Real-time Receding Horizon Control:
Application Programmer Interface Employing LSSOL

by

Sean Mark Estill

B.S. (Texas A&M University) 2002

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Master of Science, Plan II

in

Engineering - Mechanical Engineering

in the

GRADUATE DIVISION

of the

UNIVERSITY OF CALIFORNIA, BERKELEY

Committee in charge:

Professor Andrew Packard, Chair
Professor Benson Tongue

December 2003

Contents

List of Figures **v**

List of Tables **vi**

1 Introduction **1**

1.1 RHC Overview 1

1.2 Minimizing Problem Formulation Time 1

1.3 Integration with Open Control Platform (OCP) 2

2 Problem Description and Formulation **4**

2.1 LSSOL Problem Description 4

2.2 Build Specifications 5

2.3 Frame Update 6

2.4 RHC Problem Formulation for Each Time Frame 7

2.4.1 Standard Formulation 7

2.4.2 Constraint Relaxation 8

3 Example Problem **11**

4 Code Organization **14**

4.0.3 Formulation 14

4.0.4 Solving 15

4.0.5 MEX File Creation 15

4.0.6 Simulation Environment 16

5 Code Testing **17**

5.0.7 Correct Formulation 17

5.0.8 Time for Formulation 17

6 Choosing An Optimization Algorithm **19**

7 Looking Beyond the Horizon **20**

List of Figures

3.1 Comparison of relaxed and unrelaxed constraints 12
3.2 Varying initial conditions using soft constraints 13

List of Tables

2.1 Declaration of size and meaning for dynamics and cost data 9
2.2 Declaration of size and meaning for constraints data 10

5.1 Results on a Pentium 4, 1.7 GHz with 512 MB RAM, Red Hat Linux 8.0 18

Acknowledgements

The author would like to acknowledge the commitment of his co-worker Kenneth Hsu to the completion of this project. We worked together on writing the code, integrating it with the Open Control Platform (OCP), and writing the report. Also the "hands-on" guidance of Professor Andrew Packard has been essential to its success. He also played a large part in developing the code and writing this report.

Chapter 1

Introduction

1.1 RHC Overview

Receding Horizon Control (RHC) is based on theory similar to Linear Quadratic (LQ) control. The LQ Regulator (LQR) produces the optimal control sequence over an infinite time frame based on the minimization (or maximization) of a predefined quadratic function. For the given problem, no other solution is optimal. RHC is suboptimal in the sense that it performs the optimization of the control sequence over a finite time frame or horizon length and therefore can not find the optimal control sequence beyond the end of that horizon. However, an additional term is included that takes into account the remaining control sequence beyond the horizon length.

The LQR produces a solution based on analytical expressions. RHC does not have that convenience. It requires numerical solutions for it to be implemented. Therefore, it needs a quadratic program to be implemented. As a result, RHC can easily take into account constraints. However, the inclusion of constraints might affect the stability of a system.

1.2 Minimizing Problem Formulation Time

It was important to make sure the time for formulating a problem was reduced. A faster formulation time yields more time for the problem to be optimized. A key precept was to minimize the time/work needed to perform routines that are used the most. If a fraction of time could be shaved off of one routine that is used hundreds of times, a significant improvement could be realized.

The motivation behind minimizing the formulation time was to enable dynamic control. To capture changes in the system being controlled, in the control objective, or in the

constraints, the problem is reformulated at each time frame.

All the coding for the problem formulation was orginally done in Matlab. This is too slow for the solution rate that is needed. Once the concepts were well developed, the problem formulation code was further developed in C. Since C allows access to pointers, the arrays can be more quickly manipulated.

1.3 Integration with Open Control Platform (OCP)

A handful of institutes and companies is working on this DARPA funded project. The list of schools includes University of California Berkley, University of Minnesota, California Institute of Technology, Massechusetts Institute of Technology, and Georgia Institute of Technology. Each school has a unique contribution to the project.

The concept being developed is a software control platform that can take any system and apply an optimal control to it. The fundamental building block of the platform is the *OCP component*. One such component is the optimizer. Since OCP is a realtime control tool, the optimization of the control is performed in realtime as a periodic task.

The optimizer component is divided up into three phases: pre-optimize, optimize, and post-optimize. The component cycles through all three of the phases every optimizer component timeframe. During pre-optimize, a warm-start solution might be determined, and the data that the problem being solved is dependent on might be gathered. It is during this phase that the data is gathered and prepared to be given to a numerical solver. This phase has a predetermined amount of time to run for each frame.

During the optimize phase, the data is given to the numerical solver, which finds a solution to a given problem. This phase does not necessarily have a predetermined amount of time to run each frame. Although it does have a maximum time, the remaining time is given to other processes if it finishes early. The post-optimize phase takes the results of the numerical solver and issues them as commands to whatever system is being controlled.

The primary task taken on by the University of California group led by Professor Andrew Packard was the development of an application programmer interface (API) that produces from intuitive, user-defined problems the formulation that the linear quadratic optimizer LSSOL expects, which is a single quadratic function with one linear constraint function. When this OCP API was finally developed, it was written in C code. To integrate it as an OCP component written in C++, the API code was separated into various classes and functions.

With LSSOL as the numerical solver, the pre-optimize phase includes the gathering of state observations, matrix parameters, signal readings, and mode indicators. Once all of this data is gathered, a problem formulation function is called that performs the task

of translating the data from the user into a form that is recognized by LSSOL. The problem formulation is then handed over to the optimize phase where LSSOL optimizes a decision variable. If a feasible solution is not found, the constraints are relaxed, and LSSOL comes up with a feasible solution that minimizes the violation of the constraints by including a relaxation term in the decision variable.

Chapter 2

Problem Description and Formulation

2.1 LSSOL Problem Description

LSSOL is a Fortran package for constrained linear least-squares and convex quadratic programming. The general problem that LSSOL solves is:

$$\min_{v \in \mathbf{R}^n} \; F(v)$$

subject to

$$l \leq \begin{bmatrix} v \\ Cv \end{bmatrix} \leq u$$

LSSOL has a number of objective functions to choose from when specifying a problem. The one used for this problem is

$$c^T v + \frac{1}{2} v^T A v,$$

where A is symmetric and positive semi-definite.

The variable v is the decision variable. It is a vector of length n. The decision variable v and some linear combinations of it by the matrix C are constrained by the lower bound l and the upper bound u vectors.

All of the matrices can be allocated more space than actually needed. That means the quadratic cost matrix A, the linear cost vector c, the constraint matrix C, and the lower and upper bound vectors l and u can be allocated more rows and columns than actually needed or used for a given problem. LSSOL takes in both the number of rows allocated as well as the number of rows used.

This feature is particularly useful for accommodating the change in problem size when switching from having only hard constraints to having soft constraints. When the problem is realized to be infeasible, the constraint relaxation term ϵ is added to the decision variable, previously consisting of only v. This requires a change in size for many of the LSSOL parameters. Rather than free and reallocate the sizes and reformulate the entire problem in the case that constraint relaxation is needed during the middle of solving a problem, the maximum size is allocated at the beginning and the more minor changes are then applied to the matrices.

Since the constraint relaxation employed affects the cost function linearly, the quadratic cost matrix A is padded with length(ϵ) zeros across the columns and rows. The linear relaxation cost term ρ is tacked onto the bottom of the linear cost vector c.

2.2 Build Specifications

In building the component, the user must specify

- dimensions – input, state, constraint, parameter vector, mode vector;
- dynamics – in the form of mode and parameter-dependent matrices;
- cost function – in the form of mode and parameter-dependent matrices;
- constraints – in the form of mode and parameter-dependent matrices;
- weights for constraint relaxation;
- basis set for input;
- warm start function;

2.3 Frame Update

At time step k, RHC component receives 5 pieces of data from other OCP components:

- horizon length, H;
- state estimate, x^{est};
- signal estimate, $d^{\text{est}}_{k:k+H-1}$;
- mode vector, I;
- parameter vector, δ.

The hard-realtime function `Pre` does 2 operations. It

- calls the warm-start function, which has two purposes:
  - initialize the decision variable v, based on $(H, x^{\text{est}}, d^{\text{est}}, I, \delta)$, and v_{final} from the previous frame;
  - compute a candidate control action u_0.
- forms the linearly constrained, quadratic program using H, x^{est}, $d^{\text{est}}_{k:k+H-1}$, and the dynamic, constraint and cost matrices which are user-defined functions of I and δ. This is discussed further in section 2.4.

Although problem-dependent, the user can give worst-case (problem dependent) execution times for both of these steps, so these can be done in hard realtime fashion.

The anytime function `Opt` does one operation, calling LSSOL to solve the QP.

- If LSSOL returns infeasible, then the relaxed problem (see Section 2.4.2) is formed, and LSSOL is called again.
- `Opt` is terminated by the Scheduler. Cleanup will be a hard realtime task.

Finally, the hard real-time function `Post` is called.

- `Post` converts the most recent value of v into a control action, u^v_0.
- `Post` then makes a decision of which control action to use, u_0 or u^v_u. This value becomes the RHC component's output at the end of the frame.

2.4 RHC Problem Formulation for Each Time Frame

2.4.1 Standard Formulation

The following is the set up that uses the above data. For each time frame, this problem is formulated and solved. Given the optimization horizon H, initial condition x_0, dynamic model matrices, cost matrices, constraint matrices, and known signal $\{d_k\}_{k=0}^{H-1}$, consider a constrained, convex quadratic program

$$\min_{v} \quad \sum_{k=0}^{H-1} x_k^T Q x_k + u_k^T R u_k + [Cx_k - Gd_k]^T M [Cx_k - Gd_k] + v^T F v + K^T v \;\; + \;\; x_H^T \Phi x_H$$

with

$$u_{0:H-1} := \begin{bmatrix} u_0 \\ \vdots \\ u_{H-1} \end{bmatrix} = \mathcal{U} v, \qquad x_{k+1} = Ax_k + Bu_k + Ed_k \qquad k = 0, \ldots, H-1$$

subject to

$$\begin{aligned}
a_{x,box} \leq \quad & x_k & \leq b_{x,box} \quad & k = 1, \ldots, H \\
a_x \leq \quad & L_x x_k & \leq b_x \quad & k = 1, \ldots, H \\
a_{u,box} \leq \quad & u_k & \leq b_{u,box} \quad & k = 0, \ldots, H-1 \\
a_u \leq \quad & L_u u_k & \leq b_u \quad & k = 0, \ldots, H-1 \\
a_{G_u} \leq \quad & L_{G_u} \begin{bmatrix} x_{1:H} \\ u_{0:H-1} \end{bmatrix} & \leq b_{G_u} & \\
a_{G_v} \leq \quad & L_{G_v} \begin{bmatrix} x_{1:H} \\ v \end{bmatrix} & \leq b_{G_v} &
\end{aligned}$$

The columns of $\mathcal{U}$ are a basis for the control input search subspace. The signal d represents both estimated disturbances and desired trajectories. The last two constraint types are general, allowing general linear constraint on $x_1, x_2, \ldots, x_H, u_0, \ldots, u_{H-1}$ or on $x_1, x_2, \ldots, x_H, v$. It is tedious for a user to specify. The first four are special cases, but targeted towards typical operational constraints.

The formulation allows for parameter-dependent matrices. These matrices are evaluated each time the problem formulation is called.

2.4.2 Constraint Relaxation

Concisely write the quadratic optimization as

$$\min_{v} q(v) \quad \text{subject to } l_i(v) \leq 0$$

where q is the quadratic cost, and $\{l_i\}_{i=1}^{N_c}$ is the set of individual, scalar, linear constraint functions.

Associated with each constraint is an integer $k_i \geq 0$. If $k_i = 0$, then the i^{th} constraint is considered "hard," and will not be relaxed. Let $n_{\mathrm{K}} = \max_{i \leq N_c} k_i$.

Let $\rho \in \mathbf{R}^{n_K}$ be a specified weight vector with positive entries. Let $\epsilon \in \Re^{n_K}$ be a slack variable.

If the original problem is infeasible, the relaxed problem is

$$\min_{v,\epsilon} q(v) + \rho^T \epsilon \quad \text{subject to} \quad \begin{array}{ll} l_i(v) \leq \epsilon_{k_i} & \text{if } k_i \geq 0 \\ l_i(v) \leq 0 & \text{if } k_i = 0 \end{array}$$

Although this relaxed version of the quadratic problem with linear constraints has a greater opportunity to be feasible, it is not always. It is recommended that the user set up relaxation terms such that the problem is feasible when relaxed. Several choices affect this:

- number of relaxation terms
- how to apply them among the constraints
- desirable cost ratio among the elements of the cost vector ρ
- magnitude of the cost terms of the unrelaxed and relaxed formulation

Variable	Dimension/Class	Meaning
n_x	scalar `int`	State Dimension
n_u	scalar `int`	Input Dimension
n_y	scalar `int`	Output Dimension
n_d	scalar `int`	Known Signal Dimension
n_v	scalar `int`	Decision Variable Dimension
n_ϵ	scalar `int`	Relaxation Variable Dimension
H	scalar `int`	Horizon Length
x_0	$n_x \times 1$ `double`	Initial Condition
$d_{0:H-1}$	$n_d \times H$ `double`	Known Signal
A	$n_x \times n_x$ `double`	Dynamic Model
B	$n_x \times n_u$ `double`	Dynamic Model
E	$n_x \times n_d$ `double`	Dynamic Model
Q	$n_x \times n_x$ `double`	Symmetric State Step Cost
R	$n_u \times n_u$ `double`	Symmetric Input Step Cost
M	$n_y \times n_y$ `double`	Symmetric Known Signal Step Cost
C	$n_y \times n_x$ `double`	Cost-Related Matrix for State
G	$n_y \times n_d$ `double`	Cost-Related Matrix for Signal
F	$n_v \times n_v$ `double`	Quadratic Decision Variable Cost
K	$n_v \times 1$ `double`	Linear Decision Variable Cost
ρ	$n_\epsilon \times 1$ `double`	Cost-Related Matrix for Relaxation
Φ	$n_x \times n_x$ `double`	Symmetric State Terminal Cost
$\mathcal{U}$	$n_u \times n_v$ `double`	Map from Decision Variable to Input

Table 2.1: Declaration of size and meaning for dynamics and cost data

The cost and dynamics data are shown in Table 2.4.2. The constraint data is described on the next page.

$a_{x,box}$	$n_x \times 1$ `double`	Lower Bound State Box Constraint
$b_{x,box}$	$n_x \times 1$ `double`	Upper Bound State Box Constraint
$a_{u,box}$	$n_u \times 1$ `double`	Lower Bound Input Box Constraint
$b_{u,box}$	$n_u \times 1$ `double`	Upper Bound Input Box Constraint
n_{L_x}	scalar `int`	Number of Linear State Constraints
a_x	$n_{L_x} \times 1$ `double`	Lower Bound State Constraint
b_x	$n_{L_x} \times 1$ `double`	Upper Bound State Constraint
L_x	$n_{L_x} \times n_x$ `double`	State Constraint Matrix
n_{L_u}	scalar `int`	Number of Linear Input Constraints
a_u	$n_{L_u} \times 1$ `double`	Lower Bound Input Constraint
b_u	$n_{L_u} \times 1$ `double`	Upper Bound Input Constraint
L_u	$n_{L_u} \times n_u$ `double`	Input Constraint Matrix
$n_{L_{G_u}}$	scalar `int`	Number of Linear General u Constraints
a_{G_u}	$n_{L_{G_u}} \times 1$ `double`	Lower Bound General u Constraint
b_{G_u}	$n_{L_{G_u}} \times 1$ `double`	Upper Bound General u Constraint
L_{G_u}	$n_{L_{G_u}} \times (H \times (n_x + n_u))$ `double`	General u Constraint Matrix
$n_{L_{G_v}}$	scalar `int`	Number of Linear General v Constraints
a_{G_v}	$n_{L_{G_v}} \times 1$ `double`	Lower Bound General v Constraint
b_{G_v}	$n_{L_{G_v}} \times 1$ `double`	Upper Bound General v Constraint
L_{G_v}	$n_{L_{G_v}} \times (H \times n_x + n_v)$ `double`	General v Constraint Matrix

Table 2.2: Declaration of size and meaning for constraints data

The constraint data is shown in Table 2.4.2.

Chapter 3

Example Problem

To allow for a visual depiction of the performance of the RHC API, a two-state and two-input example is used. Through this example various features of the API are demonstrated. The system chosen is unstable in discrete time with a signal d in the dynamics.

$$x_{k+1} = Ax_k + Bu_k + Ed_k$$

$$x_0 = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \quad A = \begin{bmatrix} 1 & 1.1 \\ 0 & 1 \end{bmatrix} \quad B = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \quad E = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

One Matlab file that contains the example is `RHCexample.m`. It performs a comparison between running with hard constraints and running with soft constraints. The curves that goes outside of the constraints boundaries belong to the LQR results. See the results in Figure 3. Another example file is `initConds_softened.m`. A normal distribution of initial conditions is generated and tested. The results can be see in Figure 3. The setup is for relaxation on the state variable x, but hard constraints on the input variable u.

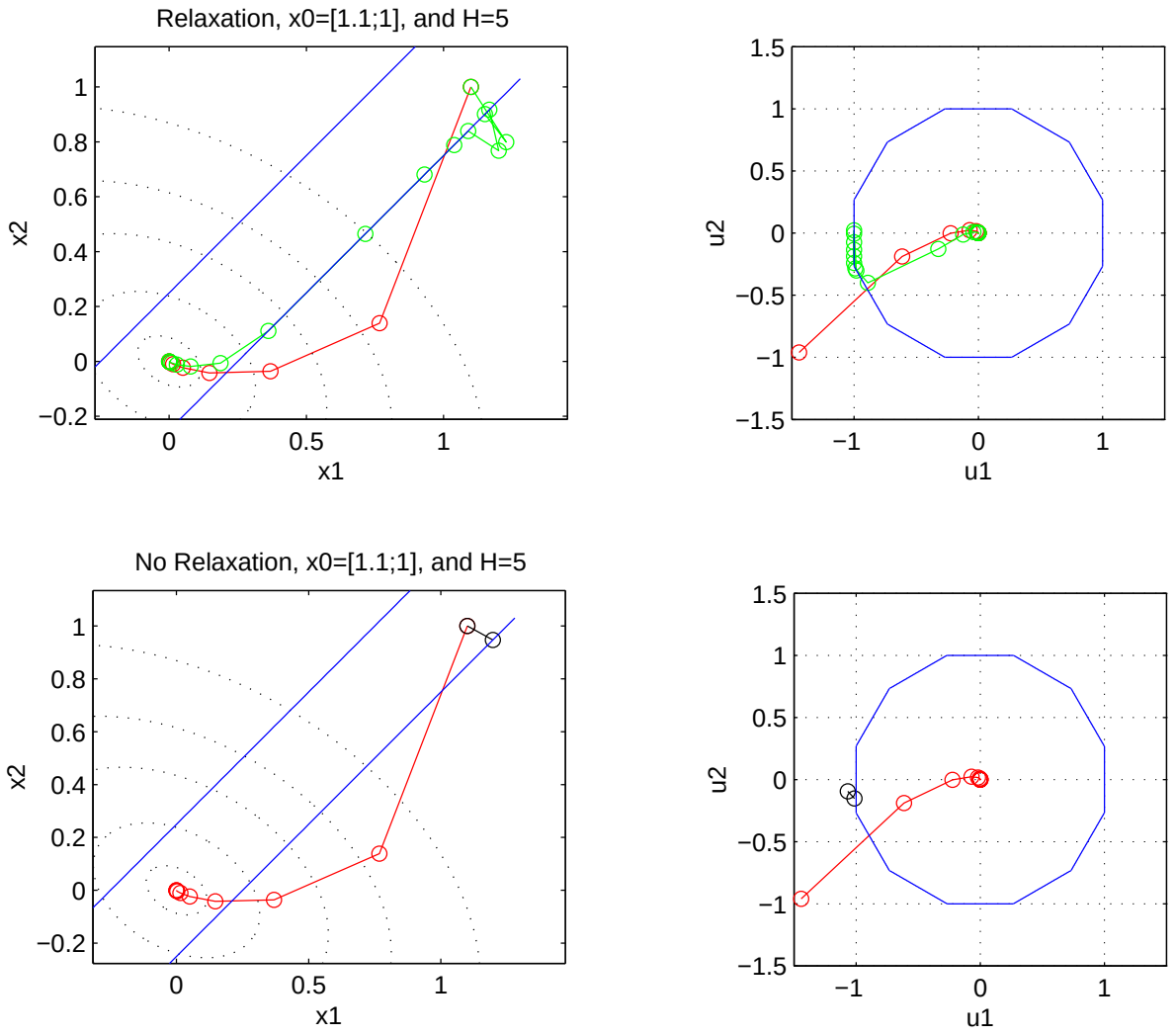

Figure 3.1: Comparison of relaxed and unrelaxed constraints

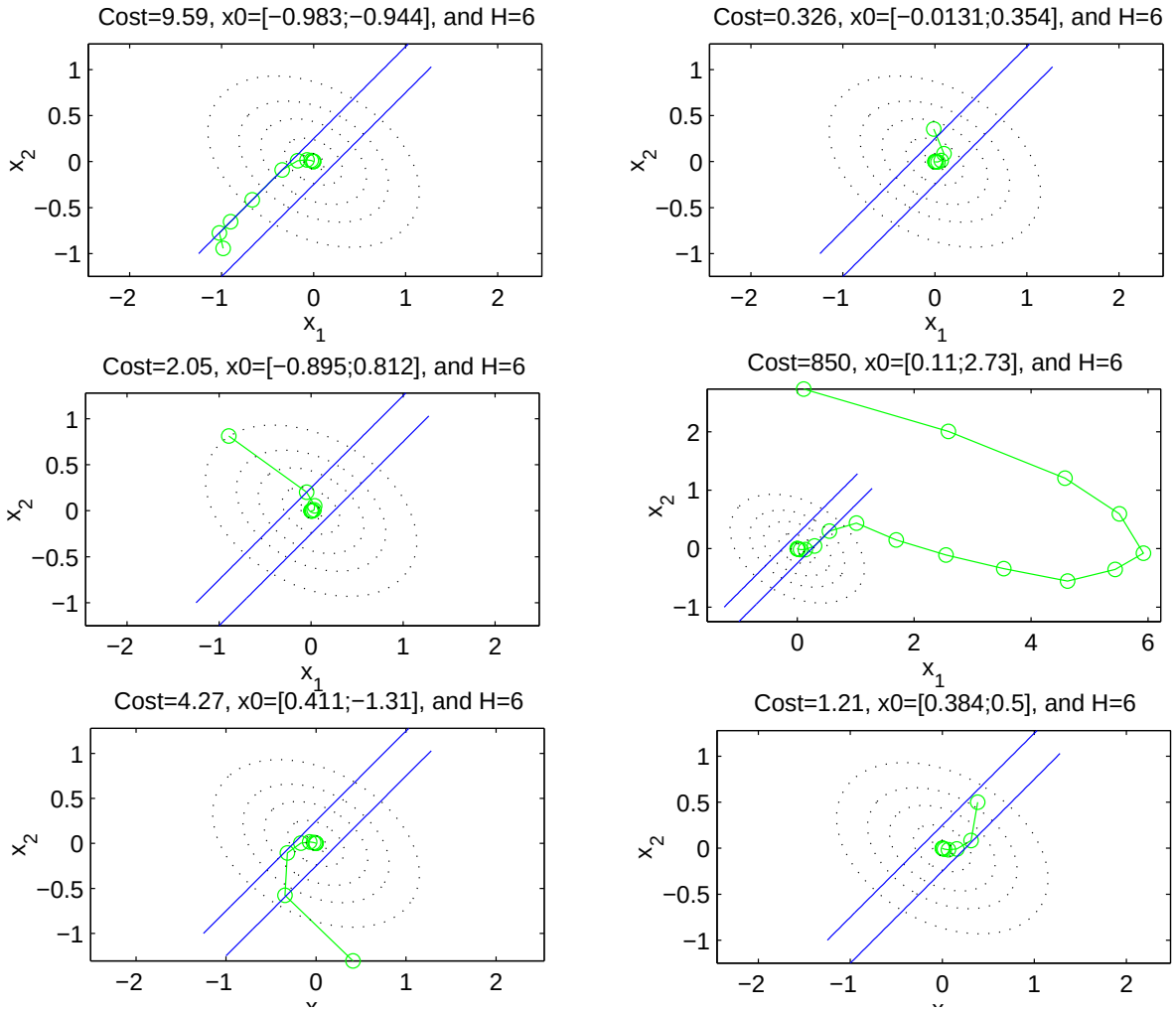

Figure 3.2: Varying initial conditions using soft constraints

Chapter 4

Code Organization

4.0.3 Formulation

The code that performs the problem formulation is in the file `QPFunctions.c`. This file contains a total of 8 functions. Only 2 are called from outside of the file: `QPformulate()` and `Reformulate()`. `QPformulate()` formulates the quadratic, linear, and scalar cost terms. It also has calls to `FillBigConst()` and `FillBoundVec()`, which respectively formulate the constraint matrix and the upper and lower bound vectors. `Reformulate()` is called only when constraint relaxation is employed. It performs the manipulations to the cost and constraint matrices necessary for relaxation.

From these primary functions the other functions are called as subroutines. `MatMultiply()` performs the matrix multiplication of any two properly sized matrices and stores the result in a third matrix. This subroutine assumes that the number of rows allocated to each matrix is the same as the number used. It is called from all of the primary functions except for `Reformulate()`. `TransMatMultiply()` performs the matrix multiplication of the transpose of the first matrix with a second matrix. This subroutine also assumes that the number of rows allocated to each matrix is the same as the number used. It is called only by `QPformulate()`.

`GenMatMultiply()` performs the matrix multiplication of any two matrices. It takes in the allocated row dimensions of the two matrices and the column dimension of the third matrix for storage, the used row dimenion of all three matrices, and pointers to the matrices. Since it takes into account the allocated dimensions, this subroutine does *not* require that the number of allocated rows be the same as the number used.

The subroutine `dosum()` is needed only in `QPformulate`. It adds the second matrix to the first, storing it in the first matrix. This is needed for the quadratic cost matrix when parts of it get added to itself.

4.0.4 Solving

The file `LDQCmdemo.c` is a C MEX script that contains the calls to formulate the problem and solve it. It is here that the LSSOL problem matrices are allocated their sizes. Also in this code, the options for running LSSOL are assigned (*i.e.*, problem type, level of output printing, and iteration limits).

If the problem is infeasible, a call to `Reformulation()` is performed, and LSSOL is re-executed. If the problem is feasible, `Reformulation()` is never called.

4.0.5 MEX File Creation

Matlab has a compiler for interacting with C code: *mex*. A C MEX script must be compiled before it can be used. In addition to `LDQCmdemo.c` and `QPFunctions.c`, other files are required that allow the LSSOL Fortran code to be utilized, even in C code (particularly for the Linux environment): `libopt.a` and `libf2c.a`. Also, a math library must be included, indicated by *-lm*. The call to create a MEX file for this program is as follows.

$$mex \;\; LDQCmdemo.c \;\; QPFunctions.c \;\; libopt.a \;\; libf2c.a \;\; -lm$$

4.0.6 Simulation Environment

The environments for performing simulations are Matlab and Simulink. The compiled MEX file is called as a normal Maltab function from `RHCexample.m` or `mdemo.m`. In these files, all the user-defined sizes and matrices are collected through function calls. The function `getABE()` collects the matrices for the dynamics; `getCostMats()` accesses the cost matrices of the quadratic program objective function as well as the matrix that maps the decision variables minus the constraint relaxation variables to the control input.

The constraints are collected through 6 separate functions, one for each constraint type. The function `getLinX` grabs the linear constraints on the state variable x. The function `getLinU` takes the linear constraints on the input u; `getBoxX` accesses the box constraints on the state x; `getLinU` takes the linear constraints on the input u; `getBoxX` accesses the box constraints on the state x; `getBoxU` collects the box constraints on the input u.

As discussed in section 2.4, the matrices are mode and parameter dependent. Therefore, each function takes in an integer that indicates mode and a pointer to an array of doubles that holds the parameter values.

The simulation may also be run in Simulink. Here the code is provided in an S-file function. The underlying formulation and optimization code remains the same. However the manner of accessing the code is closer to how it would be done in real-time.

Chapter 5

Code Testing

5.0.7 Correct Formulation

To validate the formulation of the cost and constraint matrices, it was necessary to test them. This was accomplished by peforming a formulation in Matlab and comparing the C code formulation to it. The code that does this is found in `mdemo.m`. This testing file allows either specific terms to be specified or random ones to be generated.

The test is performed over the length of the horizon, but it does not advance time frames. It is sufficient to check only one time frame since the purpose of the test is to see if the problem is correctly formulated.

5.0.8 Time for Formulation

The time performance was measured on a Pentium 4, 1.7 GHz processor with 512 Mb of RAM on Red Hat Linux 8.0. Table 5.0.8 shows the time to formulate a problem for a variety of matrix sizes.

nx	nu	nd	ny	H	L1	L2	Lgxu	Lgxv	Time
2	2	2	2	3	0	0	0	0	1.85×10^{-4}
2	2	2	2	4	0	0	0	0	2.01×10^{-4}
2	2	2	2	5	0	0	0	0	2.15×10^{-4}
2	2	2	2	7	0	0	0	0	2.64×10^{-4}
3	2	2	2	3	0	0	0	0	1.91×10^{-4}
3	2	2	2	4	0	0	0	0	2.03×10^{-4}
3	2	2	2	5	0	0	0	0	2.23×10^{-4}
3	2	2	2	7	0	0	0	0	2.72×10^{-4}
4	2	2	2	3	0	0	0	0	1.94×10^{-4}
4	2	2	2	4	0	0	0	0	2.04×10^{-4}
4	2	2	2	5	0	0	0	0	2.39×10^{-4}
4	2	2	2	7	0	0	0	0	2.87×10^{-4}
9	2	2	2	3	0	0	0	0	2.17×10^{-4}
9	3	2	2	3	0	0	0	0	2.40×10^{-4}
9	4	2	2	3	0	0	0	0	2.79×10^{-4}
9	5	2	2	3	0	0	0	0	3.20×10^{-4}
9	5	3	2	3	0	0	0	0	3.21×10^{-4}
9	5	4	2	3	0	0	0	0	3.26×10^{-4}
9	5	5	2	3	0	0	0	0	3.29×10^{-4}
9	5	10	2	3	0	0	0	0	3.39×10^{-4}
9	5	3	3	3	0	0	0	0	3.23×10^{-4}
9	5	3	4	3	0	0	0	0	3.35×10^{-4}
9	5	3	5	3	0	0	0	0	3.37×10^{-4}
9	5	3	5	20	0	0	0	0	500×10^{-4}
9	5	3	5	3	2	0	0	0	3.41×10^{-4}
9	5	3	5	3	5	0	0	0	3.56×10^{-4}
9	5	3	5	3	10	0	0	0	3.93×10^{-4}
9	5	3	5	3	10	2	0	0	4.04×10^{-4}
9	5	3	5	3	10	5	0	0	4.35×10^{-4}
9	5	3	5	3	10	10	0	0	4.74×10^{-4}
9	5	3	5	10	0	0	5	0	3.01×10^{-4}
9	5	3	5	3	0	0	10	0	3.24×10^{-4}
9	5	3	5	3	0	0	25	0	4.05×10^{-4}
9	5	3	5	3	0	0	25	5	13.9×10^{-4}
9	5	3	5	3	0	0	25	10	24.1×10^{-4}
9	5	3	5	3	0	0	25	25	55.8×10^{-4}
9	5	3	5	10	0	0	25	25	76.7×10^{-4}
9	5	3	5	20	0	0	25	25	1649×10^{-4}
9	5	3	5	10	25	25	0	0	171×10^{-4}
9	5	3	5	20	25	25	0	0	13200×10^{-4}

Table 5.1: Results on a Pentium 4, 1.7 GHz with 512 MB RAM, Red Hat Linux 8.0

Chapter 6

Choosing An Optimization Algorithm

Key to the successful integration of this project with the OCP platform is a fast convergence time for the quadratic program. The cycle of forming, testing, checking for feasibility, relaxing if necessary, and finally solving a problem had to be completed at a rate of 2 Hz.

After surveying a multitude of potential algorithms, a few were tested: NPSOL, LSSOL, and MOSEK. After performing time comparisons among them, LSSOL was found to be fastest. Although it could not take on quadratic or nonlinear constraints, linear constraints could be used to roughly approximate other constraint types. Thus, a trade off was made between precision constraint adherence and problem convergence speed.

Chapter 7

Looking Beyond the Horizon

Although the option of time-dependent matrices is under consideration, it is viewed by the pertinent parties as unnecessary. The work done on this project should lead to the further development of the OCP project, particularly as it pertains to finding the near optimal control sequence for a system.