x) = g(x)$$. This is known as the State-Dependent Coefficient (SDC) form. Note that the matrices $$A(x)$$ and $$B(x)$$ are functions of the states of the plant, and they become coefficients in the Riccati Equation (Section 2.3). It is important to notice that the SDC form is not unique. There are many possible $$A(x)$$ and $$B(x)$$ matrices. This will be explained in more detail in Section 2.2.3.

Once a SDC form has been found (which is done off-line by the designer), the SDRE approach is reduced to solving a LQR problem at each sampling instant. For a controller to exist, the conditions in the following definitions must be satisfied [14]:

10
**Definition 1** \( A(x) \) is an observable (detectable) parametrization of the nonlinear system for a given region if \([C(x), A(x)]\) are pointwise observable (detectable) for all \( x \) in that region.

**Definition 2** \( A(x) \) is a controllable (stabilizable) parametrization of the nonlinear system for a given region if \([A(x), B(x)]\) are pointwise controllable (stabilizable) for all \( x \) in that region.

If these standing assumptions do not hold, a stabilizing controller does not exist. Given these standing assumptions, the SDRE design proceeds as follows:

i. Start with a SDC form of the system to be controlled.

ii. Solve the state-dependent Riccati Equation (SDRE):

\[
A^T(x)P(x) + P(x)A(x) - P(x)B(x)R^{-1}(x)B^T(x)P(x) + Q(x) = 0 \tag{2.8}
\]

to obtain a positive, semi-definite matrix \( P(x) \).

iii. Construct the controller via:

\[
u = -R^{-1}(x)B^T(x)P(x)x \tag{2.9}\]

In cases where not all states are available we will need to design an observer to estimate the states that are not measured. In principle, a dual for the SDRE controller or other nonlinear observers could be used. In this work, we assume that full-state feedback exists.

The SDRE method is summarized in Figure 2.1 [29]. It is important to stress that the existence of the optimal control for a particular parametrization of the system is not guaranteed. Furthermore, there may be an infinite number of parameterizations of the system, so the choice of parametrization is very important. The other factor which may determine the existence of a solution to the Riccati Equation is the selection of the \( Q \) and \( R \) weighing matrices in the Riccati Equation (2.8).
2.2.3 SDC parametrization

A crucial step in the SDRE approach is the re-parametrization of the nonlinear system described by Equation 2.4 into a state-dependent form given in Equation 2.7.

The method of State-Dependent Coefficient (SDC) Parametrization is also called ‘apparent linearization’, ‘extended linearization’, or ‘state-dependent coefficient factorization’. It is simply a representation of the system so that although the system remains non-linear, linear methods such as the optimal linear quadratic regulator can be used for control [21].
It is important to realize that these parameterizations are non-unique. Furthermore, the use of a particular parametrization does not guarantee a system which can be solved using the SDRE approach [12] because the standing assumptions of stabilizability and detectability might be violated.

Although theory on whether a parametrization will work exists, it does not assist in finding a parametrization. Cloutier [14] discusses several approaches to obtain an optimal parametrization from a number of suboptimal ones. However this work will focus on an example system with a known-good SDC parametrization. See [14] for more information on obtaining a proper parametrization.

2.2.4 Selection of $Q$ and $R$ matrices

The solution to the Riccati Equation (discussed in more detail in Section 2.3) depends on the selection of the performance weights $Q$ and $R$. Furthermore, these matrices are design parameters that affect the overall performance of the closed loop system. Although there are no rigorous methods for selecting these matrices, this section provides some guidelines for their selection based on good practices.

$Q$ is a matrix of weighting coefficients used to penalize any state from becoming too large. Similarly, $R$ is used to penalize the control action. Thus, these matrices should be selected so that the states and inputs which should not get arbitrarily large (due to physical constraints, for example) are penalized the most [1]. The following guidelines are useful:

- A selection of $R$ which is a diagonal matrix simplifies calculations. If the dynamic model has been scaled (not generally true), a multiple of the identity matrix can be used. A diagonal matrix is particularly useful because this guarantees that the ‘penalty’ on each variable is independent of the others.

- If, based on the physical or design constraints of the system, a state variable or control input can not take too high of a value, the weight given to it should be large.

- $Q$ and $R$ can be chosen to be diagonal matrices with the $i^{th}$ diagonal entry of $Q^{-1}$
as \( n \) times the maximum acceptable value of \( x_i^2(t) \) and the \( i^{th} \) entry of \( R^{-1} \) \( m \) times the maximum value of \( u_i^2(t) \), where \( n \) is the number of states and \( m \) is the number of inputs. This is often referred to as Bryson's Rule [11].

- An increase in \( Q \) results in a faster response but more expensive control (the relative size of the control grows in relation to the size of the states), given that all other factors remain constant [14].

These guidelines help obtain better closed-loop performance [14]. Furthermore, they demonstrate a useful property of the SDRE approach: by penalizing the appropriate states or control action, this method allows the designer to set constraints on the system.

The previous sections presented the SDRE procedure and tunable parameters to improve the performance of the SDRE approach. We now begin looking at algorithms used to obtain the solution to the State Dependent Riccati Equation. First, we present a concise summary of Continuous-Time Algebraic Riccati Equations, and then discuss several algorithms for solving these equations numerically.

### 2.3 The Continuous-Time Algebraic Riccati Equation

Algebraic Riccati equations play a central role in Linear Quadratic control. The continuous time algebraic Riccati Equation (CARE) can be written as [1]:

\[
A^T P + PA - PBR^{-1}B^T P + Q = 0
\] (2.10)

where \( A \in \mathbb{R}^{n\times n}, B \in \mathbb{R}^{n\times m}, Q \in \mathbb{R}^{n\times n}, \) and \( R \in \mathbb{R}^{m\times m} \) are all constant.

The Hamiltonian matrix, \( M \), associated with this Riccati equation is the \( 2n \times 2n \) matrix:

\[
M = \begin{pmatrix}
A & -BR^{-1}B^T \\
-Q & -A^T
\end{pmatrix}
\] (2.11)

A Hamiltonian matrix has eigenvalues that are symmetric with respect to the imaginary
axis. To see this, note that $J^{-1}MJ = -JM = -M^T$ where

$$J = \begin{pmatrix} 0 & I \\ -I & 0 \end{pmatrix} \tag{2.12}$$

Thus, if $\lambda$ is an eigenvalue of $M$, then so is $-\lambda$. The Hamiltonian matrix is useful for numerical computation of solutions to the CARE.

The continuous time algebraic Riccati Equation has many solutions, since it is a matrix equation quadratic in the symmetric matrix $P$. Under certain stabilizability and detectability conditions (see Appendix A), there exists a unique stabilizing solution to the Riccati Equation which is the only positive-definite solution [20]. This solution has the property that the closed-loop state matrix has its eigenvalues in the open left-half plane and therefore, the closed loop system is stable. This is the solution mentioned in the previous paragraph. This solution exists and is unique if the following conditions hold true:

i. $(A, B)$ stabilizable

ii. $(A, D)$ detectable

iii. $Q \geq 0$

iv. $R > 0$

where $D$ is the square Choleski factor of $Q$ (e.g. $Q = D^T D$).

In this work, the $Q$ and $R$ matrices are selected to be constant, but in general they can be state dependent.

From a computational point of view, it is desirable to solve the Algebraic Riccati Equation analytically. However, this is not possible for higher-order systems. Because of this, numerical algorithms must be used. In the SDRE approach, these numerical algorithms must use state measurements at each sampling interval to calculate a new control action “in real-time.” Thus, for each sampling interval, an SDRE controller must read the system states, calculate the state matrices for that sampling interval, and then solve the associated State Dependent Riccati equation. The following section introduces some of the algorithms that can be used in this solution.
2.4 Riccati Equation solution algorithms

As discussed in the Introduction, there are two families of methods for numerically solving a Riccati Equation [3, 7, 25–27, 34, 37]. The first (Schur Decomposition [3, 27], matrix sign function [19], generalized eigenvector approach [3], etc) finds the steady state Riccati Equation solution from the Hamiltonian matrix, and is guaranteed to find the solution as long as it exists. The second family of methods iteratively determines the solution from an initial guess.

The SDRE approach is generally implemented using one algorithm from each of these families. The numerically robust method can not be altered because it is used as a backup when the iterative method fails, so this work focuses on improving the performance of the iterative algorithm. Three such algorithms are presented in this section. In addition, the Schur Decomposition approach for solving the Riccati Equation is discussed here because it is used to initialize and back-up the iterative algorithms in all of the previous SDRE implementations discussed in Section 2.5.

The Schur decomposition method requires about $75n^3$ computations [27], while the iterative methods also have complexity $O(n^3)$, but the constant is much smaller (for example, the Kleinman algorithm requires $6n^3$ calculations per iteration) [26].

The iterative methods are affected by two major factors: the initial guess, and the algorithm’s convergence rate. Clearly, better guesses require fewer iterations for convergence. However, because each algorithm also has a different convergence rate, a determination must be made when to fail-over to the Schur algorithm (e.g. if Kleinman’s method takes longer than 12 iterations to converge at $6n^3$ calculations per iteration, the Schur method at $75n^3$ total calculations is actually faster).

The three iterative algorithms selected for comparison in this work are: 1) Newton’s Algorithm, 2) Newton’s Algorithm with Exact Line Search, and 3) Kleinman’s Algorithm. These were selected because they are simple in complexity and individual steps within the algorithms are available as efficient library routines. Furthermore, their convergence rate is quadratic, while other algorithms converge linearly [25]. The iterative methods are
implemented in this work using Matlab and compared both for initial speed and accuracy.

2.4.1 Newton algorithm

When successful, this algorithm uses the value of the function as well as its first derivative to converge to the Riccati Equation solution. There are two stopping conditions: the algorithm has converged to a solution, or the maximum number of iterations has been reached.

The convergence of this method is not guaranteed. It is possible that the approach will move away from the correct solution because of the presence of local minimums. Also for this reason, the method may converge to a value which is not the stabilizing solution to the Riccati Equation [34]. Sometimes, the first Newton step is very large and the system takes many iterations to recover. However, when the algorithm reaches a region of convergence to a solution, the convergence is quadratic [37].

Pseudo-code for the Newton Algorithm is shown in Figure 2.2. This algorithm has a complexity of $O(n^3)$ per iteration.

![Figure 2.2: Pseudo-Code for Newton Algorithm [7]](image)
2.4.2 Newton’s method with Exact Line Search [7]

This method is a refinement of the Newton. The speed of convergence is improved and it also corrects the problem Newton’s method has with a very large initial step by using a line search to select the step size.

Benner and Byers [7] show that the added calculations have a complexity of $O(n^2)$. This is cheap compared to the Newton iterations which have complexity $O(n^3)$, especially for systems with sparse matrices. Thus, the overall algorithm complexity is still $O(n^3)$.

The following results were observed in [7] when running the Exact Line Search algorithm against the Schur method and standard Newton’s method on a set of examples:

- For cases where Newton’s method takes significantly longer than Schur decomposition, Exact Line Search was competitive or faster.
- When used as a refinement method, rather than to find the solution without a good initial guess, Exact Line Search can produce better results than Newton’s method.

The algorithm for the exact line search method is shown in Figure 2.3.

**Preconditions:** Initial Guess is stabilizing, $A$, $B$, $Q$, $R$ are invertible

**Inputs:** $A$, $B$, $R$, $P_0$ (Initial Guess, best if $P_0 = P_0^T$), Tolerance and Iteration Limit

**Outputs:** $P$ (Riccati Solution)

**Stopping Condition:** $||\Delta||_1 \leq$ Tolerance or No. of Iterations $\geq$ Iteration Limit, with $\Delta = P_n - P_{n-1}$

1. For $j = 0, 1, 2, ...$
   a. $K_j = R^{-1}(B^TP_jE + S^TC)$
   b. Solve for $N_j$ in the Lyapunov equation:
      $$(A - BK_j)^TN_jE + E^TN_j(A - BK_j) = -R(P_j).$$
   c. $V_j = E^TN_jBR^{-1}B^TN_jE$
   d. $t_j = \min_{0 \leq t \leq 2} (\text{trace}(R(P_j + tN_j)^2))$
   e. $P_{j+1} = P_j + t_jN_j$

End For
2.4.3 Kleinman iteration [25, 26]

The Kleinman algorithm is another improvement on Newton’s method. Kleinman introduces the concept of a cost matrix, and shows that the Riccati equation solution $P$ is the cost matrix of that equation. Using this approach, it is proved that if $P_0$ is a stabilizing initial guess, the iterations cause the ‘cost’ of each solution to monotonically decrease until it converges to the stabilizing solution.

The pseudo-code for this algorithm is shown in Figure 2.4. According to [29], the Kleinman algorithm requires $6n^3$ operations per iteration.

![Figure 2.4: Pseudo-Code for Kleinman Algorithm](image)

The Kleinman method is very useful as a refinement method, but requires a good guess to ensure convergence, as with the other iterative methods. In fact, Arnold and Laub [3] remark that the best technique is to find the initial solution using a method which is guaranteed to converge (for a well-conditioned problem). The Kleinman algorithm can then be used to track this solution as the system dynamics change slightly.
2.4.4 Schur Decomposition

The guaranteed method for solving Riccati equations suggested in [3] is the Schur Decomposition method. Although other numerically robust methods exist (for example, Sign Function method and the generalized eigenvalue approach), the Schur method is the de-facto standard to solve Riccati Equations. These numerically robust methods are necessary because although their computation time is longer, they succeed in finding a solution when the iterative methods fail. The Schur decomposition method is composed of two steps [27]:

1. The first step is the reduction of the $2n \times 2n$ Hamiltonian matrix $M$ associated with the CARE to an ordered real Schur form, where:

   $M = \begin{pmatrix} A & -B \\ -Q & -A^T \end{pmatrix} \in \mathbb{R}^{2n \times 2n}$ \hspace{1cm} (2.13)

   and $n$ is the number of states in the system. The real Schur form is:

   $U^T MU = S = \begin{pmatrix} S_{11} & S_{12} \\ 0 & S_{22} \end{pmatrix}$ \hspace{1cm} (2.14)

   where $U \in \mathbb{R}^{2n \times 2n}$ and $S_{ij} \in \mathbb{R}^{n \times n}$. This step can be performed using a number of routines in the LAPACK library [2], as described in Figure 2.5.

2. The second step is the solution of an $n^{th}$ order linear matrix equation to obtain $P$:

   $PU_{11} = U_{21}$ \hspace{1cm} (2.15)

   where $U_{11}, U_{22}$ are $n \times n$ partitions of the matrix $U = \begin{pmatrix} U_{11} & U_{12} \\ U_{21} & U_{22} \end{pmatrix}$.

   The condition number $\kappa$ (the ratio of the largest to the smallest singular value of the matrix) of $U_{11}$ should be analyzed during computation since it may point to a badly conditioned Riccati Equation. Condition numbers of $10^t$ result in a loss of about $t$ digits of accuracy in $P$. The entire algorithm requires about $75n^3$ calculations [27] to find the stabilizing solution, where $n$ is the number of states in the system. The
algorithm is described in Figure 2.5. All of the functions listed are available in the LAPACK library.

Properties of the four algorithms presented here are summarized in Table 2.1. Note that although all of the algorithms have a complexity of $O(n^3)$ ($n$ is the number of system states), the Schur approach is about 12 times slower than a single iteration of the other methods [7].

Table 2.1: CARE solution algorithms summary

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Complexity</th>
<th>Convergence Rate</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Newton</td>
<td>$O(n^3)$</td>
<td>quadratic</td>
<td>Initial large step; non guaranteed convergence</td>
</tr>
<tr>
<td>Newton with Exact Line Search</td>
<td>$O(n^3)$</td>
<td>quadratic</td>
<td>Accuracy, speed depend on the step size (t); non guaranteed convergence</td>
</tr>
<tr>
<td>Kleinman</td>
<td>$O(n^3)$</td>
<td>quadratic</td>
<td>Convergence guaranteed if initial guess is stabilizing</td>
</tr>
<tr>
<td>Schur</td>
<td>$O(n^3)$</td>
<td>not iterative</td>
<td>Correct solution guaranteed if it exists</td>
</tr>
</tbody>
</table>

All existing implementations of the SDRE method have used the combination of Kleinman and Schur methods because the Schur method always finds the solution to the Riccati Equation if it exists and the Kleinman method is faster than the Schur approach. In this work, we found that the Kleinman approach is more accurate and faster than the other two iterative algorithms (see Chapter 3).

2.5 Previous SDRE implementations

Similar SDRE implementations to the one presented here have been reported by several researchers. Some of these (for example, [4, 21]) include simulations which demonstrate the capabilities of the SDRE method. The first real-time implementation was reported by
Langson [17] where two 75MHz computers were connected together. One simulated the plant while the other executed the controller. The first implementation using a physical plant with online SDRE solution is by Erdem [17], where a double inverted pendulum is controlled around its unstable equilibrium using a sampling rate of 100Hz with a 60MHz digital signal processor. These and other previous SDRE implementations have been summarized in Table 2.3.

Menon, et al. obtained timing comparisons between the main algorithms for Riccati solvers (Schur Decomposition and Kleinman iteration) on different computers (see Table 2.2).

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Processor</th>
<th>Operating System</th>
<th>Execution Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Schur Method</td>
<td>Pentium II, 300 MHz</td>
<td>Windows NT®</td>
<td>2.38 ms</td>
</tr>
<tr>
<td>Schur Method</td>
<td>Pentium III, 450 MHz</td>
<td>Windows NT</td>
<td>1.57 ms</td>
</tr>
<tr>
<td>Schur Method</td>
<td>PowerPC 604e</td>
<td>dSPACE Proprietary</td>
<td>1.56 ms</td>
</tr>
<tr>
<td>Kleinman Method</td>
<td>Pentium II, 300 MHz</td>
<td>Windows NT</td>
<td>1.46 ms</td>
</tr>
<tr>
<td>Kleinman Method</td>
<td>Pentium III, 450 MHz</td>
<td>Windows NT</td>
<td>0.91 ms</td>
</tr>
<tr>
<td>Kleinman Method</td>
<td>PowerPC 604e</td>
<td>dSPACE Proprietary</td>
<td>0.476 ms</td>
</tr>
</tbody>
</table>

As mentioned earlier, although several researchers have already implemented real-time SDRE algorithms, their results indicate the need to improve the performance of the SDRE algorithm so that it can be applied to more complex systems. This work develops methodologies for analyzing the SDRE algorithm and presents several such improvements.
**Preconditions:** well-conditioned Riccati Equation  
**Inputs:** $A, B, Q, R$  
**Outputs:** $P$ (Riccati Solution), Condition Number

1. Form the Hamiltonian Matrix $M$ of the Riccati Equation using the inputs.
2. Balance $M$ using a routine like BALANC.
3. Reduce $M$ to upper Hessenberg form using a routine like ORTHES.
4. Accumulate the transformations from ORTHES using ORTRAN.
5. Find the real Schur form (RSF) of the Hessenberg matrix using HQR3 or similar.
6. Backtransform the matrix so it corresponds to the original matrix using BALBAK or similar.
7. Compute Condition Number of $U_{11}$.
8. Solve $PU_{11} = U_{21}$ using a linear solver routine.

Figure 2.5: Pseudo-code for Schur Algorithm [27]

### Table 2.3: Previous Work Summary

<table>
<thead>
<tr>
<th>Author(s)</th>
<th>Work Summary</th>
<th>Hardware Environment</th>
<th>Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hammett, Hall, Ridgely</td>
<td>Matlab simulations of SDRE</td>
<td>N/A</td>
<td>1998</td>
</tr>
<tr>
<td>Beeler</td>
<td>Matlab simulations of SDRE and several variations</td>
<td>N/A</td>
<td>2004</td>
</tr>
<tr>
<td>Langson</td>
<td>Physical SDRE controller implementation, simulated plant</td>
<td>Two 75MHz computers</td>
<td>1991</td>
</tr>
<tr>
<td>Erdem, Alleyne</td>
<td>Physical plant (levitated ball), controller uses a DSP together with Matlab, analytical SDRE solution</td>
<td>dSpace DS1102 DSP, Computer with Matlab</td>
<td>1999</td>
</tr>
<tr>
<td>Erdem</td>
<td>Physical plant, controller uses a DSP together with Matlab, online SDRE solution</td>
<td>dSpace DSP, Computer with Matlab</td>
<td>2001</td>
</tr>
<tr>
<td>Menon, et al.</td>
<td>Self-contained SDRE implementation (using workstation PC or PowerPC DSP Board), simulated plant (missile control)</td>
<td>Pentium II (300 MHz), Pentium III (450 MHz), Motorola 604e PowerPC DSP Board</td>
<td>2002</td>
</tr>
<tr>
<td>Bogdanov, et al.</td>
<td>Physical plant (small unmanned helicopter), fully-embedded controller, online SDRE solution</td>
<td>Geode GX1 CPU (x86),</td>
<td>2003</td>
</tr>
</tbody>
</table>
Chapter 3

Iterative Method Comparison

As described in Chapter 2, all previous SDRE implementations to solve the Riccati equation have used a combination of the Schur method and the Kleinman method. Here, we compare the Kleinman approach to Newton’s method and Newton’s method with Exact Line Search in order to determine which iterative Riccati Equation solution method results in the most accurate solutions in the fastest time. The three methods were implemented and tested in Matlab. Two facets of the algorithms are considered: the effectiveness of each for solving Continuous-time Algebraic Riccati Equations, and their relative speeds. The effectiveness is measured as a failure percentage, where a failure means that the algorithm either did not converge or converged to the wrong solution (as compared to the known-correct value returned by the Matlab ‘care’ function).

To ensure that all three algorithms are tested the same way, the same set of initial guesses is used for every test. The Riccati Equations used are from the CAREX benchmark examples for CARE [5]. This collection includes twenty Riccati Equation examples and all of them except number 20 were used (the system in example 20 has 421 states – too large for the scope of this project). The Matlab function ‘care’ is used to find the solution of each Riccati Equation, as it is based on a Hamiltonian matrix approach to calculate the result. This function also balances the matrices to improve their numerical conditioning. Once a solution is obtained, it is scaled by random numbers to simulate the changes in system states between the previous sampling interval and the current one. Four guesses were used to initialize the iterative algorithm. One was the solution obtained by the Hamiltonian
approach and the other three where scaled variations of this solution, with the scaling factor between 0 and 2.

3.1 Accuracy analysis

Each iterative method’s solution was compared with the result returned by the Matlab ‘care’ function. The error is computed as the 2-norm of the difference between the iterative solution, $T$, and the ‘care’ solution, $C$:

$$error = ||T - C||_2$$ (3.1)

The first test performed is an accuracy test. All three iterative approaches were used, and the results were analyzed to determine which method is most likely to converge to the correct solution. A solution is considered correct if the error is less than 0.0001.

The accuracy results are presented in Table 3.2 and summarized in Figure 3.1. A failure for this test is any Riccati Equation solution with more than 0.0001 error, as defined by Equation 3.1. Only those CAREs where the algorithm failed to converge to the correct solution are shown.

The Kleinman algorithm failed in 21 out of the 76 tests (27.6%), while Newton’s algorithm and Exact Line Search with a 0.00001 step size failed in 46 and 44 tests (60.5% and 57.9%), respectively. (The Exact Line Search step size is discussed further shortly).

The Kleinman approach converged to the wrong solution in 15 of its 21 (71.4%) failures. The Newton converged to the wrong value in 20 of 46 (43.5%) failures, and Exact Line Search did in 40 of 44 (90.9%) failures. In this situation, it is better to fail to converge than to give an incorrect solution, since a convergence failure can be easily detected. The results are summarized in Figure 3.1.

Kleinman [25] indicates that the initial guess for his algorithm is stabilizing if the closed loop state matrix is Hurwitz. Several of the tests in this section were started with a non-stabilizing initial guess. Failed tests which were initialized in such a way are shown in bold in Table 3.2. If the Kleinman algorithm is altered to include the initial guess verification,
the Kleinman approach results in only seven failed attempts to find the solution, with only four tests converging to a wrong (non-positive definite) solution. This is significantly better than both the Newton and Exact Line Search methods, as shown in Figure 3.1. Of course, with initial guess verification, tests started with a non-stabilizing initial guess are not run, and an algorithm such as the Schur method needs to be used to solve these Riccati equations in a real system.

A difficulty with the Exact Line Search approach is its sensitivity to the step-size parameter \( t \). Table 3.1 summarizes accuracy results for various step sizes. As the step size increases, the number of failures also increases (from 43 with a step size of 0.001 to 45 with a step size of 0.5). A smaller step size means more accurate results, but also more computation time. For the example presented here, the algorithm needs to search through...
2000 points to determine the local minimum when the step size is 0.001, and through only 4 points when the step size is 0.5. However, because the Exact Line Search approach doesn’t perform well (converges to the wrong value too frequently, as seen in Figure 3.1) for this example, it was not considered further.

<table>
<thead>
<tr>
<th>Step Size (t)</th>
<th>Search Space</th>
<th>Tests Failed (out of 76)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.001</td>
<td>2000</td>
<td>43 (56.6%)</td>
</tr>
<tr>
<td>0.01</td>
<td>200</td>
<td>44 (57.9%)</td>
</tr>
<tr>
<td>0.1</td>
<td>20</td>
<td>43 (56.6%)</td>
</tr>
<tr>
<td>0.5</td>
<td>4</td>
<td>45 (59.2%)</td>
</tr>
</tbody>
</table>

When the Exact Line Search step size becomes larger, the number of failed tests increases. Having a smaller step size results in more computation time because each iteration of the algorithm needs to search through a search space of a larger size to find a local minimum.

### 3.2 Speed analysis

The previous section showed that the Exact Line search and Newton methods performed poorer with respect to accuracy than the Kleinman approach. Although the accuracy of the Exact Line Search approach improved slightly as the step size parameter became smaller, a small step size increases computation time. Benner and Byers [7] claim that the purpose of Exact Line Search is not for speed, but rather to make the method converge monotonically.

Because the Exact Line Search algorithm does not achieve any speed improvements over Newton’s method, only Newton’s method was compared to Kleinman’s algorithm for speed. The additional computation time involved in adding Kleinman initial guess verification, as described in Section 3.1, is also examined. In order to conduct the timing tests, the algorithms were implemented in Matlab in such a way that all of the initialization and iteration code was the same, except for the lines of computation code within each iteration, as shown in Figures 3.2 and 3.3.

The results of these comparisons are summarized in Figures 3.6, 3.4 and 3.5. These
\[
K = (B'\times R) / R; \\
\text{Right} = -K \times R \times K' - Q; \\
Ak = A + B \times K'; \\
\%	ext{ solve Lyapunov equation} \\
P = \text{lyap}(Ak', \text{Right}); \\
\]

Figure 3.2: Matlab code for one Kleinman Algorithm iteration

\[
K = (B' \times P) / R; \\
\text{LyapA} = A - B \times K; \\
D = (B' \times P)' \times K; \\
\text{LyapQ} = (Q + A' \times D + P + A - D); \\
N = \text{lyap} (\text{LyapA}', \text{LyapQ}); \\
P = P + N; \\
\]

Figure 3.3: Matlab code for one Newton Algorithm iteration

figures show that although the computational complexity of each of the algorithms (Kleinman, Kleinman with Initial Guess Verification, and Newton) is approximately equal, the Kleinman algorithm requires fewer iterations in just about every test. Furthermore, if initial guess verification is used, the average number of iterations required drops even further. This is due to the fact that a non-stabilizing initial guess means that the iterative algorithm is not attempted for the solution. This is important because the algorithm would have failed in these cases, causing the total solution time to be the iterative solution time (leading to failure) plus the numerically robust method solution time. With initial guess verification, the iterative solution time is eliminated from this computation. The results of these tests are presented in Table 3.3.
This table presents accuracy test results for all three algorithms considered. Only failed tests are displayed, with the values shown representing the solution error as defined by Equation 3.1. ‘No Conv.’ means that the test did not converge within the 100 iterations allowed. The $R_1$, $R_2$, and $R_3$ matrices are randomly generated matrices with element values between 0 and 2, and the multiplication between these matrices and the $care$ matrix is performed pointwise. The $care$ matrix is the solution returned by the Matlab ‘care’ command. Tests in bold are initialized with non-stabilizing guesses. The Kleinman algorithm would be skipped for these cases, and the Schur approach used immediately, saving the computation time of running the Kleinman algorithm and having it fail.
This figure shows that when the algorithms converge to a solution, the Kleinman algorithm requires fewer iterations than the Newton algorithm. In fact, this is true for every test example except 6, 10, and 11. There is no Newton method comparison for examples 14 and 17 because that method did not converge to a correct solution with any of the initial guesses.
This graph shows the total number of iterations required, including tests when the algorithm did not converge to a solution within the iteration limit (100 for this test). The number of iterations is significantly higher for the Newton method because it fails to converge a lot more frequently than the Kleinman approach.
Table 3.3: Iterative Method Timing Results (seconds)

<table>
<thead>
<tr>
<th>Example</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
<th>18</th>
<th>19</th>
</tr>
</thead>
<tbody>
<tr>
<td>States</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>8</td>
<td>9</td>
<td>30</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>39</td>
<td>64</td>
<td>21</td>
<td>100</td>
<td>60</td>
</tr>
<tr>
<td>Iterations</td>
<td>6</td>
<td>10</td>
<td>5</td>
<td>3</td>
<td>3</td>
<td>10</td>
<td>7</td>
<td>23</td>
<td>9</td>
<td>19</td>
<td>46</td>
<td>101</td>
<td>4</td>
<td>31</td>
<td>7</td>
<td>4</td>
<td>21</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>Iter. Time</td>
<td>0.006</td>
<td>0.005</td>
<td>0.005</td>
<td>0.005</td>
<td>0.006</td>
<td>0.011</td>
<td>0.004</td>
<td>0.004</td>
<td>0.008</td>
<td>0.004</td>
<td>0.004</td>
<td>0.004</td>
<td>0.004</td>
<td>0.004</td>
<td>0.004</td>
<td>0.004</td>
<td>0.004</td>
<td>0.004</td>
<td></td>
</tr>
</tbody>
</table>

| Guess Verification | 0.005 | 0.006 | 0.005 | 0.005 | 0.008 | 0.014 | 0.006 | 0.006 | 0.006 | 0.006 | 0.006 | 0.006 | 0.006 | 0.006 | 0.006 | 0.006 | 0.006 | 0.006 |

This table shows the average amount of time and iterations required to complete an example. Iteration counts of fewer than 101 mean that the algorithm converged to a solution. Iteration counts of 101 mean that the iteration limit was reached. The methods compared are the Kleinman approach ('Kleinman'), Kleinman approach with initial guess verification ('Guess Verification') and the Newton method. For the Kleinman method with guess verification, 'Bad P0' implies that the initial guess was not stabilizing. The number of iterations in this case is 0, since the Kleinman method is not applied. The last three lines of each section ('Conv. Iter.', 'Total Iter.', and 'Iter. Time') are averages of the number of iterations required for convergence, the total number of iterations, and the amount of time per iteration, respectively. These averages are analyzed further in Figures 3.6, 3.4 and 3.5.
Timing results are provided for those tests in which the algorithm converged to a solution in fewer than the maximum number of iterations which in this case was 100 (whether it was the correct solution or not). The Kleinman algorithm with Guess Verification is also presented so that the overhead of the additional check can be analyzed. The computational complexity of all three algorithm variations are presented in Figure 3.6. This figure shows the per-iteration speed of the algorithms as the number of system states increases. Kleinman [26] showed that his algorithm takes $6n^3$ operations (where $n$ is the number of system states), and this trend line fits with the data shown in Figure 3.6.

Our experimental results suggest that the Kleinman approach has a higher success rate (converges to the correct solution more frequently) than the other two algorithms, that it has per-iteration speed comparable to, and requires fewer iterations than the Newton approach. Also, adding initial guess verification to the Kleinman approach significantly improves the accuracy of the algorithm because all tests which are started use a stabilizing guess. These are reasons to conclude that the Kleinman algorithm is the best iterative approach (of the tested methods) a real-time implementation of the SDRE controller.

Because the Kleinman algorithm accounts for the majority of the execution time of the software, we looked at reducing the overall SDRE algorithm execution time by limiting the maximum number of iterations of the Kleinman algorithm. As Section 3.3 shows, a very low limit (7 iterations) is enough for minimal average computation time per Riccati equation solution of the application tested. We use the full SDRE algorithm, including both the Kleinman and the Schur methods to perform this analysis.
The Newton, Kleinman, and Kleinman with Initial Guess Verification per-iteration times are very close to each other. The two exceptions, with 60 and 64 states are likely due to the fact that the Newton algorithm required over 16 iterations to converge (for the three randomly scaled initial guesses) while the Kleinman approach required 4 iterations. The additional iterations for the Newton method allowed the initial overhead of starting the simulation to be less noticeable. The computational complexity trendline ($n^3$) suggested by Kleinman [26] is plotted against the data (it is scaled down to fit the 100 states case). This trendline shows that the complexity results obtained in this work correspond to those presented by Kleinman. We also see that guess verification does not significantly increase the complexity of the Kleinman algorithm.
3.3 Kleinman maximum iterations

The choice of maximum number of iterations for the Kleinman algorithm can significantly affect the performance of the overall SDRE control system. If the limit is too low, the Kleinman algorithm will often fail and result in overhead without actually producing answers. On the other hand, if the number of maximum iterations is too high, cases when the Kleinman method doesn’t converge to the solution may take a long amount of time.

Considering the maximum number of iterations as a design parameter allows us to make the total execution time of the SDRE controller shorter without significantly changing the accuracy of any of the solutions. We analyzed the number of Kleinman iterations required, for a particular application, and then investigated what happens to the execution time when that number is limited. The experimental results presented indicate that the Kleinman algorithm tends to converge in fewer than seven iterations, and any solutions which take longer than this generally do not converge.

According to the authors of the algorithms, the Schur algorithm is about twelve times slower than Kleinman’s approach. This gives an indication of the proper maximum number of iterations (the Kleinman approach can’t regularly take more than twelve iterations), but more analysis is required. In particular, the effect of the number of iterations on the accuracy of the solution as well as on the speed of the algorithm was studied.

3.3.1 Effect on accuracy

In order to perform this analysis, we used the Simulink Inverted Pendulum (see Chapter 4). This simulation was modified to count the number of times that the Kleinman algorithm succeeds, the number of times it doesn’t converge, and the number of times it converges to the incorrect value. This data is collected for different maximum numbers of iterations of the Kleinman algorithm. The results are found in Figures 3.7 and 3.8.

Figure 3.7 shows that the Kleinman algorithm is successful most of the time, and the Schur algorithm accounts for only 17% of the overall invocations. Furthermore, in as few
as seven Kleinman iterations, the Schur algorithm is only required 19.5% of the time. Thus, it appears that between 7 and 10 iterations is sufficient for the Kleinman algorithm. The second plot (Figure 3.8) shows that the Schur algorithm is sometimes required to correct the Kleinman algorithm. This means that a method must be established to detect failure of the Kleinman algorithm. That plot also shows that for this data set, the Kleinman algorithm appears to take longer to converge to incorrect values than it does to correct values (there are zero incorrect Kleinman answers below seven iterations).

One way to detect failure of the Kleinman algorithm is through applying the Initial Guess Verification approach discussed in the previous section. This method is ideal because it detects likely failure before the algorithm is even run. The only resulting cases of failure when this approach is used should be due to ill-balanced input matrices.
This figure shows the breakdown of Kleinman algorithm failures for various choices of maximum number of iterations. From an SDRE algorithm point of view, it is acceptable to have no convergence, but incorrect convergence can break the controller. There are no incorrect solutions for fewer than 7 maximum iterations. The Initial Guess Verification approach can be used to lower the number incorrect convergence results.

### 3.3.2 Effect on performance

A second test uses the optimized C code in Section 5.1 to examine the effect of a maximum iteration limit on the performance of the SDRE system.

A large data set (33334 state vectors) is generated using the Matlab model. This set corresponds to 1000 seconds of simulation time, so the samples are collected every 29 ms (of simulation time). The simulation uses a random disturbance high enough to cause the controller to be ineffective. This is done to ensure that a large number of different states are present in the simulation, rather than just those when the controller is holding the pendulum upright (see Chapter 4 for a full discussion of the inverted pendulum). This data set is then solved using the software version of the SDRE solver. Table 3.3.2 shows the full results,
Table 3.4: Kleinman iterations timing test

<table>
<thead>
<tr>
<th>Kleinman Iterations</th>
<th>Kleinman Solved</th>
<th>Schur Solved</th>
<th>Total Equations</th>
<th>Time (s)</th>
<th>Std. Dev. Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>33059</td>
<td>274</td>
<td>33334</td>
<td>20.313</td>
<td>0.13258</td>
</tr>
<tr>
<td>9</td>
<td>33050</td>
<td>283</td>
<td>33334</td>
<td>20.326</td>
<td>0.11453</td>
</tr>
<tr>
<td>8</td>
<td>33029</td>
<td>304</td>
<td>33334</td>
<td>20.373</td>
<td>0.10489</td>
</tr>
<tr>
<td>7</td>
<td>32991</td>
<td>342</td>
<td>33334</td>
<td>20.661</td>
<td>0.35065</td>
</tr>
<tr>
<td>6</td>
<td>32913</td>
<td>420</td>
<td>33334</td>
<td>23.357</td>
<td>0.13551</td>
</tr>
<tr>
<td>5</td>
<td>32869</td>
<td>464</td>
<td>33334</td>
<td>20.306</td>
<td>0.14861</td>
</tr>
<tr>
<td>4</td>
<td>32832</td>
<td>501</td>
<td>33334</td>
<td>20.476</td>
<td>0.12423</td>
</tr>
<tr>
<td>3</td>
<td>32800</td>
<td>533</td>
<td>33334</td>
<td>20.344</td>
<td>0.07827</td>
</tr>
<tr>
<td>2</td>
<td>30608</td>
<td>2725</td>
<td>33334</td>
<td>21.950</td>
<td>0.19820</td>
</tr>
<tr>
<td>1</td>
<td>1003</td>
<td>32330</td>
<td>33334</td>
<td>46.279</td>
<td>0.28845</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>33333</td>
<td>33334</td>
<td>38.180</td>
<td>0.18040</td>
</tr>
</tbody>
</table>

This data shows the full results for the data presented in Figure 3.3.2. Raising the maximum number of Kleinman iterations for this application does not increase the required computation time because the majority of the Kleinman solutions succeed. However, limiting the maximum number of iterations in a system where the Kleinman algorithm tends to fail to converge can improve the overall execution time.

which are summarized in Figure 3.9.

Several important points are identified. First, there is a spike at 1 Kleinman iteration, since this causes the solver to be attempted (causing overhead) but it just about always fails. However, allowing Kleinman as few as 3 iterations takes the same amount of time as allowing more iterations. The calculation times for maximum values higher than two iterations are almost the same for this system. Comparing these results to the Kleinman algorithm accuracy results from Section 3.1 indicates that there should be a tradeoff between the maximum number of iterations and the likelihood that the iterative method will fail. If the states of the controlled system are expected to stay almost constant, the iterative method will succeed most of the time, so setting a larger number of maximum iterations is acceptable. If, however, the states are expected to fluctuate resulting in poorly balanced input matrices, the Schur method will likely have to be used more frequently. Because of this, the number of maximum Kleinman iterations should be made smaller, or
the algorithm will spend a lot of time performing calculations which do not converge to the correct solution.

The results in this section agree with the accuracy results (Section 3.3.1. Both show that 7-10 iterations is required so that the Kleinman algorithm provides the correct answer most of the time (for this application). Furthermore, we can develop a method for determining the maximum number of iterations to use for a system if its behavior is known \textit{a priori}. This method can also be used to decide on the computing power required for a given control system.

If the number of iterations required for each Kleinman algorithm solution under normal system behavior is recorded, the optimal number of maximum iterations to use is given by Equation 3.2:

\[
\min_{I_{\text{max}}} \left( \sum_{I=0}^{I_{\text{max}}} IP_I T_K + \sum_{I=I_{\text{max}}+1}^{\infty} I_{\text{max}} P_I T_K + \sum_{I=I_{\text{max}}+1}^{\infty} P_I T_S \right) \tag{3.2}
\]
Table 3.5: Maximum Iterations Computation using Equation 3.2

<table>
<thead>
<tr>
<th>Iterations</th>
<th>Number of Solutions</th>
<th>Percent of occurrence</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1003</td>
<td>2.953%</td>
<td>12.65</td>
</tr>
<tr>
<td>2</td>
<td>29605</td>
<td>87.171%</td>
<td>2.28</td>
</tr>
<tr>
<td>3</td>
<td>2192</td>
<td>6.454%</td>
<td>1.48</td>
</tr>
<tr>
<td>4</td>
<td>32</td>
<td>0.094%</td>
<td>1.50</td>
</tr>
<tr>
<td>5</td>
<td>37</td>
<td>0.109%</td>
<td>1.51</td>
</tr>
<tr>
<td>6</td>
<td>44</td>
<td>0.130%</td>
<td>1.53</td>
</tr>
<tr>
<td>7</td>
<td>708</td>
<td>2.085%</td>
<td>1.18</td>
</tr>
<tr>
<td>8</td>
<td>38</td>
<td>0.112%</td>
<td>1.17</td>
</tr>
<tr>
<td>9</td>
<td>21</td>
<td>0.062%</td>
<td>1.17</td>
</tr>
<tr>
<td>10</td>
<td>9</td>
<td>0.027%</td>
<td>1.17</td>
</tr>
<tr>
<td>11</td>
<td>9</td>
<td>0.027%</td>
<td>1.17</td>
</tr>
<tr>
<td>12</td>
<td>3</td>
<td>0.009%</td>
<td>1.18</td>
</tr>
<tr>
<td>13</td>
<td>4</td>
<td>0.012%</td>
<td>1.18</td>
</tr>
<tr>
<td>14</td>
<td>1</td>
<td>0.003%</td>
<td>1.19</td>
</tr>
<tr>
<td>15</td>
<td>1</td>
<td>0.003%</td>
<td>1.20</td>
</tr>
<tr>
<td>16</td>
<td>1</td>
<td>0.003%</td>
<td>1.20</td>
</tr>
<tr>
<td>17</td>
<td>1</td>
<td>0.003%</td>
<td>1.21</td>
</tr>
<tr>
<td>18</td>
<td>1</td>
<td>0.003%</td>
<td>1.22</td>
</tr>
<tr>
<td>∞</td>
<td>252</td>
<td>0.742%</td>
<td>1.17</td>
</tr>
</tbody>
</table>

The number of solutions obtained with each number of iterations is obtained during simulation of the sample system. ∞ iterations implies that the Kleinman algorithm was not successful in finding the solution, so these cases would iterate until the maximum number of allowed iterations regardless of what that maximum is.

where \( I_{\text{max}} \) is the maximum number of iterations allowed, \( P_t \) is the the rate (percentage) at which \( I \) iterations are required for a solution, \( T_K \) is the Kleinman iteration complexity, and \( T_S \) is the Schur algorithm complexity. This equation is a sum of the total complexity of running the Kleinman algorithm when it succeeds and when it fails at the maximum number of iterations (and the Schur algorithm is required).

Applying this equation to the system evaluated in the simulations above, we obtain the results in Table 3.5. These results are presented graphically in Figure 3.10. The minimum point in the curve is at 8 iterations for this system, so this is the maximum number of iterations which will be used.
This figure uses Equation 3.2 to plot complexities associated with different Maximum Iterations for the Kleinman algorithm. The values $T_K = 1$ and $T_S = 6$ are used because these are the relative complexity differences between the two algorithms.

The test data used for this section was obtained from running a Matlab simulation. This simulation was created in order to ensure that the SDRE implementation being designed in this work is correct. A test application which can be used to verify the implementation must be simple but still have characteristics exhibited by SDRE control target applications. The inverted pendulum is used because it is non-linear but simple enough to demonstrate all of the main SDRE control issues. To explore some of the issues mentioned, the pendulum is simulated using Matlab. This allows us to more easily and carefully examine any portions of the test system than we would be able to using a physical setup.
Chapter 4

Application: Inverted Pendulum

The simulation model being used as an evaluation method for the SDRE implementation is based on the Quanser Consulting Inc. Linear Inverted Pendulum. This is a setup where a free-swinging pendulum is attached to a cart on a track. Sensors are available to measure the position of the cart on the track and the angle of the pendulum. Figures 4.2 and 4.3 show a photo and a diagram of the pendulum, including sensor and actuator locations. The goal of the controller is to stabilize the pendulum in its inverted position, and the cart at the center of the track. The system dynamics are governed by the following non-linear equations [33]:

\[
(m_p + m_c)\ddot{x} + m_p\dot{\theta}L_p \cos(\theta) - m_p\dot{\theta}^2L_p \sin(\theta) = F \tag{4.1}
\]
\[
m_pL_p \cos(\theta)\ddot{x} + m_p\dot{\theta}L_p^2 - m_pgL_p \sin(\theta) = 0 \tag{4.2}
\]

where \(F\) is the input force to the cart (\(N\)), \(m_p\) is the mass of the pendulum (\(kg\)), \(m_c\) is the mass of the cart (\(kg\)), and \(L_p\) is the length of half of the pendulum (center of gravity location) (\(m\)). These equations are derived from the pendulum block diagram in Figure 4.1.

The SDC parametrization of the inverted pendulum model presented here is derived in the next section. The parameterized model can then be used to test an SDRE controller.

### 4.1 Inverted Pendulum SDRE derivation

The inverted pendulum dynamic equations are given in Equations 4.1 and 4.2.
Figure 4.1: Pendulum Parameters Block Diagram

Putting these equations into state-space form, the following is obtained: The states selected for the pendulum are pendulum angle, angular velocity, cart position, and cart velocity. This results in state vector $X = (s_1, s_2, s_3, s_4)^T = (\theta, \dot{\theta}, x, \dot{x})^T$. Thus, the state-space equations are:

\[
\begin{align*}
\dot{s}_1 &= s_2 \\
\dot{s}_2 &= \frac{\cos(s_1)s_2^2L_p\sin(s_1) + \cos(s_1)F - g\sin(s_1)(m_c + m_p)}{L_p(-m_p - m_c + m_p\cos(s_1)^2)} \\
\dot{s}_3 &= s_4 \\
\dot{s}_4 &= \frac{m_p\cos(s_1)g\sin(s_1) - m_ps_2^2L_p\sin(s_1) - F}{-m_p - m_c + m_p\cos(s_1)^2} 
\end{align*}
\] (4.3)

One possible parametrization can be obtained by separating the inputs from the states.
For this parametrization (there is an infinite number, as described earlier), a $\theta$ is added to the denominator of two terms in the $A$ matrix. This creates a $\frac{\sin(\theta)}{\theta}$ term, which we can define as a $sinc(\theta)$ function such that:

$$sinc(\theta) = \frac{\sin(\theta)}{\theta} \text{ if } \theta \neq 0$$

$$sinc(\theta) = 1 \text{ if } \theta = 0$$

This removes the singularity when $\theta$ is zero. The $C$ matrix is simply the identity matrix because all four states are assumed measured by sensors. Finally, there is no direct coupling between the inputs and the outputs, so the $D$ matrix is zero. The final state-space matrix
equations are shown below:

\[
\dot{X} = \begin{pmatrix}
0 & 1 & 0 & 0 \\
\frac{gsin(\theta)(m_p+m_c)}{L_pC_1\theta} & \frac{-m_p\dot{\theta}L_p\sin(\theta)\cos(\theta)}{L_pC_1} & 0 & 0 \\
0 & 0 & 1 & 0 \\
\frac{-gsin(\theta)m_p\cos(\theta)}{C_1\theta} & \frac{\sin(\theta)\dot{\theta}m_pL_p}{C_1} & 0 & 0 \\
\end{pmatrix}
\begin{pmatrix}
\theta \\
\dot{\theta} \\
x \\
\dot{x} \\
\end{pmatrix}
+ \begin{pmatrix}
0 \\
\frac{-\cos(\theta)}{L_pC_1} \\
x \\
\frac{1}{C_1} \\
\end{pmatrix}F
\]

\[y = \begin{pmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 \\
\end{pmatrix}
\begin{pmatrix}
\theta \\
\dot{\theta} \\
x \\
\dot{x} \\
\end{pmatrix}
\]

(4.4)

where \(C_1 = (m_p + m_c - m_p\cos(\theta)^2)\).

The parameters associated with the Quanser IP02 Inverted Pendulum are presented in Table 4.1.
Table 4.1: Quanser Inverted Pendulum parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Abbreviation</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>gravitational constant</td>
<td>$g$</td>
<td>9.81 m/s$^2$</td>
</tr>
<tr>
<td>pendulum mass</td>
<td>$m_p$</td>
<td>0.210 kg</td>
</tr>
<tr>
<td>cart mass</td>
<td>$m_c$</td>
<td>0.5 kg</td>
</tr>
<tr>
<td>pendulum center of mass</td>
<td>$L_p$</td>
<td>0.305/2 m</td>
</tr>
<tr>
<td>cart gearbox ratio</td>
<td>$K_g$</td>
<td>3.71</td>
</tr>
<tr>
<td>DC motor torque constant</td>
<td>$K_m$</td>
<td>0.00767 Nm/A</td>
</tr>
<tr>
<td>motor pinion radius</td>
<td>$r$</td>
<td>0.0127 m</td>
</tr>
<tr>
<td>motor armature resistance</td>
<td>$R$</td>
<td>2.6 Ω</td>
</tr>
</tbody>
</table>

Using both the state-space equations (Equation 4.3) to represent the actual (non-linear) inverted pendulum, and the SDC parameterized version (Equation 4.4), a Simulink model is constructed, as shown in Figure 4.4.

### 4.2 Matlab simulation

![Inverted Pendulum Simulink Model](image)

The Simulink model includes the two representations of the inverted pendulum (Figure 4.5, as well as three other blocks to more accurately represent the whole system. The control action is calculated by a Matlab function containing the SDRE solver, which runs...
every 0.03 seconds of simulation time (33 times per simulation second). The control action is clipped by a saturation block which represents the physical voltage limits of the pendulum system. (This system has a maximum voltage rating of 6V). Finally, the voltage input into the system is converted into force on the cart using the following equation:

\[
F = \frac{K_m K_a V}{R r} - \frac{K_m^2 K_a^2 \ddot{x}}{R r^2} 
\]

where \( V \) is the input voltage and \( \ddot{x} \) is the cart velocity. The second term represents the reverse electro-motive force caused by the motion of the cart. Equation 4.5 is represented by the ‘Voltage to Force’ block in the diagram in Figure 4.4.
The plant block itself is constructed as the two separate s-Functions. The non-linear equations (4.1 and 4.2) represent the dynamics of the actual inverted pendulum being controlled, while the SDC model is being used to calculate the matrices required for the SDRE controller. Note that to compute the SDC model it is necessary to measure all of the states of the inverted pendulum.

During the simulation, when the SDRE calculation fails, the controller outputs a small control action. In the case of the Matlab simulation, it is easy to detect failure, since the ‘care’ routine used to solve the Riccati equation actually returns an error. This is particularly useful to start the pendulum moving, since the SDRE method can not stabilize the pendulum to its unstable equilibrium from its stable equilibrium, and the ‘care’ solution fails at this system state. When this occurs, a minimal control signal \(u = 0.01\) is output by the controller in order to change the states so that the SDRE approach can be engaged.

The plots in Figures 4.6 and 4.7 represent the pendulum angle and cart position. The pendulum starts at the bottom equilibrium point (180 degrees) and begins swinging (at first because of the minimal control action, and then because of the SDRE control) until it is vertical (0 degrees). The position of the cart also returns to zero as the pendulum is stabilized at the top. Figure 4.8 shows the control effort calculated by the SDRE, and that which is actually input to the system because of the saturation region.

Once the pendulum is stabilized at the top, a pulse train disturbance is input to the system. An appropriate disturbance magnitude is inferred from the observed controller force input into the system. The maximum force observed is about 6\(\,N\) in either the positive or negative direction, so half of this value 3\(\,N\) is selected for the pulse train. The controller successfully keeps the pendulum vertical. The angle, position, and control signals for this steady state are shown in Figures 4.9, 4.10, and 4.11.

The performance of the simulation is satisfactory because of the correct selection of weighing matrices. The decisions involved in this selection are presented in the following section.
This plot shows the pendulum angle behavior from the bottom equilibrium point to steady-state at the vertical equilibrium. In this time, the angle has gone from 180 degrees to 0 degrees.

4.3 Selection of the $Q$ and $R$ matrices

The simulations described in the previous section use the weighing matrices in Equation 4.6. The $Q$ values represent the large penalties on angle and position and small penalties on angular and cart velocities. The weighing matrices are initially found using Bryson’s rule. By selecting constraints for the angle and position states (0.05 degrees and 5 cm, respectively), we obtain a $Q$ matrix of $diag(400, 1, 40000, 1)$. The $R$ matrix can be found in a similar manner (the control should be constrained to 6V). These initial weighing matrices are good selections for stabilizing the inverted pendulum if it starts at its upper equilibrium, but they need to be adjusted to force the system to go from its lower equilibrium (or close to it) to its upper equilibrium. The decisions which resulted in these matrices are presented below.
The position of the cart leaves the center while the pendulum is swinging. Once the pendulum is regulated at the top equilibrium, the cart returns to the center of the track.

\[
Q = \begin{pmatrix}
500 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 10000 & 0 \\
0 & 0 & 0 & 1
\end{pmatrix}
\] (4.6)

\[
R = 3
\]

Equation 4.6 shows a proper selection of \(Q\) and \(R\) matrices for this control application. We can now demonstrate what happens when an improper selection is made. Because the \(Q\) and \(R\) matrices are weights for the states and the control, we analyze the effect of varying one weight at a time (from the “known good” matrices listed above) on each of the parameters. Not surprisingly, the effect of lowering any weight in \(Q\) is comparable to the effect of raising the weight in \(R\), and vice-versa. This is because relative sizes of \(Q\) and \(R\) actually affect the performance of the controller.

The first weight varied is the \(Q(1, 1)\) weight, which penalizes the angle of the pendulum.
The control action required to regulate the pendulum to its upper equilibrium point. The control actions are large as the pendulum is swinging, and become small once it is near the equilibrium.

There are two extremes in each of these scenarios: to underpenalize and to overpenalize. As seen in Figure 4.12, having an angle penalty of 1 results in the pendulum never reaching the upper stable point. This is because the controller is essentially regulating the other states, and the value of the angle is not monitored. On the other extreme, if the angle is over-penalized (10000), two things happen. The angle penalty overshadows the position penalty, so the pendulum no longer stays in the middle of the track. Also, the system attempts to correct the angle too quickly, and overshoots every time. The angle and position for this scenario are shown in Figures 4.13 and 4.14.

Similar analysis done with the cart position penalty shows that over-penalizing forces the cart to stay close to the track center, but doesn’t allow the cart to go far-enough out to swing the pendulum. Under-penalizing the cart position causes the cart to drift off of the track. Furthermore, the angle requirement is never met in that situation either.

Under-penalizing the control-input causes the physical magnitude of the control signal to be very high. This means that the signal is always clipped either at the high saturation
Figure 4.9: Simulation: Pendulum angle at steady state with pulse disturbance

The pendulum swings sharply due to a disturbance, but is regulated back to the equilibrium (after some overshoot).

point or at the low saturation point. Although it is able to swing the pendulum up to the upper equilibrium, it overshoots and falls back down. Having a high penalty for the controller forces the control signal to be very low and the controller isn’t able to push the pendulum to the upper equilibrium.

For both the angle and cart velocity penalties, under-penalization doesn’t have a significant effect because the penalties are already very low since the speed of the cart or pendulum do not matter in attaining the final state. However, overpenalization of both of these states causes the system (either angle or position) to be forced to move slower. This results in the system never reaching its upper equilibrium point since it never moves fast enough to swing the pendulum up.

This analysis demonstrates how important the selection of the $Q$ and $R$ matrices is in the SDRE solution. The designer must find the lowest possible penalties which make the system work because the numerical magnitudes within the mathematical algorithms must remain low to keep maximum precision. At the same time, the penalties need to be as
The cart moves to keep the pendulum upright, and then returns to the center of the track. To compensate for the pulse train disturbance, the cart only moves about 4 cm. as high as possible so that the physical signals within the system remain within their allowed bounds. This section also demonstrates one of the benefits to the SDRE approach. This approach allows for the use of designer intuition in creating the control. By analyzing the plant, we are able to make decisions about weights using knowledge about the state constraints.

In this chapter we have presented, via simulations, a complete analysis of an SDRE controller designed for an inverted pendulum. The results once again emphasize the significance of the proper selection of parametrization and weighing matrices. Furthermore, we have established a test environment for a full implementation of this controller. The next chapter details this implementation as we begin analyzing the software structure of the SDRE algorithm.
Figure 4.11: Simulation: Pendulum control action at steady state with pulse disturbance

The controller compensates for the pulse disturbance by issuing a control action in the opposite direction.

Figure 4.12: $Q$ & $R$ matrix analysis: Angle under-penalization

The pendulum never swings high enough to be stabilized at the top (the maximum swing is about 95 degrees).
Figure 4.13: $Q$ & $R$ matrix analysis: Angle over-penalization – Pendulum Angle

The pendulum swings too quickly and overshoots the reference angle.

Figure 4.14: $Q$ & $R$ matrix analysis: Angle over-penalization – Cart Position

The controller does not ‘monitor’ the position because the angle penalty is so large by comparison. Because of this, the cart moves very far from the center.
Chapter 5

SDRE Software Implementation

This chapter presents a C design for the SDRE control implementation (Section 5.1), and this design is analyzed for possible performance improvement.

5.1 Software algorithm design

One of the main objectives of this work is to develop an implementation of the SDRE algorithm that can be deployed in embedded systems. There are two goals that such an implementation must accomplish: 1) It must be efficient, that is, the performance of individual operations within the implementation should require as little execution time as possible, and 2) It must be modular. It is important that any portion can be substituted with a different component at a later time.

This program is written in such a way that it will be easy to replace portions of software code with a hardware device. For example, the functions which read inputs derive those inputs from a file in the software version. The hardware version will have those inputs derived from sensors on the system being controlled. The program was compiled with maximum gcc optimization (e.g. ‘-O9’). It can be noted that other compilers may produce better optimized code, but for the purposes of this work, gcc is used as the baseline because it is widely available for most platforms).

Figure 5.1 shows the high-level program structure. Note that it is very similar in flow to the Matlab Simulink diagram.
A detailed call graph for the implementation is presented on the following pages, broken up into three figures. Figure 5.2 shows the function call tree for the Kleinman algorithm, Figure 5.3 shows the function call tree for the Schur algorithm, and Figure 5.4 shows the tree for the rest of the program (kleinman() and schur() are represented as single nodes in this tree). The percentage value next to each function name is the total amount of time spent in that function, including its children. As expected, 83.8% of the total time is spent in the Kleinman function, while only 0.8% is spent inside of the Schur function. (For a 1000 second inverted pendulum simulation, with random system disturbances).
Figure 5.3: Schur Algorithm Call Graph
Figure 5.4: High Level Software Call Graph
The software implementation must be further analyzed to find other areas for performance improvement. In particular, there are two types of improvements which would benefit this algorithm the most. The first is to reduce the number of memory and I/O operations required for the program. While it performs well (faster than realtime) for the inverted pendulum system on a Pentium 4 workstation, an embedded implementation would likely have less processing power and less memory bandwidth. In addition, the inverted pendulum system is very simple and most real-world examples have more states, requiring more computation. The second improvement is to move certain time-consuming portions of the algorithms into a hardware component for computation. If done correctly, this would help with the first improvement (I/O and memory operations) as well, since the hardware would be responsible for its own I/O and memory for that component.

In Chapter 6, we continue to analyze the software implementation. In particular, it is benchmarked and bottlenecks are identified. The largest bottleneck (software function with the highest execution time) is selected for improvement. This function is analyzed and a hardware implementation which uses systolic arrays, simplifying the software version’s computational intensity is suggested.
Chapter 6

Hardware Co-design

The software package in the previous chapter was designed such that it is ready for an embedded implementation. We now further analyze this program for specific bottlenecks which can then be addressed by utilizing knowledge about the individual algorithms. For example, some portions can be made faster if implemented in hardware [15].

In order to create a hardware-software co-design, a natural partition in the current software implementation must be identified. One possible partition is to implement mathematical operations in hardware and sequential operations (like loops) in software. Before any partition is selected, the computational complexity of the additional synchronization overhead between software and hardware components must be analyzed. The complexity of transferring data to the hardware block and synchronizing the results output must be less than the computational performance gain from re-designing a component in hardware [35]. This is because the transfer and synchronization of data can be time-consuming both to design and to run. However, if the algorithm being moved to hardware is significantly sped-up, this overhead is easily overcome by the net performance gain.

The overall system for a co-design of an SDRE implementation including the controller and the plant is shown in Figure 6.1. The software implementation is able to perform quick calculations using specialized hardware components.

For the SDRE implementation, software benchmarking shows which functions are the most time-consuming for the inverted pendulum application. However, it is important to analyze the complexity of these functions with other applications (more states) also. For
Figure 6.1: System Block Diagram
example, general matrix I/O functions tend to have a complexity of \( O(n^2) \) (the computational workload increases in proportion to the number of elements in a matrix). Other functions, which may have a complexity of \( O(n^3) \) will actually have comparable or better performance than the I/O functions for small matrix computations, but quickly become slower for large matrix computations (see Figure 6.2).

![Figure 6.2: \( O(N^2) \) vs. \( O(N^3) \) complexity for low N](image)

### 6.1 Design considerations

Before any hardware is designed, the functions which are the most time consuming are selected and analyzed. Since the Kleinman algorithm takes the longest amount of execution time in the system, this is the algorithm targeted for speedup. Memory management functions and other non-algorithm related functions are not included in this analysis because they are tied very closely to the architecture of the system running the software. Time consuming functions in the Kleinman algorithm are shown in the call graph in Figure 6.3. Some of these functions have been selected for analysis in Table 6.1.
The functions shown here are in the call tree of the Kleinman algorithm. However, the execution time percentages reflect total function execution, including other portions of the system. Because of this, any improvement to these functions has the potential to speed up other areas of the system as well.

The \textit{dhseqr} function is selected for further analysis in this work, as it has the highest execution time for the inverted pendulum case (47.2%). In addition, a simple “synthetic” test is run on a nine-state system (with randomly generated state matrices). It is found that the \textit{dhseqr} function accounts for about 60% of the execution time in that case. Thus, it appears that this is a good candidate for speedup because as the matrix complexity grows, the function begins to account for more of the execution time.

We see that there are two uses for this function: one use requires the computation of the Schur decomposition, and the other does not – it only requires the computation of the eigenvalues of $H$. To see the separate effects of these two capabilities of the function, all of its uses within the software implementation are analyzed. The functions which use \textit{dhseqr} are \textit{dgees} and \textit{dgeev}. \textit{dgeev} is only called in one place: the Kleinman algorithm. It is used there to verify the validity of the initial guess, so its use is required. In that location, we only need the eigenvalues, not the Schur form or the eigenvectors. The \textit{dgees} function is called in two places, and both of them require the Schur form. The first place is the Schur algorithm (\textit{sb02md}), and the second one is the solution to the Lyapunov equation (\textit{sb03md}).
<table>
<thead>
<tr>
<th>Function</th>
<th>% execution time</th>
<th>Function Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>matrixMultiply</td>
<td>7.0</td>
<td>Matrix multiplication. This is used by other portions of the software as well (like calculating the actual output signal). Matrix multiplication is a $O(n^3)$ operation in general. However, the Level 3 BLAS routine dgemm is being used here. This routine accounts for 3.5% of the total matrixMultiply time. The rest of the time used by memory management functions. Most of these (the ones which deal with memory allocation for the initial matrices and for the result) can not be improved.</td>
</tr>
<tr>
<td>sb03md</td>
<td>45.30</td>
<td>Solution of continuous or discrete time Lyapunov equations.</td>
</tr>
<tr>
<td>dhseqr</td>
<td>47.2</td>
<td>Computes the eigenvalues of real upper Hessenberg matrix $H$, and optionally, the matrices $T$ and $Z$ from the Schur decomposition $H = ZT Z^T T$ where $T$ is an upper quasi-triangular matrix (the Schur form) and $Z$ is the orthogonal matrix of Schur vectors. [2]</td>
</tr>
<tr>
<td>dgeev</td>
<td>29.2</td>
<td>Used by dhseqr to compute the eigenvalues and optionally the eigenvectors of an N by N real nonsymmetric matrix.</td>
</tr>
</tbody>
</table>

used in the Kleinman algorithm.

In order to find the computation time required for both cases of the dhseqr routine, it is split into two separate functions. One function only calculates the eigenvalues of a matrix, while the other one calculates the Schur form as well. Profiling is repeated and it is determined that the “simple” (eigenvalues-only) case accounts for about half of the execution time of both functions for both the inverted-pendulum (4 state) and synthetic (9-state) case. This demonstrates that the simple case of the dhseqr function is a good candidate for hardware speedup.
6.2 *dhseqr* Hardware components

The first step in creating a hardware design is to partition the algorithm into logical blocks. Before we do that, we must select an algorithm to use which performs the same function as *dhseqr* but is more applicable for a hardware implementation. There are many different algorithms to calculate the eigenvalues of a matrix, so the approach taken here is to find a hardware implementation of a similar algorithm and modify it to fit the function required. In particular, a systolic array implementation of the matrix inverse function [16] is selected because it’s both simple and expandable.

In El Amawy’s design [16], two systolic arrays are used – first to calculate the QR decomposition of the dense matrix input, then to calculate the $R$ inverse matrix and finally find the inverse of the original matrix. Out of this design, only the first portion is interesting for an eigenvalue finder. This portion is shown in Figure 6.4. The structural elements each have two modes, determined by whether the inputs to that element carry a flag which is carried by the diagonal elements of the input matrix ($a_{11}, a_{22}, etc$). The modes are defined by Equations 6.1 - 6.7.

- **Circle Element:**
  \[ Y_{out} = X_{in} \text{ and } Y_{out\text{flag}} = X_{in\text{flag}} \]  
  \[ (6.1) \]

- **Rectangle Element:**
  \[ y_{in} \text{ has flag:} \]

  \[ Y_{out} = \sqrt{x_{in}^2 + y_{in}^2} \]  
  \[ (6.2) \]
  \[ X_{out} = 0 \]  
  \[ (6.3) \]
  \[ c = Y_{in}/\sqrt{x_{in}^2 + y_{in}^2} \]  
  \[ (6.4) \]
  \[ s = X_{in}/\sqrt{x_{in}^2 + y_{in}^2} \]  
  \[ (6.5) \]
Figure 6.4: Systolic Array Structure for QR decomposition[16]
$y_{in}$ has no flag:

\[ Y_{out} = cY_{in} + sX_{in} \quad (6.6) \]
\[ X_{out} = -sY_{in} + cX_{in} \quad (6.7) \]

In addition, $X_{out}$ carries a flag if $X_{in}$ carried it, and $Y_{out}$ carries a flag if $Y_{in}$ carried it.

The systolic array described in [16] provides the first two pieces for eigenvalue computation by QR decomposition: the $Q$ and $R$ matrices. These matrices can then be used to find the eigenvalues of the input matrix through an iterative process, as demonstrated by the Matlab code in Figure 6.5.

```matlab
oldA=A+1;
while A ~= oldA
    oldA = A;
    [Q,R] = qr(A)
    A=R*Q
end
```

Figure 6.5: Matlab code for QR iteration

As Figure 6.5 shows, the second portion of the hardware implementation must be a matrix multiplier. Once again, a systolic array architecture fits very well for this approach. One implementation is shown in Figure 6.8 [24]. By passing the elements of matrices $A$ and $B$ through the array structure, the results are accumulated within the array. Each array block has a simple function: $accum = accum + X_{in} \times Y_{in}$.

In addition to the matrix multiplication and QR decomposition components, several control signals and signal conditioning elements are needed.

- A buffer component is needed so that the $R$ and $Q$ matrices can be presented to the multiplier at the same time (the $R$ matrix is output first from the decomposition array).
- **Validator** components are required so that the input to the multiplication array is valid. These components filter the input to the multiplication array unless they are told that the inputs are correct.

- A control signal for clearing the array before the next multiplication is to begin.

Loop elements are best left in software, so the QR iteration itself can be performed in software while the computations are performed in hardware. However, if the entire QR iteration is to be performed in hardware, the following components are required as well:

- A multiplexer element to decide whether the matrix input for the QR decomposition array should come from the outside or from the multiplication array.

- A comparator to determine when the QR iteration can stop (the eigenvalues have been found).

The entire structure is shown in Figure 6.6. The VHDL implementation mirrors this structure. It consists of the following entities:

**Syst_Arr** This is the implementation of the QR decomposition structure shown in Figure 6.4. It includes generic arrays of the following components:

**A1rect** As described in [16], this entity implements the rectangular element of the systolic array.

**A1circ** Also as described in [16], this entity implements the circular element of the systolic array.

**BufferElement** This entity is a simple delay buffer. It accepts an input, waits one clock cycle, and then outputs it. By placing these in a generic array (of size $N$), it is possible to delay the $R$ matrix so that it can enter the multiplication structure at the same time as the $Q$ matrix.

This element is synchronous, and the architecture is structural: it only includes port mappings between the different elements and a short “process” block describing
when to show the output vectors ($outQ$ and $outR$). These are output on the falling edge of the clock to ensure that the signals have settled by the time the multiplier attempts to read them at the rising edge of the clock.

**Validator** This is an asynchronous component which performs an “and” operation between the input array and the valid array. This has the effect of zeroing any elements in the input array which are not valid.

**Mult_Arr** This is the multiplication array as shown in Figure 6.8. Like the QR decomposition array, this one is also implemented as a generic array. The main element in the
array is:

**MultElement** This is a multiplication element which contains an accumulator. The function is to multiply both of inputs together, add the result to the accumulator, and then pass the inputs on to the outputs. $X_{in}$ is passed to $X_{out}$ and $Y_{in}$ is passed to $Y_{out}$. This is a structural architecture.

**QRsystem** This is the overarching system, which includes all of the rest of the elements, as shown in Figure 6.6. This is also a structural architecture – it only contains port mappings.

### 6.2.1 Validating the design

A testbench is required to ensure that the design is functionally valid. First, each of the major components is tested individually to verify its functionality. Then, a global testbench is built for the QRsystem entity.

All of the testbenches read a data file from the file system of the computer the simulation is being run on. For the QRsystem testbench, this datafile contains a matrix and flags for that matrix (as described in [16]). A sample simulation (for a $2 \times 2$ matrix) for this testbench is shown in Figure 6.7. The simulation shows that the matrix $$\begin{pmatrix} 2 & 3 \\ 5 & 6 \end{pmatrix}$$ along with the identity matrix ($I_2$) is fed into design. Although the $qmatrix$ values begin changing earlier, the valid signal ensures that the $Q$ and $R$ matrices aren’t entered into the multiplication array too early. The valid signal feeds the skewed $Q$ and $R$ matrices into the multiplication array, and at the same time the reset signal becomes low, meaning that the buffers in the array can begin accumulating. After flowing through the multiplication array, the multiplication result is available (near the end of the simulation). Verification shows that this result $A_{new} = \begin{pmatrix} 8.2069 & -2.5172 \\ -0.5172 & -0.2069 \end{pmatrix}$ is the correct result for a single iteration of the algorithm.

Of course, this simulation only verifies the functionality of the behavioral model. In
order to determine the actual performance of the block, it must be synthesized and timed. However, because the Fixed-Point library routines used in this design have not yet been standardized [8], the division algorithm is not synthesizeable by the Xilinx ISE 8.2i tool. In addition, the algorithm requires a square root function, which is not present in the library. In order to demonstrate the synthesis process, the matrix multiplication component of the design is synthesized.
6.3 Matrix multiplication

To show the advantage of using an FPGA hardware component for certain pieces of the SDRE design, the matrix multiplication algorithm is implemented in VHDL, synthesized, and timed.

The software version uses the Level 3 BLAS operation (dgemm) to perform the multiplication. This operation requires $O(N^2)$ memory accesses and $O(N^3)$ floating-point operations. The best approach to improving this performance in hardware is to “exploit [the] inherent data flow of the operations” [35]. Because of this, the most straightforward approach for a hardware matrix multiplication component is to operate on multiple rows and multiple columns concurrently. This approach is utilized in the systolic array design presented in the previous section.

### 6.3.1 Design

The multiplication array consists of several *MultElement* components. These act as buffers and accumulators. $X_{in}$ is passed to $X_{out}$ of each element, and $Y_{in}$ is passed to $Y_{out}$. Because this array is intended to be used as part of the *dhseqr* function, it actually performs the function $(A \ast B^T)^T$. This is another example of how hardware can be easily manipulated to produce the exact result needed, rather than (in this case) having to take transposes in software.

The multiplication array architecture actually consists of four generic arrays connected together. This is required because the first row and the first column of the array have a slightly different function than the rest (the inputs for these elements come from outside of the array rather than from the previous column or row). The general structure of the array, along with the order of matrix inputs is shown in Figure 6.8.

The design is created using the VHDL Analysis and Standardization Group’s fixed point math package [8]. This is a synthesizeable package which includes all of the basic mathematical operations. (For matrix multiplication, only fixed-point addition and multiplication
6.3.2 Functional results

A testbench is created to verify the functionality of the matrix multiplication block. This testbench reads a set of floating point matrices to multiply from a file, converts them to a fixed-point representation, performs the multiplication using the design under test, and then converts the results back to floating point to simplify verification.

The test matrices used are:
\[ A = \begin{pmatrix} -0.3714 & -0.9285 \\ -0.9285 & 0.3714 \end{pmatrix} \]

\[ B = \begin{pmatrix} -5.3852 & -6.6850 \\ 0 & -0.5571 \end{pmatrix} \]

The result, \( C = \begin{pmatrix} 8.2071 & 2.5173 \\ 0.5173 & -0.2069 \end{pmatrix} \) is shown in Figure 6.9, which shows the testbench output.

Figure 6.9: Multiplication Array Testbench Functional Simulation

**Matrix Multiplication Precision**

The functional performance of the Matrix Multiplication unit depends a great deal on the size of the buses used for the fixed-point calculations. The results in Figure 6.9 use a 64-bit fixed-point representation, with 32 bits for the real portion of the number and 32 bits for the fraction. However, although it produces very precise results, this is not a feasible representation size because the synthesized device becomes too large. (64 bits of precision means that most operations need 64 operational units, which means that the design occupies too much space on an FPGA).
Table 6.2: Inverted Pendulum Maximum Value

<table>
<thead>
<tr>
<th>Max Value</th>
<th>$2^A$</th>
<th>Overflows</th>
<th>Kleinman</th>
<th>Schur</th>
</tr>
</thead>
<tbody>
<tr>
<td>1024</td>
<td>10</td>
<td>33333</td>
<td>0</td>
<td>33334</td>
</tr>
<tr>
<td>2048</td>
<td>11</td>
<td>33333</td>
<td>0</td>
<td>33334</td>
</tr>
<tr>
<td>4096</td>
<td>12</td>
<td>33333</td>
<td>0</td>
<td>33334</td>
</tr>
<tr>
<td>8192</td>
<td>13</td>
<td>33333</td>
<td>0</td>
<td>33334</td>
</tr>
<tr>
<td>16384</td>
<td>14</td>
<td>463</td>
<td>32842</td>
<td>492</td>
</tr>
<tr>
<td>32768</td>
<td>15</td>
<td>379</td>
<td>32883</td>
<td>451</td>
</tr>
<tr>
<td>65536</td>
<td>16</td>
<td>309</td>
<td>32925</td>
<td>409</td>
</tr>
<tr>
<td>131072</td>
<td>17</td>
<td>234</td>
<td>32973</td>
<td>361</td>
</tr>
<tr>
<td>524288</td>
<td>19</td>
<td>168</td>
<td>33005</td>
<td>329</td>
</tr>
<tr>
<td>1048576</td>
<td>20</td>
<td>115</td>
<td>33032</td>
<td>302</td>
</tr>
<tr>
<td>2097152</td>
<td>21</td>
<td>79</td>
<td>33047</td>
<td>287</td>
</tr>
<tr>
<td>4194304</td>
<td>22</td>
<td>64</td>
<td>33052</td>
<td>282</td>
</tr>
<tr>
<td>8388608</td>
<td>23</td>
<td>51</td>
<td>33054</td>
<td>280</td>
</tr>
<tr>
<td>16777216</td>
<td>24</td>
<td>39</td>
<td>33057</td>
<td>277</td>
</tr>
<tr>
<td>4294967296</td>
<td>32</td>
<td>9</td>
<td>33059</td>
<td>275</td>
</tr>
</tbody>
</table>

Two situations are analyzed to figure out the precision that should be used in hardware design. First, the specific application (inverted pendulum) is analyzed to understand what the largest values that occur for matrix multiplication are. Second, this information is used to analyze the performance of the matrix multiplication hardware unit itself.

In order to analyze the inverted pendulum magnitudes, the software is instrumented such that:

- An overflow value is selected for each test. This value ranges from $2^{10}$ to $2^{32}$.
- The matrix multiplication function checks both its inputs and its output for overflow.
- A single overflow is enough to invalidate the Kleinman algorithm’s result and cause the system to use the Schur method.
- Because it is intended to always succeed, the Schur method does not check for overflows. Its implementation can be kept in software and virtually arbitrary precision is possible.
The results are summarized in Figure 6.10 and Table 6.2. Both show that a maximum magnitude of $2^{14}$ is sufficient for this application. In fact, the majority of the cases when the Kleinman algorithm fails to converge are cases when an overflow occurs, so detecting overflow early is actually a way to save time by applying the Schur algorithm immediately. As seen in the graph in Figure 6.10 (note that the number of occurrences is on a logarithmic axis), although the number of overflows becomes exponentially smaller as the maximum value grows, the algorithm still requires approximately the same number of solutions from the Schur algorithm.

The maximum magnitude required for this application has been determined, so we are now able to analyze the effect of the bus width on the Matrix Multiplication unit. The effect
of bus width on precision is analyzed by looking at the percent error of the results, using the 64-bit results as a baseline. The FPGA ratio is also analyzed, again using the 64-bit ratio as a baseline.

Test matrices are randomly generated to perform the precision test. Since the maximum value for both the multiplication input and output can be $2^{14}$, the maximum values of the input matrices can either be $2^{14}$ and $I_N$ or $\sqrt{2^{14}/N}$ and $\sqrt{2^{14}/N}$. The test matrices used in the precision experiment are randomly generated normal matrices in $[-1, 1]$. One is scaled by $2^{14}$, one is scaled by $\sqrt{2^{14}/N}$, and one isn’t scaled. The dimensions of all three matrices are $4 \times 4$ because this is the most common matrix multiplication size for the inverted pendulum controller.

Multiplication precision results are obtained by performing four combinations of matrix multiplications: small, small; small, medium; small, large; medium, medium. As discussed above, it is not possible to perform a large,large multiplication or a medium,large multiplication because these would overflow the real portion size.

The results from the precision tests are shown in Table 6.3. These show that the appropriate number of bits for the fractional portion is 8, since fewer bits can result in significant precision loss. The medium,medium multiplication retained the most precision when the number of fraction bits was decreased because the overall size of the multiplication values is significantly larger than the loss in precision. The multiplications which included the small magnitude matrix lost precision quickly due to multiplication by values smaller than one.

The number of bits required for the fractional component of the matrix multiplication is eight. Combined with the 14 bits required for the real component, this yields a total requirement of 22 bits. This requirement is significantly lower than the 64 bit precision used in the functional verification, and synthesis components with this bus size is much easier.
### Table 6.3: Multiplication Unit Fraction Precision Results

<table>
<thead>
<tr>
<th>Matrices</th>
<th>Real Bits</th>
<th>Frac. Bits</th>
<th>% Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Small x Small</td>
<td>14</td>
<td>32</td>
<td>0.0000</td>
</tr>
<tr>
<td></td>
<td>14</td>
<td>16</td>
<td>0.0037</td>
</tr>
<tr>
<td></td>
<td>14</td>
<td>8</td>
<td>1.2328</td>
</tr>
<tr>
<td></td>
<td>14</td>
<td>6</td>
<td>3.2508</td>
</tr>
<tr>
<td>Small x Medium</td>
<td>14</td>
<td>32</td>
<td>0.0000</td>
</tr>
<tr>
<td></td>
<td>14</td>
<td>16</td>
<td>0.0207</td>
</tr>
<tr>
<td></td>
<td>14</td>
<td>8</td>
<td>0.3033</td>
</tr>
<tr>
<td></td>
<td>14</td>
<td>6</td>
<td>1.0004</td>
</tr>
<tr>
<td>Small x Large</td>
<td>14</td>
<td>32</td>
<td>0.0000</td>
</tr>
<tr>
<td></td>
<td>14</td>
<td>16</td>
<td>0.0216</td>
</tr>
<tr>
<td></td>
<td>14</td>
<td>8</td>
<td>2.8376</td>
</tr>
<tr>
<td></td>
<td>14</td>
<td>6</td>
<td>28.9193</td>
</tr>
<tr>
<td>Medium x Medium</td>
<td>14</td>
<td>32</td>
<td>0.0000</td>
</tr>
<tr>
<td></td>
<td>14</td>
<td>16</td>
<td>0.0015</td>
</tr>
<tr>
<td></td>
<td>14</td>
<td>8</td>
<td>0.0148</td>
</tr>
<tr>
<td></td>
<td>14</td>
<td>6</td>
<td>0.0207</td>
</tr>
</tbody>
</table>

### 6.3.3 Synthesis results

While the design presented here is synthesizeable, there are several additional components which would be required to make it feasible for an FPGA. For example, the matrix multiplication array currently accepts all of its inputs in parallel. While this certainly increases the speed of delivering the inputs, it also means that each bit of each input and output requires an individual pin on the FPGA (e.g. a precision of 24 bits for a $4 \times 4$ matrix multiplication would require 288 pins). Clearly this is not a realistic solution, so before any such design can be fully implemented, serial to parallel (and parallel to serial) converter components are required so that the inputs and outputs can be fed into the chip serially.

The discussion here does not take into account any of the design, chip area, or performance overheads associated with additional synchronization and I/O components as described in the last paragraph. Synthesis is performed only for the computational portion of the matrix multiplication array. Once the design is synthesized, the effect of precision on the design size can be studied. Table 6.4 summarizes these results.

80
Table 6.4: Matrix Multiplication Synthesis Results

<table>
<thead>
<tr>
<th>Real Bits</th>
<th>Frac. Bits</th>
<th>Total Bits</th>
<th>Slices</th>
<th>Max Clock (MHz)</th>
<th>Min Arrival (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>382</td>
<td>147.07</td>
<td>8.214</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>5</td>
<td>660</td>
<td>74.78</td>
<td>14.972</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>10</td>
<td>1966</td>
<td>59.645</td>
<td>21.715</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>17</td>
<td>5442</td>
<td>42.874</td>
<td>26.491</td>
</tr>
<tr>
<td>14</td>
<td>8</td>
<td>22</td>
<td>7643</td>
<td>40.293</td>
<td>26.714</td>
</tr>
<tr>
<td>16</td>
<td>16</td>
<td>32</td>
<td>14733</td>
<td>35.657</td>
<td>30.478</td>
</tr>
</tbody>
</table>

This table shows the tradeoff between matrix multiplication precision and chip space. Also, as the amount of space consumed on the chip increases, the maximum clock speed that can be used decreases and the amount of time required for signals to settle before being clocked increases. This is all due to the increase in complexity in the design. When there are more elements for a signal to propagate through, the design must operate at a slower speed. Likewise, more complexity means that it takes longer for signals to settle.

6.3.4 Timing results

By implementing the systolic array design described in the previous sections, the complexity of the matrix multiplication algorithm has gone from $O(N^3)$ to $O(N)$, where a multiplication happens in 1 unit of time. In fact, for an increase in matrix dimension, the number of total cycles to calculate the product increases by three. (Cycles here does not imply clock cycles, as every multiplication element requires several clock cycles to perform each multiplication). Figure 6.11 shows a graphical representation of this improvement in performance. Of course, if the multiplication unit used by the software implementation is significantly faster than that used by the hardware implementation, the performance increase isn’t as great.

The simulation and synthesis results indicate that a design like the one presented in this chapter is feasible for improvement of an eigenvalue finder function as compared to software implementations. In software, this function has a complexity of $O(n^3)$ while the theoretical hardware complexity is $O(n)$. Additional performance gain would be obtained from using fixed-point calculations instead of floating point, as done in the software implementation. As described earlier, the additional complexity of synchronization and I/O
overheads are not taken into account in this design, therefore the actual performance improvement will not be as great.

We have demonstrated the improvement that implementing a portion of a design in hardware can bring to a software algorithm. Even if the iterative loop for the QR decomposition routine is allowed to remain in software, the complexity of each iteration has decreased significantly. The SDRE algorithm’s performance can be greatly improved if this hardware component is integrated. As discussed earlier, other factors that will help increase the speed of the algorithm are correctly choosing the required precision, $Q$ and $R$ weighing matrices, SDC parametrization, and maximum Kleinman algorithm iterations. With all of these modifications, the SDRE approach will be applicable to systems with higher performance requirements and more states.
Chapter 7

Conclusions

This work presented the SDRE method as used by many control engineers. We discussed the advantages of this method over other non-linear control techniques: wider applicability and a low-complexity, systematic method. While the SDRE approach has these advantages, previous implementations show that performance improvements are necessary. The need for a faster, embeddable implementation of the SDRE algorithm that could control more complex systems in “real-time” was presented.

We discussed the challenges in improving the performance of this algorithm, and presented several potential improvements. First we discussed the need to make proper off-line selections (SDC parametrization and weighing matrix selection) before applying the SDRE algorithm. The latter is analyzed with a sample application and poor choices are demonstrated.

Previous implementations of the SDRE approach use the Schur algorithm to solve the associated Riccati Equation with the Kleinman iterative algorithm used for refinement. Although two different iterative algorithms (Newton’s method and Newton’s method with Exact Line Search) are profiled for use as the refinement algorithm, it was determined through both timing and accuracy tests that the current approach of using the Kleinman algorithm is best.

Once the Schur/Kleinman SDRE algorithm was implemented using a modular C software design, the performance of this algorithm was further examined. In particular, we looked at the maximum number of iterations for the Kleinman algorithm and found that
as few as eight iterations is effective (for the sample application) for both Kleinman accuracy and overall system speed. The experimental results presented here indicated that the maximum number of Kleinman iterations should be guided by whether the plant exhibits regular behavior, with its states not varying by large amounts between sampling times.

Finally, the software implementation was profiled and algorithm bottlenecks identified. A methodology was presented for designing a hardware component for the *dhseqr* function, which finds the eigenvalues of a matrix. Functional results were presented demonstrating that the design works, and a portion of this design, matrix multiplication, was synthesized. This design, if combined with the SDRE software implementation would significantly improve the performance of the algorithm.

All of our results indicate that the SDRE algorithm has potential for performance improvement by applying the methodologies presented here and further analyzing the algorithm’s bottlenecks. In order to explore this potential fully, there are several directions for future work.

### 7.1 Future work

A good methodology for the analysis and improvement of the SDRE algorithm is presented. However, there are several directions which this work does not explore.

The ultimate goal is to improve the performance of the SDRE algorithm so that it is applicable for a wider subset and for more complex systems. Two things must happen to accomplish this goal: 1) A complete analysis of the scalability of the SDRE algorithm as the number of states increases, and 2) Application of the hardware design methodology developed here to other software algorithm bottlenecks. Also, an efficient way to interconnect the hardware and software components must be determined. One way to do this is to use an FPGA with a processor implemented on it.

Another possibility is to convert the entire algorithm to fixed-point to remove most of the overhead from the floating point calculations. Libraries such as the one in [8] for
hardware design and others for software are making this much easier.

We showed several methods of detecting failure of the algorithm in this work. However, a more rigorous analysis of failure occurrences should be completed. This would also allow for a better understanding of the scope of control applications for the SDRE method. Another approach that can be applied to minimize the failure of the Kleinman approach may be a back-off algorithm. In this scenario, the frequency of Kleinman failure can be recorded and if it passes a certain threshold, the Kleinman approach is not used for a specified number of sampling intervals.

Although we compared three different iterative methods for solving Algebraic Riccati Equations, this work does not include the so-called Quasi-Newton Method, presented in [23]. This method should be independently analyzed and ranked against the other three iterative methods compared here. The authors claim that this method performs better than the Kleinman approach.
Bibliography


Appendix A

Controllability, stabilizability, observability, detectability [20]

This work relies heavily on the assumptions that the parametrization of the input-affine system being controlled is observable (detectable) and controllable (stabilizable). (See Chapter 2.2 for more details on the SDRE approach). This Appendix defines the four terms in these assumptions.

A.1 Controllability and stabilizability

The controllability property of a system determines whether an initial state \( x_0 \) can be steered to the origin using the input (control) \( u(t) \) in a finite amount of time. This is reflected in the following definition:

**Definition 3** A state \( x_0 \) is controllable if there exists a finite interval \([0, T]\) and an input \( \{u(t), t \in [0, T]\} \) such that \( x(T) = 0 \). If all states in the system are controllable, the system is controllable.

The test for controllability is presented in Theorem 1.

**Theorem 1** Consider the state space model:

\[
\delta x[k] = A_\delta x[k] + B_\delta u[k] \\
y[k] = C_\delta x[k] + D_\delta u[k]
\]

90
i. The set of all controllable states is the range space of the controllability matrix \( \Gamma_c[A, B] \), where
\[
\Gamma_c[A, B] \triangleq \begin{pmatrix} B & AB & A^2B & \ldots & A^{n-1}B \end{pmatrix}
\]

ii. The model is completely controllable if and only if \( \Gamma_c[A, B] \) has full row rank.

A system can be either completely controllable or not completely controllable. If it is not completely controllable, the system can be decomposed into the sum of controllable and completely uncontrollable subsystems.

A state-space model of a system is stabilizable if its uncontrollable subspace is stable. The stability of this subspace is determined by the location of the eigenvalues of the uncontrollable subsystem.

### A.2 Observability and detectability

Observability involves being able to determine the plant states when given the system output. In particular, if a system has a non-zero state and zero input, but produces zero output, that state is deemed unobservable. This is formally defined below:

**Definition 4** The state \( x_0 \neq 0 \) is said to be unobservable if, given \( x(0) = x_0 \), and \( u[k] = 0 \) for \( k \geq 0 \), then \( y[k] = 0 \) for \( k \geq 0 \). The system is completely observable if there exists no nonzero initial state that is unobservable.

Similarly to controllability, there is a test for observability, presented in Theorem 2.

**Theorem 2** Consider the state-space model:
\[
\begin{align*}
\delta x[k] &= A\delta x[k] + B\delta u[k] \\
y[k] &= C\delta x[k] + D\delta u[k]
\end{align*}
\]
i. The set of all unobservable states is equal to the null space of the observability matrix
\[ \Gamma_0[A, C], \]
where
\[
\Gamma_0[A, C] \triangleq \begin{pmatrix}
C \\
CA \\
\vdots \\
CA^{n-1}
\end{pmatrix}
\]

ii. The system is completely observable if and only if \( \Gamma_0[A, C] \) has full column rank \( n \).

A state-space model of a system is detectable if its unobservable subspace is stable.

The similarity between Theorems 1 and 2 results in a duality property, formalized in Theorem 3.

**Theorem 3** Consider a state space model described by \((A, B, C, D)\). Then the system is completely controllable if and only if the dual system \((A^T, C^T, B^T, D^T)\) is completely observable.

### A.3 Example

The inverted pendulum in Chapter 4 exhibits the following state vector during its Matlab simulation at certain time \( t \):
\[
x(t) = \begin{pmatrix}
3.112562 & -1.891076 & -0.004449 & -0.291277
\end{pmatrix}
\]

Using the SDC parametrization presented in that chapter, we obtain the following state
matrix $A$ and input matrix $B$ at time $t$:

$$
A = \begin{pmatrix}
0 & 1 & 0 & 0 \\
0.85155 & -0.02304 & 0 & 0 \\
0 & 0 & 0 & 1 \\
0.03839 & -0.00351 & 0 & 0
\end{pmatrix}
$$

$$
B = \begin{pmatrix}
0 \\
13.10459 \\
0 \\
1.99929
\end{pmatrix}
$$

Using Theorem 2, we obtain the controllability matrix:

$$
\Gamma_c = \begin{pmatrix}
0 & 12.9707 & -2.0127 & 42.8764 \\
12.9707 & -2.0127 & 42.8764 & -13.2578 \\
0 & 1.9900 & -0.3088 & 1.9563 \\
1.99 & -0.3088 & 1.9563 & -1.3170
\end{pmatrix}
$$

This matrix has rank of 4, so at this point in time and for this particular state the system is controllable. As defined in Section 2.2, in order for the entire SDC parametrization to be controllable, this procedure would have to be applied pointwise for all possible state trajectories $x$ in a set that defines the domain of operation of the system.
Appendix B
Software Documentation

This work implements a software package which contains an SDRE controller. This package is further explained in this Appendix. In particular, the details of the main functions within the implementation are discussed, and instructions for using the package are given.

B.1 Using runSDRE

The SDRE software implementation discussed in this work is made up of several components which build into a binary called runSDRE. This binary reads its inputs from a data file and performs the SDRE calculations. There are several calculation and debugging options:

- The input file can contain either state matrix and weighing matrix values \((A, B, Q, R)\) or plant state values. Figures B.1 and B.2 show examples of these two input files. Each matrix is specified on a single line, with the first two values in the line defining the matrix dimensions. (Figure B.1 contains states for a plant which has four states, so the matrix dimensions are \(4 \times 1\). Figure B.2 contains the \(A\) and \(B\) state matrices followed by the \(Q\) and \(R\) matrices).

Whichever input type is selected, using the \(-s\) (State Vector) or \(-r\) (Riccati Matrices) options, the runSDRE program solves every set of matrices present in the file and then terminates. (The “-f” option is used to actually specify the input file name).

- The “-n [number]” option can be specified to run the entire program more than once.
• The “-k [number]” option can be used to set the maximum number of Kleinman iterations before resorting to Schur. (“-O” only uses the Schur algorithm, never even calling Kleinman).

• The “-m [number]” option can be used to limit the maximum value in any Kleinman algorithm matrix multiplications. This causes the Kleinman result to be considered invalid if an overflow occurred at any point during the calculation.

• There are several output flags that can be used: “-p” prints the result (control output) for every solution. “-v” enables verbosity. This option can be specified several times for more information output. “-d” enables debug messages for internal functions, such as memory allocations.

• Additionally, the program can be built either with or without the “-DHW” flag in the Makefile. This causes hardware components to be used instead of software modules. Currently, this flag causes the system to use the C software simulation for the QR-iteration component instead of the dhseqr function.

A full example run is presented in Figure B.3. This example calls runSDRE on a data file containing two pendulum state vectors. It sets the maximum Kleinman iterations to seven, the maximum matrix multiplication magnitude to one million, and specifies several of the output options. (Some of the output has been removed because it is repetitive).

```
4 1 3.141593 0.000000 0.000000 0.000000
4 1 3.143410 0.159714 0.000277 0.024436
4 1 3.125253 -1.411111 -0.002490 -0.216053
4 1 3.052412 -3.137173 -0.013786 -0.493806
4 1 2.938865 -4.317728 -0.032137 -0.718139
```

Figure B.1: Sample inverted pendulum states input file
B.2 Software functions

This section describes the major functions in the runSDRE package. This package is comprised of several I/O functions (needed to get data from either a data file or the actual plant), low-level matrix manipulation functions, and matrix algebra functions (mostly wrappers for functions from the LAPACK, SLICOT, and BLAS libraries). The top-level module is “runSDRE.c”. It is responsible for command-line arguments, initial memory allocations for static matrices, etc. If this program is converted to run in an embedded environment, most of the changes would take place in this module.

B.2.1 I/O functions

**readABQR** This function reads the $A, B, Q, R$ matrices from a data file. It returns 0 as long as there is more data to be read, and !0 as soon as the file is empty.

**readStates** Similar to readABQR, but used when state vectors are the system input.

**readMatrix** Takes a file and a position in that file as arguments, reads a matrix located at that position into a Matrix structure variable.

**getQR** Used when state vectors are the system input, this function returns the $Q$ and $R$ matrices to be used.
Starting next set of data.

Matrix A:

0.00000 1.00000 0.00000 0.00000
-24.23036 -0.38835 0.00000 0.00000
0.00000 0.00000 0.00000 1.00000
-0.72054 -0.08983 0.00000 0.00000

Matrix B:

0.00000
6.98719
0.00000
1.61623

Matrix Q:

500.00000 0.00000 0.00000 0.00000
0.00000 1.00000 0.00000 0.00000
0.00000 0.00000 10000.00000 0.00000
0.00000 0.00000 0.00000 1.00000

Matrix R:

3.00000

Matrix Guess:

[All Zeros]

Performing Schur decomposition

Matrix Riccati Solution:

938.73598 201.74140 -2011.19880 -769.54816
201.74140 46.00810 -449.79190 -177.44952
-2011.19880 -449.79190 6062.75251 1837.34840
-769.54816 -177.44952 1837.34840 702.16740
-156.19229

Starting next set of data.

Matrix A:

[Same as above]

Matrix B:

[Same as above]

Matrix Q:

[Same as above]

Matrix R:

3.00000

Matrix Guess:

[Same as Riccati Solution above]

Performing Kleinman iterations

convr: 0.001000, difference, 0.000001, numIter 1, iterLimit 7
Kleinman done successfully. 1 iterations, 0.000001 convergence

Matrix Riccati Solution:

938.73598 201.74140 -2011.19880 -769.54816
201.74140 46.00810 -449.79190 -177.44952
-2011.19880 -449.79190 6062.75251 1837.34840
-769.54816 -177.44952 1837.34840 702.16740
-156.19229

Schur solved: 1 equations, Kleinman solved 1 equations, Total was 2 equations.

0.020875 seconds

Figure B.3: Example execution of runSDRE program
B.2.2 Matrix functions

There are functions which operate on matrices without requiring any outside library routines. Any of these functions which return a new matrix leave the caller responsible to free that matrix.

**createMatrixStruct** The majority of places in this program use a Matrix structure, rather than a simple array of numbers. This is useful so that the dimensions of the matrix don’t need to be passed around separately – they are always together with the data. The matrix structure also contains a name for the matrix, useful when debugging.

The createMatrixStruct function allocates and clears memory for a matrix structure, sets the matrix dimensions and the title. The caller is responsible for freeing the memory allocated by this function.

**freeMatrixStruct** Used to free a Matrix structure.

**computeSS** This is one of the few functions which are specific to the inverted pendulum application. It takes the state vector as an input and calculates the $A$ and $B$ matrices.

**truncateMatrix** This function is used to figure out whether an overflow would occur. It sets the global overflowed variable if any of the elements in the passed-in matrix are higher than the maximum allowed value.

**skewMatrixLeft** This function is used to skew a matrix so that the result has the same number of rows ($m$) and $m - 1$ extra columns. The extra spaces are filled with a passed-in padding value. This function is used to create the input values for the QR algorithm.

**skewMatrixRight** Same as skewMatrixLeft but skews the matrix in the opposite direction.

**flipHorizontal** Returns the input matrix flipped horizontally. (Column 1 becomes column N).
**identity**  Returns an identity matrix of dimensions $N \times N$. $N$ is the only parameter.

**matrixAdd**  Adds two matrix structures, puts the result in the passed-in placeholder.

**concatMatrices**  Concatenates two matrices together. This function takes two square matrices of dimension $M \times N$ and returns one matrix of dimension $2M \times N$.

**maxElement**  Returns the highest element present in the matrix.

**minElement**  Returns the lowest element present in the matrix.

**matrixEquals**  Determines whether two matrices are equal. This function is very slow, but is only used in hardware module simulations. A hardware implementation of a parallel comparator would be very simple.

This function only checks for equality up to digits precision.

**matrixEqualsAbs**  Determines whether the matrices are equal if the absolute values of all elements are taken before checking equality.

There are also several simple helper functions: getValue, getValuePtr, modifyOne, modifyAll, addMatrices, subMatrices, deleteMatrix, copyFrom, copyFromTo.

### B.2.3 Matrix algebra functions

Any of the functions below which require a BLAS, LAPACK, or SLICOT library call are responsible for data conversion and alignment to fit the library routine’s requirements. They also set up the workspace for the routines which require additional memory to perform their calculations.

**kleinman**  The kleinman algorithm wrapper. This function takes $A$, $B$, $Q$, $R$, $X$ as inputs ($X$ is the matrix where the result is stored between datasets). Also requires a convergence value and the maximum number of iterations to use. The convergence is calculated by subtracting solutions from consecutive iterations and looking at the
highest absolute value of the elements. The algorithm fails if the maximum number of iterations is reached before the solution converges.

This function uses the \textit{dgeev} function to detect negative eigenvalues in the guess (determine whether the guess is stabilizing) and the \textit{lyap} function to calculate the Lyapunov solution.

\textbf{schur} Similar to the kleinman algorithm wrapper, this is the Schur wrapper. Accepts $A$, $B$, $Q$, $R$ and $X$ as inputs. This function uses the SLICOT \textit{sb02md} function to perform the Schur calculation.

\textbf{inverse} Calculates the inverse of a matrix. This function uses the LAPACK \textit{dgetrf} routine to perform LU factorization, and the \textit{dgetri} routine to calculate the matrix inverse on the factors.

\textbf{matrixMultiply} This function is a wrapper for the BLAS \textit{dgemm} routine. It accepts two Matrix structures and a result placeholder as parameters. In addition, similar to the \textit{dgemm} function, it is possible to specify whether to transpose either of the matrices before multiplying them.

\textbf{lyap} Solves the Lyapunov equation required for the Kleinman algorithm. Takes $A$, $Q$ and a solution placeholder as parameters. This function uses the SLICOT \textit{sb03md} function to perform the calculation.

\textbf{transposeStruct} Transposes a Matrix structure.

\textbf{transpose} Transposes a matrix array (without the structure).

\section*{B.2.4 Debugging functions}

\textbf{printMatrixStruct} Prints the passed-in matrix to stdout. This function requires verbosity of two or greater.
DEBUG Printf-style function which is only output to stdout if the “-d” command-line parameter is used.

printMatrix Similar to printMatrixStruct but requires the dimensions of the matrix.

printMatrixTitled Similar to printMatrix but also prints a passed-in title.

B.2.5 Library functions

The functions in this section are SLICOT, LAPACK, or BLAS routines which are used within the runSDRE software package. Only the top-level functions (functions directly called by runSDRE code) are described, however all functions which are required for compilation are listed. All descriptions are taken from the function descriptions in the specific libraries [2, 6, 9].

dgeev (LAPACK function) DGEEV computes for an $N$-by-$N$ real nonsymmetric matrix $A$, the eigenvalues and, optionally, the left and/or right eigenvectors.

The right eigenvector $v(j)$ of $A$ satisfies $A * v(j) = \lambda(j) * v(j)$ where $\lambda(j)$ is its eigenvalue. The left eigenvector $u(j)$ of $A$ satisfies $u(j)^T * A = \lambda(j) * u(j)^T$ where $u(j)^T$ denotes the conjugate transpose of $u(j)$.

The computed eigenvectors are normalized to have Euclidean norm equal to 1 and largest component real.

sb03md (SLICOT function) Solves for $X$ either the real continuous-time Lyapunov equation $op(A)' * X + X * op(A) = scale * C$ or the real discrete-time Lyapunov equation $op(A)' * X * op(A) - X = scale * C$ and/or estimate an associated condition number, called separation, where $op(A) = A$ or $A^T$ and $C$ is symmetric ($C = C^T$). $A$ is $N$-by-$N$, the right hand side $C$ and the solution $X$ are $N$-by-$N$, and $scale$ is an output scale factor, set less than or equal to 1 to avoid overflow in $X$.

dgemm (BLAS function) DGEMM performs one of the matrix-matrix operations $C = \alpha * op(A) * op(B) + \beta * C$, where $op(X)$ is one of $op(X) = X$ or $op(X) = X^T$, $\alpha$
and $\beta$ are scalars, and $A$, $B$ and $C$ are matrices, with $op(A)$ an $m \times k$ matrix, $op(B)$ a $k \times n$ matrix and $C$ an $m \times n$ matrix.

**sb02md** (SLICOT function) Solves for $X$ either the continuous-time algebraic Riccati equation $Q + A^T X + X A - X B R^{-1} B^T X = 0$ or the discrete-time algebraic Riccati Equation $X = A^T X A - A' X B \left( R + B' X B \right)^{-1} B'^T X A + Q$ where $A$, $B$, $Q$ and $R$ are $N$ by $N$, $N$ by $M$, $N$ by $N$ and $M$ by $M$ matrices respectively, with $Q$ symmetric and $R$ symmetric nonsingular; $X$ is an $N - by - N$ symmetric matrix. The matrix $G = B * R^{-1} B'$ must be provided on input, instead of $B$ and $R$, that is, for instance, the continuous-time equation $Q + A' X + X A - X G X = 0$ is solved, where $G$ is an $N - by - N$ symmetric matrix. SLICOT Library routine SB02MT should be used to compute $G$, given $B$ and $R$. SB02MT also enables to solve Riccati Equations corresponding to optimal problems with coupling terms.

The routine also returns the computed values of the closed-loop spectrum of the optimal system, i.e., the stable eigenvalues $\lambda(1), ..., \lambda(N)$ of the corresponding Hamiltonian or symplectic matrix associated to the optimal problem.

**dgetrf** (LAPACK function) DGETRF computes an LU factorization of a general $M - by - N$ matrix $A$ using partial pivoting with row interchanges.

The factorization has the form $A = P * L * U$ where $P$ is a permutation matrix, $L$ is lower triangular with unit diagonal elements (lower trapezoidal if $m > n$), and $U$ is upper triangular (upper trapezoidal if $m < n$).

This is the right-looking Level 3 BLAS version of the algorithm.

**dgetri** (LAPACK function) DGETRI computes the inverse of a matrix using the LU factorization computed by DGETRF.

This method inverts $U$ and then computes $A^{-1}$ by solving the system $A^{-1} * L = U^{-1}$ for $A^{-1}$.