Techniques for solving Nonlinear Programming Problems with Emphasis on Interior Point Methods and Optimal Control Problems.

Catherine Buchanan

Master of Philosophy
Department of Mathematics and Statistics
University of Edinburgh
2007

“To Almighty God: whose love kept me safe through the toughest of days.”

Declaration

Abstract

The primary focus of this work is a thorough research into the current available techniques for solving nonlinear programming problems. Emphasis is placed on interior-point methods and the connection between optimal control problems and nonlinear programming is explored.

The document contains a detailed discussion of nonlinear programming, introducing different methods used to find solutions to NLP problems and then describing a large variety of algorithms from the literature. These descriptions make use of a unified notation, highlighting key algorithmic differences between solvers. Specifically, the variations in problem formulation, chosen merit functions, ways of determining stepsize and dealing with nonconvexity are shown. Comparisons between reported results on standard test sets are made.

The work also contains an understanding of optimal control problems, beginning with an introduction to Hamiltonians, based on their background in calculus of variations and Newtonian mechanics. Several small real-life problems are taken from the literature and it is shown that they can be modelled as optimal control problems so that Hamiltonian theory and Pontryagin's maximum principle can be used to solve them. This is followed by an explanation of how Runge-Kutta discretization schemes can be used to transform optimal control problems into nonlinear programs, making the wide range of NLP solvers available for their solution.

A large focus of this work is on the interior point LP and QP solver `hopdm`. The aim has been to extend the solver so that the logic behind it can be used for solving nonlinear programming problems. The decisions which were made when converting `hopdm` into an nlp solver have been listed and explained. This includes a discussion of implementational details required for any interior point method, such as maintenance of centrality and choice of barrier parameter. `hopdm` has successfully been used as the basis for an SQP solver which is able to solve approximately 85% of the CUTE set and work has been carried out into extending it into an interior point NLP solver.

Acknowledgements

I would like to acknowledge the help of all those whose support, guidance, inspiration and correction, mixed with unquenchable faith in me, made the writing of this thesis possible. Specifically, I would like to thank my supervisors, Jacek Gondzio and Ken McKinnon; and fellow PhD students Marco Colombo and David Stern.

On a personal note, I am grateful to many friends who gave of their time and energy to encourage me in my life and work. In particular, I offer thanks to those who have provided me with safe places to live in these years: my parents, Natasha Lovik, Tim & Vanessa Allsop, Adam & Lindsey Beagles, Andy & Angela Houghton, Chris & Bridget Johnston, Jackie Keeley, Fatima Araujo and Stephen & Helen Wood.

May those whose lives have so greatly enriched mine be truly blessed.

Table of Contents

List of Notation **v**

List of Figures **viii**

List of Tables **x**

Chapter 1 Introduction **1**

Chapter 2 Methods of Solving Nonlinear Optimization Problems **4**

2.1 Penalty methods 4

2.1.1 Quadratic penalty 6

2.1.2 Log barrier penalty 7

2.1.3 Exact penalty 9

2.2 Interior point 10

2.2.1 Use of log barrier terms 10

2.2.2 First order optimality conditions 11

2.2.3 Newton's method 12

2.3 Sequential quadratic programming 14

2.4 Filter methods 14

Chapter 3 Sequential Quadratic Programming **16**

3.1 The quadratic model 16

3.1.1 Formulation 16

3.1.2 Solution 18

3.2 Optimization tools 22

3.2.1 Factorization of $\mathbb{H}\mathbb{L}$ 22

3.2.2 Linesearch methods 23

3.2.3 Trust region methods 25

3.3 hopdmSQP 26

3.3.1 Dealing with `hopdm` error codes and nondescent directions 29

3.4 CUTE 30

3.4.1 Small subset 30
3.4.2 Complete set 31
3.5 Possible further improvements to `hopdmSQP` 32

Chapter 4 Nonlinear Programming Algorithms 33
4.1 Penalty methods 33
4.2 Interior point methods 35
4.2.1 Linesearch methods 36
4.2.2 Trust region methods 39
4.2.3 Hybrid methods 41
4.2.4 Filter methods 42
4.3 Sequential quadratic programming 44
4.3.1 Solving the quadratic model 44
4.3.2 Choosing the step 46
4.4 Comparisons 46
4.4.1 Results from individual solvers 46
4.4.2 Results from comparison papers 48

Chapter 5 The Potential for `hopdm` to be a Nonlinear Interior Point Solver. 50
5.1 The problem formulation used by `hopdm` 51
5.2 `hopdmNLP` 51
5.2.1 Linear algebra 52
5.2.2 Central path and barrier parameter 53
5.2.3 Merit function 55
5.2.4 Variables 56
5.2.5 Dealing with nonconvexity 59
5.3 Summary of possible future work 60

Chapter 6 Optimal Control Problems 61
6.1 Optimal control theory 61
6.2 Calculus of variations 63
6.2.1 Euler-Lagrange equations 64
6.3 Hamiltonians 65
6.3.1 In Newtonian mechanics 65
6.3.2 In calculus of variations 66
6.3.3 In optimal control theory 67
6.4 Examples 70
6.4.1 Mountain pass 70

6.4.2 Bicycle . . . 71
6.4.3 Sailing . . . 73

Chapter 7 Optimal Control Problems as Nonlinear Programming Problems 75

7.1 Runge-Kutta discretization schemes . . . 75
7.2 `hopdmSQP` used on models of small OCPs . . . 77
7.2.1 Mountain pass . . . 77
7.2.2 Bicycle . . . 78
7.2.3 Sailing . . . 80
7.3 Sequential nonlinear programming . . . 82

Chapter 8 Further Work 84

8.1 Improving `hopdmSQP` . . . 84
8.2 Coding `hopdmNLP` . . . 85
8.3 Sequential nonlinear programming . . . 85
8.4 Small optimal control problems . . . 86

Appendix A Cute Results 88

Appendix B Ampl models 112

B.1 Mountain Pass . . . 112
B.2 Bicycle . . . 113
B.3 Sailing . . . 113
B.4 Golf . . . 114
B.4.1 1-dimensional Euler discretization . . . 114
B.4.2 2-dimensional Trapezoidal discretization . . . 115
B.4.3 3-dimensional Runge-Kutta discretization . . . 116

Appendix C Golf 121

C.1 The problem . . . 121
C.2 Problem instances solved . . . 122
C.2.1 1D line . . . 123
C.2.2 2D flat plane . . . 124
C.2.3 2D tilted line . . . 125
C.2.4 2D curve . . . 126
C.2.5 3D tilted plane . . . 128
C.2.6 3D bowl shape . . . 129
C.2.7 3D ramp . . . 130
C.3 Comments and conclusions . . . 131

Appendix D Sailing **135**

D.1 Variations of the problem 135

D.1.1 Wind fields 135

D.1.2 Water movement 136

D.1.3 Shape of the water 136

D.1.4 Boat type 139

D.2 Dynamic programming 139

D.2.1 Vanderbei's adaptation 140

D.2.2 Our adaptation 143

D.3 Numerical results 145

D.3.1 Problems solved 147

Bibliography **152**

List of Notation

The following is a list of the notation used throughout this thesis. Some symbols have two meanings, where the different usages should be apparent from context. Specifically, meanings given to symbols in chapters 1–5 may be replaced with new meanings in chapters 6 and 7.

Symbol	Meaning
a	linear constraint matrix coefficient
b	right hand side constant
c	constraint
d	direction
e	vector of 1s
f	objective
g	generic linear vector in objective
	gravitational constant
h	small vector/small valued function
	infeasibility variables
	integral stepsize
i	row index
j	column index
k	iteration counter
	order of Runge-Kutta schemes
l	lower bound
m	no. of constraints
	mass
n	no. of variables
p	a second set of slacks[1]
	number of integration steps in OCP
q	a second set of Lagrange multipliers[1]
r	constraint range
s	slack variables
t	constant for subspace determination[2]
	time
u	upper bound
	control variables

[1]used in `Loqo` [71]
[2]used in [12]

Symbol	Meaning
v	velocity
w	Lagrange multipliers for upper bounds
x	primal variables
	state variables
y	vector of state and control variables
z	Lagrange multipliers for lower bounds
$B(x,\mu)$	mixed penalty function
$B(x,\lambda)$	approximation to $\mathbb{HL}$
$H(x,u,t)$	Hamiltonian
$\mathbb{HL}(x,\lambda)$	Hessian of Lagrangian, also written as $\mathbb{HL}$
J	performace measure
$P(x,\mu)$	log barrier penalty function
Q	generic quadratic matrix
$Q(x;\mu)$	quadratic penalty function
W	weighting matrix
α	stepsize
β	combination of TR directions[3]
	constants used in Runge-Kutta schemes
γ	problem dependent scalar[4]
δ	trust region radius
	variations
ϵ	small scalar
ζ	parameter used in choice of α by [80]
η	parameter used in choice of α by [80]
θ	addition to $\mathbb{HL}$ in augmented system, caused by variable bounds
	an unknown angle
ϑ	small number used for initializing variables
κ	small constants[5]
λ	Lagrange multipliers/dual variables
	adjoint variables
μ	barrier parameter
ν	penalty parameter
ξ	right hand sides of Newton equations
ϖ	maximum point on a given mountain pass
ρ	infeasibility penalty
σ	centering parameter
ς	duality measure
τ	convergence tolerance
	endpoints of subdivision of integration steps
φ	continuously differentiable function
ω	infimum of mountain pass
Γ	constant diagonal matrix
Δ	Newton direction

[3]used in `NuOpt` [80]
[4]used in [12]
[5]used in IPOPT [77]

Symbol	Meaning
Θ	complementarity product
Λ	diagonal formed from λ[6].
Φ	merit function
Ω	set of all possible mountain passes
$\mathcal{A}$	active set
$\mathcal{E}$	set of equality constraints
$\mathcal{I}$	set of inequality constraints
$\mathcal{I}_1$	set of $\geq$ constraints
$\mathcal{I}_2$	set of $\leq$ constraints
$\|\mathcal{I}\|$ etc.	size of set of inequality constraints etc.

[6]Generally, a capital letter, Z say, represents the diagonal matrix formed from the corresponding vector, i.e. z

List of Figures

2.1 Newton's Method 5
2.2 $-\ln c_i(x)$ 8
2.3 A Filter Update 14

3.1 A Trust-Region Dogleg. 26

4.1 Showing the need for a logarithmic linesearch. 35

5.1 A typical central path 54

6.1 An example of a terminal control problem. 63
6.2 Wind polar 73

7.1 Subdivision of the integral step from t_i to t_{i+1}. 76
7.2 Contours of the six-hump camel back function. 78
7.3 An optimal mountain pass between minima of the six-hump camel back function. 79
7.4 Optimal trajectory for cyclist wishing to maximize suntan. 79
7.5 The sailing problem to be solved. 80
7.6 Two possible sailing trajectories. 81
7.7 Optimal sailing trajectory. 81

C.1 Showing the forces present on a golf ball on the green. 122
C.2 A golf green shaped as a damped sin curve. 126
C.3 Optimal golf trajectory on a 3D planar green. 133
C.4 Optimal golf trajectory on a 3D bowl shaped green. 133
C.5 Optimal golf trajectory on a 3D ramp. 134

D.1 A spatial wind field in the Apostle Islands 136
D.2 Tidal currents near the Isle of Wight in the hours before high tide. 137
D.3 Tidal currents near the Isle of Wight in the hours after high tide. 138
D.4 Showing the 7 directions a boat can sail in in Vanderbei's formulation. 140
D.5 Optimal route 1 for sailing problem found by DP. 142

D.6 Optimal route 2 for sailing problem found by DP. 143
D.7 Optimal route 3 for sailing problem found by DP. 143
D.8 Optimal route 4 for sailing problem found by DP. 144
D.9 Optimal route 5 for sailing problem found by DP. 144
D.10 Optimal route 6 for sailing problem found by DP. 145
D.11 Approximate wind polar for Boat 1. 146
D.12 Wind polar for Boat 2 147
D.13 Optimal routes for Boats 1 & 2 sailing with the wind 148
D.14 An optimal route found for Boat 1 sailing in variable wind. 148
D.15 An optimal route found for Boat 2 sailing in wind which varies discretely. 149
D.16 Continuously varying wind field 150
D.17 Optimal route for Boat 1 sailing in wind which varies continuously 151
D.18 Optimal route for Boat 2 sailing in wind which varies continuously 151

List of Tables

3.1 Changes made to `hopdmSQP` to improve its performance. 31

4.1 Comparison of key features of interior point solvers 45
4.2 Rankings of solvers compared on HS test set. 47
4.3 Rankings of solvers compared on large scale test sets. 47
4.4 Comparing `hopdmSQP` with robust solvers on large scale problems. 49

5.1 How definition of θ_i depends on type of variable x_i. 52

7.1 Different optimal sailing trajectories found by `hopdmSQP` 82

A.1 Increasing success rate as `hopdmSQP` is improved. 88
A.2 Results of running `hopdmSQP` on entire CUTE set. 90
A.3 Problems for which `hopdm` fails to find an optimal solution. 107

C.1 Solving the golf problem with a 1D green. 123
C.2 Solving the golf problem with a 2D flat green. 124
C.3 Solving the golf problem with a 2D tilted green. 125
C.4 Solving the golf problem with a 2D curved green. 127
C.5 Solving the golf problem with a 3D tilted planar green. 128
C.6 Solving the golf problem with a 3D bowl shaped green. 130
C.7 Solving the golf problem with a 3D ramp shaped green. 131

D.1 An example of wind directions at stages of a dynamic programming model of the sailing problem. 142
D.2 Data for a wind polar which approximates Vanderbei's sailing times (Boat 1). 146
D.3 Data for a wind polar which approximates that found by Hennessey *et al.* through practical experimentation (Boat 2). 147

Chapter 1

Introduction

We start this discussion with a description of optimization and, more specifically, nonlinear optimization problems.

Optimization occurs naturally in many industries. For example, airline and train companies alter their fares according to factors including how many tickets have been sold, in order to maximize their revenue; telecommunication companies determine *optimal* networks based on the twin goals of ensuring coverage between homes, cities or countries and minimizing the expense of providing and laying cables; warehouse managers determine their stock levels in order to keep the storage space required small whilst having sufficient goods in stock to be able to quickly satisfy customer demands.

To effectively find the optimal strategies for each of these problems, as well as many more, the optimizer must determine a mathematical model for the real life problem. This model will usually be written in terms of *variables* which are mathematical representations of real life objects or criteria that affect the decision making process. In the case of warehouse stock levels, variables would be chosen to represent each stored item.

An *objective* function is then needed to provide a measure of desirability for each possible arrangement of the objects/criteria represented by the variables. Also, any physical *constraints* on the variables, or on the system to be optimized should be included in the model. Constraints on variables include that they may have to be nonnegative to accurately represent countable objects such as telecommunication cables or airline tickets. Or, for an example of a constraint on the system itself, there may be an overall space limit in the warehouse restricting the amount of stock that can be kept.

Throughout this work, we will consider minimization problems as the techniques used to solve them can easily be reversed to tackle maximization problems.

Optimization problems can be classified according to the nature of the objective function and constraints. If these are both linear (each variable has constant coefficients), like this

$$\begin{array}{lccccccc}
\min & x_1 & + & 2x_2 & - & \frac{1}{2}x_3 & & \\
\text{s.t.} & & & x_2 & + & 3x_3 & \leq & 1 \\
 & x_1 & & & - & \frac{1}{2}x_3 & = & 0
\end{array}$$

then the problem is known as a *linear* programming problem (LP). If a problem is not linear then it is classified as a *nonlinear* programming problem (NLP). Nonlinear programs vary widely. The objective and constraint functions can include mathematical functions such as sin and cos, they can include logarithms and variables raised to high powers or to fractional powers. They can be differentiable or non-differentiable.

There is one type of nonlinear programming problem which is typically considered separately. In this case, the objective is a quadratic function and the constraint functions are linear, like this

$$\begin{array}{lccccccc}
\min & \frac{1}{2}x_1^2 & + & 4x_2x_3 & - & 0.3x_3 & & \\
\text{s.t.} & x_1 & & & - & \frac{1}{2}x_3 & \leq & 1 \\
 & & & 2x_2 & + & x_3 & = & -6.
\end{array}$$

Problems of this form are known as *quadratic programming* problems (QP) and there are standard methods for solving them, which will be discussed later, see section 3.1.2.

Nonlinear programs are written generally as follows:

$$\begin{array}{lll}
\min & f(x) & \\
\text{s.t.} & c_i(x) = 0 & i \in \mathcal{E} \\
 & c_i(x) \geq 0 & i \in \mathcal{I} \\
 & l_i \leq x_i \leq u_i & i \in 1..n
\end{array} \tag{1.1}$$

where f and c_i $\forall i \in \mathcal{E} \cup \mathcal{I}$ (the objective and constraint functions respectively) are functions mapping $\mathbb{R}^n$ to $\mathbb{R}$, $x \in \mathbb{R}^n$ represents the problem variables and $l, u \in \mathbb{R}^n$ represent lower and upper bounds on x. $\mathcal{E}$ is the set of equality constraints and $\mathcal{I}$ is the set of inequality constraints.

In this work, we research techniques for solving nonlinear programming problems and consider a specific set of optimization problems, describing traditional techniques for finding their solution and showing how they can be reformulated as nonlinear programming problems and thence solved. We assume that $f(x)$ and $c_i(x)$ $\forall i \in 1..m$ are continuous and second-order differentiable.

First, in Chapter 2, nonlinear programming is explained and a variety of solution techniques, including penalty methods and interior point methods are described.

In Chapter 3, it is shown how a sequence of quadratic approximations to NLP can be used to find directions between iterates which converge to a solution of the NLP. This technique is known as sequential quadratic programming (SQP). Chapter 3 includes analysis of the choices made when implementing an SQP method using the interior point LP and QP solver `hopdm`. `hopdmSQP`'s success on the CUTE [14] problem set is shown.

A discussion of published NLP algorithms is provided in Chapter 4. The details and choices made when implementing an NLP solver are wide and varied and this chapter highlights some of the main differences between algorithms, looking at specific choices and some of the reasoning behind them. The reported success of these algorithms is considered, with reference to some comparison papers and to results obtained with `hopdmSQP`.

The final chapter on NLP methods describes the beginning of work towards extending `hopdm` into an NLP interior point solver. There is discussion about some of the issues involved with choices relating to interior point methods, such as centrality and the barrier parameter and ways in which `hopdmNLP` would differ from `hopdmSQP` are considered.

The next two chapters refer to a specific type of optimization problem, known as optimal control problems (OCP). In Chapter 6, these problems are introduced, along with the traditional solution technique which uses Hamiltonian theory. Then Chapter 7 shows how OCP can be reformulated, using Runge-Kutta discretization schemes, as NLP problems which can be solved using the techniques from Chapters 2 – 5. Small, practical optimal control problems are referenced, modelled and solved.

Ideas for future work will be discussed in Chapter 8. They include completing the extension of `hopdm` to `hopdmNLP` and exploring how it could be tuned to be especially efficient for OCPs.

Chapter 2

Methods of Solving Nonlinear Optimization Problems

This chapter provides a discussion of a selection of optimization techniques for solving nonlinear programming problems. Specific algorithms which use these techniques will then be outlined in Chapter 4.

We begin by introducing Newton's method, which is a vital tool used, usually in its pure form, in all of the nonlinear programming algorithms discussed in this chapter. We then describe a variety of penalty method algorithms, commenting on some of the ill-conditioning inherent in these techniques. The theory behind interior point methods for nonlinear programming is then explained and sequential quadratic programming is briefly introduced. We finish this chapter with a description of a typical filter method.

2.1 Penalty methods

The first nonlinear optimization technique described here is the technique of converting a constrained optimization problem into an unconstrained one. This is done by appending any violation of the constraints to the objective function and removing the constraints. Using these techniques, a series of optimization problems is solved, with the penalty on constraint violation increasing with each successive problem. There are several methods of penalising the constraints, some of which are considered in the next subsections. The understanding of these methods is taken mainly from Nocedal & Wright [62].

Before considering the different penalty functions we will describe the necessary condition for a point to be a minimum of a function.

A function $f(x)$ defined on X has a minimum at a point x^* if $f(x^*) \leq f(x)\ \forall x \in X$.

Theorem 2.1. *A necessary condition for x^* to be a minimum of $f(x)$ is that $\nabla f(x) = 0$.*

Proof. The proof uses the Taylor expansion of f about $x = x^*$ given by

$$f(x^* + h) = f(x^*) + h^T \nabla f(x^*) + \frac{1}{2} h^T \nabla^2 f(x^* + \epsilon h) h.$$

Then, using the fact that $f(x^*) \leq f(x^* + h)$ if x^* is a minimum of $f(x)$, this can be rewritten as

$$f(x^* + h) - f(x^*) = h^T \nabla f(x^*) + \frac{1}{2} h^T \nabla^2 f(x^* + \epsilon h) h \geq 0$$

for all h sufficiently close to 0. Now for h small enough, we can see that the sign of the change in f is dominated by the first order term $h^T \nabla f(x^*)$. Since this can be made either positive or negative, according to an indiscriminate choice of h, $f(x^* + h) - f(x^*)$ is only non-negative for all h if $\nabla f(x^*) = 0$, as required. □

Now, a common iterative method for finding the root (the value of x for which $\nabla f(x) = 0$) of a system of nonlinear equations is Newton's method, which is shown in Figure 2.1.

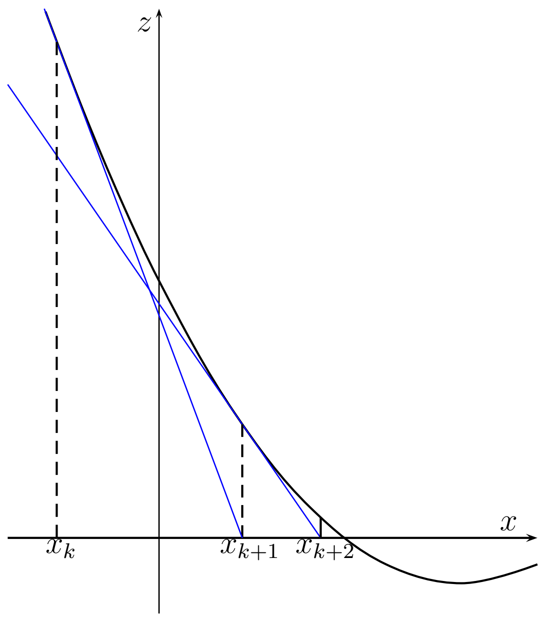


Figure 2.1: Newton's method. The blue lines are tangent lines to the function and the points where they cross the x-axis represent the next trial point.

At the current point, x_k, a tangent line

$$z = \nabla f(x_k) + \nabla^2 f(x_k)(x - x_k)$$

is a local approximation to the function $\nabla f(x)$. Extending this line until it intersects the line $z = 0$ suggests this intersection point as the next point to consider. Algebraically, the new point is given as

$$x_{k+1} = x_k - (\nabla^2 f(x_k))^{-1} \nabla f(x_k).$$

A new tangent line is then constructed about the point x_{k+1} and the process continues iteratively until two consecutive points are sufficiently close together for the algorithm to terminate and return a solution.

Newton's method can be used when trying to find the minimum of an unconstrained problem. When starting the search within a neighbourhood of the root, Newton's method demonstrates quadratic convergence (the number of digits of accuracy doubles at each step).

Before describing algorithms which use Newton's method, we define the *Lagrangian* function,

$$L(x, \lambda) = f(x) - \sum_{i \in \mathcal{E} \cup \mathcal{I}} \lambda_i c_i(x), \tag{2.1}$$

which is used to identify optimal points in constrained optimization. ($f(x)$, $c_i(x)$, $\mathcal{E}$ and $\mathcal{I}$ are defined as in (1.1) and λ_i are known as Lagrange multipliers.) In this definition, and for the remainder of this section on penalty methods, we will include bound constraints as general inequality constraints and not treat them specially.

2.1.1 Quadratic penalty

The simplest of the penalty methods is based on the *quadratic penalty function* which was first proposed by Courant [22] in 1943. The penalty terms are the squares of the constraint violations so that the constrained nonlinear problem (1.1) is now written as the unconstrained problem

$$\min Q(x; \mu) = f(x) + \frac{1}{2\mu} \sum_{i \in \mathcal{E}} c_i^2(x) + \frac{1}{2\mu} \sum_{i \in \mathcal{I}} ([c_i(x)]^-)^2,$$

where $\mu > 0$ is the penalty parameter and $[c_i(x)]^- = \max(-c_i(x), 0)$. As μ tends towards zero the constraint violation is penalised more severely.

If there are only equality constraints, then the quadratic penalty function is at least as differentiable as the original NLP problem and its smoothness makes a range of unconstrained optimization techniques available for its solution. However, if inequality constraints are present then the $[c_i(x)]^-$ terms mean that $Q(x; \mu)$ has a discontinuous second derivative at all points on the boundary of the region which is feasible for (1.1).

This lack of smoothness is one of the difficulties that arises when using the quadratic penalty function; another is caused by the need to decrease μ to zero. The minimization of $Q(x;\mu)$ becomes more difficult because the Hessian $\nabla^2_{xx}Q(x;\mu)$ is ill-conditioned when μ is small.

An algorithm which uses the quadratic penalty method is quite straightforward for equality constrained problems, that is, problems for which $\mathcal{I} = \emptyset$. Newton's method is a popular method used to find increasingly accurate approximations to $\nabla Q(x;\mu_k) = 0$, the first order necessary condition for a minimum (Theorem 2.1).

Algorithm 2.1.

```
Choose starting parameter $\mu_0$, starting tolerance $\tau_0$ and starting point $x_0^s$.
For $k = 0, 1, 2, \ldots$
    Minimize $Q(x;\mu_k)$ approximately, starting at $x_k^s$, to find $x_k$.
        Determine that $x_k$ is found when $\|\nabla Q(x;\mu_k)\| < \tau_k$.
    If final convergence test is satisfied
        STOP with solution $x_k$.
    Choose new penalty parameter $\mu_{k+1} \in (0, \mu_k)$.
    Choose new tolerance $\tau_{k+1} \in (0, \tau_k)$.
    Choose new starting point $x_{k+1}^s$.[1]
End for
```

It can be proved that if the exact minimizer of $Q(x;\mu_k)$ is found at each iteration, or if the tolerance used to find the approximate minimizer tends to zero as k tends towards infinity then any limit point of the sequence $\{x_k\}$ is a solution of the NLP problem (1.1).

2.1.2 Log barrier penalty

Logarithmic barrier methods were introduced by Frisch [33] and developed by Fiacco & McCormick [28]. The logarithmic barrier penalty function is best suited to problems which only have inequality constraints. That is, problems of the form (1.1) where $\mathcal{E} = \emptyset$.

In this method, the penalty terms are based on the natural logarithms of the constraints. They have the following properties, shown in Figure 2.2:

- They are smooth when the constraint is strictly satisfied. That is, when the value of $c_i(x)$ is strictly greater than zero.

[1]The current solution x_k can be used here.

- Their value approaches infinity as the boundary of the constraint is approached. That is, as the value of $c_i(x)$ nears zero.
- They are infinite (or undefined) when the constraint is violated. That is, when the value of $c_i(x)$ is less than zero.

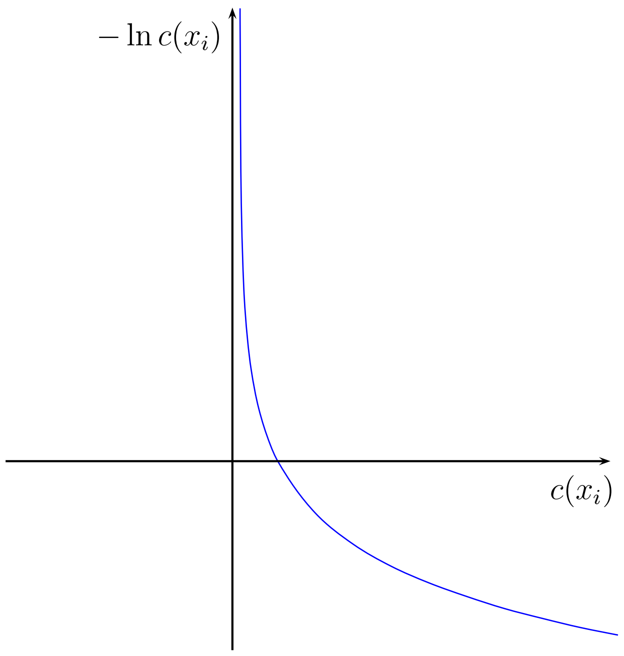


Figure 2.2: $-\ln c_i(x)$

So the NLP problem (1.1) represented as an unconstrained minimization problem using the log barrier approach is

$$\min P(x;\mu) = f(x) - \mu \sum_{i\in\mathcal{I}} \ln c_i(x), \tag{2.2}$$

where $\mu > 0$ is the penalty parameter.

The penalty on constraint violation is not explicitly increased as μ decreases, but since any point which violates a constraint always contributes a penalty of order infinity for each constraint that it violates, the increase is not strictly necessary. As μ decreases, the minimization of $P(x;\mu)$ more closely resembles minimization of the objective function. It differs in that it has sharp peaks towards infinity at constraint boundaries. These peaks constitute bad scaling, making this unconstrained minimization problem increasingly difficult to solve as the value of μ approaches zero.

However, $P(x;\mu)$ is differentiable and so, similarly to the quadratic penalty function, its minimizer can be found using a range of unconstrained optimization techniques, including Newton's method.

Algorithms that use the log barrier penalty function are very similar to algorithms using the quadratic penalty function and are similarly straightforward:

Algorithm 2.2.

```
Choose starting parameter $\mu_0$, starting tolerance $\tau_0$ and starting point $x^s_0$.
For $k = 0, 1, 2, \ldots$
    Minimize $P(x; \mu_k)$ approximately, starting at $x^s_k$, to find $x_k$.
        determine that $x_k$ is found when $\|\nabla P(x; \mu_k)\| < \tau_k$.
    If final convergence test is satisfied
        STOP with solution $x_k$.
    Choose new penalty parameter $\mu_{k+1} \in (0, \mu_k)$.
    Choose new tolerance $\tau_{k+1} \in (0, \tau_k)$.
    Choose new starting point $x^s_{k+1}$.[2]
End for
```

Under certain conditions it can be proved that any sequence of approximate minimizers of $P(x;\mu)$ converges to a minimizer of the inequality constrained NLP problem to be solved, as μ tends towards zero.

The simplest way to extend log barrier penalty functions to handle equality constraints is to use a penalty function which appends quadratic penalty terms for violation of the equality constraints. A combined penalty function of this kind has the form:

$$B(x; \mu) = f(x) - \mu \sum_{i \in \mathcal{I}} \ln c_i(x) + \frac{1}{2\mu} \sum_{i \in \mathcal{E}} c_i^2(x), \tag{2.3}$$

where $\mu > 0$ is again the penalty parameter.

Algorithms which use $B(x;\mu)$ are formed in the same way as algorithms which use each of $Q(x;\mu)$ and $P(x;\mu)$. That is, alternating reduction of the penalty parameter with using Newton's method to find an approximate minimizer to the current penalty function.

Combined penalty functions of this type are differentiable and so can be solved using a range of techniques for unconstrained optimization. They still suffer from ill-conditioning and poor scaling as μ tends towards zero.

It is possible to prove that any sequence of approximate minimizers of $B(x;\mu)$ converges to a minimizer of the NLP problem as μ tends to zero.

2.1.3 Exact penalty

To find the solution to (1.1) using either the quadratic penalty method or the log barrier penalty method requires the solution of a sequence of unconstrained

[2]A good point can be chosen by extrapolating along the path $x_0, x_1, x_2, \ldots$

minimization problems. There also exists a class of penalty functions, known as *exact* penalty functions, for which only a single minimization is required.

One of these is the l_1 penalty function

$$\Phi_1(x;\mu) = f(x) + \frac{1}{\mu}\sum_{i\in\mathcal{E}}|c_i(x)| + \frac{1}{\mu}\sum_{i\in\mathcal{I}}[c_i(x)]^- \tag{2.4}$$

which is not differentiable, so cannot be solved using algorithms which use the necessary condition of Theorem 2.1.

Another is the Augmented Lagrangian function, an extension of the quadratic penalty function, based on the Lagrangian (2.1), which reduces the need to decrease μ to zero and so does not suffer from problems caused by ill-conditioning of the Hessian.

We will return to these penalty functions and their uses later.

2.2 Interior point

Next, we will consider interior point methods and their use in solving nonlinear programming problems. They have most often been used in constrained linear and quadratic programming, but research has been carried out into extending their success in these areas into more general nonlinear programming. Some of this work will be discussed in Chapter 4.

Interior point methods were initially recognised as a successful technique for solving optimization problems following the publication of a paper by Karmarkar [51] in 1984. Much of the understanding used in this discussion is taken from Wright [79] and some of the details come from lectures given by Gondzio [38].

2.2.1 Use of log barrier terms

A significant feature of interior point methods is that an optimal solution is approached from the interior of the feasible region, but is never reached exactly. This feature is caused by the replacement of inequality constraints and variable bounds by log barrier terms in the objective function. These terms share with the log barrier penalty function the properties which are listed above and shown in Figure 2.2.

The NLP problem (1.1) can be rewritten as

$$\begin{array}{lll} \min & f(x) & \\ \text{s.t.} & c_i(x) = 0 & i \in 1,2,\ldots,m \\ & l_i \le x_i \le u_i & i \in 1,2,\ldots,n, \end{array} \tag{2.5}$$

where slack variables are used to change inequalities into equalities. The slack variables are included in the model as non-negative variables which are appended to x. The constraints, $c_i(x)$, are altered accordingly $\forall i \in \mathcal{I}$, $m = |\mathcal{E}| + |\mathcal{I}|$ and other notation is the same as in (1.1).

Then the inequalities $x_i \geq l_i$, $u_i \geq x_i$ can be replaced by log barrier penalty terms, giving the *barrier problem*:

$$\begin{array}{ll} \min & f(x) - \mu \sum_{i=1}^{n} \ln(x_i - l_i) - \mu \sum_{i=1}^{n} \ln(u_i - x_i) \\ \text{s.t.} & c_i(x) \quad = \quad 0 \qquad i \in 1, 2, \ldots, m. \end{array}$$

The penalty parameter μ is reduced at each iteration according to rules which vary between different interior point algorithms. As it tends towards 0, the objective function to be minimized more closely represents the objective of the NLP problem.

2.2.2 First order optimality conditions

The next step in solving an NLP problem using an interior point method is to determine the *first order optimality* conditions. This is done using the Lagrangian function defined above (2.1).

$$L(x, \lambda, z; \mu) = f(x) - \sum_{i=1}^{m} \lambda_i c_i(x) - \mu \sum_{i=1}^{n} \ln(x_i - l_i) - \mu \sum_{i=1}^{n} \ln(u_i - x_i).$$

Conditions for a minimum are then determined by differentiating the Lagrangian function with respect to each of its variables and claiming, as in Theorem 2.1, that each of these differentials must be zero at a stationary point. It is possible to prove that these are first order necessary conditions.

$$\begin{aligned} \nabla_x L(x, \lambda, z; \mu) &= \nabla f(x) - \sum_{i=1}^{m} \lambda_i \nabla c_i(x) - \sum_{i=1}^{n} \frac{\mu}{x_i - l_i} + \sum_{i=1}^{n} \frac{\mu}{u_i - x_i} = 0 \\ \nabla_{\lambda_i} L(x, \lambda, z; \mu) &= c_i(x) \quad = \quad 0 \qquad i \in 1, 2. \ldots, m. \end{aligned}$$

To make the notation clearer, we will denote the constraints as column vectors:

$$c(x) \in \mathbb{R}^m = \begin{bmatrix} c_1(x) \\ c_2(x) \\ \vdots \\ c_m(x) \end{bmatrix} \quad \text{and } \nabla c(x) \in \mathbb{R}^{m \times n} = \begin{bmatrix} \nabla c_1^T(x) \\ \nabla c_2^T(x) \\ \vdots \\ \nabla c_m^T(x) \end{bmatrix}.$$

Then, setting

$$\frac{\mu}{x_i - l_i} = z_i, \ \frac{\mu}{u_i - x_i} = w_i \qquad \forall i \in 1, 2, \ldots, n$$

and reversing the sign of the first equation, the first order optimality conditions can be rewritten as follows:

$$\begin{aligned} -\nabla f(x) + \nabla c^T(x)\lambda + z - w &= 0 \\ c(x) &= 0 \\ (X–L)Ze &= \mu e \\ (U–X)We &= \mu e, \end{aligned} \tag{2.6}$$

where $z, w \in \mathbb{R}^n$ are Lagrange multipliers associated with bound constraints, capital letters (i.e. W, Z, L, U) represent diagonal matrices formed from the vectors (w, z, l, u) and e is a vector of 1s of appropriate dimension. $(u_i - x_i, x_i - l_i, z_i, w_i \geq 0 \ \forall i \in 1, 2, \ldots, n.)$

2.2.3 Newton's method

The first order conditions for nonlinear programming given by (2.6) can be solved using Newton's method (Figure 2.1), since they form a large system of nonlinear equations:

$$F(x, \lambda, z, w; \mu) = 0$$

where

$$F(x, \lambda, z, w; \mu) = \begin{bmatrix} -\nabla f(x) + \nabla c^T(x)\lambda + z - w \\ c(x) \\ (X–L)Z - \mu e \\ (U–X)W - \mu e \end{bmatrix}.$$

For a given point, (x, λ, z, w), the *Newton direction* is found by solving the system of linear equations

$$\nabla F(x, \lambda, z, w; \mu) \begin{bmatrix} x_{k+1} & - & x_k \\ \lambda_{k+1} & - & \lambda_k \\ z_{k+1} & - & z_k \\ w_{k+1} & - & w_k \end{bmatrix} = -F(x, \lambda, z, w; \mu),$$

where

$$\nabla F(x, \lambda, z, w; \mu) = \begin{bmatrix} -H\!\!L(x, \lambda) & \nabla c^T(x) & I & -I \\ \nabla c(x) & 0 & 0 & 0 \\ Z & 0 & X–L & 0 \\ -W & 0 & 0 & U–X \end{bmatrix}$$

with $H\!\!L(x, \lambda) = \nabla^2 f(x) - \sum_{i=1}^{m} \lambda_i \nabla^2 c_i(x)$.

This can also be written as

$$\begin{bmatrix} -H\!\!L(x, \lambda) & \nabla c^T(x) & I & -I \\ \nabla c(x) & 0 & 0 & 0 \\ Z & 0 & X–L & 0 \\ -W & 0 & 0 & U–X \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta \lambda \\ \Delta z \\ \Delta w \end{bmatrix} = \begin{bmatrix} \xi_c \\ \xi_b \\ \xi_z \\ \xi_w \end{bmatrix}, \tag{2.7}$$

where $\xi_c = \nabla f(x) - \nabla c^T(x)\lambda - z + w$, $\xi_b = -c(x)$, $\xi_z = \mu e - (X-L)Ze$ and $\xi_w = \mu e - (U-X)We$.

Finding the Newton direction is one step in a class of interior point algorithms for NLP which take the following form:

Algorithm 2.3.

```
Choose starting point $(x_0, \lambda_0, z_0, w_0)$ such that $l - x_0 > 0$, $z_0 > 0$, $u - x_0 > 0$, $w_0 > 0$.
Calculate merit[3] of starting point.
For $k = 0, 1, 2, \ldots$
    If final convergence test is satisfied
        STOP with solution $x_k$.
    Compute $H\!L(x_k, \lambda_k)$, $\nabla f(x_k)$, $c(x_k)$ and $\nabla c(x_k)$.
    Calculate average of complementarity pairs
        $$\varsigma_k = \frac{1}{2n}\sum_{i=1}^{n}\left[(x_{i_k} - l_i)z_{i_k} + (u_i - x_{ik})w_{i_k}\right]. \qquad (2.8)$$
    Choose $\mu = \sigma_k \varsigma_k$, where $\sigma_k \in (0,1)$.
    Determine Newton direction $\begin{bmatrix} \Delta x_k \\ \Delta \lambda_k \\ \Delta z_k \\ \Delta w_k \end{bmatrix}$ using (2.7).
    Choose a step $\alpha \in [0,1]$ to be taken in the Newton direction[4], ensuring that:
        $$\begin{array}{rcrcl} x_k - l & + & \alpha\Delta x_k & > & 0 \\ z_k & + & \alpha\Delta z_k & > & 0 \\ u - x_k & - & \alpha\Delta x_k & > & 0 \\ w_k & + & \alpha\Delta w_k & > & 0. \end{array}$$
    Make step:
        $$\begin{array}{rcl} x_{k+1} & = & x_k + \alpha\Delta x_k \\ \lambda_{k+1} & = & \lambda_k + \alpha\Delta\lambda_k \\ z_{k+1} & = & z_k + \alpha\Delta z_k \\ w_{k+1} & = & w_k + \alpha\Delta w_k. \end{array}$$
End for
```

In this algorithm, if a stepsize of 1 can be taken then all primal and dual infeasibility is removed.

Determining how to control the parameters σ_k, μ and α so that they guide the sequence of iterates towards a minimum is an important part of any interior point algorithm. Discussion of how these parameters are chosen, along with other details which must be considered when implementing an interior point algorithm, is left to Chapter 5.

[3]Ways to measure merit are discussed in section 3.2.2.1.
[4]Other criteria for choosing α (using merit) will be discussed in section 3.2.2.3.

2.3 Sequential quadratic programming

In this chapter, we will only briefly mention the technique known as *Sequential Quadratic Programming* (SQP). It consists of simplifying the NLP (1.1) by using a quadratic approximation and then using the solution to the quadratic model to make a step towards a new point where another quadratic model is formed. A sequence of quadratic models are solved, giving the name of the method. A very good discussion on the issues arising in SQP is given by Boggs and Tolle [13].

The choices which need to be made when implementing an SQP method will be discussed in greater detail in Chapter 3.

2.4 Filter methods

There is also a set of algorithms, known as filter methods, which were originally devised by Fletcher & Leyffer [29].

Basically, a *filter* is chosen and a new point is accepted if it passes through the filter. The *filter* comprises points which can be represented in two dimensions with one axis representing the value of the objective function and the other representing the violation of the constraints. An acceptable point is one which improves in either direction, either lowering the value of the objective function or reducing the violation of the constraints.

When a point is accepted, it is added to the filter and any points in the filter which are *dominated* by the new point are removed. A point is said to be dominated by a new point if it has both a higher objective value and a greater violation of the constraints than the new point. This is illustrated in Figure 2.3.

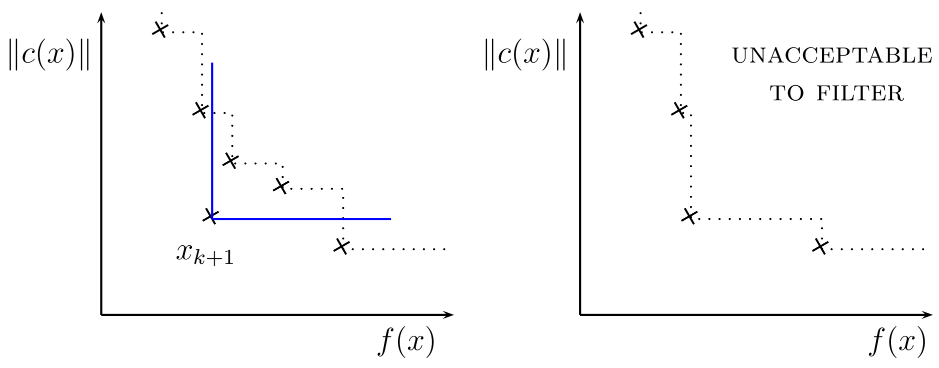


Figure 2.3: The diagram on the left shows the point x_{k+1} dominating two of the points in the original filter. These points are removed, leaving a larger region of unacceptable points. The diagram on the right shows the modified filter.

At no stage should the filter contain any point which is dominated by any other.

If a problem is feasible and a minimum exists then the filter method converges to an optimal point, where there is no constraint violation and the objective function cannot be improved further.

Chapter 3

Sequential Quadratic Programming

Sequential quadratic programming (SQP) is first mentioned in section 2.3 as a method for solving NLP problems of the form (1.1). Details of SQP techniques and of the choices which must be made when implementing an SQP method have been left to this chapter. We will first discuss the structure of general SQP methods and the broader choices to be made for any SQP algorithm and then move on to discuss specific choices which were made when implementing an SQP method based on the LP and QP solver `hopdm`, [39], of Gondzio. We will also describe results obtained when testing this solver on the *Constrained and Unconstrained Testing Environment* set (CUTE [14]) of test problems for linear and nonlinear optimization.

For the purpose of this work we will restrict our attention to those NLP problems which have objective and constraint functions which are at least twice continuously differentiable.

Much of the understanding of SQP which is used here is taken from the report by Boggs and Tolle [13] and the book of Nocedal and Wright [62], which also provides information about methods used to solve QP problems.

3.1 The quadratic model

3.1.1 Formulation

Any SQP method involves the solution of a sequence of quadratic approximations to the nonlinear program, hence the name. Therefore, one of the first and key steps in the implementation of an SQP method is to determine a way to form the quadratic model at each point in the sequence. The quadratic model must be of the form

$$\begin{array}{lll} \min & \frac{1}{2}d^TQd + g^Td & \\ \text{s.t.} & a_i^Td = b_i & i \in \mathcal{E} \\ & a_i^Td \geq b_i & i \in \mathcal{I} \\ & l_i \leq d_i \leq u_i & i \in 1, 2, \ldots, n, \end{array} \tag{3.1}$$

where Q is a symmetric $n \times n$ matrix, $g, a_i\ i \in \mathcal{E} \cup \mathcal{I}$ are vectors in $\mathbb{R}^n$ and $b_i\ i \in \mathcal{E} \cup \mathcal{I}$ are in $\mathbb{R}$. $d \in \mathbb{R}^n$ represents the problem variables, which, in the case of SQP, are a direction in which a step will be taken to move to the next point in the sequence. Throughout this chapter we will use the notation d to represent QP problem variables. We will use x to represent the points in $\mathbb{R}^n$ which form the sequence.

At each sequence point, x, the most obvious choice for approximating (1.1) by a model of form (3.1), which has a quadratic objective function and linear constraints, is to take a quadratic approximation to the NLP objective and a linear approximation to the NLP constraints as follows:

$$\begin{array}{lll} \min & \frac{1}{2}d^T\nabla^2 f(x)d + \nabla f^T(x)d & \\ \text{s.t.} & \nabla c_i^T(x)d + c_i(x) = 0 & i \in \mathcal{E} \\ & \nabla c_i^T(x)d + c_i(x) \geq 0 & i \in \mathcal{I} \\ & l_i^d \leq d_i \leq u_i^d & i \in 1, 2, \ldots, n. \end{array}$$

Upper and lower bounds on the direction are calculated simply from the current point and the variable bound:

$$\begin{array}{rcl} l_i^d & = & l_i - x_i \\ u_i^d & = & u_i - x_i. \end{array}$$

This approximation, however, fails to take into account any nonlinearity in the constraint functions and so easily breaks down on problems with nonlinear constraints.

In order for nonlinearities in the constraints to be considered, they must be included in the second order terms in the objective function. Instead of taking the quadratic model objective to be an approximation to the NLP objective, it is partly made an approximation to the Lagrangian (2.1).

The first order terms in the objective approximation remain the same and the second order terms are an approximation to the Hessian of the Lagrangian ($H\!L(x, \lambda) = \nabla^2 L(x, \lambda)$, written from here on as $H\!L$). The quadratic model becomes:

$$\begin{aligned} \min \quad & \tfrac{1}{2}d^T B(x,\lambda)d \ + \nabla f^T(x)d & & \\ \text{s.t.} \quad & \nabla c_i^T(x)d + c_i(x) = 0 & i \in \mathcal{E} & \\ & \nabla c_i^T(x)d + c_i(x) \geq 0 & i \in \mathcal{I} & \\ & l_i^d \leq d_i \leq u_i^d & i \in 1, 2, \ldots, n, & \end{aligned} \tag{3.2}$$

where $B(x,\lambda)$ is either $\mathbb{H}L$ or an approximation to it. Methods of choosing $B(x,\lambda)$ are considered in section 3.2.1.

3.1.2 Solution

Before considering techniques for solving the QP approximations (3.2), we will state the first order necessary conditions for a point x^* to be an optimal solution of a general NLP problem (1.1). As a QP problem is a specific type of NLP problem, these conditions also hold for direction vectors d^* which are optimal solutions of general QP problems (3.1).

Theorem 3.1. *Suppose that x^* is a local solution of* (1.1) *and that the linearly independent constraint qualification*[1] *(LICQ) holds at x^*. Then there is a Lagrange multiplier vector λ^*, with components λ_i^*, $i \in \mathcal{E} \cup \mathcal{I}$, such that the following conditions are satisfied at (x^*, λ^*)*

$$\nabla_x L(x^*, \lambda^*) = 0, \tag{3.3a}$$

$$c_i(x^*) = 0, \quad i \in \mathcal{E} \tag{3.3b}$$

$$c_i(x^*) \geq 0, \quad i \in \mathcal{I} \tag{3.3c}$$

$$\lambda_i^* \geq 0, \quad i \in \mathcal{I} \tag{3.3d}$$

$$\lambda_i^* c_i(x^*) = 0, \quad i \in \mathcal{E} \cup \mathcal{I}. \tag{3.3e}$$

Equations (3.3) are commonly known as the *Karush-Kuhn-Tucker*, or KKT, conditions. In this formulation, we have again included variable bound constraints as general inequality constraints. An extension of the KKT conditions to include bound constraints explicitly is straightforward.

Once the quadratic model (3.2) has been formulated, it has to be solved to find the direction variables d. There are several techniques for solving QP problems. The focus of the latter part of this chapter is on an SQP method based on the interior point solver `hopdm` [39], but we will briefly mention an alternative technique for solving QP problems before showing how interior point methods

[1]Linearly Independent Constraint Qualification - states that the constraint gradients of all constraints which are *active* (defined later) at this optimal solution are independent.

are used in QP and how they are similar to interior point methods for NLP as discussed in section 2.2.

3.1.2.1 Active set

Now, the alternative technique to interior point methods that we will mention here is *active set* methods.

At any given point, x, within a system with constraints, a subset of the constraints is satisfied at equality and is known as the *active set* ($\mathcal{A}$). This set includes all of the equality constraints and may contain some (or all) of the inequality constraints.

$$\mathcal{A} = \{i \in \mathcal{E} \cup \mathcal{I} : c_i(x) = 0\} \tag{3.4}$$

Considering only the active set reduces the KKT conditions (3.3), removing (3.3c) and (3.3d), as they refer to inequality constraints, and removing (3.3e), as $\lambda_i^* c_i^*(x) = 0 \ \forall i \in \mathcal{A}$ at a feasible point because of (3.4). This leaves a system of equations

$$\begin{bmatrix} B(x,\lambda) & -\nabla c_{\mathcal{A}}^T(x) \\ \nabla c_{\mathcal{A}}(x) & 0 \end{bmatrix} \begin{bmatrix} d^* \\ \lambda^* \end{bmatrix} = \begin{bmatrix} -\nabla f^T(x) \\ -c_{\mathcal{A}} \end{bmatrix}, \tag{3.5}$$

where $B(x,\lambda), x, d, \lambda$ and f are as in (3.2) and $c_{\mathcal{A}}(x) \in \mathbb{R}^{|\mathcal{A}|} = \begin{bmatrix} c_1(x) \\ \vdots \\ c_{|\mathcal{A}|}(x) \end{bmatrix}$,

$\nabla c_{\mathcal{A}}(x) \in \mathbb{R}^{|\mathcal{A}| \times n} = \begin{bmatrix} \nabla c_1^T(x) \\ \vdots \\ \nabla c_{|\mathcal{A}|}^T(x) \end{bmatrix}$ represent the constraints in $\mathcal{A}$. The system (3.5) can then be rearranged and solved directly.

If it were possible to identify the active set from the problem statement, then this method of solving QP problems would require the solution of only one set of linear equations. Instead, an initial guess of $\mathcal{A}$ is made. Every time (3.5) is solved $\mathcal{A}$ is updated using a descent method which is designed to guide the algorithm to the optimal active set $\mathcal{A}^*$. Active set algorithms are constructed to ensure that the same active set is never considered twice, so that, as there are only a finite number of possible combinations of constraints which can form active sets, the algorithm is guaranteed to terminate at the solution.

Problems can arise when the constraint gradients are not linearly independent at the point currently being considered. If constraint gradients are linearly dependent then the claim that the same active set is never repeated is no longer valid and it is possible for the algorithm to cycle, never terminating at an optimal point.

3.1.2.2 Interior point

An interior point algorithm applied directly to the NLP problem is described in section 2.2. An interior point algorithm used to solve each QP in a sequence of approximations to the NLP problem has many similarities.

The first order optimality conditions for the QP model are derived in the same way as the first order optimality conditions for the NLP problem (2.6):

- First, subtract a slack term from each of the inequality constraints in (3.2) to convert them into equalities, and alter $c_i(x)$ accordingly $\forall i \in \mathcal{I}$:

$$\begin{array}{lcccccl} \min & \frac{1}{2}d^T B(x,\lambda)d & + & \nabla f^T(x)d & & & \\ \text{s.t.} & \nabla c_i^T(x)d & + & c_i(x) & = & 0 & i \in 1,2,\ldots,m \\ & l_i^d \;\le\; d_i \;\le\; u_i^d & & & & & i \in 1,2,\ldots,n. \end{array}$$

  The slack variables are included in the model as non-negative variables which are appended to d.

- Then replace the bound constraints on d_i with barrier terms in the objective:

$$\begin{array}{ll} \min & \frac{1}{2}d^T B(x,\lambda)d + \nabla f^T(x)d - \mu\sum_{i=1}^{n}\ln(d_i - l_i^d) - \mu\sum_{i=1}^{n}\ln(u_i^d - d_i) \\ \text{s.t.} & \nabla c_i^T(x)d \;+\; c_i(x) \;=\; 0 \qquad\qquad i \in 1,2,\ldots,m. \end{array}$$

- Find the Lagrangian for this problem:

$$\begin{aligned} L(d,\lambda;x,\mu) =& \frac{1}{2}d^T B(x,\lambda)d + \nabla f^T(x)d - \sum_{i=1}^{m}\lambda_i(\nabla c_i^T(x)d + c_i(x)) \\ & - \mu\sum_{i=1}^{n}\ln(d_i - l_i^d) - \mu\sum_{i=1}^{n}\ln(u_i^d - d_i). \end{aligned}$$

- Differentiate $L(d,\lambda;x,\mu)$ with respect to each of d and λ, setting the derivatives equal to zero to get the first order optimality conditions for the QP approximations:

$$\begin{aligned} \nabla_d L(d,\lambda_{\mathcal{E}},\lambda_{\mathcal{I}},z;x,\mu) &= B(x,\lambda)d + \nabla f(x) - \sum_{i=1}^{m}\lambda_i\nabla c_i(x) \\ &\quad - \sum_{i=1}^{n}\frac{\mu}{d_i - l_i^d} + \sum_{i=1}^{n}\frac{\mu}{u_i^d - d_i} = 0 \\ \nabla_{\lambda_{\mathcal{E}}} L(d,\lambda_{\mathcal{E}},\lambda_{\mathcal{I}},z;x,\mu) &= c_i(x) = 0 \qquad i \in \mathcal{E} \end{aligned}$$

- Then, setting

$$\frac{\mu}{d_i - l_i^d} = z_i, \;\; \frac{\mu}{u_i^d - d_i} = w_i \quad \forall i \in 1,2,\ldots,n$$

and reversing the sign of the first equation, we get a set of first order optimality conditions for QP approximation problems which are analogous to (2.6) for NLP problems.

$$\begin{aligned} -B(x,\lambda)d - \nabla f(x) + \nabla c^T(x)\lambda + z - w &= 0 && \text{(3.6a)}\\ \nabla c(x)^T d + c(x) &= 0 && \text{(3.6b)}\\ (D\text{–}L^d)Ze &= \mu e && \text{(3.6c)}\\ (U^d\text{–}D)We &= \mu e && \text{(3.6d)} \end{aligned}$$

$$(d_i - l_i, z, u_i - d_i, w_i \geq 0 \ \forall i \in 1, 2, \dots, n.)$$

An interior point method using (3.6) is similar to Algorithm 2.3 for solving (2.6). The first order optimality conditions (3.6) are a system of linear equations which can be solved using Newton's method

$$\nabla F(d,\lambda,z,w;x,\mu) \begin{bmatrix} d_{k+1} & - & d_k \\ \lambda_{k+1} & - & \lambda_k \\ z_{k+1} & - & z_k \\ w_{k+1} & - & w_k \end{bmatrix} = -F(d,\lambda,z,w;x,\mu) \qquad (3.7)$$

with

$$F(d,\lambda,z,w;x,\mu) = \begin{bmatrix} -B(x,\lambda) - \nabla f(x) + \nabla c^T(x)\lambda + z - w \\ \nabla c^T(x)d + c(x) \\ (D\text{–}L^d)Ze - \mu e \\ (U^d\text{–}D)We - \mu e \end{bmatrix}$$

and

$$\nabla F(d,\lambda,z,w;x,\mu) = \begin{bmatrix} -B(x,\lambda) & \nabla c^T(x) & I & -I \\ \nabla c(x) & 0 & 0 & 0 \\ Z & 0 & D\text{–}L^d & 0 \\ -W & 0 & 0 & U^d\text{–}D \end{bmatrix}$$

to give an optimal direction of improvement (the Newton direction):

$$\begin{bmatrix} d_{k+1} & - & d_k \\ \lambda_{k+1} & - & \lambda_k \\ z_{k+1} & - & z_k \\ w_{k+1} & - & w_k \end{bmatrix}.$$

A step is taken in this direction, ensuring that the new values of z, w, $d - l^d$ and $u^d - d$ are greater than zero.

A series of Newton systems are solved, with μ being updated after each solution of (3.6). Methods of updating μ are left to Chapter 5.

It is interesting here to compare the KKT conditions (3.3) with the first order optimality conditions derived for use in equality constrained interior point algorithms (3.6):

- (3.3a) and (3.6a) are equivalent.
- (3.3b) and (3.6b) are also equivalent.
- (3.3c) and (3.3d) are only included for bound constraints and the Lagrange multipliers associated with them, which are kept nonnegative by interior point logic.
- (3.6c) and (3.6d) represent the same concept, and differ from (3.3e) only because of the barrier term μ which is added in the interior point method. As the interior point algorithm progresses, μ invariably $\rightarrow 0$ and so (3.6c), (3.6d) $\rightarrow$ (3.3e).

It can be seen that an interior point method searches for a solution where the KKT conditions are satisfied.

3.2 Optimization tools

Determining which method to use to solve the QP approximations in an SQP algorithm is just one of many choices which need to be made. Other decisions made when implementing our own SQP algorithm are detailed in section 3.3. Before this discussion, we consider some optimization tools which will be referred to.

3.2.1 Factorization of $\mathbb{HL}$

A presentation of the linear algebra inside `hopdm` is generally beyond the scope of this work. (See [2] for more details.) However, it is relevant to note that in order to solve the system of equations (3.7), `hopdm` uses Cholesky factorization.

Cholesky factorization is used to solve a system of equations of the form $Bx = b$ by finding the lower triangular matrix L such that $B = LL^T$. This is only possible of the matrix B is symmetric. If B is also positive definite then this factorization can be extended to $B = LDL^T$, where D is a diagonal matrix with positive entries.

If the matrix $\mathbb{HL}$ represents a nonconvex problem then it is impossible to find matrices L and D such that the diagonal elements of D are all positive. This would imply that the stationary point of the QP model is not a minimum and that the Newton direction found as the solution of (3.7) is not a descent direction.

This potential difficulty can be overcome by replacing $\mathbb{HL}$ with a positive definite approximation. This can be calculated using Quasi-Newton approaches such as the BFGS and DFP methods (see Nocedal & Wright [62] or Fletcher [30] for further details) or by adding a positive multiple of the identity (ΓI) to $I\!H\!L$ so that all the elements in the diagonal matrix D of the LDL^T factorization of $I\!H\!L + \Gamma I$ are positive.

3.2.2 Linesearch methods

There are two fundamental strategies for moving from the current point x_k to a new point x_{k+1}. The first to be considered here is linesearch methods. In these, the system of equations (3.7) is solved to find the Newton direction for the quadratic model (d_k) and then a decision is made about how far to travel in that direction to obtain improvement in the nonlinear program.

3.2.2.1 Merit functions

Merit functions are used to measure the *merit* of points on the line defined by $x_k + \alpha_k d_k$. These functions closely resemble the penalty functions defined in section 2.1 but represent entirely different concepts. They are used to compare points with respect to improvements in each of the objective and constraint feasibility. In this section, and for the remainder of this work, the penalty parameter $\frac{1}{\mu}$ is replaced with ν to avoid confusion with the interior point barrier parameter μ.

Exact merit functions are those for which a penalty parameter, ν, can be found such that minimization of the merit function is equivalent to finding a minimum of the original nonlinear problem. In [44], Han and Mangasarian discuss values of ν which have this property for an l_1 merit function. They prove that if ν is larger than the maximum absolute value of the dual variables of the problem, that is:

$$\nu > |\lambda_i| \quad \forall i \in \mathcal{E} \cup \mathcal{I} \tag{3.8}$$

then the property holds. This value of ν is bounded if the LICQ is satisfied. Exact merit functions include l_1 merit functions and Augmented Lagrangian merit functions.

Inexact merit functions, such as quadratic merit functions, can also be used, but a penalty parameter which ensures that the minimum of the merit function is also the minimum of the NLP problem cannot be found.

3.2.2.2 The Maratos effect

Merit functions of the form

$$\Phi(x; \nu) = f(x) + \nu \|c(x)\|$$

can suffer from the Maratos effect [54] which occurs because curvature in the constraints is not adequately represented by linearization in the QP model. This can result in cases where the direction found by solving (3.7) would cause increase in both the objective function and the constraint violation whilst being consistent with the quadratic convergence expected from a Newton's method. Nocedal & Wright [62] gives an example of this, taken from [67]. The Maratos effect can dramatically reduce the rate at which an SQP method converges.

The problem caused by the Maratos effect can be tackled by allowing a non-monotonic decrease in the merit function. Grippo, Lampariello & Lucidi [43] implement this concept by storing merit function values from the previous M iterations and insisting upon improvement on the worst of these. (M is a constant.)

The problem can also be tackled by using a second order correction. A second direction d'_k, which satisfies the linear constraints at $x_k + d_k$ is calculated and the linesearch is carried out with respect to the direction $d_k + d'_k$. For further details see Fletcher [30] or Nocedal & Wright [62].

3.2.2.3 Choosing a stepsize

The stepsize α_k could be chosen to be the minimizer of the merit function evaluated at $x_k + \alpha_k d_k$. That is, the minimizer of $\Phi(\alpha_k)$ where

$$\Phi(\alpha_k) = f(x_k + \alpha_k d_k) + \nu \sum_{i\in\mathcal{E}} |c(x_k + \alpha_k d_k)| + \nu \sum_{i\in\mathcal{I}} |c(x_k + \alpha_k d_k)|^-.$$

However, finding the value of α_k which exactly minimizes $\Phi(\alpha_k)$ is a relatively expensive operation and it is usually more efficient to find a good, approximate value.

If the search direction, d_k is descent with respect to the merit function then there are values of α_k for which the merit function value decreases. It is necessary to ensure that

a. the decrease made is not negligible,

b. the step taken is not too small.

This can be done by using a backtracking linesearch which decreases α_k after every trial. For each trial value of α_k, the point $x_k + \alpha_k d_k$ is tested to see if sufficient decrease would be made in the merit function if a step of length α_k were taken. The following condition is checked:

$$\Phi(\alpha_k) \leq \Phi(0) + c_1 \alpha_k \nabla\Phi_k(0)^T d_k, \tag{3.9}$$

where $c_1 \approx 0.0001$ is a small constant and $\nabla\Phi_k(0)^T d_k$ is the directional derivative of the merit function at point x_k with respect to search direction d_k.

This condition (3.9) is often called the "Armijo" condition and is one of two which are collectively known as Wolfe's conditions. The second condition ensures that α_k does not get too small by insisting that the merit function is decreasing less rapidly at $x_k + \alpha_k d_k$ than at x_k. The condition

$$\nabla\Phi_k(\alpha_k)^T d_k \geq c_2 \nabla\Phi_k(0)^T d_k, \tag{3.10}$$

where $c_2 \in (c_1, 1)$ (c_1 as in (3.9)), is known as the curvature condition. Its use allows for linesearches which are not backtracking.

Backtracking linesearches can start from any value of α_k, usually > 1. In the case of linesearches based on Newton's method, the best starting point is chosen by setting $\alpha_k = 1$. New trial values of α_k are then chosen between 0 and the current value. This can be done by successively halving the current value or by interpolating known function and derivative values of $\Phi(\alpha_k)$ and using the minimizer of the interpolating polynomial as the next value of α_k.

3.2.3 Trust region methods

The second fundamental strategy for moving from the current point x_k to a new point x_{k+1} is the use of a trust region. A point is found which approximately minimizes a quadratic model at x_k, with an additional constraint that restricts the distance between x_k and x_{k+1}. This constraint is called a trust region constraint and takes the form $\Delta x_k \leq \delta$, where δ is the current trust region's size.

One method of finding this new point is to calculate the Cauchy point, x^{CP}, (the best step in the steepest descent direction) and the Newton point, x^N, (taking a step of 1 in the Newton direction) and then choose the best point on the line joining x_k, x^{CP} and x^N. This line, known as a *dogleg*, is shown in Figure 3.1. The improvement predicted by the step in the quadratic model is compared with the improvement which is actually made in the nonlinear program and a decision about whether to take the step or adjust the trust region is made as follows:

```
**If** actual reduction $\geq C_1 \times$ predicted reduction.
    Make step and increase trust region size.
**Else if** actual reduction $\geq C_2 \times$ predicted reduction.
    Make step.
**Else**
    Reduce trust region size and do not make step.
```

$1 > C_1 > C_2 > 0$ are constants which vary between algorithms and don't seem to be critical to their success. More of the underlying theory of trust region methods can be found in e.g. [9, 30, 35, 62].

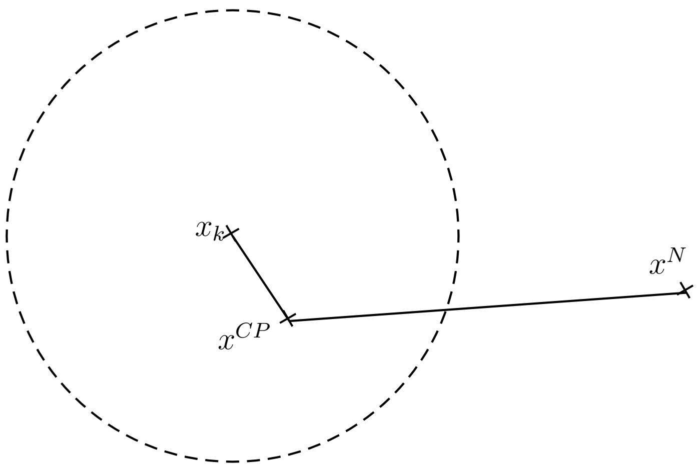


Figure 3.1: Showing the trajectory which can be searched when looking for an approximate minimizer inside a given trust region. The point which would be chosen if this method were used is the point where the *dogleg* trajectory crosses the trust region boundary.

3.3 hopdmSQP

In this section, we give a basic outline of a linesearch SQP algorithm, which we will consider step by step to show the decisions made when implementing our own SQP solver, using the LP and QP solver `hopdm` [39]. We used the *Constrained and Unconstrained Testing Environment* (CUTE [14]), to tailor the algorithm.

Algorithm 3.1.

```
Choose a starting point $x_0$, $\lambda_0$.
For $k = 0, 1, 2, \ldots$
    Calculate $f(x_k), \nabla f(x_k)$, $B(x_k, \lambda_k)$ and $c_i(x_k), \nabla c_i(x_k)$ $\forall i \in \mathcal{E} \cup \mathcal{I}$.
    Choose accuracy tolerance $\tau_k$.
    If final convergence test is satisfied
        STOP with solution $x_k, \lambda_k$.
    Form QP approximation (3.2).
    Solve (3.2) to get direction $d_k$ and new Lagrange multipliers $\lambda_{k_{new}}$.
    Choose a steplength $\alpha_k$.
    Make step
      $x_{k+1} = x_k + \alpha_k d_k$
      $\lambda_{k+1} = \lambda_k + \alpha_k(\lambda_{k_{new}} - \lambda_k)$
End for
```

- Choose a starting point x_0, λ_0.

  With the exception of cases when the primal starting points, x_0, given by the

CUTE models are outside their variable bounds, they are left unchanged. Otherwise, they are brought within the bounds, using the following algorithm:

```
If primal starting points are less than the variable lower bounds
  If a finite upper bound exists
    Choose $x_0$ to be between the lower and upper bounds in a $10\% - 90\%$ ratio.
  Else
    Choose $x_0$ to be $\vartheta$ greater than the lower bound[2].
    $(x_0 = l + \vartheta)$
Else if primal starting points are more than the variable upper bounds
  If a finite lower bound exists
    Choose $x_0$ to be between the lower and upper bounds in a $90\% - 10\%$ ratio.
  Else
    Choose $x_0$ to be $\vartheta$ less than the upper bound[2].
    $(x_0 = u - \vartheta)$
```

Dual starting points, λ_0, were always left unchanged.

- Calculate $f(x_k), \nabla f(x_k)$, $B(x_k, \lambda_k)$ and $c_i(x_k), \nabla c_i(x_k)$ $\forall i \in \mathcal{E} \cup \mathcal{I}$.
  The CUTE problems have been written as `ampl` models. `Ampl` [32] is an automatic differentiation tool which provides first and second derivatives for optimization problems. Using the guidelines set out in [34] we wrote an interface between `ampl` and `hopdm` which enables us to use the exact Hessian of the Lagrangian as an initial $B(x_k, \lambda_k)$.

- Choose accuracy tolerance τ_k.
  As the QP model is an approximation to the NLP problem which we are trying to solve, we do not need to aim for high accuracy until we believe that we are close to the solution of the NLP problem. Therefore, we ask for an accuracy of 1.0×10^{-6} (6 decimal places) for the first quadratic model, and thereafter choose the accuracy requested according to change in objective function.

$$\begin{array}{lllllll}
\textbf{if} & |(f_{k-1} - f_k)| & < & 0.1 & \textbf{then} & \tau_k & = & 5.0 \times 10^{-7} \\
\textbf{if} & |(f_{k-1} - f_k)| & < & 0.01 & \textbf{then} & \tau_k & = & 5.0 \times 10^{-8} \\
\textbf{if} & |(f_{k-1} - f_k)| & < & 0.001 & \textbf{then} & \tau_k & = & 5.0 \times 10^{-9}
\end{array}$$

[2] ϑ is chosen as in `Loqo` [74] to be 1.0.

Of course, if the objective function is a constant, then this method of choosing τ_k is inappropriate, because it will ask for high accuracy on every iteration. In these cases, we use the reduction of constraint violation to determine τ_k.

- **if** final convergence test is satisfied
  STOP with solution x_k, λ_k.
  Theorem 3.1 shows that if x_k is a solution and the LICQ holds at x_k, then the KKT conditions (3.3) are satisfied. Therefore, in most cases, checking these conditions is a suitable way to determine whether x_k is a solution of the problem.

  If the objective function is a constant, then we only need to check that the constraints are not violated.

- Form QP approximation (3.2).
  The QP approximation which we use here is not as straightforward as (3.2).
  - We place artificial bounds on the primal direction variables. Generally, we allow them to have a maximum size of 20. However, if the requested accuracy for the QP model is less than 1.0×10^{-6} (because we believe we are close to the solution of the NLP problem) then they are allowed a maximum size of just 5. These bounds act as a basic trust region.
  - Elements in the Hessian and Jacobian with size less than 1.0×10^{-8} are removed completely. This affects the sparsity structure of each of these matrices, reducing the number of floating point operations required when solving the first order optimality conditions and therefore improving the efficiency of the algorithm.
  - We add extra variables to the problem, two for each equality constraint, to ensure that the problem is primal feasible. In each pair, both variables can take any positive value, but one is added to the constraint and the other is subtracted. Inequality constraints receive one new positive variable, which is added. If the new variables (h^+ and h^-) are required, then they are penalised in the objective function, with a penalty parameter ρ.

$$
\begin{array}{llcrcl}
\min & \frac{1}{2}d^T B(x,\lambda)d + \nabla f^T(x)d & + & \rho \sum_{i\in\mathcal{E}\cup\mathcal{I}} h_i^+ & + & \rho \sum_{i\in\mathcal{E}} h_i^- \\
\text{s.t.} & \nabla c_i^T(x)d + c_i(x) & + & h_i^+ - h_i^- = 0 & & i \in \mathcal{E} \\
 & \nabla c_i^T(x)d + c_i(x) & + & h_i^+ \geq 0 & & i \in \mathcal{I} \\
 & & & h_i^+ \geq 0 & & i \in \mathcal{E}\cup\mathcal{I} \\
 & & & h_i^- \geq 0 & & i \in \mathcal{E} \\
 & & & l_i^d \leq d_i \leq u_i^d & & i \in 1,2,\ldots,n
\end{array}
$$

Inspired by Benson & Shanno [5], ρ is given an initial value ten times greater than the largest primal or dual variable. As the algorithm progresses, ρ is gradually increased, to make the use of these artificial variables less attractive as a solution is neared. A good explanation of the reasoning behind adding extra parameters to the linearized constraints is given by Tone [70].

- Solve (3.2) to get direction d_k and new Lagrange multipliers $\lambda_{k_{new}}$.
  At this point, the altered version of (3.2) is sent to `hopdm` and a solution is found. There are several factors which may cause `hopdm` to terminate with an error code which states that a solution could not be found.

  The solution found is tested to check that it is a descent direction with respect to the merit function. Strategies for continuing when the solution is not descent, or when `hopdm` terminates with an error code are discussed in section 3.3.1.

- Choose a steplength α_k.
  α_k is chosen with a linesearch strategy that uses the l_1 merit function with a penalty parameter chosen as in (3.8). A backtracking linesearch is used, with new trial values of α_k chosen by quadratic interpolation. The nonmonotone strategy of Grippo *et al.* [43] is implemented to handle the Maratos effect.

- Make step

$$
\begin{aligned}
x_{k+1} &= x_k + \alpha_k d_k \\
\lambda_{k+1} &= \lambda_k + \alpha_k(\lambda_{k_{new}} - \lambda_k)
\end{aligned}
$$

3.3.1 Dealing with `hopdm` error codes and nondescent directions

If `hopdm` returns an error code or if the direction that it finds is nondescent with respect to the merit function then extra work must be carried out to enable the algorithm to continue. This section describes the methods implemented in `hopdmSQP` in such cases.

Primal or dual infeasible problem

If the error code returned states that the QP model is primal or dual infeasible then we adjust the penalty parameter ρ, which was introduced to guarantee primal feasibility, and return the problem to `hopdm`.

Requested accuracy not reached

If the problem is not solved to the requested accuracy then a multiple of the identity (ΓI) is added to $H\!\!L$ and the problem is returned to `hopdm`.

The first time this addition is required, Γ is calculated by adding together the maximum absolute value of off-diagonal elements and the size of the most negative diagonal element. If this gives $\Gamma < 1$ then Γ is taken to be 1. On each subsequent $H\!\!L$ update, Γ is doubled.

Nondescent directions

If the direction found is nondescent with respect to the merit function then there are a number of possibilities available to us. The first thing we try is to make our request for accuracy more demanding and return the problem to `hopdm`.

If the direction returned is nondescent after progressively increasing the accuracy requested to a maximum of 11 decimal places, we try to regularize the Hessian approximation in the same way as when the solution found by `hopdm` did not reach the requested accuracy.

If this regularization has not resulted in a descent direction after Γ has been doubled 3 times, then we use the steepest descent direction with respect to the objective ($\nabla f(x_k)$) as a possible step direction. We need to be careful to ensure that this direction does not take variables outside their bounds, as this could cause numerical difficulties. We use the size of the previous iteration's step ($\|d_{k-1}\|$) as a guide to determine how far along the steepest descent direction we should aim to travel.

The steepest descent direction is also used if changing ρ when the problem formulation is infeasible is unsuccessful or if requested accuracy is not obtained after 4 regularization attempts.

3.4 CUTE

3.4.1 Small subset

The *Constrained and Unconstrained Testing Environment* (CUTE [14]) was used to test the success of `hopdmSQP`. A test set of 96 problems was randomly chosen from the 732 problems which have been written as `ampl` models by Benson [8]. When the first trial version of `hopdmSQP` was tested on these 96 problems, 52 (54%) were solved to the requested accuracy. The algorithm's behaviour on each of the problems was observed, patterns were detected and changes were made to attempt to improve the success rate. These changes are explained in Table 3.1

and the specific problems which are solved with each version of `hopdmSQP` are shown in Appendix A (Table A.1). Changes E and J were not considered to be successful alterations, so have not been kept. The current version of `hopdmSQP` is able to solve 81 of the problems (84%).

	Description of Change	% Solved
Start		54
A	Reduced restriction on termination conditions, replacing requirement that KKT condition (3.3a) be satisfied to 6 decimal places with the possibility that if objective function is no longer being significantly improved and 5 decimal place accuracy is achieved on 3 successive iterations, or 4 decimal place accuracy is achieved on 10 successive iterations, the algorithm will terminate.	58
B	Insisted that primal variables are initialized within their bounds.	57
C	Increased penalty parameter ρ by a multiple of 2 at each iteration.	62
D	Delayed the increase of ρ until iteration 10.	79
E	Placed a fake upper bound on infeasibility variables.	59
F	Changed response to `hopdm` not reaching requested accuracy.	70
G	Introduced the possibility of regularizing matrix when direction is nondescent.	81
H	Reduced the use of nonmonotonicity, so that the possibility of accepting a point which does not make an improvement on the current merit function is only included once $\alpha < 0.1$.	79
I	Some reordering of parameter settings and subroutines.	84
J	Stabilized merit function so that comparisons with past iterations use the same penalty parameter as the current iterate.	82

Table 3.1: Changes made to `hopdmSQP` to improve its performance.

3.4.2 Complete set

`hopdmSQP` was then tested on all 732 problems. The time taken to solve each problem, the number of NLP iterations and the constraint violation and objective value at the solution are shown in Table A.2. The solver was allowed to run for four hours and was allowed a maximum of 500 iterations. In all, 612 problems were solved (84%).

3.5 Possible further improvements to `hopdmSQP`

Those problems for which `hopdmSQP` fails to converge are listed in Table A.3 along with a description of their behaviour during the iteration sequence, or a guess at why the problem is not solved. From this, it is possible to see that 19 problems converge to a known solution of the problem, but do not terminate. This indicates that more work needs to be carried out into determining appropriate termination conditions for the algorithm. Possibly, it would be advisable to alter these conditions so that they are related to the size of the objective value.

It can be seen that there are many problems for which $\alpha \to 0$ and no further progress can be made. It would be worth investigating the causes of this. It is possible that this behaviour is caused by the Maratos effect, despite the nonmonotone procedure which has been implemented. It is also possible that it would be more effective to update the merit function penalty parameter in a different way, maybe giving less penalty to constraint violation. However, these are conjectures and the actual causes for this behaviour still need to be researched.

There are 2 problems (`semicon1` and `vanderm1`) for which the steepest descent method returns an error. It has not been implemented correctly when it is called because ρ is increased above its maximum value.

Also, the method for updating $H\!\!L$ when the problem is thought to be nonconvex is inefficient. It would be more efficient to determine the nature of $H\!\!L$ before 200 redundant QP iterations have been carried out and `hopdm` is unable to find a solution with the requested accuracy. Also, it would be efficient to test each trial identity addition $H\!\!L + \Gamma I$ for convexity before the QP iterations are carried out.

It is also of concern that the steepest descent method *ever* needs to be implemented. Ideally, the merit function and $H\!\!L$ should be constructed such that the solution of the QP approximation can provide sufficient decrease in the merit function.

Now, there are several problems for which the objective value is still discernibly decreasing after 500 iterations. At least three of these (`hues-mod, palmer1d, palmer2c`) can be solved if the algorithm is allowed to run longer. If we include these problems, and those for which the solver converges but does not terminate as successes, then `hopdmSQP` solves 86.6% of the `CUTE` set.

Chapter 4

Nonlinear Programming Algorithms

This chapter provides an overview of the work on solution methods to NLP problems which are found in the literature. To this end, we consider the different solution methods described in Chapter 2 and comment on the choices made by authors of the widely varying solution techniques within each section. Also, although each author has chosen different notation for their work, we will use the same notation as in previous chapters when commenting on each solver, unifying the algorithms in the literature so that comparisons can be made more easily.

We will begin by mentioning penalty methods, considering the need to avoid the ill-conditioning inherent in these techniques. We will then move on to show how interior point methods, although similar, avoid the problem of ill-conditioning when applied directly to NLP problems. Here, we provide a lengthy discussion about the details of a selection of primal-dual interior point solvers. Following this, we will mention several SQP methods from the literature, remarking on the wide variety of techniques available.

Finally, we consider the merit of some of the algorithms presented, relating the authors' own conclusions, referring to some comparison papers and including reference to the success of our `hopdmSQP` relative to the conclusions drawn in these papers.

4.1 Penalty methods

Penalty methods for solving NLP problems have been being researched for a number of decades. A good summary text, published in 1968, is the book by Fiacco & McCormick [28] which gives a thorough historical survey of sequential unconstrained methods for solving constrained minimization problems before describing a variety of such algorithms in detail. As mentioned in Chapter 2, these solution techniques tend to suffer from ill-conditioning as the penalty parameter μ approaches zero.

Nash & Sofer have written an interesting paper [61] which employs techniques to combat these problems, using a log-barrier function of the form (2.2), for inequality constrained problems. As a pure Newton method of the form

$$d_k = -H\!\!L^{-1}\nabla f(x_k), \quad x_{k+1} = x_k + d_k,$$

would struggle as μ approaches zero because of ill-conditioning, they use a truncated Newton method, avoiding problems caused by the ill-condition of the Hessian matrix, $H\!\!L$, by calculating an approximation to the Newton direction. It is shown that their approximation becomes more accurate as μ decreases.

It is necessary to incorporate a linesearch to find a suitable stepsize, α, such that

$$x_{k+1} = x_k + \alpha d_k$$

is a good new point. Murray & Wright [59] show that standard linesearch methods (usually based on polynomial interpolation) are often ineffective for log-barrier functions. They suggest that the interpolating function used to estimate a good stepsize should include logarithmic terms. In fact, numerical results are given in both [61] and [59] which show that altering the linesearch to reflect the presence of logarithmic terms leads to a significant improvement in the efficiency of a log-barrier algorithm. An example of the problems encountered when using a polynomial interpolant to a log-barrier function is shown in Figure 4.1. It can be seen that the minimum of the polynomial interpolant does not provide a good estimate of the minimum of the log-barrier function.

Other work on how to overcome ill-conditioning and poor scaling as μ approaches zero has been carried out by Gould [42] and Dussault [25]. Also, Forsgren & Gill [31] have considered the use of a mixed penalty function of the form (2.3) and have shown how ill-conditioning can be avoided by use of primal-dual interior point methods.

Before we move on to discuss those methods, it is important to remember the Augmented Lagrangian function, which was mentioned in section 2.1 as a penalty function which does not suffer from ill-conditioning as μ approaches zero. It was first proposed by Hestenes [48] and Powell [66], its key properties are described by Fletcher [30] and it is the basis of the successful algorithm LANCELOT [20, 21] by Conn, Gould and Toint.

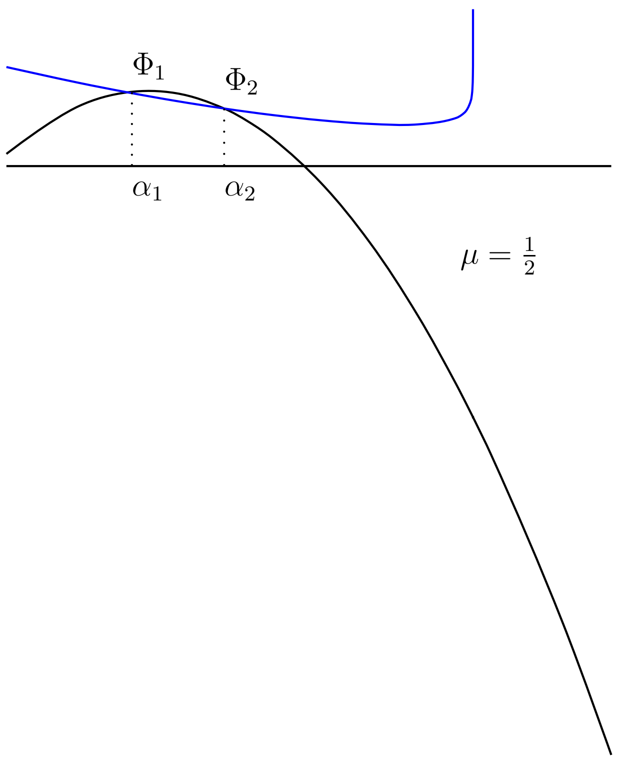


Figure 4.1: Showing the inadequacy of a polynomial interpolant to a logarithmic barrier function. The blue line shows the logarithmic barrier function and the black line is its polynomial interpolant at points α_1 and α_2. (Calculated from $\Phi(\alpha_1), \nabla\Phi(\alpha_1)$ and $\Phi(\alpha_2)$.)

4.2 Interior point methods

It is interesting to note that although the KKT conditions for the logarithmic barrier problem

$$\begin{aligned} \nabla f(x) - \mu \sum_{i \in \mathcal{I}} \frac{\nabla c_i(x)}{c_i(x)} &= 0 \\ c(x) &\geq 0 \end{aligned}$$

are equivalent to the perturbed KKT conditions used in interior point methods

$$\begin{aligned} \nabla f(x) - \sum_{i \in \mathcal{I}} \lambda_i \nabla c_i(x) &= 0 \\ c(x) &\geq 0 \\ \lambda_i c_i(x) &= \mu \qquad \forall i \in \mathcal{I}, \end{aligned}$$

the iterates found when applying Newton's method to each case do not coincide. This result, which is easy to extend to nonlinear problems, has been shown by El Bakry *et al.* [26] for linear programming.

In this section we will look at a selection of algorithms which use the perturbed KKT conditions and so are not prone to suffer from ill-conditioning.

Algorithm 2.3 provides a structure for an interior point method for solving an NLP problem. There are many choices which have to be made in the implementation of such a method. In this section we will focus on 5 key features:

- The problem formulation to be solved. For example, we have already mentioned formulations (1.1) and (2.5) and considered the possibility of $\mathcal{E} = \emptyset$ or $\mathcal{I} = \emptyset$, or instances where there are no bound constraints, or when the bound constraints are included in $\mathcal{I}$.
- The merit function.
- The step direction and stepsize, α.
- The barrier parameter, μ.
- The method for dealing with nonconvexity, which is often handled by choosing a positive definite approximation, $B(x, \lambda)$, to the Hessian of the Lagrangian.

We have first divided the interior point solvers into four groups. We consider traditional linesearch and trust region techniques, a recent algorithm which combines the two and finally a linesearch algorithm based on Fletcher & Leyffer's filter mechanism [29]. In all, we discuss the algorithmic details of six interior point NLP solvers and this section is completed with a comparison of these solvers in relation to the 5 features listed above.

4.2.1 Linesearch methods

Linesearch methods were introduced in section 3.2.2 and the issues which need to be addressed when writing a linesearch algorithm are outlined in the subsequent discussion. Essentially, a system of linear equations is solved to find the Newton direction and a merit function is used to determine a suitable step to be taken in this direction. Here we discuss two interior point linesearch solvers in detail.

`Loqo` [71]

Vanderbei & Shanno have used the QP solver `Loqo` [74] as a building block for an interior point algorithm for nonlinear programming. Some details of the QP solver will be mentioned later, in Chapter 5.

Loqo solves:

$$\begin{array}{lcccl} \min & f(x) & & & \\ \text{s.t.} & 0 & \leq c_i(x) & \leq & r_i \quad i \in \mathcal{I} \\ & 0 & \leq c_i(x) & \leq & 0 \quad i \in \mathcal{E} \end{array}$$

with bound constraints also included, but eliminated here for simplicity of formulation. $r \in \mathbb{R}^m \in [0, \infty)$ represents the range which an inequality constraint can take. Equality constraints are written this way, by setting $r_i = 0$.

The above formulation can be altered to

$$\begin{array}{lcccccll} \min & f(x) & & & & & & \\ \text{s.t.} & c_i(x) & - & s_i & & = & 0 & i \in \mathcal{E} \cup \mathcal{I} \\ & & & s_i & + \; p_i & = & r_i & i \in \mathcal{I} \\ & & & s_i & + \; p_i & = & 0 & i \in \mathcal{E} \\ & & & s_i, & p_i & \geq & 0, & \end{array}$$

where $s,\ p \in \mathbb{R}^m$ are slack variables.
This formulation gives the Lagrangian

$$\begin{aligned} L(x, s, p, \lambda, q; \mu) =& f(x) - \lambda^T(c(x) - s) - \sum_{i \in \mathcal{I}} q_i(s_i + p_i - r_i) \\ & - \sum_{i \in \mathcal{E}} q_i(s_i + p_i) - \mu \sum_{i \in \mathcal{E} \cup \mathcal{I}} \ln s_i - \mu \sum_{i \in \mathcal{E} \cup \mathcal{I}} \ln p_i, \end{aligned}$$

where $\lambda,\ q \in \mathbb{R}^m$ are Lagrange multipliers associated with constraints.

Working only with equality constraints makes this formulation advantageous when solving the first order optimality conditions to find the Newton direction. For more details, see [74] or [71].

The merit function used is

$$f(x) - \mu \sum_{i \in \mathcal{E} \cup \mathcal{I}} (\ln s_i + \ln p_i) + \frac{\nu}{2} \left\{ \|c(x) - s\|_2^2 + \|s_{\mathcal{I}} + p_{\mathcal{I}} - r_{\mathcal{I}}\|_2^2 + \|s_{\mathcal{E}} + p_{\mathcal{E}}\|_2^2 \right\}.$$

ν is initialised at zero and increased if the direction found is nondescent or if α tends to zero, which could imply that the algorithm is converging to an infeasible optimum. Although the theory states that this merit function, being inexact, could require ν to appraoch infinity, in practice it increases rarely and often remains at zero.

The stepsize, α, is chosen by successive halving. If the point with α_k does not improve the merit function then $\alpha_{k+1} = \frac{\alpha_k}{2}$.

The barrier parameter, μ, is chosen by an efficient heuristic which is based on the reasoning that a sequence of iterates converges more quickly if the values of complementarity products $s_i\lambda_i$ converge uniformly to zero. Distance from uniformity is measured by comparing the value of each complementarity product against the average. When far from uniformity, the value of μ at the next iteration is chosen to be close to the current value to promote uniformity.

Finally, if $H\!L$ is not positive definite, a positive multiple of the identity is added such that

$$B(x, \lambda) = H\!L + \Gamma I$$

is positive definite. Positive definiteness is ensured by repeatedly doubling Γ until the diagonal elements of the symmetric factorization (see section 3.2.1) of $B(x, \lambda) = LDL^T$ are all greater than zero.

<u>`A.L.Tits, A.Wächter, S.Bakhtiari, T.J.Urban & C.T.Lawrence`</u> [**69**]

Tits *et al.* have proposed a primal dual interior point algorithm which was written specifically to deal with a group of problems for which interior point algorithms have been shown to consistently fail (see [76]).

This algorithm solves

$$\begin{array}{rlcll} \min & f(x) & & & \\ \text{s.t.} & c_i(x) & = & 0 & i \in \mathcal{E} \\ & c_i(x) & \geq & 0 & i \in \mathcal{I} \end{array}$$

by replacing the equality constraints with an l_1 penalty in the objective[1]

$$\begin{array}{rllll} \min & f(x) & + & \nu \sum_{i \in \mathcal{E}} c_i(x) & = f_\nu(x) \\ \text{s.t.} & c_i(x) & \geq & 0 & i \in \mathcal{E} \cup \mathcal{I}. \end{array}$$

Using $\lambda_i = \dfrac{\mu}{c_i(x)}$, this formulation gives the Lagrangian

$$\begin{aligned} L(x, \lambda; \nu, \mu) =& f(x) - \mu \sum_{i \in \mathcal{E} \cup \mathcal{I}} \ln c_i(x) + \nu \sum_{i \in \mathcal{E}} c_i(x) \\ \nabla_x L(x, \lambda; \nu, \mu) =& \nabla f(x) + (\nu e - \lambda_{\mathcal{E}})^T \nabla c_{\mathcal{E}}(x) - \lambda_{\mathcal{I}}^T \nabla c_{\mathcal{I}}(x), \end{aligned}$$

where $\lambda_{\mathcal{E}}$, $\lambda_{\mathcal{I}}$ are Lagrange multipliers associated with equality and inequality constraints respectively and $\nabla c_{\mathcal{E}}(x)$, $\nabla c_{\mathcal{I}}(x)$ are Jacobian matrices associated with equality and inequality constraints.

This formulation already includes an l_1 penalty function which is used as a merit function. It only penalizes equality constraints as the constraints are prevented from taking negative values by the logarithmic terms, which means that inequality constraints cannot take infeasible values. As Han and Mangasarian proved in [44], if $\nu > \max_{i \in \mathcal{E}} |\lambda_i|$ then the minima of this penalty function are equal to minima of the NLP problem. However, to prevent ν increasing too quickly, it is only updated when several stringent conditions are met.

[1] $| \, . \, |$ signs are redundant in this formulation as $c_i(x)$ is always positive.

The stepsize, α, is chosen carefully. It is computed as the first α in the sequence $\{1, \eta, \eta^2, \dots\}$, $\eta \in (0,1)$ such that

$$\begin{aligned} f_\nu(x + \alpha\Delta x + \alpha^2\Delta\tilde{x}) &\le f_\nu(x) + \zeta\alpha\nabla f_\nu(x)^T\Delta x \\ c_i(x + \alpha\Delta x + \alpha^2\Delta\tilde{x}) &\ge 0 \qquad i \in \mathcal{E} \cup \mathcal{I}, \end{aligned}$$

where $\zeta \in (0, \frac{1}{2})$ and $\Delta\tilde{x}$ is a second order correction which is only included when the iteration sequence is near to a solution, in order to avoid the Maratos effect [54].

The barrier parameter is selected such that μ

- is large enough to prevent α tending to zero due to infeasibility.
- is small enough that significant decrease for f is achieved.
- approaches zero fast enough for the local convergence properties associated with the Newton method ($\mu = 0$) to be exploited.

No mention is made of how they choose their modification to the Hessian of the Lagrangian in order to deal with nonconvexity.

4.2.2 Trust region methods

Trust region methods were introduced in section 3.2.3. An additional constraint is added to the quadratic model to ensure that the suggested step is not larger than a given trust region radius. Here, we discuss the features of two individual algorithms which use trust region logic.

<u>`NuOpt`</u> [**80**]

`NuOpt` extends work by Yamashita [81]. It works to solve problems of the form:

$$\begin{array}{rrcll} \min & f(x) & & & \\ \text{s.t.} & c_i(x) & = & 0 & i \in \mathcal{E} \cup \mathcal{I} \\ & x & \ge & 0 & i \in 1, 2, \dots, n, \end{array}$$

where inequality constraints have been converted to equality constraints by the addition of slack variables with lower bounds of zero. This gives the Lagrangian:

$$L(x, \lambda; \mu) = f(x) - \lambda^T c(x) - \mu \sum_{i=1}^{n} \ln x_i.$$

An l_1 merit function is used:

$$f(x) - \mu \sum_{i=1}^{n} \ln x_i + \nu \sum_{i \in \mathcal{E} \cup \mathcal{I}} |c_i(x)|$$

with $\nu \geq \max_{i \in \mathcal{E} \cup \mathcal{I}} |\lambda_i|$ as in previous algorithms. The Maratos effect is handled with a nonmonotone technique which is implemented if the interior point barrier parameter, μ, is less than some chosen value ϵ.

Often in trust region methods, two directions are calculated as solutions to each QP approximation. Here, the first is the steepest descent direction $(\Delta x^{SD}, \Delta\lambda^{SD})$ and the second is the Newton direction $(\Delta x^N, \Delta\lambda^N)$. The direction taken is a linear combination of these two directions

$$(\Delta x, \Delta\lambda) = \beta(\Delta x^{SD}, \Delta\lambda^{SD}) + (1-\beta)(\Delta x^N, \Delta\lambda^N).$$

The combination to be used is chosen in an iterative way.

```
Set $\beta$ to 0
While $\alpha$ not accepted
    Choose largest $\alpha$ in direction $(\Delta x, \Delta\lambda)$ such that:
    - the trust region radius is not exceeded.
    - no variable bounds are violated.
    Find $\alpha^* \in [0, \alpha)$ that minimizes QP approximation in direction $(\Delta x, \Delta\lambda)$.
    If $\alpha^*$ makes sufficient improvement on the Cauchy point.
        Accept $\alpha^*$.
    Else
        Increase $\beta$ by 0.1.
```

If possible, this algorithm accepts the Newton direction.

The barrier parameter μ is controlled by several problem dependent parameters. The strategy for choosing it changes after it has been reduced below the level at which a nonmonotone strategy is introduced.

A positive diagonal matrix is added to $I\!H\!L$ if it is nonsingular.

KNITRO [17]

KNITRO works with the formulation

$$\begin{array}{rlcll} \min & f(x) & & & \\ \text{s.t.} & c_i(x) & = & 0 & i \in \mathcal{E} \\ & c_i(x) & \geq & 0 & i \in \mathcal{I}, \end{array}$$

adding positive slack variables to inequality constraints such that $c_i(x) \geq 0$ is replaced with $c_i(x) + s_i = 0$, $s_i \geq 0 \; \forall \; i \in \mathcal{I}$. This gives the Lagrangian:

$$L(x, s, \lambda; \mu) = f(x) + \lambda_{\mathcal{E}}^T c_{\mathcal{E}}(x) + \lambda_{\mathcal{I}}^T (c_{\mathcal{I}}(x) + s) - \mu \sum_{i \in \mathcal{I}} \ln s_i.$$

The merit function chosen uses the Euclidean norm:

$$f(x) - \mu \sum_{i \in \mathcal{I}} \ln s_i + \nu \| c_{\mathcal{E}}(x), c_{\mathcal{I}}(x) + s \|_2$$

which is nondifferentiable, like the l_1 merit function. It is also prone to the Maratos effect, which is avoided here by the use of second order correction terms.

The step direction is determined in two parts. First, the optimal point on a dogleg trajectory such as the one shown in Figure 3.1 is found. This direction is called the *normal* direction and lies in the range space of the linearized constraints. A second component of the direction is chosen by finding the step in the null space of the constraints which makes the most improvement in the quadratic approximation to the Lagrangian. (That is, the best step which does not alter the amount by which the linearized constraints are violated.) The methods used to calculate this second component of direction are designed to cope with problems which are nonconvex. For more details, see [17].

The penalty parameter ν is chosen such that the predicted improvement in the quadratic model which is made by a step in the composite direction is at least a fraction (0.3) of the predicted improvement made by a step in the normal direction.

The method implemented in KNITRO is not a straightforward trust region method for solving nonlinear programs. In fact, the authors incorporate some logic from SQP methods. As the barrier parameter, μ, is reduced towards zero, a tolerance τ^μ is also reduced to zero. For each value of μ, a sequence of trust region problems is solved, until the KKT conditions (3.3) for the barrier problem are satisfied to within tolerance τ^μ. When this tolerance is reached, μ and τ^μ are decreased such that

$$\mu_{k+1} = \frac{\mu}{5}, \quad \tau^\mu_{k+1} = \frac{\tau^\mu_k}{5}.$$

Finally, it is worth noting that the authors of [17] have experimented with different trust region shapes, concluding that the best shape is one which is designed to prevent slack variables from approaching zero prematurely. The trust region is scaled with S^{-1} to penalize steps near to the boundary. (S is the diagonal matrix formed from s.) That is, the unscaled trust region $\|\Delta x, \Delta s\|_2 \leq \delta$ is replaced with $\|\Delta x, S^{-1}\Delta s\|_2 \leq \delta$.

4.2.3 Hybrid methods

Although most solvers can be classified as either linesearch or trust region methods, recently (2006) Waltz *et al.* [78] mixed linesearch and trust region iterations together in a way which utilizes the advantages inherent in both techniques whilst avoiding the disadvantages.

<u>`KNITRO-Direct`</u> [78]

KNITRO-Direct is a KNITRO-based algorithm which is intended to be more robust than either a pure trust region or a pure linesearch method. It works with the same formulation as KNITRO [17] and also has the same merit function, using second order correction terms to avoid the Maratos effect.

Similarly to KNITRO, μ is held constant until the KKT conditions (3.3) for the barrier problem are satisfied with a tolerance of τ^{μ}.

Now, in practice, a linesearch method, which requires only one direction calculation, is used at every iteration, whilst the trust region method described in [17], which requires an expensive null-space decomposition each time it is used, is only implemented when the linesearch is shown or predicted to be unsuccessful. That is, a trust region step is made when $\mathbb{H}\mathbb{L}$ is not positive definite, or when the steplength α approaches zero. As stated above, the direction chosen by the trust region method of KNITRO is designed to be able to handle nonconvexity effectively.

In order to make smooth transitions between linesearch and trust region iterations, parameters such as α, δ, μ, and τ^{μ} are updated with different strategies according to whether the immediately preceding step was a linesearch or a trust region step.

4.2.4 Filter methods

Filter methods were introduced by Fletcher & Leyffer in 1997 [29] in the context of active set trust region SQP. Described already in section 2.4, a filter represents points which would not be accepted. For each point that defines the filter, the barrier objective value and the constraint violation are stored and any further trial point which has a higher value by both of these measures is not accepted. See Figure 2.3 for an example of a filter.

The next solver to be described uses a filter method incorporated into an interior point NLP solver.

<u>**IPOPT**</u> [77]

Inequality constraints are removed from the NLP formulation by the addition of positive slack variables, using the now familiar problem structure

$$\begin{array}{rrcll} \min & f(x) & & & \\ \text{s.t.} & c_i(x) & = & 0 & i \in \mathcal{E} \cup \mathcal{I} \\ & x & \geq & 0 & i \in 1, 2, \ldots, n \end{array}$$

which gives the Lagrangian

$$L(x, \lambda; \mu) = f(x) - \lambda^T c(x) - \mu \sum_{i=1}^{n} \ln x_i.$$

Instead of using a merit function, IPOPT combines the use of a filter method with a standard linesearch method.

After each QP approximation is formed and solved, α is determined first to keep variables within their bounds and then the point $x + \alpha d$ is tested, and, if necessary, updated with a variation on a backtracking Armijo approach which includes the filter:

```
While $\alpha$ is not accepted.
  If $\alpha < \alpha_{\min}$.
    Move to a feasibility restoration phase[2].
  Else if $x + \alpha d$ is within the filter.
    Reject point.
    $\alpha = \frac{\alpha}{2}$.
  Else
    If $\|c(x_k)\| <$ a chosen minimum constraint violation.
    And a 'switching'[3] condition holds.
      If Armijo condition (3.9) is met.
        Accept $\alpha$.
      Else
        $\alpha = \frac{\alpha}{2}$.
    Else if there is sufficient decrease in barrier objective function.
    Or a sufficient decrease in constraint violation.
      Accept $\alpha$.
    Else
      $\alpha = \frac{\alpha}{2}$.
```

If the accepted point $x + \alpha d$ does not meet the switching condition, or does not make sufficient decrease in the barrier objective function, then it is added to the filter.

[2]Description of this is beyond the scope of this work. See [77] for further details.

[3]The switching condition is met if the search direction is descent and its directional derivative $(\nabla\varphi^T(x)d)$ satisfies the relationship

$$\alpha(\nabla\varphi^T(x)d)^{\kappa_3} > \kappa_5\|c(x)\|^{\kappa_4}$$

with respect to α and to the constraint violation. (Here $\kappa_3, \kappa_4 > 1$ and $\kappa_5 > 0$ are constants and $\varphi(x) = f(x) - \mu\sum_{i=1}^{n} \ln x_i$.)

Similarly to KNITRO and KNITRO-Direct, IPOPT holds μ constant until the KKT conditions (3.3) for the barrier problem are satisfied to within a given tolerance τ^μ. When they are satisfied to this level, μ is reduced using the following formula:

$$\mu_{k+1} = \max\left\{\frac{\tau^\mu}{10}, \min\left\{\kappa_1\mu_k, \mu_k^{\kappa_2}\right\}\right\},$$

τ^μ is reduced and the filter is reset. ($\kappa_1 \in (0,1)$, $\kappa_2 \in (1,2)$ are chosen constants.)

Any nonconvexity in the problem is dealt with by inertia correction. The inertia (number of positive and negative eigenvalues) of the matrix is found by calling Harwell libraries (such as the factorization routine MA27 from [45]) and any indefiniteness is removed by the addition of multiples of the identity.

See Table 4.1 for a comparison of these solvers with regard to the 5 features mentioned at the beginning of this section.

4.3 Sequential quadratic programming

There are many different SQP methods described in the literature. In this section, we look at three, to illustrate the variety of algorithms available. We consider SNOPT [36], a method of Boggs, Kearsley & Tolle [12] and `filterSQP` [29]. The key features of an SQP algorithm are the method used to solve the quadratic model and the strategies used for ensuring that progress is made in the iteration sequence. That is, choosing a suitable stepsize, or deciding how to update a filter.

4.3.1 Solving the quadratic model

Both SNOPT and `filterSQP` solve the quadratic approximation using an active set method. The model each solves is, however, different, SNOPT forming a first derivatives approximation to $H\!\!L$ whilst `filterSQP` uses the exact Hessian of the Lagrangian where possible.

In [12] Boggs *et al.* use a method for solving the QP approximation which has not previously been mentioned here. They call their method the `O3D` algorithm ("optimizing over 3-dimensional subspaces"). Essentially, they find three search directions, d_i, using the formula:

$$\left[-\nabla c(x_k)C^{-2}\nabla c^T(x_k) + \frac{H\!\!L}{\gamma}\right] d_i = t_i, \quad i = 1,2,3,$$

where γ is a scalar depending on the current iterate and C is the diagonal matrix formed by $c(x_k)$. t_i are chosen such that at least one of the directions, d_i, is descent with respect to the objective function. The algorithm then searches for

Solver	`Loqo` [71]	Tits *et al.* [69]	`NuOpt` [80]	KNITRO [17]	KNITRO -Direct [78]	IPOPT [77]
Algorithm Type	Linesearch	Linesearch	Trust region	Trust region	Linesearch Trust Region	Filter Linesearch
Constraints	Inequalities	Inequalities	Equalities	Equalities	Equalities	Equalities
Variable Bounds	Nonnegative slacks	None	Nonnegative slacks	Nonnegative slacks	Nonnegative slacks	Nonnegative slacks
Merit Function	Quadratic	l_1	l_1	Euclidean norm	Euclidean norm	none
Stepsize (α)	Successive halving	From sequence $\{1, \eta, \eta^2, \ldots\}$ $\eta \in (0,1)$	Trust region boundary calculated radially along dogleg	depends on direction	depends on linesearch/ trust region	Successive halving
Barrier Parameter (μ) updated	Every iteration	Every iteration	Every iteration	When tolerance τ^μ is attained	When tolerance τ^μ is attained	When tolerance τ^μ is attained
Nonconvexity	Diagonal added by heuristic	no mention	no mention	Null space decomposition	Null space decomposition	Diagonal added by eigenvalue calculation

Table 4.1: Comparison of Key Features of Interior Point Solvers

the best direction which is a linear combination of d_1, d_2 and d_3 and which is not larger than a given trust region radius.

4.3.2 Choosing the step

Both [36] and [12] are linesearch methods (although [12] makes some use of trust region logic when determining the step) which use an augmented Lagrangian merit function with a backtracking linesearch. In both algorithms, the penalty parameters in the merit function are adjusted to ensure that α can be chosen to allow sufficient decrease in the merit function.

`FilterSQP` is a trust region method which does not use a merit function. Instead, α is always chosen to be 1 and the new point $x_{k+1} = x_k + d_k$ is tested. If it is acceptable to the current filter, then it is added to the filter (see Figure 2.3) and the trust region radius, δ, may be increased. If it is not acceptable to the filter then it is rejected and δ is decreased.

4.4 Comparisons

For many of the algorithms above, numerical results have been presented which show the success of the solver on test problems from well known test sets (Hock & Schittkowski [50], Mittelmann's quadratic programming set [56], Vanderbei's large scale engineering set [72], `CUTE` [14] and `COPS` [24]). In this section, we will relate the successes (or failures) of each solver and the conclusions reached by the algorithms' authors. We will follow this with some comments on comparison papers which discuss the relative successes of a selection of the above algorithms when run on the same computing machines.

4.4.1 Results from individual solvers

Loqo [71] is compared with LANCELOT [21] and MINOS [60] on the Hock & Schittkowski test set and on a selection of large scale problems from [56] and [72]. On the small problems from [50], `Loqo` and MINOS are shown to be competitive, with LANCELOT falling slightly behind, although solution times are generally in fractions of seconds. Each solver was ranked according to its speed when solving these problems and the results are compared in Table 4.2.

On the large scale problems from [56] and [72], `Loqo` is evidently the most appropriate solver of the three. The solvers are again ranked in accordance with their speed in solving these problems and the results are shown in Table 4.3.

Solver	1st	2nd	3rd
Loqo	44	55	13
LANCELOT	4	30	78
MINOS	72	31	9

Table 4.2: Rankings of Loqo [71], LANCELOT [21] and MINOS [60] when compared on the Hock & Schittkowski test set [50].

Solver	1st	2nd	3rd
Loqo	21	4	2
LANCELOT	0	12	15
MINOS	4	14	9

Table 4.3: Rankings of Loqo [71], LANCELOT [21] and MINOS [60] when compared on the large scale test sets [56, 72].

It is worth noting here that, of these three solvers, Loqo is the only one with access to second derivatives, and so would be expected to have the best performance.

Tits, *et al.*'s interior point method [69] is run on 63 carefully chosen problems from the Hock & Schittkowski test set. It is shown that, in terms of iteration count, this solver is better than Loqo on 39 of these 63 problems.

NUOPT [80] is not compared with any of the other solvers, but, with judicious choice of parameters for computing μ, is able to solve all but 1 of the problems from the Hock & Schittkowski test set and succeeds in finding a solution to 31 of 33 problems from CUTE. These 33 problems are chosen such that only one is selected from each family of similar problems, excluding any problem with no objective function or less than 1000 variables. Extensive numerical results are reported.

KNITRO [17] is compared with LANCELOT [21] on the Hock & Schittkowski test set and on 15 problems from CUTE which have been selected for variety. On the small problems from [50], KNITRO does not perform as well as LANCELOT, but on the larger problems from [14] it is competitive.

KNITRO-Direct [78] is a recent algorithm (published November 2005). It has not been compared with other algorithms, but numerical testing on CUTE shows an improvement on previous versions of KNITRO.

IPOPT [77] is compared with KNITRO and Loqo on problems from CUTE. The authors state that they believe that their termination criteria are stricter than those of either KNITRO or Loqo and then show that, using Dolan & Moré's performance measure [24], IPOPT's success rate is slightly higher than those of the other solvers when success is measured by iteration count, number of function evaluations or CPU time.

SNOPT [36] is not compared with any of the other solvers. However, it is shown to solve all problems in the COPS test set and 92.8% of problems in CUTE for which $n - m < 2000$.

filterSQP [29] is compared with LANCELOT on problems from CUTE. The problems are divided into two categories: small problems with $n, m < 25$ and large problems with either $n \geq 25$ or $m \geq 25$. For comparison, only those problems where filterSQP and LANCELOT find the same optimal solution are considered. On both sets of problems, filterSQP proves to be more reliable and faster than LANCELOT.

4.4.2 Results from comparison papers

The authors of the three solvers Loqo [71], KNITRO [17] and SNOPT [36] have made a thorough comparison of their solvers, given in full by the Table [23]. The results of these three solvers on large scale problems from the test sets CUTE, COPS and [72] have been compared in specific detail by Benson, Shanno & Vanderbei [7] and comparisons between these three solvers and filterSQP [29] on problems from CUTE have been made by Morales, Nocedal *et al.* in [57]. These two comparison papers take very different angles in their comparisons, although each divides the problems into groups according to the type of constraints which are present in the problem formulation. In [7] a small subset of varied problems is chosen and the details of formulation and structure which cause the three solvers to behave as they do are discussed. (Results from other large scale problems are also included.) In [57] the entire CUTE set is divided into four sets (unconstrained, equality constrained, inequality constrained and generally constrained problems) and the performance measure [24] is used to draw graphs which compare the behaviour of the four algorithms.

We would like to compare our solver hopdmSQP with these robust solvers and choose to do so by comparing the number of iterations required in order to find an optimal solution on the problems detailed in [7]. For hopdmSQP we count the number of outer NLP iterations and the total number of QP iterations required.

These comparisons, shown in Table 4.4, show that hopdmSQP performs better on these large scale problems than SNOPT, which is designed for problems with

[1] There are bound constraints.

	Dimensions					hopdmSQP	
	n	m	Loqo	KNITRO	SNOPT	NLP	QP
equality QP							
dtoc3	14996	9997	54	x	11271	2	7
inequality QP							
mosarqp1	2500	700	18	28	4578	4	36
yao	2500	1999	202	20	2	24	298
cvxbqp1	10000	0^1	18	x	10000	3	22
biggsb1	1000	0^1	30	23	x	13	154
mixed QP							
cvxqp3	10000	7500	38	x	10217	x	x
gridneta	8964	6724	24	24	8773	4	32
unconstrained QP							
tridia	10000	0	12	8	x	5	27
equality NLP							
gilbert	1000	1	37	33	1046	26	224
dtoc4	14996	9997	20	x	x	7	59
inequality NLP							
svanberg	5000	5000	20	x	x	10	89
mixed NLP							
clnlbeam	1499	1000	119	22	1466	5	32
dallasl	837	598	40	263	x	192	1677
unconstrained NLP							
curly10	10000	0	18	23	x	74	429
penalty1	1000	0	56	46	1170	68	502

Table 4.4: Comparing the number of iterations required by the solvers Loqo, KNITRO, SNOPT and hopdmSQP to solve a selection of large scale problems from the CUTE set. (x represents a problem which cannot be solved)

$n - m < 2000$ and does not use second order information, but not as well as Loqo and KNITRO.

Both [7] and [57] conclude that different solvers have the best performance for different groups of problems. For example, of the four algorithms compared, KNITRO is clearly the most efficient for solving equality constrained problems ([7] notes that Loqo's practice of converting equality constraints into inequality constraints is not ideal). However, Loqo is the most efficient solver for unconstrained problems; filterSQP, Loqo and KNITRO are competitive for inequality constrained problems; and KNITRO and SNOPT are competitive for generally constrained problems.

It is important to remember here, that IPOPT [77], although not included in these comparison papers, reports results which show that its performance is favourable when compared with that of Loqo and KNITRO.

Chapter 5

The Potential for `hopdm` to be a Nonlinear Interior Point Solver.

In Chapter 4 we discussed a large variety of NLP algorithms from the literature. We considered interior point methods and sequential quadratic programming methods, but did not find any solvers which combine these two techniques in the way that `hopdmSQP` does. We suggest that, although `hopdmSQP` is ultimately successful in solving problems from `CUTE`, using an interior point code which works directly with the nonlinear program is likely to be more efficient than the combination of interior point with SQP. Each system of linear equations corresponding to the quadratic model would then only be solved once; rather than several times, reducing μ to zero to obtain a high degree of accuracy. This seems appropriate given that the model is an approximation. We propose to use the experience gained in implementing `hopdmSQP` to write an interior point NLP solver (`hopdmNLP`) which uses `hopdm` [39] as a building block in much the same way as `Loqo` [71] uses [74].

In this chapter, we consider the structure of `hopdm` in greater detail than before, and we begin by stating the formulation that it uses. We then elaborate some of the finer details of the implementation of an interior point method, which are handled by the complete LP/QP solver `hopdm` in `hopdmSQP`, but which will now be dealt with explicitly, such as the control of the barrier parameter μ.

We finish this chapter with a list of factors which should be further investigated before `hopdmNLP` is fully implemented. The algorithm is currently a work in progress.

5.1 The problem formulation used by `hopdm`

$$\begin{array}{lll} \min & f(x) & \\ \text{s.t.} & c_i(x) = 0 & i \in \mathcal{E} \\ & c_i(x) \geq 0 & i \in \mathcal{I}_1 \\ & c_i(x) \leq 0 & i \in \mathcal{I}_2 \\ & l_i \leq x_i \leq u_i & i \in 1 \dots n. \end{array} \tag{5.1}$$

where $\mathcal{I}_1$, $\mathcal{I}_2$ represent a partition of $\mathcal{I}$ into $\geq$ and $\leq$ constraints, respectively, and all other notation is the same as in (1.1).

There are also some range constraints

$$-r_i \leq c_i(x) \leq 0.$$

These are included as a special case of $\mathcal{I}_2$. Additionally, not all variables x_i have both upper and lower bounds. Some may have neither. Some may have identical lower and upper bounds and be *fixed*.

5.2 `hopdmNLP`

In extending `hopdm` into a direct NLP solver we are able to use its robust linear algebra techniques for solving the Newton system of equations. However, unlike the extension to an SQP solver, we are no longer able to rely on previously written code to handle the necessary interior point logic. This logic is considered in section 5.2.2, where the concept of centrality is properly introduced and methods of updating the barrier parameter are discussed.

Before this discussion, we will introduce adjustments made to the Newton system of equations (2.7) which assist in its Cholesky decomposition. We will go on to show how the merit function can be changed to incorporate the barrier parameter and will look carefully at the different types of variables included in the problem structure and at techniques which have been proposed for handling each of them. Finally, we will once more consider techniques available for dealing with problems which are nonconvex. This is an important feature of any nonlinear programming solver and, although mentioned in Chapters 3 and 4, warrants further mention here.

Throughout this section we will draw attention to areas which should be given careful thought when the algorithm is finally determined. They are aspects of the solver which will benefit from numerical experience with different algorithmic or parameter choices. We consider details which may influence the choices to be made and alternative ideas which can be implemented and then compared.

5.2.1 Linear algebra

In general, the linear algebra which forms the basis of `hopdm` is outside the scope of this work. Interested readers are referred to [2], which explains several techniques for solving the Newton equations, as well as discussing issues such as scaling, preprocessing, choice of a starting point and the optimal ordering of a matrix for Cholesky decomposition. Here, we will introduce the augmented system approach to solving the Newton system of equations as, initially, we intend to use this approach in `hopdmNLP`.

Starting with the Newton system of equations (2.7)

$$\begin{bmatrix} -H\!L(x,\lambda) & \nabla c^T(x) & I & -I \\ \nabla c(x) & 0 & 0 & 0 \\ Z & 0 & X\!-\!L & 0 \\ -W & 0 & 0 & U\!-\!X \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta\lambda \\ \Delta z \\ \Delta w \end{bmatrix} = \begin{bmatrix} \xi_c \\ \xi_b \\ \xi_z \\ \xi_w \end{bmatrix},$$

it is possible to determine Δz and Δw in terms of Δx and hence remove them from the equations, leaving the augmented system

$$\begin{bmatrix} -H\!L - \Theta^{-1} & \nabla c^T(x) \\ \nabla c(x) & 0 \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta\lambda \end{bmatrix} = \begin{bmatrix} \xi'_c \\ \xi_b \end{bmatrix} \tag{5.2}$$

where $\xi'_c = \nabla f(x) - \nabla c^T(x)\lambda - (X\!-\!L)^{-1}\mu e + (U\!-\!X)^{-1}\mu e$ is the update of ξ_c which reflects the elimination of Δz and Δw; and Θ is the diagonal matrix formed from θ where the definition of θ_i, dependent on the type of variable x_i, is shown in Table 5.1. Problems which could arise from the ∞s present in the formulation of

Variable Type	θ_i^{-1}	θ_i
Lower bounded	$\dfrac{z_i}{x_i - l_i}$	$\dfrac{x_i - l_i}{z_i}$
Upper bounded	$\dfrac{w_i}{u_i - x_i}$	$\dfrac{u_i - x_i}{w_i}$
Upper & lower bounded	$\dfrac{z_i}{x_i - l_i} + \dfrac{w_i}{u_i - x_i}$	$\dfrac{(x_i - l_i)(u_i - x_i)}{(u_i - x_i)z_i + w_i(x_i - l_i)}$
Free	0	∞
Fixed	∞	0

Table 5.1: How definition of θ_i depends on type of variable x_i.

θ_i related to free and fixed variables will be considered in section 5.2.4.

As outlined in section 3.2.1, the system of equations (2.7), is solved by using Cholesky decomposition. In practice, the augmented system of equations (5.2) is solved, finding the lower triangular matrix L and diagonal matrix D such that

$$LDL^T = \begin{bmatrix} -H\!\!L - \Theta^{-1} & \nabla c^T(x) \\ \nabla c(x) & 0 \end{bmatrix} \tag{5.3}$$

and using successive backsolves to determine Δx and $\Delta\lambda$. It is possible to use the block structure of the augmented system to exploit the sparsity structures of the Jacobian and the Hessian of the Lagrangian.

In fact, in `hopdmNLP`, we are able to further exploit this sparsity. In `hopdmSQP`, at each successive quadratic approximation, we removed all small elements from $H\!\!L$ and $\nabla c(x)$, changing the matrix sparsity where possible, to reduce the number of floating point operations required. Here, we propose to alter small elements to zero without removing them from the sparsity structure of the matrices. This has the advantage of giving exactly the same sparsity pattern to each successive augmented system, allowing us to determine an optimal row reordering and symbolic factorization of (5.3) once, at the beginning of the algorithm. Thus we can use the sparsity pattern of an optimally reordered augmented system to determine a sparsity structure of the lower triangular matrix L which can be reused at every iteration.

Many of the advantages of reusing the same matrix structure and row ordering at each iteration are beyond the scope of this work. However, whilst not investigating the advantages of sparsity exploitation techniques deeply, it is worth acknowledging that they are key to the linear algebra which is implemented in `hopdm` and that understanding sparsity patterns and how they can be utilized is important when trying to improve the efficiency of an optimization algorithm.

5.2.2 Central path and barrier parameter

An important feature of any interior point algorithm is the speed at which the barrier parameter μ approaches zero. In Chapter 4 we briefly discussed the logic which some of the NLP solvers in the literature use to control the decrease of μ, but we have left most of the discussion of this aspect of interior point methods to this section.

First, we will refer to the literature in order to describe the central path in the context of linear and quadratic programming and then we will consider possible ways in which these concepts can be extended to nonlinear programming algorithms.

5.2.2.1 Linear and quadratic programming

In linear and convex quadratic programming, the central path, $\mathcal{C}$, is an arc of points $(x, \lambda, z, w; \mu) \geq 0$ which is controlled by the parameter μ. For each value of μ, there is a unique point (x, λ, z, w) which is primal-dual feasible and for which every complementarity product $((x_i - l_i)z_i,\ (u_i - x_i)w_i)$ is identical and equal to μ. As μ approaches zero, the central path leads to an optimal solution of the LP/QP problem.

Interior point algorithms for solving LP and QP problems make use of this central path by defining a neighbourhood of $\mathcal{C}$ (see Figure 5.1) and setting μ so

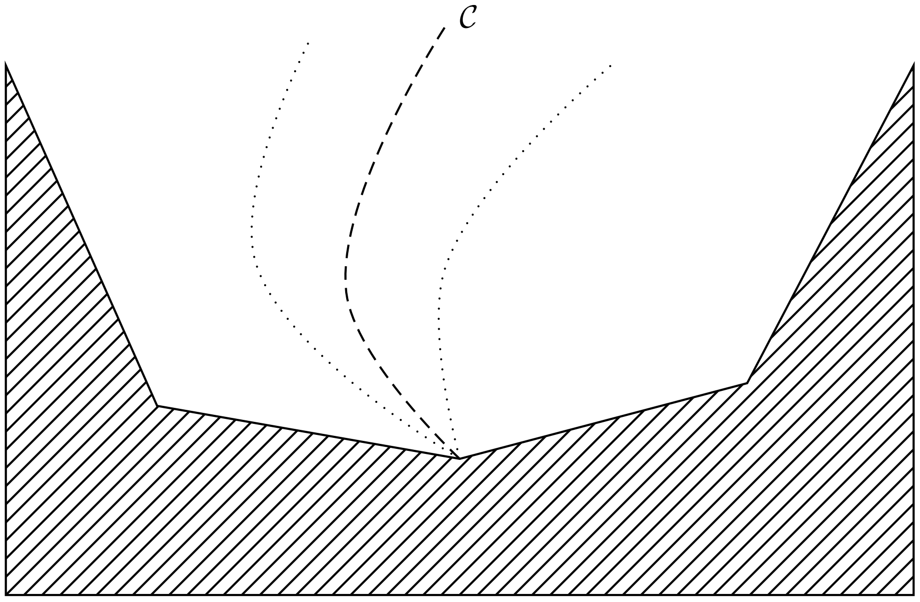


Figure 5.1: A typical neighbourhood of a central path $\mathcal{C}$.

that large stepsizes can be taken whilst keeping the sequence of iterates within the chosen neighbourhood. This is typically done by calculating the average of the complementarity products at the current point, ς (2.8), and choosing $\mu = \sigma\varsigma$ where $\sigma \in (0, 1)$. The choices of neighbourhood and of σ are crucial to the efficiency of the algorithm. A large amount of research has been carried out in this field and rather than try to summarize here, we refer the reader to several good texts on the subject.

Firstly, we recommend the book [79] by Wright, which includes discussion of the central path, different neighbourhoods of the path and various algorithms which cause the series of iterates to stay within these neighbourhoods. Two of the path-following algorithms described are the predictor-corrector method of Mehrotra [55] and its extension to multiple centrality correctors by Gondzio [40]. Briefly, the predictor-corrector method involves finding a *predictor* direction by solving the augmented system with $\mu = 0$. The maximum stepsize in this direction which keeps the iterate within the neighbourhood is considered and the improvement in ς which would be made if this step was taken is recorded. The augmented system is then solved with a new right hand side, based on this value of ς, giving a *corrector* direction which is added to the predictor. Corrector directions can also be calculated which correct for errors introduced by linearization of the nonlinear complementarity constraints ($X–L^T Z = \mu e$, $U–X^T W = \mu e$).

The recent paper by Colombo & Gondzio [18], which introduces a new neighbourhood and includes further ideas relating to multiple centrality correctors, also provides a clear description of the various neighbourhoods of the central path and a thorough explanation of corrector directions.

5.2.2.2 Nonlinear programming

Unfortunately, the central path is not well-defined for nonlinear programming. However, it is possible that some of the techniques for choosing σ and μ which have been proven to be effective in LP/QP may also prove effective if applied to nonlinear programming problems. This could be done by attempting to choose μ such that all complementarity pairs ($(x_i - l_i)z_i$, $(u_i - x_i)w_i$) converge to zero uniformly. Questions which should be considered include:

- The amount of effort which should be applied to the search for a good centring direction for the QP approximation, remembering that it is only an approximation. It will be very interesting to research the progress in the NLP which can be achieved by using multiple centrality correctors whilst following the central path of a local QP approximation.
- Whether it would be advantageous to alternate NLP linesearches with choices of corrector directions, hoping to achieve better progress for each numerical factorization of the augmented system.
- If it would be a good strategy to keep μ constant until the KKT conditions have been satisfied to a given tolerance in the same way as KNITRO, KNITRO-Direct and IPOPT do.

5.2.3 Merit function

In `hopdmNLP`, we again choose to use the l_1 merit function. However, this is not entirely straightforward. Firstly, we are no longer solving the NLP problem (5.1) by taking steps calculated by `hopdm`. Instead, we are finding solutions to successive approximations to the barrier problem

$$\begin{array}{lrcll}
\min & \multicolumn{3}{l}{f(x) - \mu \sum_{i=1}^{n} \ln(l_i - x_i) - \mu \sum_{i=1}^{n} \ln(u_i - x_i)} & \\
\text{s.t.} & c_i(x) & = & 0 & i \in \mathcal{E} \\
 & c_i(x) & \geq & 0 & i \in \mathcal{I}_1 \\
 & c_i(x) & \leq & 0 & i \in \mathcal{I}_2.
\end{array}$$

The merit function must, therefore, be changed to include the barrier terms which are now present in the NLP objective function. That is, the l_1 merit function is of the form:

$$\Phi(x;\mu,\nu) = f(x) - \mu\sum_{i=1}^{n}\ln(x_i - l_i) - \mu\sum_{i=1}^{n}\ln(u_i - x_i) + \nu\sum_{i\in\mathcal{E}}|c_i(x)| + \nu\sum_{i\in\mathcal{I}_1}|c_i(x)|^- + \nu\sum_{i\in\mathcal{I}_2}|c_i(x)|^+,$$

where $|c_i(x)|^- = \max(0, -c_i(x))$ and $|c_i(x)|^+ = \max(0, c_i(x))$.

Secondly, it is necessary to remember that the l_1 merit function may be subject to the Maratos effect. In implementing a nonmonotone strategy (such as that of Grippo *et al.* [43]) to combat this effect, we should consider how to determine the values of $\Phi(x;\mu,\nu)$ from previous iterations which will be compared with those of the new trial points. In `hopdmSQP` we considered separating the storage of the objective values and constraint violations at each accepted point so that the merit value of a trial point ($\Phi(x_{trial};\nu_k)$) could be compared with the merit value of previous points with the same penalty for constraint violation[1] ($\Phi(x_{k-1};\nu_k)$, $\Phi(x_{k-2};\nu_k)$). Without this change to the way that points from different iterations are compared, the merit value of the trial point would be compared with the merit value of previous points with the penalty parameters appropriate to the iterations at which they were accepted ($\Phi(x_{k-1};\nu_{k-1})$, $\Phi(x_{k-2};\nu_{k-2})$).

In the case of `hopdmSQP`, this change in comparisons was not immediately successful and so it was not included in the solver which was used for our trials with `CUTE`. However, to implement a nonmonotone strategy for the barrier objective function, it will be vital to consider whether comparisons with points from previous iterations should be made with the current values of the barrier and penalty parameters.

5.2.4 Variables

A primal-dual interior point method works with variables which can be separated into different types. Here, we consider five types of variable and the different ways in which we propose to handle them in `hopdmNLP`. It is especially important to consider how the variables are initialized at the beginning of the algorithm, and how they are updated after each quadratic approximation is solved. In this section, we will also consider the problems which arise from using the augmented system approach when there are free or fixed variables in the problem formulation, as these can give rise to division by zero, or the inclusion of ∞ in the linear equations to be solved. (See Table 5.1.)

[1]This was change `J` in Table 3.1.

Primal variables, x**,** are initialized in the same way as for `hopdmSQP`, that is, they keep the initial value given by the problem, unless that value falls outside the variable bounds[2].

At each iteration, a backtracking linesearch is carried out to find a stepsize, α, which provides sufficient improvement in the merit function. Primal variables are updated using this value of α, so that $x_{k+1} = x_k + \alpha\Delta x_k$.

If primal variables are not free or fixed, then there are no difficulties in handling them. However, if they are free or fixed then special precautions must be taken to avoid numerical difficulties in the augmented system.

Handling fixed variables

`hopdm` deals with fixed variables ($l_i = u_i$) by removing them from the problem structure. The value of a fixed variable, x_{fi}, is assigned, $x_{fi} = u_i$. All constraints are updated to account for the removal of the fixed variable and right hand sides are adjusted. For example, a constraint $Ax + 3x_{fi} = b_1$ where $b_2 \le x_{fi} \le b_2$ becomes $Ax = b_1 - 3b_2$. Doing this, the ∞s which are added to the diagonal in positions corresponding to the fixed variables are no longer a problem. This can be seen if we separate the fixed variables from other variables in the augmented system as follows (using the subscript fi to denote quantities relating to the fixed variables):

$$\begin{bmatrix} -H\!\!L - \Theta^{-1} & & \nabla c^T(x) \\ & -H\!\!L - \infty & \nabla c^T_{fi}(x) \\ \nabla c(x) & \nabla c_{fi}(x) & \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta x_{fi} \\ \Delta\lambda \end{bmatrix} = \begin{bmatrix} \xi'_c \\ \xi'_{c_{fi}} \\ \xi_b \end{bmatrix}.$$

By removing the columns of the Jacobian which are associated with the fixed variables, adjusting the right hand side and noticing that $\Delta x_{fi} = 0$, since the variables are fixed, terms including ∞ are only present in redundant equations

$$\begin{bmatrix} -H\!\!L - \Theta^{-1} & & \nabla c^T(x) \\ & -H\!\!L - \infty & \\ \nabla c(x) & & \end{bmatrix} \begin{bmatrix} \Delta x \\ 0 \\ \Delta\lambda \end{bmatrix} = \begin{bmatrix} \xi'_c \\ 0 \\ \xi'_b \end{bmatrix}$$

and can be excluded from the calculations.

This technique works well in `hopdm` but it is worth noting that there are other methods for dealing with fixed variables. The QP solver `Loqo` [74], for example, adds a positive slack variable to a fixed variable ($x_{fi} + s_i = u_i$, $x_{fi} \ge 0$, $s_i \ge 0$), allowing x_{fi} to vary and hence removing the ∞s.

[2]The algorithm used to choose an initial value for the primal variables if they fall outside the variable bounds is given on page 27.

Handling free variables

`hopdm` handles free variables by splitting them into the difference of two positive variables

$$x_{fr} = x_{fr}^{+} - x_{fr}^{-},$$

$$x_{fr}^{+},\ x_{fr}^{-} \geq 0$$

which are then added to the Hessian and Jacobian matrices.

However, this technique appears time-consuming and less attractive when considering applying it to a succession of quadratic approximations which will each be solved only once. Also, it has been shown (Lustig *et al.* [53]) that both x_{fr}^{+} and x_{fr}^{-} may become extremely large whilst their difference remains bounded.

In [74], Vanderbei proposes two alternative techniques for handling free variables. In early versions of `Loqo`, diagonal elements of the augmented system which correspond to free variables and which could be zero because $\theta_i^{-1} = 0$ are given a low priority as pivot elements for the Cholesky decomposition, in the hope that earlier calculations will remove the zero on the diagonal and prevent numerical errors caused by an attempt to divide by that zero. In more recent versions, free variables are replaced by

$$x_{fr} + x_{fr}^{+} - x_{fr}^{-},$$

$$x_{fr}^{+},\ x_{fr}^{-} \geq 0$$

where the variables x_{fr}^{+} and x_{fr}^{-} contribute nonzero terms to θ^{-1} in the diagonal elements of the augmented system which correspond to x_{fr}, but do not add extra rows or columns to the Jacobian and Hessian in the way that a straightforward split of the variables does.

Dual variables associated with constraints, λ, are initialized according to whether the constraints they are associated with are violated or not.

```
If constraint $i$ is not violated
    $\lambda_i = 0$
Else if upper bound on constraint is violated.
    $\lambda_i = \vartheta$.[3]
Else
    $\lambda_i = -\vartheta$.
```

[3]ϑ is a small value, currently set to 1 here, and elsewhere in the variable initialization process.

At each iteration, λ is updated by taking a step of size α in the direction calculated by solution of the augmented system.

Slack variables, s**,** are initialized at every iteration as every QP approximation has different linearized constraints. If the linearized constraint is not violated at the current point, then the slack is chosen appropriately to cause the constraint to be exactly satisfied. If the constraint is violated, then the slack is given a positive value of ϑ. Care must be taken with range constraints.

However, if the slack variables (and their associated duals) are redefined at every iteration then any centrality suggested by the previous iteration's solution is lost. This has potential for causing the algorithm to stall, especially if elements of s are set close to zero. An alternative would be to initialize slack variables at the first iteration and then update them using $s_{k+1} = s_k + \alpha \Delta s_k$, finding α previously, using a backtracking linesearch.

Dual variables associated with bounds, z, w**,** are initialized to ensure uniformity between complementarity pairs $(x_i - l_i)z_i$ and $(u_i - x_i)w_i$. Currently, the bound dual values are chosen such that $(x_i - l_i)z_i = \mu$ and $(u_i - x_i)w_i = \mu$ for some chosen initial value of μ. The choice of z and w is important because the behaviour of interior point methods is very sensitive to the choice of starting point. Although this choice has been proposed as theoretically sound, more work needs to be carried out into what makes a good starting point in practice.

z and w associated with primal variables are updated at each iteration by taking a step of size α in the direction calculated by the solution of the augmented system. Dual variables, z, associated with slack variable bounds are chosen at each iteration so that the complementarity pairs $s_i z_i = \mu$.

Infeasibility variables, h**,** which were useful in `hopdmSQP` are no longer included in the problem formulation.

5.2.5 Dealing with nonconvexity

`hopdmSQP` deals with nonconvexity by testing the direction found by `hopdm` to see if it is a descent direction with respect to the chosen merit function and adding a regularizing term to the Hessian of the Lagrangian if it is not. In section 3.5 we conjectured that using the exact Hessian of the Lagrangian when it is not convex, with the risk of finding a nondescent direction, is an inefficient technique as it could result in the need to solve several systems of linear equations before a descent direction is found. However, if the solution found when using an exact $\mathbb{H}\mathbb{L}$ which is not positive definite is a descent direction, then it is likely to be a better direction than one chosen by using a positive definite approximation.

In Chapter 4 we investigated techniques for dealing with nonconvexity which are employed by successful interior point solvers from the literature. These techniques include adding a regularization term to $\mathbb{H}\mathbb{L}$ during the Cholesky factorization, removing any negative elements from the diagonal D; and analysing the inertia of the matrix to determine the size of the regularization term which needs to be added.

In implementing `hopdmNLP`, it will be necessary to consider whether one of these techniques should be automatically applied to ensure that a positive definite approximation to any indefinite $\mathbb{H}\mathbb{L}$ is used, or whether it would be efficient to first test to see whether the direction found using the exact $\mathbb{H}\mathbb{L}$ is descent, as is done in `hopdmSQP`.

5.3 Summary of possible future work

The implementation of `hopdmNLP` depends on several key decisions, mentioned with possible solutions in the above sections. Here, we gather together the aspects of the code which require further thought and which will benefit from comparison of various techniques once a working code is available.

- Choosing how to update the penalty parameter, μ, at each iteration.
- Deciding how to compare merit function values from different iterations, with regard to whether current or previous values of the parameters μ and ν should be used when comparing previous points with the current trial point.
- Deciding what method to use to deal with free variables in order to prevent division by zero in the Cholesky factorization when $\theta_i^{-1} = 0$.
- Determining an effective starting point, especially with regard to initialization of slack variables and dual variables associated with variable bounds.
- Selecting a technique to implement when the Hessian of the Lagrangian is not positive definite, and deciding whether to use that technique automatically, or to allow the system of equations to be solved with an indefinite Hessian because of the potential for greater accuracy if a descent direction is found when using the exact $\mathbb{H}\mathbb{L}$.

Chapter 6

Optimal Control Problems

This chapter introduces a group of optimization problems with specific properties, known as optimal control problems (OCPs). They tend to arise in dynamic systems where the user is required to operate the system *optimally* by choosing values of certain *controls*. Problems of this nature are typically found in systems where the motion of an object is to be controlled, but can also be found in other fields such as biology, chemistry and economics.

We describe the properties of a problem which make it classifiable as an OCP and describe traditional methods of solving such problems. We introduce the theory of Hamiltonians and Pontryagin's maximum principle, showing how Hamiltonians are derived through both calculus of variations and Newtonian mechanics. We then describe the way in which four practical problems can be modelled as optimal control problems and show how Hamiltonian theory can be used to find the solutions to these problems. In Chapter 7 we demonstrate how to represent optimal control problems as nonlinear programs so that the solvers described in preceding chapters can be used.

Much of the understanding in this chapter is taken from "*Optimal Control Theory*" by Kirk [52] and from Reitzenstein's Ph.D. thesis [68]. Throughout the next two chapters, vectors $\mathbf{x} = (x_1, x_2, \ldots x_n)$ will be denoted in boldface.

6.1 Optimal control theory

Using the definition given by Kirk, an optimal control problem is to find an *admissible control*, $\mathbf{u}$, which causes the system to follow an *admissible trajectory*, $\mathbf{x}$, that minimizes[1] a performance measure. $\mathbf{x}^*$ is called an *optimal trajectory* and $\mathbf{u}^*$ is called an *optimal control.*

[1] Again, we will only consider minimization problems in this chapter, as the techniques are easily reversed to solve maximization problems.

An optimal control problem is of the form

$$\begin{aligned} \min \quad J = \quad & \phi(\mathbf{x}(t_F), t_F) + \int_{t_0}^{t_F} F(\mathbf{x}(t), \mathbf{u}(t), t)\; dt && \text{(6.1a)} \\ \text{s.t.} \quad \dot{\mathbf{x}}(t) = \quad & f(\mathbf{x}(t), \mathbf{u}(t), t) && \text{(6.1b)} \\ & \mathbf{x}(t) \in X,\ \ \mathbf{u}(t) \in U\ \forall t, \end{aligned}$$

where the sets X and U contain all admissible values of the state and control vectors respectively, and may include initial or terminal conditions, or both. The performance measure J (6.1a) is a functional which is made up of two parts. $\phi(\mathbf{x}(t_F), t_F)$ represents goals for the final state of the system and $F(\mathbf{x}(t), \mathbf{u}(t), t)$ represents goals which are aimed at for the duration of the process to be controlled. If J is in the form (6.1a) then it is said to be in *Bolza form.* If $\phi \equiv 0$ then J is in *Lagrange form* and if $F \equiv 0$ it is in *Mayer form.* J can represent a variety of different targets, examples of which are given here, with possible performance measures:

Minimum time: On a racing straight a car needs to get from point A to point B in the smallest possible time.
$J = t_F$ is in Mayer form.
$J = \int_{t_0}^{t_F} 1\; dt$ is in Lagrange form.

Minimum control effort: On an interplanetary exploration $u(t)$ is the thrust of the engine and fuel consumption is proportional to the thrust. The aim is to use as little fuel as possible.
$J = \int_{t_0}^{t_F} |u(t)|\; dt$ or $J = \int_{t_0}^{t_F} (u(t))^2\; dt$ are in Lagrange form.

Terminal control: When firing a missile it is important that it does not deviate from its target (see Figure 6.1).
$J = |\mathbf{x}_F - \mathbf{r}_F|$ or $J = (\mathbf{x}_F - \mathbf{r}_F)^2$, where $\mathbf{x}_F$ is position at final time and $\mathbf{r}_F$ is desired position at final time, are in Mayer form.

When hitting a golf ball towards a hole, it is important that it reaches the hole with a small speed so that it falls in.
$J = \sqrt{v_{x_1}(t_F)^2 + v_{x_2}(t_F)^2}$ is in Mayer form.
v_{x_1} and v_{x_2} are components of velocity.

Regulator: A manned spacecraft's angular position, $\theta(t)$, is to be maintained near 0.
$J = \int_0^\infty [a\theta(t) + b\dot{\theta(t)}]\; dt$ is in Lagrange form.
a and b are weighting factors.
In choosing the performance measure it is possible to include a term to minimize control effort. That is, to use a *mixed* objective.

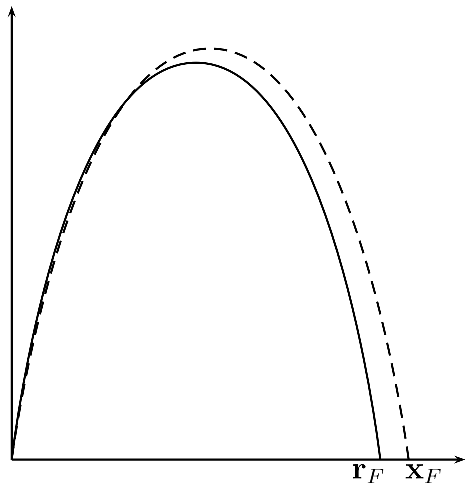


Figure 6.1: Showing a desired missile trajectory, and how a slight change in the angle it is projected with can cause a significant error in the final position. The solid line shows a trajectory to the target point $\mathbf{r}_F$ and the dashed line shows a trajectory given by a small error in the projection angle.

Tracking: A system $\mathbf{x}(t)$ must be maintained as close to a desired state $\mathbf{r}(t)$ as possible in the time period $[t_0, t_F]$.
$J = \int_{t_0}^{t_F} \|\mathbf{x}(t) - \mathbf{r}(t)\|^2_{W(t)}$, where $W(t)$ is a matrix selected to weight the importance of different components of the system, is in Lagrange form.

A nontrivial part of finding the solution to any OCP is the model chosen to represent the process. This includes determining an appropriate performance measure to represent the aim of the physical system.

6.2 Calculus of variations

As previously stated, this chapter will show how Hamiltonians have been used to solve OCPs, before we demonstrate how to reformulate OCPs as NLPs in Chapter 7. We begin our introduction to Hamiltonian theory by explaining the basics of calculus of variations, a technique whose usage dates back to Queen Dido of Carthage in 814BC.

Calculus of variations is used to find the maxima or minima of functionals which are defined similarly to J when it is in the Lagrange form:

$$J = \int_{t_0}^{t_F} F(\mathbf{x}(t), \mathbf{u}(t), t)\ dt.$$

However, instead of defining J in terms of the control variable $\mathbf{u}$, it is defined in terms of the first derivative of the state variable:

$$J = \int_{t_0}^{t_F} F(\mathbf{x}(t), \dot{\mathbf{x}}(t), t)\ dt$$

and $\mathbf{x}$ is no longer written as an explicit function of t:

$$J = \int_{t_0}^{t_F} F(\mathbf{x}, \dot{\mathbf{x}}, t)\ dt.$$

6.2.1 Euler-Lagrange equations

To find the extrema of these functionals, J, we use the *Euler-Lagrange* equations which we derive here following the steps taken in Arthurs [3].

If we choose a function $\boldsymbol{\varphi}$ which is continuously differentiable with respect to t and a small constant $\epsilon > 0$ and assume that F is continuously differentiable with respect to $\mathbf{x}$ and $\dot{\mathbf{x}}$, we can expand $J(\mathbf{x} + \epsilon\boldsymbol{\varphi})$ in a Taylor series to get

$$\begin{aligned} J(\mathbf{x} + \epsilon\boldsymbol{\varphi}) &= \int_{t_0}^{t_F} F(\mathbf{x} + \epsilon\boldsymbol{\varphi}, \dot{\mathbf{x}} + \epsilon\dot{\boldsymbol{\varphi}}, t)\ dt \\ &= \int_{t_0}^{t_F} \left\{ F(\mathbf{x}, \dot{\mathbf{x}}, t) + \epsilon\boldsymbol{\varphi}\frac{\partial F}{\partial \mathbf{x}}(\mathbf{x}, \dot{\mathbf{x}}, t) + \epsilon\dot{\boldsymbol{\varphi}}\frac{\partial F}{\partial \dot{\mathbf{x}}}(\mathbf{x}, \dot{\mathbf{x}}, t) + O(\epsilon^2) \right\}\ dt \\ &= J(\mathbf{x}) + \epsilon\delta J + O(\epsilon^2), \end{aligned}$$

where

$$\delta J = \int_{t_0}^{t_F} \left\{ \boldsymbol{\varphi}\frac{\partial F}{\partial \mathbf{x}}(\mathbf{x}, \dot{\mathbf{x}}, t) + \dot{\boldsymbol{\varphi}}\frac{\partial F}{\partial \dot{\mathbf{x}}}(\mathbf{x}, \dot{\mathbf{x}}, t) \right\}\ dt$$

denotes the linear term and is called the *first variation of J.*

By integrating the second term of δJ by parts and simplifying by assuming that both endpoints are fixed ($\boldsymbol{\varphi}(t_0) = 0$ and $\boldsymbol{\varphi}(t_F) = 0$) we get that

$$\delta J = \int_{t_0}^{t_F} \boldsymbol{\varphi} \left\{ \frac{\partial F}{\partial \mathbf{x}}(\mathbf{x}, \dot{\mathbf{x}}, t) - \frac{d}{dt}\frac{\partial F}{\partial \dot{\mathbf{x}}}(\mathbf{x}, \dot{\mathbf{x}}, t) \right\}\ dt$$

The following theorem for optimization of functionals is analogous to theorem 2.1 for optimization of functions.

Theorem 6.1. *A necessary condition for J to have an extremum at* $\mathbf{x}^*$ *is that* $\mathbf{x}^*$ *be a solution of*

$$\delta J = \int_{t_0}^{t_F} \boldsymbol{\varphi} \left\{ \frac{\partial F}{\partial \mathbf{x}}(\mathbf{x}, \dot{\mathbf{x}}, t) - \frac{d}{dt}\frac{\partial F}{\partial \dot{\mathbf{x}}}(\mathbf{x}, \dot{\mathbf{x}}, t) \right\}\ dt = 0.$$

By showing that if a function $h(t)$ is continuous and

$$\epsilon \int_{t_0}^{t_F} \boldsymbol{\varphi} h(t)\ dt = 0$$

for every function $\boldsymbol{\varphi}$ that is continuous in the interval $[t_0, t_F]$ it can be proved that $h(t)$ must be zero everywhere in the interval $[t_0, t_F]$.

Then, replacing $h(t)$ with δJ, we get that

Theorem 6.2. *A necessary condition for J to have an extremum at $\mathbf{x}^*$ is that $\mathbf{x}^*$ be a solution of*

$$\frac{\partial F}{\partial \mathbf{x}}(\mathbf{x}, \dot{\mathbf{x}}, t) - \frac{d}{dt}\frac{\partial F}{\partial \dot{\mathbf{x}}}(\mathbf{x}, \dot{\mathbf{x}}, t) = 0 \qquad t_0 \le t \le t_F \tag{6.2}$$

with $\mathbf{x}(t_0) = \mathbf{x}_0$ and $\mathbf{x}(t_F) = \mathbf{x}_F$.

The *Euler-Lagrange* equations (6.2) can be very difficult to solve. Generally, they are nonlinear, ordinary second-order differential equations.

6.3 Hamiltonians

It is possible to rewrite the second order *Euler-Lagrange* equations formed through calculus of variations as a system of twice as many first order differential equations. It is done by using the concept of a *Hamiltonian.* We demonstrate the format of a Hamiltonian in Newtonian mechanics before showing how the *Euler-Lagrange* equations can be translated into the same structure and how this structure can be used to solve OCPs.

6.3.1 In Newtonian mechanics

In standard mechanics, an autonomous (independent of time) *Newtonian* system (which follows Newton's Laws) is also called *Hamiltonian*, and has a *Hamiltonian* function whose value is always conserved and equal to the energy value of the Newtonian system.

This can be explained as follows (derivation taken from Percival & Richards [64]). If x is the displacement of a particle of mass m in a given direction, λ is its linear momentum in that direction and $F(x,t)$ is the force on the particle then the equation of motion and definition of momentum (momentum $= mv$ where v is the speed of the particle in the given direction) are given by the equations

$$\begin{aligned} \dot{x} &= \frac{\lambda}{m} \\ \dot{\lambda} &= F(x,t). \end{aligned}$$

Then if we define

$$V(x,t) = -\int_{x_0}^{x} F(x,t)\ dx$$

we can define the *Hamiltonian* as

$$H(x,\lambda,t) = \frac{\lambda^2}{2m} + V(x,t)$$

so that the equations of motion and definition of momentum can be rewritten

$$\dot{x} = \frac{dx}{dt} = \frac{\partial H}{\partial \lambda}(x,\lambda,t) \tag{6.3a}$$

$$\dot{\lambda} = \frac{d\lambda}{dt} = -\frac{\partial H}{\partial x}(x,\lambda,t) \tag{6.3b}$$

It is then possible to show that the value of the *Hamiltonian* is conserved, by taking the derivative of the *Hamiltonian* and using equations (6.3):

$$\frac{dH}{dt} = \frac{\partial H}{\partial x}\dot{x} + \frac{\partial H}{\partial \lambda}\dot{\lambda} = \frac{\partial H}{\partial x}\frac{dx}{dt} + \frac{\partial H}{\partial \lambda}\frac{d\lambda}{dt} = \frac{\partial H}{\partial x}\frac{\partial H}{\partial \lambda} + \frac{\partial H}{\partial \lambda}\left(-\frac{\partial H}{\partial x}\right) = 0.$$

6.3.2 In calculus of variations

Now we can see how it is possible to use the format of the *Hamiltonian* to solve problems in calculus of variations. This derivation is taken from Arthurs [3] which shows how to reduce the system of *Euler-Lagrange* equations (6.2) to the system of first order differential equations (6.3).

We introduce a new variable

$$\boldsymbol{\lambda} = \frac{\partial F}{\partial \dot{\mathbf{x}}}(\mathbf{x},\dot{\mathbf{x}},t) \tag{6.4}$$

which is said to be conjugate to $\mathbf{x}$. Assuming that (6.4) can be solved to give $\dot{\mathbf{x}}$ as a function of t, $\mathbf{x}$ and $\boldsymbol{\lambda}$ then we can define a *Hamiltonian* by the equation

$$H(\mathbf{x},\boldsymbol{\lambda},t) = \boldsymbol{\lambda}^T\dot{\mathbf{x}} - F(\mathbf{x},\dot{\mathbf{x}},t)$$

Considering the differential of H, which is given by

$$dH = \boldsymbol{\lambda}^T d\dot{\mathbf{x}} + \dot{\mathbf{x}}^T d\boldsymbol{\lambda} - \frac{\partial F}{\partial t}(\mathbf{x},\dot{\mathbf{x}},t)dt - \left(\frac{\partial F}{\partial \mathbf{x}}^T(\mathbf{x},\dot{\mathbf{x}},t)d\mathbf{x} + \frac{\partial F}{\partial \dot{\mathbf{x}}}^T(\mathbf{x},\dot{\mathbf{x}},t)d\dot{\mathbf{x}}\right)$$

$$= -\frac{\partial F}{\partial t}(\mathbf{x},\dot{\mathbf{x}},t)dt + \dot{\mathbf{x}}^T d\boldsymbol{\lambda} - \frac{\partial F}{\partial \mathbf{x}}^T(\mathbf{x},\dot{\mathbf{x}},t)d\mathbf{x}$$

(cancelling terms in $d\dot{\mathbf{x}}$ because of definition of $\boldsymbol{\lambda}$ (equation (6.4)) we find that

$$\frac{d\mathbf{x}}{dt} = \frac{\partial H}{\partial \boldsymbol{\lambda}}(\mathbf{x},\boldsymbol{\lambda},t) \tag{6.5a}$$

$$-\frac{d\boldsymbol{\lambda}}{dt} = \frac{\partial H}{\partial \mathbf{x}}(\mathbf{x},\boldsymbol{\lambda},t), \tag{6.5b}$$

where the second of these equations comes from observing that $\frac{\partial H}{\partial \mathbf{x}}(\mathbf{x}, \boldsymbol{\lambda}, t) = -\frac{\partial F}{\partial \mathbf{x}}^T(\mathbf{x}, \dot{\mathbf{x}}, t)$ and using both the *Euler-Lagrange* equations (6.2) and the definition of $\boldsymbol{\lambda}$.

Equations (6.5) are known, in this setting, as the *canonical Euler equations* and can be seen to be equivalent to the equation of motion and definition of momentum (6.3).

These are a system of one dimensional differential equations which should be easier to solve than the *Euler-Lagrange* equations.

6.3.3 In optimal control theory

The *Hamiltonian* function can be used, by way of Pontryagin's *maximum principle* [65] (described later), to help solve problems arising from optimal control theory. We will look at some examples of this later, but first look at how the *Hamiltonian* function for optimal control theory is defined. Most of this definition is taken from Kirk [52].

First we use the method of Lagrange multipliers to attach a measure for violation of constraints (6.1b) to the performance measure J (6.1a), such that

$$J_a = \phi(\mathbf{x}(t_F), t_F) + \int_{t_0}^{t_F} F(\mathbf{x}(t), \mathbf{u}(t), t)\ dt - \int_{t_0}^{t_F} \boldsymbol{\lambda}^T \left(\dot{\mathbf{x}} - f(\mathbf{x}(t), \mathbf{u}(t), t)\right)\ dt,$$

where the Lagrange multipliers $\boldsymbol{\lambda}$ are also known as *adjoint variables.*

Now it is possible to rewrite performance measures which are written in Bolza form as measures in Lagrange or Mayer form. In this case, we will rewrite the terms in ϕ so that they are inside the integral and J_a is in Lagrange form.

So, taking

$$\phi(\mathbf{x}(t_F), t_F) = \phi(\mathbf{x}(t_0), t_0) + \int_{t_0}^{t_F} \frac{d}{dt}\left[\phi(\mathbf{x}(t), t)\right] dt,$$

where we can ignore the term $\phi(\mathbf{x}(t_0), t_0)$ because it does not affect the minimization as $\mathbf{x}(t_0)$ and t_0 are fixed, we can write

$$J_a = \int_{t_0}^{t_F} \left\{ F(\mathbf{x}(t), \mathbf{u}(t), t) - \boldsymbol{\lambda}^T \left(\dot{\mathbf{x}} - f(\mathbf{x}(t), \mathbf{u}(t), t)\right) + \frac{d}{dt}\left[\phi(\mathbf{x}(t), t)\right] \right\}\ dt$$

or, by using the chain rule of differentiation on terms in ϕ and writing

$$\begin{aligned} F_a(\mathbf{x}(t), \dot{\mathbf{x}}(t), \mathbf{u}(t), \boldsymbol{\lambda}(t), t) \ \equiv\ & F(\mathbf{x}(t), \mathbf{u}(t), t) - \boldsymbol{\lambda}^T \left(\dot{\mathbf{x}} - f(\mathbf{x}(t), \mathbf{u}(t), t)\right) \\ & + \left[\frac{\partial \phi}{\partial \mathbf{x}}(\mathbf{x}(t), t)\right]^T \dot{\mathbf{x}}(t) + \frac{\partial \phi}{\partial t}(\mathbf{x}(t), t), \end{aligned}$$

we can further simplify the notation to

$$J_a = \int_{t_0}^{t_F} F_a(\mathbf{x}(t), \dot{\mathbf{x}}(t), \mathbf{u}(t), \boldsymbol{\lambda}(t), t)\ dt.$$

Now we know from Theorem 6.1 that for an extremal, the first variation $\delta J_a = 0$. Thus, if we introduce variations $\delta\mathbf{x}$, $\delta\dot{\mathbf{x}}$, $\delta\mathbf{u}$, and $\delta\boldsymbol{\lambda}$ we have the following condition for an extremal:

$$\begin{aligned}\delta J_a = \int_{t_0}^{t_F} \Bigg\{ & \left[\frac{\partial F_a}{\partial \mathbf{x}}(\mathbf{x}(t), \dot{\mathbf{x}}(t), \mathbf{u}(t), \boldsymbol{\lambda}(t), t)\right]^T \delta\mathbf{x}(t) \\ & + \left[\frac{\partial F_a}{\partial \dot{\mathbf{x}}}(\mathbf{x}(t), \dot{\mathbf{x}}(t), \mathbf{u}(t), \boldsymbol{\lambda}(t), t)\right]^T \delta\dot{\mathbf{x}}(t) \\ & + \left[\frac{\partial F_a}{\partial \mathbf{u}}(\mathbf{x}(t), \dot{\mathbf{x}}(t), \mathbf{u}(t), \boldsymbol{\lambda}(t), t)\right]^T \delta\mathbf{u}(t) \\ & + \left[\frac{\partial F_a}{\partial \boldsymbol{\lambda}}(\mathbf{x}(t), \dot{\mathbf{x}}(t), \mathbf{u}(t), \boldsymbol{\lambda}(t), t)\right]^T \delta\boldsymbol{\lambda}(t) \Bigg\}\, dt \\ = 0. &\end{aligned}$$

By integrating the second term of δJ_a by parts and assuming that endpoints are fixed ($\delta\mathbf{x}(t_0) = \delta\mathbf{x}(t_F) = 0$), similarly to the derivation of the *Euler-Lagrange* equations in section 6.2.1, we get

$$\begin{aligned}\delta J_a = \int_{t_0}^{t_F} \Bigg\{ & \Bigg[\left[\frac{\partial F_a}{\partial \mathbf{x}}(\mathbf{x}(t), \dot{\mathbf{x}}(t), \mathbf{u}(t), \boldsymbol{\lambda}(t), t)\right]^T \\ & \quad - \frac{d}{dt}\left[\frac{\partial F_a}{\partial \dot{\mathbf{x}}}(\mathbf{x}(t), \dot{\mathbf{x}}(t), \mathbf{u}(t), \boldsymbol{\lambda}(t), t)\right]^T\Bigg] \delta\mathbf{x}(t) \\ & + \left[\frac{\partial F_a}{\partial \mathbf{u}}(\mathbf{x}(t), \dot{\mathbf{x}}(t), \mathbf{u}(t), \boldsymbol{\lambda}(t), t)\right]^T \delta\mathbf{u}(t) \\ & + \left[\frac{\partial F_a}{\partial \boldsymbol{\lambda}}(\mathbf{x}(t), \dot{\mathbf{x}}(t), \mathbf{u}(t), \boldsymbol{\lambda}(t), t)\right]^T \delta\boldsymbol{\lambda}(t) \Bigg\}\, dt \\ = 0. &\end{aligned}$$

By considering only the terms in F_a which include ϕ we can establish that they add to zero in all cases. Then, separating terms back out from the definition of F_a, we retain:

$$\begin{aligned}0 = \int_{t_0}^{t_F} \Bigg\{ & \Bigg[\left[\frac{\partial F}{\partial \mathbf{x}}(\mathbf{x}(t), \mathbf{u}(t), t)\right]^T + \boldsymbol{\lambda}^T(t)\left[\frac{\partial f}{\partial \mathbf{x}}(\mathbf{x}(t), \mathbf{u}(t), t)\right] - \frac{d}{dt}\left[-\boldsymbol{\lambda}^T(t)\right]\Bigg] \delta\mathbf{x}(t) \\ & + \Bigg[\left[\frac{\partial F}{\partial \mathbf{u}}(\mathbf{x}(t), \mathbf{u}(t), t)\right]^T + \boldsymbol{\lambda}^T(t)\left[\frac{\partial f}{\partial \mathbf{u}}(\mathbf{x}(t), \mathbf{u}(t), t)\right]\Bigg] \delta\mathbf{u}(t) \\ & + \left[[f(\mathbf{x}(t), \mathbf{u}(t), t) - \dot{\mathbf{x}}(t)]^T\right] \delta\boldsymbol{\lambda}(t) \Bigg\}\, dt.\end{aligned}$$

We first observe that the constraints

$$\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}(t), \mathbf{u}(t), t)$$

must be satisfied at an extremal, so the coefficient of $\delta\boldsymbol{\lambda}(t)$ is 0. The Lagrange multipliers $\boldsymbol{\lambda}(t)$ are arbitrary, so can be chosen to make the coefficients of $\delta\mathbf{x}(t) = 0$, that is

$$\dot{\boldsymbol{\lambda}}(t) = -\frac{\partial F}{\partial \mathbf{x}}(\mathbf{x}(t), \mathbf{u}(t), t) - \left[\frac{\partial f}{\partial \mathbf{x}}(\mathbf{x}(t), \mathbf{u}(t), t)\right]^T \boldsymbol{\lambda}(t). \tag{6.6}$$

The remaining variation $\delta\mathbf{u}(t)$ is independent, so its coefficients must be 0

$$0 = \frac{\partial F}{\partial \mathbf{u}}(\mathbf{x}(t), \mathbf{u}(t), t) + \left[\frac{\partial f}{\partial \mathbf{u}}(\mathbf{x}(t), \mathbf{u}(t), t)\right]^T \boldsymbol{\lambda}(t).$$

These conditions can be extended to the cases where the final time and point are not fixed. (See [52] for further details.)

If we now define a *Hamiltonian*

$$H(\mathbf{x}(t), \mathbf{u}(t), \boldsymbol{\lambda}(t), t) \equiv F(\mathbf{x}(t), \mathbf{u}(t), t) + \boldsymbol{\lambda}^T(t)\,[f(\mathbf{x}(t), \mathbf{u}(t), t)] \tag{6.7}$$

then we can see that equations (6.3) and (6.5) are echoed here

$$\dot{\mathbf{x}} = \frac{dx}{dt} = \frac{\partial H}{\partial \lambda}(\mathbf{x}(t), \mathbf{u}(t), \boldsymbol{\lambda}(t), t) \tag{6.8a}$$

$$\dot{\boldsymbol{\lambda}} = \frac{d\lambda}{dt} = -\frac{\partial H}{\partial x}(\mathbf{x}(t), \mathbf{u}(t), \boldsymbol{\lambda}(t), t) \tag{6.8b}$$

and can be used to solve the optimal control problem.

We also have the equation from the $\delta\mathbf{u}(t)$ coefficient, which can be written as

$$0 = \frac{\partial H}{\partial \mathbf{u}}(\mathbf{x}(t), \mathbf{u}(t), \boldsymbol{\lambda}(t), t)$$

and which is the property of the *Hamiltonian* function which is used in Pontryagin's *maximum principle.* The *maximum principle* is more generally written as

$$\mathbf{u}(t) = \arg\max_{\mathbf{u}\in U} H(\mathbf{x}(t), \mathbf{u}(t), \boldsymbol{\lambda}(t), t)$$

and formally (from Pontryagin *et al.* [65]) as

Theorem 6.3. *Let $\mathbf{u}(t)$, $t_0 \leq t \leq t_F$, be an admissible control which transfers the phase point from the position $\mathbf{x}_0$ at time t_0 to some position $\mathbf{x}_F$ which is defined either*

a) as a specific point $\mathbf{x}_F = \mathbf{x}_f$

b) within a given set $\mathbf{x}_F \in X_F$

c) as free

and let $\mathbf{x}(t)$ *be the corresponding trajectory. In order that* $\mathbf{u}(t)$ *and* $\mathbf{x}(t)$ *be optimal it is necessary that there exist a nonzero continuous vector function* $\boldsymbol{\lambda}(t)$ *(see* (6.6)*) such that for every* $t \in [t_0, t_F]$ *the function* $H(\mathbf{x}(t), \mathbf{u}(t), \boldsymbol{\lambda}(t), t)$ *attains its maximum at the point* $\mathbf{u}(t)$ *where* $\mathbf{u}(t)$ *is chosen to maximize* $H(\mathbf{x}(t), \mathbf{u}(t), \boldsymbol{\lambda}(t), t)$ $\forall t$*.*

The sense of Theorem 6.3 can easily be reversed to consider a minimization requirement on J, i.e. $\mathbf{u}(t)$ chosen such that $\mathbf{u}(t) = \arg\min_{\mathbf{u}\in U} H(\mathbf{x}(t), \mathbf{u}(t), \boldsymbol{\lambda}(t), t)$.

Generally in optimal control, the optimal state trajectory $\mathbf{x}^*(t)$ can be calculated by integration once the optimal control trajectory $\mathbf{u}^*(t)$ has been determined.

6.4 Examples

We will now show three examples of the definition of a *Hamiltonian* used to solve problems with practical application. They are taken from Reitzenstein's work on mountain pass problems [68], Jan Olsder's work on bicycle routing [63] and Hennessey *et al.*'s work on sailing in steady winds [46].

6.4.1 Mountain pass

6.4.1.1 The problem

The mountain pass problem is very comprehensively described by Moré and Munson in [58].

Mountain passes are paths across a region defined by a continuously differentiable function, $f(\mathbf{x})$, which join two points with specific properties. At one end of the mountain pass is a point, $\mathbf{x}_a$, which is a local minimizer of the function, at the other is a point, $\mathbf{x}_b$, with a lower function value than that of the local minimizer.

These definitions ensure that any path joining $\mathbf{x}_a$ and $\mathbf{x}_b$ crosses a point with a function value higher than that of either of the end points. Each possible connecting path has a maximum point, in terms of the function evaluated across the length of the path. The optimal path searched for is the one which has the lowest maximum.

More formally, the mountain pass problem is to find ω where

$$\omega = \inf_{x\in\Omega}\{\max\{f(\mathbf{x}(t)) : t \in [0,1]\}\}$$

and Ω is the set of all paths which connect $\mathbf{x}_a$ with $\mathbf{x}_b$ ($\mathbf{x}(0) = \mathbf{x}_a$, $\mathbf{x}(1) = \mathbf{x}_b$).

This can also be solved by defining a new variable

$$\varpi = \max_{t\in[0,1]} f(\mathbf{x}(t))$$

and minimizing over all possible paths, $\mathbf{x} \in \Omega$.

Reitzenstein [68] reformulates the mountain pass problem as an optimal control problem in several ways, the first and simplest being

$$\begin{array}{lrcll} \min & & & \varpi & \\ \text{subject to} & \dot{\mathbf{x}}(t) & = & \mathbf{u}(t) & \\ & f(\mathbf{x}(t)) & \leq & \varpi & \forall t \in [0,1] \\ & \mathbf{x}(0) = \mathbf{x}_a, & & \mathbf{x}(1) = \mathbf{x}_b, & \end{array}$$

where $\mathbf{u}(t)$, the control variable, is a velocity vector which steers the trajectory. Control variables are bounded, i.e. $|u_j(t)| \leq r_j,\ j = 1, 2, \ldots, n,\ \forall t \in [0,1]$

Alternative formulations given include defining the control variable $\mathbf{u}(t)$ to represent acceleration and including a regularization term in the performance measure to help find a unique, smooth solution.

6.4.1.2 The Hamiltonian

The *Hamiltonian* for Reitzenstein's first formulation of the mountain pass problem is determined here. $F(\mathbf{x}(t), \mathbf{u}(t), t) = 0$ (the performance measure is in Mayer form) and $\mathbf{f}(\mathbf{x}(t), \mathbf{u}(t), t) = \mathbf{u}(t)$ so the *Hamiltonian* is

$$H(\mathbf{x}(t), \mathbf{u}(t), \boldsymbol{\lambda}(t), t) = \boldsymbol{\lambda}(t)^T \mathbf{u}(t)$$

In order to solve the problem, measures must also be taken to include the constraint $f(x(t)) \leq \varpi\ \forall t \in [0,1]$ using Lagrange multipliers, but the details are beyond the scope of this discussion.

6.4.2 Bicycle

6.4.2.1 The problem

The problem considered by Jan Olsder in [63] is that of a cyclist who leaves his house at sunrise and cycles throughout the day at a constant positive speed in such a way that he returns to his house at sunset. The objective is to maximize his suntan by cycling into the sun as much as possible.

If the land has a standard (x_1, x_2) coordinate system (the positive x_2 direction points north, the positive x_1 direction east) with the house of the cyclist at the origin, the equations of motion of cyclist and bicycle, with initial and final conditions are

$$\begin{array}{rclrcl} \dot{x_1}(t) & = & \cos\theta(t), & x_1(0) & = & x_1(\pi) = 0 \\ \dot{x_2}(t) & = & \sin\theta(t), & x_2(0) & = & x_2(\pi) = 0, \end{array}$$

where time, t, is scaled in such a way that sunrise occurs at $t = 0$ and sunset at $t = \pi$ and the cycling direction, $\theta(t)$, is a function of time.

To simplify the problem, assumptions are made regarding flatness of the land, lack of obstacles, length of the day, motion of the sun and the projection of the sun onto the horizontal plane.

The problem is defined as follows:

$$\begin{array}{rrcccl}
\max & J(\theta) & = & \displaystyle\int_0^\pi \cos(t)\cos(\theta(t)) - \sin(t)\sin(\theta(t))\ dt = \int_0^\pi \cos(t+\theta(t))\ dt \\
\text{s.t.} & \dot{x_1}(t) & = & \cos(\theta(t)) \\
& \dot{x_2}(t) & = & \sin(\theta(t)) \\
& x_1(0) & = & x_1(\pi) & = & 0 \\
& x_2(0) & = & x_2(\pi) & = & 0,
\end{array}$$

where the integral represents the inner product between the direction of cycling and the direction of the sun and the differential equation constraints represent the motion of the cyclist and bicycle, given that the cyclist has a unit speed.

6.4.2.2 The Hamiltonian

To define this problem in terms of the *Hamiltonian*, with the control variable $\mathbf{u}(t) = \theta(t)$, we note that $F(\mathbf{x}(t), \theta(t), t) = \cos(t)\cos(\theta(t)) - \sin(t)\sin(\theta(t))$, and $\mathbf{f}(\mathbf{x}(t), \theta(t), t) = \begin{bmatrix} \cos(\theta(t)) \\ \sin(\theta(t)) \end{bmatrix}$.
The *Hamiltonian* is

$$\begin{aligned}
H(\mathbf{x}(t), \theta(t), \boldsymbol{\lambda}(t), t) &= \cos(t)\cos(\theta(t)) - \sin(t)\sin(\theta(t)) \\
&\qquad + \lambda_1(t)\cos(\theta(t)) + \lambda_2(t)\sin(\theta(t)) \\
&= \cos(\theta(t))[\lambda_1(t) + \cos(t)] + \sin(\theta(t))[\lambda_2(t) - \sin(t)].
\end{aligned}$$

Using the second of the canonical equations (6.3), (6.5), (6.8) it can be seen that

$$\dot{\boldsymbol{\lambda}}(t) = 0 \Rightarrow \boldsymbol{\lambda}(t) = \boldsymbol{\lambda} \text{ (a constant).}$$

The *Hamiltonian* must then be maximized with respect to the control variable, $\theta(t)$, in order to find the optimal trajectory for the cyclist. The *maximum principle* is used to show that $\theta(t)$ should be chosen such that

$$\begin{pmatrix} \cos(\theta(t)) \\ \sin(\theta(t)) \end{pmatrix} \Big|\Big| \begin{pmatrix} \lambda_1 + \cos(t) \\ \lambda_2 - \sin(t) \end{pmatrix},$$

where the symbol || means “is parallel to”.

See [63] for more details and versions of the problem with different assumptions and emphases.

6.4.3 Sailing

6.4.3.1 The problem

The sailing problems, known generally as Zermelo [83] problems and discussed by Bryson & Ho [15], take a boat which is initially at the origin and try to find the shortest time possible to reach a target zone. The velocity of the boat is determined by the strength and direction of water currents and the wind.

The case considered by Hennessey *et al.* [46] is of the same form. The speed of the boat relative to the angle between the direction it sails in and the direction of the wind is determined by the use of a *wind polar*, as shown in Figure 6.2.

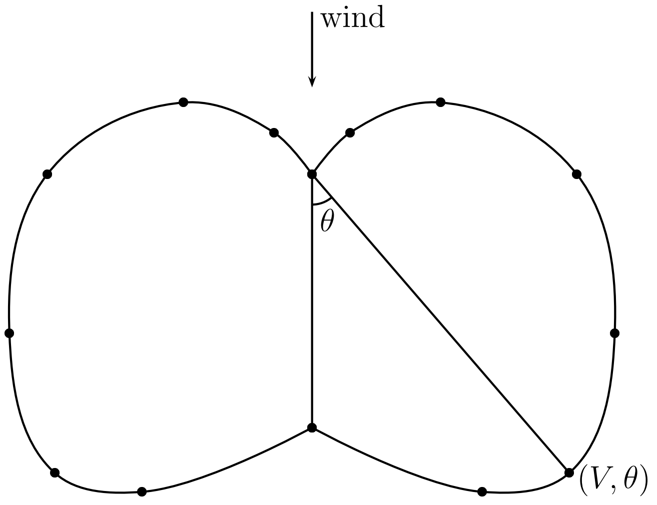


Figure 6.2: Wind polar, showing speed (V) and angle between boat and wind (θ) in polar coordinates.

The boat has a constant velocity relative to the water flow.

In the simplest problem considered, the wind field is considered to be constant and the water is assumed to be still. Given a standard (x_1, x_2) coordinate system, the equations of motion for this case are given by

$$\begin{aligned} \dot{x}_1(t) &= V(\theta(t))\cos(u(t)) \\ \dot{x}_2(t) &= V(\theta(t))\sin(u(t)), \end{aligned}$$

where $u(t)$ is the angle between the x_1 axis and the direction of the boat at time t and is the variable which can be *controlled* by the sailor. $\theta(t)$ can be determined from $u(t)$ when the wind speed is known.

The problem to be solved is

$$\begin{array}{lrcl} \min & t_F & = & \int_0^{t_F} 1 \; dt \\ \text{s.t.} & \dot{x}_1(t) & = & V(\theta(t))\cos(u(t)) \\ & \dot{x}_2(t) & = & V(\theta(t))\sin(u(t)) \\ & \mathbf{x}(0) & = & \mathbf{x}_0 \\ & \mathbf{x}(t_F) & = & \mathbf{x}_F, \end{array}$$

where the initial and final conditions $\mathbf{x}_0$ and $\mathbf{x}_F$ represent a fixed starting point and a target zone.

6.4.3.2 The Hamiltonian

To define this problem in terms of the *Hamiltonian*, we note that $F(\mathbf{x}(t), \mathbf{u}(t), t) = 1$ and $\mathbf{f}(\mathbf{x}(t), \mathbf{u}(t), t) = \begin{bmatrix} V(\theta(t))\cos(u(t)) \\ V(\theta(t)) sin(u(t)) \end{bmatrix}$.
The *Hamiltonian* is

$$H((\mathbf{x}(t), \mathbf{u}(t), \boldsymbol{\lambda}(t), t) = 1 + \lambda_1(t)V(\theta(t))\cos(u(t)) + \lambda_2(t)V(\theta(t))\sin(u(t)).$$

The solution is found by using calculus of variations to find the value of $\mathbf{u}(t)$ which minimizes $H(\mathbf{x}(t), \mathbf{u}(t), \boldsymbol{\lambda}(t), t)$.

Solutions to specific examples of these three problems will be calculated in Chapter 7 by modelling them as nonlinear programming problems and solving using `hopdmSQP`.

Chapter 7

Optimal Control Problems as Nonlinear Programming Problems

In this chapter, we will show how optimal control problems can be approximated in such a way that the NLP solvers which were described in Chapters 2 – 5 can be used to solve them. This is done by converting the continuous functions ($\mathbf{x}(t)$) present in the OCP formulation (6.1) into variables which can be handled by the solvers.

First, we introduce a group of numerical methods known as Runge-Kutta discretization schemes, which can be used to approximate continuous functions with variables. Specifically, we show how to model the three problems described in section 6.4 as NLP problems and use `hopdmSQP` to find solutions.

Finally, we discuss how refinements of the discretization strategy can be used to find increasingly accurate approximations to the solution of the continuous problem. We consider the merits of different types of NLP solvers when applied to a sequence of nonlinear programs generated by increasing the number of variables used to approximate the problem.

7.1 Runge-Kutta discretization schemes

In their standard form, optimal control problems cannot be solved by the NLP techniques of Chapter 2. This is because NLP algorithms are used to find optimal values for variables rather than optimal functionals. That is, a possible solution $\mathbf{x}$ is made up of distinguishable values $x_1, x_2 \ldots x_n$. In OCPs, a possible solution $(\mathbf{x}(t), \mathbf{u}(t))$ is made up of continuous trajectories, $x_1(t), x_2(t), \ldots, u_1(t), u_2(t), \ldots$, which are functionals of t.

In order to make NLP algorithms appropriate for solving OCPs, the continuous trajectories $x_1(t), x_2(t)\ldots$ must be approximated by discrete variables. This can be done using Runge-Kutta discretization schemes which are described here, following the derivation given in Betts [10].

Each trajectory is divided into p intervals and evaluated at the end points $t_0, t_1, t_2, \ldots t_p$. The intervals are of variable size h_i, where $h_i = t_i - t_{i-1}$, $i = 1\ldots k$. In this way,

$$\dot{\mathbf{x}}(t) = f(\mathbf{x}(t), \mathbf{u}(t), t)$$

becomes

$$\mathbf{x}(t_{i+1}) = \mathbf{x}(t_i) + \int_{t_i}^{t_{i+1}} f(\mathbf{x}(t), \mathbf{u}(t), t)\ dt \qquad i = 0\ldots p{-}1.$$

To evaluate the integral term, we further divide the integration step into k subintervals $(t_i, \tau_1), (\tau_1, \tau_2), \ldots, (\tau_{k-1}, \tau_k)$ of nonnegative length, as shown in Figure 7.1.

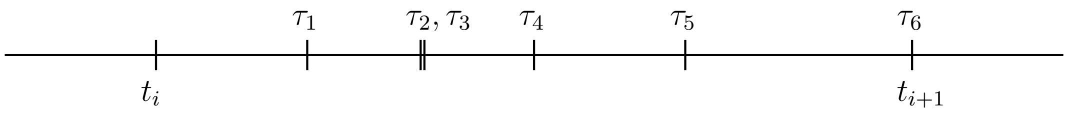


Figure 7.1: Possible subdivision of the integral step from t_i to t_{i+1} with $k = 6$.

Then the function values at the intermediate points, $\hat{f}_j = f(\mathbf{x}(\tau_j), \mathbf{u}(\tau_j), \tau_j)$ are used to approximate the original integration step

$$\int_{t_i}^{t_{i+1}} f(\mathbf{x}(t), \mathbf{u}(t), t)\ dt \approx h_i \sum_{j=1}^{k} \beta_j \hat{f}_j$$

with given constants β_j. To evaluate $\hat{f}_j$, it is necessary to also approximate the values of the state and control variables at each intermediate step. With suitable such approximations, we obtain the Runge-Kutta family of one-step discretization schemes of the form:

$$\mathbf{x}(t_{i+1}) = \mathbf{x}(t_i) + h_i \sum_{j=1}^{k} \beta_j \hat{f}_j.$$

Here, writing the state and control variables as $\mathbf{y}(t) = (\mathbf{x}(t), \mathbf{u}(t))$, we give three common examples of Runge-Kutta discretization schemes, which each uses a different number of subintervals, k:

- **Euler:** (k=1)
$$\mathbf{y}(t_{i+1}) = \mathbf{y}(t_i) + h_i f(\mathbf{y}(t_i), t_i). \tag{7.1}$$

- **Trapezoidal:** (k=2)

$$\mathbf{y}(t_{i+1}) = \mathbf{y}(t_i) + \frac{h_i}{2}\big(f(\mathbf{y}(t_i), t_i) + f(\mathbf{y}(t_{i+1}), t_{i+1})\big). \tag{7.2}$$

- **Classical Runge-Kutta:** (k=4)

$$\begin{aligned}
\mathbf{k}_1(t_i) &= h_i f\left(\mathbf{y}(t_i), t_i\right) \\
\mathbf{k}_2(t_i) &= h_i f\left(\mathbf{y}(t_i) + \frac{1}{2}\mathbf{k}_1(t_i), t_i + \frac{h_i}{2}\right) \\
\mathbf{k}_3(t_i) &= h_i f\left(\mathbf{y}(t_i) + \frac{1}{2}\mathbf{k}_2(t_i), t_i + \frac{h_i}{2}\right) \\
\mathbf{k}_4(t_i) &= h_i f\left(\mathbf{y}(t_i) + \mathbf{k}_3(t_i), t_{i+1}\right)
\end{aligned}$$

$$\mathbf{y}(t_{i+1}) = \mathbf{y}(t_i) + \frac{1}{6}\big(\mathbf{k}_1(t_i) + 2\mathbf{k}_2(t_i) + 2\mathbf{k}_3(t_i) + \mathbf{k}_4(t_i)\big). \tag{7.3}$$

The approximation to the continuous functions becomes more accurate as either the number of integration steps, p, or the number of subintervals, k, increases. In fact, the expected error at each integration step i is of the order h_i^{k+1}. Therefore, the accumulated error of a Runge-Kutta scheme with k subintervals at each integration step is of order h^k, where h is the average stepsize.

7.2 `hopdmSQP` used on models of small OCPs

In this section, we demonstrate the use of the Euler discretization scheme (7.1) by using it to model the three OCPs introduced in section 6.4 as NLP problems. `hopdmSQP` is then used to find approximate solutions to each of these problems. In Appendix C we introduce a further optimal control problem and make comparisons between the three Runge-Kutta discretization schemes (7.1), (7.2), (7.3).

7.2.1 Mountain pass

Specific examples of the family of mountain pass problems described in section 6.4.1 are given in Moré & Munson [58]. We choose

$$f(x_1, x_2) = \left(4 - 2.1x_1^2 + \frac{1}{3}x_1^4\right)x_1^2 + x_1x_2 + 4(x_2^2 - 1)x_2^2, \tag{7.4}$$

which is known as the *six-hump camel back* function. It has six local minimizers, shown in Figure 7.2. There are several ways to choose the endpoints of the mountain pass. Here, we choose the minima at $(-1.5, -0.6)$ and $(0.0, 0.8)$ and search for the path between them which has the smallest maximum point. In the

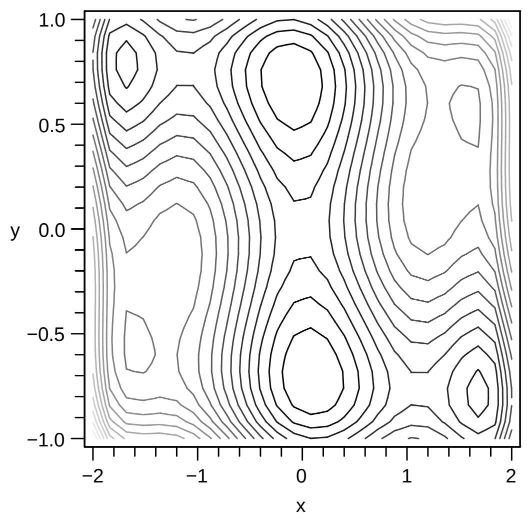


Figure 7.2: Contours of the six-hump camel back function.

conversion of the continuous OCP model to a discrete NLP model, the differential equation constraints

$$\dot{\mathbf{x}}(t) = \mathbf{u}(t)$$

are discretized to become

$$\begin{aligned} x_1(t_{i+1}) &= x_1(t_i) + h_i u_1(t_i) \\ x_2(t_{i+1}) &= x_2(t_i) + h_i u_2(t_i). \end{aligned}$$

The complete `ampl` model is in Appendix B.1. The path found by `hopdmSQP` when there are 200 integration steps of equal length ($p = 200$, $h_i = \frac{1}{p}\ \forall i$) is shown in Figure 7.3. The maximum point, or *mountain pass*, is at $(-1.28, -0.56)$ with a value of 2.24. This is the same as the solution found in [58], using their elastic string theory.

7.2.2 Bicycle

The bicycle problem taken from Jan Olsder [63] and described in section 6.4.2 has two differential equation constraints and an integral objective function which must be discretized before NLP algorithms can be used to solve it. This is done, using an Euler discretization scheme (7.1), as follows:

$$\max \int_0^{\pi} \cos(t + \theta(t))\ dt$$

becomes

$$\max \sum_{i=0}^{p} \cos(t_i + \theta(t_i));$$

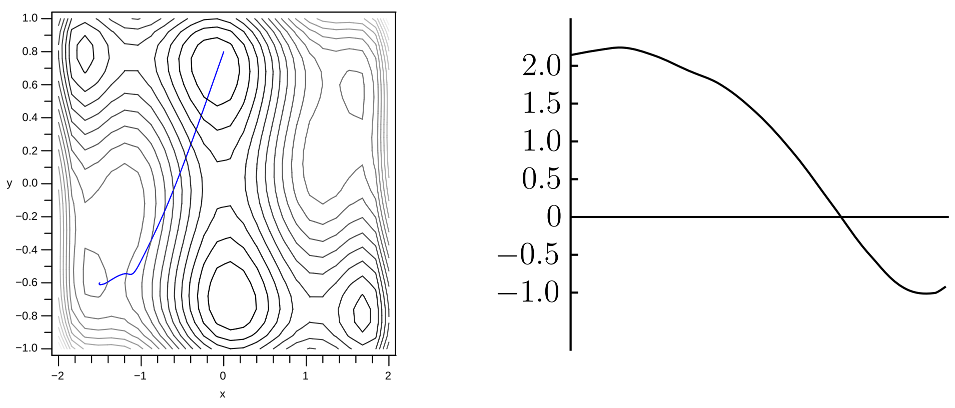


Figure 7.3: An optimal mountain pass between the minimizers $(-1.5, -0.6)$ and $(0.0, 0.8)$ of the six-hump camel back function. The diagram on the left shows the path that the optimal mountain pass takes. The diagram on the right shows the function values along this optimal path.

and

$$\begin{aligned}\dot{x_1}(t) &= \cos(\theta(t))\\ \dot{x_2}(t) &= \sin(\theta(t))\end{aligned}$$

become

$$\begin{aligned}x_1(t_{i+1}) &= x_1(t_i) + h_i\cos(\theta(t_i))\\ x_2(t_{i+1}) &= x_2(t_i) + h_i\sin(\theta(t_i)).\end{aligned}$$

The complete `ampl` model is in Appendix B.2. The optimal trajectory found by

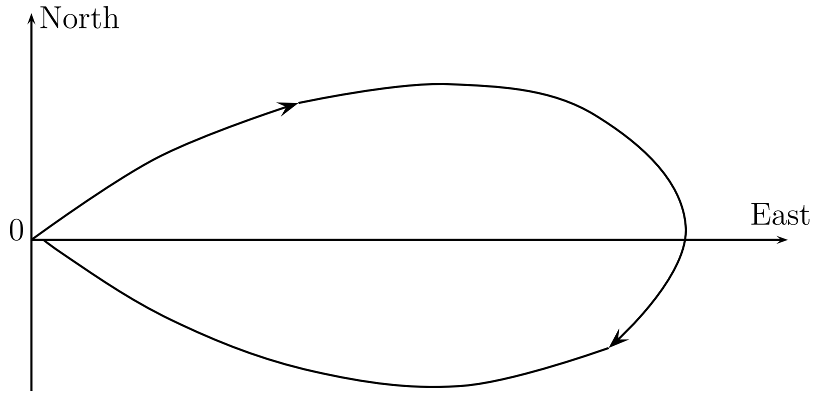


Figure 7.4: Optimal trajectory for cyclist wishing to maximize suntan.

`hopdmSQP` in the case where there are 50 integration steps of equal length ($p = 50$, $h_i = \frac{\pi}{p}\ \forall i$) is shown in Figure 7.4. The cyclist starts by cycling North-East and reaches the furthest point at midday. This is the same solution as is found in [63] by using Hamiltonians.

7.2.3 Sailing

There are many possible examples of the sailing problem given by Hennessey *et al.*'s interpretation of the Zermelo problem [46]. For the purpose of demonstrating how an Euler discretization of the problem can be used to find an optimal trajectory for the sailboat, we use the simplest problem possible. That is, we assume that the water is still and that the wind is constant, in the same direction as the boat needs to travel, see Figure 7.5.

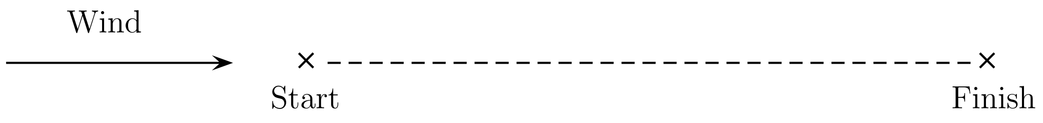


Figure 7.5: The sailing problem to be solved.

In order to convert the OCP sailing problem given in section 6.4.3 into an NLP problem, it is necessary to discretize the two differential equation constraints

$$\begin{aligned} \dot{x}_1(t) &= V(\theta(t))\cos(u(t)) \\ \dot{x}_2(t) &= V(\theta(t))\sin(u(t)). \end{aligned}$$

As the wind field is parallel to the x_1 axis, $\theta(t) = u(t)$ and we calculate $V(\theta)$ using the equation for a quintic curve (7.5) given in [46] and found by practical experimentation with a C&C yacht on Lake Superior.

$$\begin{gathered} V(\theta) = \textstyle\sum_{i=0}^{5} C_i|\theta|^i \\ C_5 = -0.3765,\ C_4 = 1.0479,\ C_3 = 0.9402, \\ C_2 = -4.7994,\ C_1 = 3.0336,\ C_0 = 4.8401 \end{gathered} \tag{7.5}$$

Using an Euler discretization scheme (7.1), we get the discretized equations

$$\begin{aligned} x_1(t_{i+1}) &= x_1(t_i) + h_i V(\theta(t_i))\cos(\theta(t_i)) \\ x_2(t_{i+1}) &= x_2(t_i) + h_i V(\theta(t_i))\sin(\theta(t_i)). \end{aligned}$$

The complete `ampl` model is shown in Appendix B.3. Absolute values, $|\theta|$, have been replaced with $\sqrt{\theta^2}$ to remove discontinuity from the constraints.

Before using `hopdmSQP` to find solutions to the discretized problem, we consider the form which we would expect the solution to take. We consider the problem of sailing between the points $(0, 0)$ and $(100, 0)$, with the boat speed given by (7.5).

In [46], the authors prove that the optimal sailing trajectory between two points can always be given as one or two line segments. Two possible trajectories are the straight line which joins the two points or a triangle travelling away from the start point at the angle $-\theta^{max}$, covering half of the horizontal distance in this direction, and then sailing towards the finish point at an angle of θ^{max}. (θ^{max} (or $-\theta^{max}$ due to symmetry) is the angle with the wind which allows sailing at the maximum possible speed.) These two trajectories are shown in Figure 7.6. The

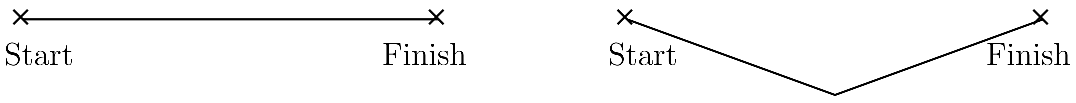


Figure 7.6: Two possible sailing trajectories between $(0, 0)$ and $(100, 0)$.

straight line trajectory would take 20.661 units of time to sail, whilst the two-segment trajectory found by sailing at the maximum speed with respect to the wind would take 19.799 units. The two-segment trajectory shown is equivalent, in terms of time taken to travel between the start and finish points, to any trajectory with more segments in which the boat only travels at the angles $\pm\theta^{max}$.

Using `hopdmSQP` to solve the above problem in the case where there are 40 integration steps of equal length ($p = 40$, $h_i = \frac{t_F}{p}$), we obtain the multi-segmented trajectory shown in Figure 7.7. It can be represented as a two-segment trajectory, which is also shown in Figure 7.7. It would take 19.382 units of time to sail this

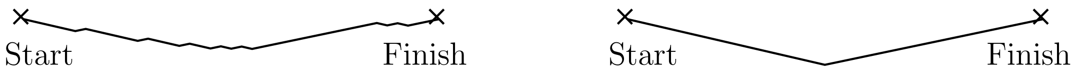


Figure 7.7: Optimal sailing trajectory between $(0, 0)$ and $(100, 0)$ found by `hopdmSQP`. The diagram on the left shows the exact solution found by `hopdmSQP` while the diagram on the right shows its representation as a 2-segment trajectory.

trajectory, which is faster than sailing either of the proposed trajectories shown in Figure 7.6.

It is important to remember here that, as `hopdmSQP` searches for a local minimum, it is possible that there may be a better solution to the problem than this. Indeed, if slight alterations are made to the problem formulation, different trajectories are found by the solver. For example, Table 7.1 shows the optimal trajectories found when:

a. instead of restricting the sailing angle to the angles which it is possible for the boat to travel in ($-\frac{5\pi}{6} \leq \theta \leq \frac{5\pi}{6}$), we further restrict the sailing angle such that the boat can only travel *forward* ($-\frac{\pi}{2} \leq \theta \leq \frac{\pi}{2}$).

b. instead of considering the problem of travelling from $(0, 0)$ to $(100, 0)$, we consider the equivalent problem of travelling from $(0, 50)$ to $(100, 50)$.

Start	Finish	Restriction on θ	θ_1	θ_2	Time
$(0, 0)$	$(100, 0)$	$\pm\frac{5\pi}{6}$	-0.224281	0.203531	19.3815
$(0, 0)$	$(100, 0)$	$\pm\frac{\pi}{2}$	-0.234789	0.193263	19.3896
$(0, 50)$	$(100, 50)$	$\pm\frac{5\pi}{6}$	-0.278126	0.152511	19.4772
$(0, 50)$	$(100, 50)$	$\pm\frac{\pi}{2}$	-0.336878	0.101123	19.7208

Table 7.1: Different optimal sailing trajectories found by `hopdmSQP` when slight changes are made to the problem formulation. The two angles θ_1 and θ_2 are the angles the boat sails at with respect to the wind on each of the two segments of its journey.

If `hopdmSQP` searched for a global minimizer rather than a local minimizer then each of these trajectories would be the same.

In Appendix D we relate further work carried out into finding an optimal sailing trajectory.

7.3 Sequential nonlinear programming

The solution found by using an NLP solver to solve a discretized approximation to an optimal control problem is, by definition, an approximation. It is necessary to assess the accuracy of this estimate of the solution.

In section 7.1 we noted that the expected error for a k-stage Runge-Kutta discretization scheme is h^k, where h is the average size of an integration step. However, although decreasing h reduces the expected error, the number of NLP variables increases as h decreases and so the problem becomes increasingly difficult to solve.

As the difficulty of the problem increases as h decreases, it is common to initially use a coarse discretization scheme, with large integration stepsizes. It is then possible to construct an approximate solution to a finer discretization of the OCP by interpolating the solution to the coarse discretization. This constructed solution can be used as a starting point for the finer discretization of the problem. Significant work has been carried out into determining how best to choose a new discretization such that finer discretizations are employed mainly over the integration steps where the error is largest.

This technique, known as *Sequential Nonlinear Programming*, is described by Betts [10, 11] and summarized in Algorithm 7.1.

Algorithm 7.1.

```
Choose a coarse discretization ($h$ large).
Determine a suitable starting point.
While solution not found.
    Use an NLP solver to solve current discretization of the problem.
    Assess the accuracy of the problem.
    If solution is acceptably accurate.
        Accept current solution.
    Else
        Refine the discretization of the problem.
        Interpolate the current solution to provide a new starting point.
```

In [11], Betts gives numerical experience with different NLP solvers used in sequential nonlinear programming. He compares active set SQP methods with interior point NLP methods and reports that the SQP algorithms are more appropriate for solving a sequence of discretized problems. This is because it is straightforward to use the interpolation of a solution to a coarse discretization as a starting point in an active set method, but interior point methods cannot be restarted, or *warm-started* as easily.

However, a large amount of research has been carried out into warm-starting interior point methods from solutions to similar problems. The problems which occur when trying to use the solution to one problem as a starting point for a similar problem are described by Colombo *et al.* [19]. Essentially, a small perturbation to a problem may move the previous optimal solution far from centrality causing an interior-point method to make very slow progress towards a new solution. Colombo *et al.* provide a short review of techniques which have been proposed for combating this difficulty in the case of linear programming, referring to [5, 41, 37, 82] and Benson & Shanno [6] have shown some success in applying warm-start techniques to the case of nonlinear programming. It would be interesting to investigate how this research can be applied to sequential nonlinear programming to make interior-point methods competitive with active set SQP methods in optimal control theory.

Chapter 8

Further Work

There are many ways in which the work which has been described in this thesis can be continued. In this concluding chapter, we have divided the possible future directions into four sections, here amalgamating suggestions from previous chapters with still more ideas.

8.1 Improving `hopdmSQP`

First, we list suggestions for improving the algorithm implemented in `hopdmSQP`, including ideas first mentioned in section 3.5.

- Investigate the tendency for the stepsize, α to approach 0 in many of the problems. It is possible that this is caused by the Maratos effect [54] and possible that a different technique for updating the merit function penalty parameter, ν, may prevent this from happening.
- Improve the steepest descent method which is implemented.
- Consider the logic behind the update of $H\!\!L$ when the direction found by solving the quadratic approximation is nondescent with respect to the merit function. It may be that the current implementation of `hopdmSQP` is too slow to determine that $H\!\!L$ is not positive definite and to add a regularization term.
- Decide whether to adjust the termination conditions so that they are related to the objective value.
- Implement a method for determining if a problem is infeasible. For example, SNOPT [36] begins by testing the linear constraints present in the problem, classifying problems which do not have feasible solutions with respect to the linear constraints as infeasible. It would be possible to use the existing presolve in `hopdm` to carry out this test.

8.2 Coding `hopdmNLP`

In Chapter 5, we introduced the interior point NLP solver `hopdmNLP` and discussed some of the issues which have to be resolved before the algorithm can be fully implemented. Some of these issues (the logic behind the update of $\mathbb{H}\mathbb{L}$, possible adjustment to the termination conditions and implementation of a method to identify infeasibility) are shared with `hopdmSQP`. Here we list other issues which will be considered during the implementation of `hopdmNLP`. Some of these have been mentioned in section 5.3.

- Choose how to update the penalty parameter, μ.
- Determine how best to compare merit function values from different iterations, where the penalty parameters, μ and ν, may differ.
- Decide how to deal with free/fixed variables which may introduce ∞s into the linear algebra.
- Determine a well-centred starting point.
- Instead of limiting the linear algebra to the Augmented System approach (5.2) consider extending the nonlinear code so that it can use the normal equations approach as well.
- Consider the possibility of experimenting with different preconditioners.

Once `hopdmNLP` is fully implemented it would be interesting to compare its performance with some of the NLP solvers described in Chapter 4.

8.3 Sequential nonlinear programming

There are three main stages involved in extending an NLP solver so that it can be used to solve discretized OCPs in a sequential nonlinear programming algorithm such as Algorithm 7.1. These 3 stages lead to the following possibilities for future work:

1. Research ways of evaluating the errors in a solution found by solving a coarse discretization of an OCP and methods of using these errors to choose the next discretization of the problem, aiming to make the integration steps smaller over areas of the solution trajectory where the errors are largest.
2. Decide how to use the solution found from the coarse discretization in order to find a good starting point for the refined discretization. It should be

possible to find a good starting point for primal variables by straightforward interpolation of the solution of the coarse discretization, but care should be taken when determining new dual variable estimates.

3. Investigate warm-start techniques that have been proposed for use with interior-point methods. Consider how they could be used appropriately in interior point SNLP.

Once this research has been done and the code `hopdmNLP` has been completed, `hopdmNLP` could be incorporated into an algorithm which includes each of the 3 stages listed here, making it able to solve optimal control problems.

8.4 Small optimal control problems

In Chapter 6 we introduced 3 small optimal control problems. In Chapter 7 we chose simple versions of each of these problems and showed how to model them using an Euler discretization (7.1) scheme.

Further work could include using other Runge-Kutta discretization schemes (e.g. (7.2), (7.3)) to form models for each of these problems. Work could then be carried out into investigating the efficiency of each scheme, taking into account the time taken to solve each model and the accuracy of the solution found. (This work has been begun in Appendix C.)

Additional work could also be carried out into finding appropriate models for more advanced versions of each problem. For example

Mountain pass problem: several mountain pass problems which are more complicated than the *six-hump camel-back* function (7.4) are given in [58]. These are problems relating to the potential energy surfaces of chemical reactions.

Cycling problem: the formulation of the cycling problem given in Chapter. 6 allows for negative suntan. In [63], Jan Olsder gives further formulation of the problem which, more realistically, ensures that the cyclist's suntan does not decrease when he is cycling away from the sun.

Sailing problem: this is an incredibly versatile problem. There are many more versions of it to consider, including considering time and space-dependent wind fields, moving water (tides, river currents) and start/finish zones rather than points. (This work has been begun in Appendix D.)

Optimal control problems can also be found in many other industries and applications. For example, in the space industry, finding optimal trajectories for satellites sent to photograph and examine planets in our solar system is an optimal control problem and in the manufacturing industry, optimal control is used to optimize chemical reactions.

Good aims for the completion of the work started here would be to write a robust piece of code which implements an interior-point NLP method in the context of sequential nonlinear programming. This code should be competitive with solvers such as IPOPT [77], KNITRO-Direct [78] and LOQO [71] and be able to solve a range of optimal control problems of all difficulties, which will have been modelled using several Runge-Kutta discretization schemes.

Appendix A

Cute Results

Table A.1: Increasing success rate as `hopdmSQP` is improved.

	After Change										
Problem Name	**start**	**A**	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**	**J**
`allinitu`	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
`arwhead`	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
`aug3d`	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
`batch`				✓	✓	✓	✓	✓	✓	✓	✓
`bdvalue`	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
`bigbank`		✓	✓	✓	✓	✓	✓	✓			
`bloweyb`								✓	✓	✓	✓
`booth`	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
`box2`	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
`bqp1var`	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
`brainpc4`								✓	✓	✓	
`brainpc6`								✓	✓	✓	
`brainpc8`								✓	✓	✓	✓
`broydnbd`								✓	✓	✓	✓
`catenary`							✓	✓	✓	✓	✓
`cb3`	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
`chandheq`	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
`chemrcta`											
`chemrctb`											
`clnlbeam`							✓	✓	✓	✓	✓
`clplatea`	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
`csfi1`				✓	✓		✓	✓	✓	✓	✓
`degenlpb`				✓	✓	✓	✓	✓	✓	✓	✓
`denschnf`	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
`dixchlnv`									✓		
`dixmaanb`	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
`dixmaanf`	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
`djtl`											

Continued on Next Page…

Problem Name	After Change start	A	B	C	D	E	F	G	H	I	J
drcav2lq											
dtoc1l	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
dtoc1nd											
eigenc2											
engval1	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
engval2	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
explin	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
explin2	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
extrasim	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
flosp2th											
fminsurf								✓	✓	✓	✓
gilbert	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
gottfr	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
gouldqp2	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
growthls		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
hatfldg								✓	✓	✓	✓
hatfldh	✓			✓	✓	✓	✓	✓	✓	✓	✓
heart6ls											
himmelbg	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
himmelbi	✓	✓	✓				✓	✓	✓	✓	✓
himmelbk	✓	✓	✓		✓		✓	✓	✓	✓	✓
hs011	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
hs015				✓	✓	✓	✓	✓	✓	✓	✓
hs022	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
hs026	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
hs031	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
hs036				✓	✓	✓	✓	✓	✓	✓	✓
hs042	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
hs046	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
hs065	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
hs075							✓	✓	✓	✓	✓
hs080	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
hs083	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
hs100lnp		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
hs106											
hs110	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
hs3mod	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
hvycrash											
liswet2		✓	✓	✓	✓	✓	✓	✓		✓	✓
liswet6		✓	✓	✓	✓	✓	✓	✓		✓	✓
liswet7							✓	✓	✓	✓	✓
lootsma									✓	✓	✓
lotschd	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
matrix2							✓	✓	✓	✓	✓

Continued on Next Page…

Problem Name	After Change										
	start	A	B	C	D	E	F	G	H	I	J
maxlika	✓	✓		✓	✓	✓	✓	✓	✓	✓	✓
mccormck	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
methanb8	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
mifflin1	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
minsurf	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
mosarqp2	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
msqrta									✓	✓	✓
msqrtb											
ncvxqp4								✓		✓	✓
ncvxqp8											
noncvxu2								✓	✓	✓	✓
nonscomp							✓	✓	✓	✓	✓
obstclal	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
palmer5c	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
palmer7c							✓	✓	✓	✓	✓
pfit4	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
pfit4ls	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
polak3	✓	✓	✓	✓	✓		✓	✓	✓	✓	✓
polak4	✓	✓	✓		✓		✓	✓	✓	✓	✓
portfl4	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
reading1								✓	✓	✓	✓
sinquad									✓	✓	✓
sipow2	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
steenbre											

Table A.1: Increasing success rate as `hopdmSQP` is improved.

Table A.2: Shows `hopdmSQP`'s success on whole of CUTE set, including problem sizes, iteration count, time taken[2], objective found and final constraint violation.

Problem Name	n	m	Outer Iters	Time (secs)	Objective Value	Constraint Violation
3pk	30	0	27	0.06	1.7203E+00	0.0000E+00
aircrfta	5	0	3	0.00	0.0000E+00	6.0845E-06
aircrftb	5	0	17	0.02	4.9070E-12	0.0000E+00
airport	84	42	20	0.25	4.7953E+04	0.0000E+00
aljazzaf	3	1	**IL**			
allinit	3	0	7	0.00	1.6706E+01	0.0000E+00
allinitc	3	1	**IL**			
allinitu	4	0	16	0.07	5.7444E+00	0.0000E+00
alsotame	2	1	6	0.00	8.2085E-02	2.8888E-13
argauss	3	0	6	0.01	0.0000E+00	3.3817E-04

Continued on Next Page...

Problem Name	n	m	Outer Iters	Time (secs)	Objective Value	Constraint Violation
arglina	100	0	2	0.04	1.0000E+02	0.0000E+00
arglinb	10	0	14	0.05	4.6341E+00	0.0000E+00
arglinc	8	0	6	0.01	6.1351E+00	0.0000E+00
argtrig	100	0	4	1.14	0.0000E+00	1.6217E-05
artif	5000	0	**IL**			
arwhead	5000	0	7	1.24	-1.3612E-09	0.0000E+00
aug2d	20192	9996	18	$>$	1.6874E+06	3.1188E-05
aug2dc	20200	9996	20	$>$	1.8184E+06	3.6615E+00
aug2dcqp	20200	9996	33	$>$	6.4982E+06	2.5975E-05
aug2dqp	20192	9996	**IL**			
aug3d	3873	1000	4	0.51	5.5407E+02	3.5069E-06
aug3dc	3873	1000	4	0.51	7.7126E+02	1.2651E-07
aug3dcqp	3873	1000	5	1.48	9.9336E+02	5.3912E-09
aug3dqp	3873	1000	13	3.93	6.7524E+02	4.2309E-11
avgasa	6	6	2	0	-4.17E+00	0.0000E+00
avgasb	6	6	3	0	-4.1328E+00	0.0000E+00
avion2	49	15	**IL**			
bard	3	0	8	0	8.2149E-03	0.0000E+00
batch	46	69	48	0.35	2.5918E+05	8.4601E-09
bdexp	5000	0	13	1.22	2.4035E-04	0.0000E+00
bdqrtic	1000	0	12	0.34	3.98E+03	0.0000E+00
bdvalue	5000	0	2	0.48	0.0000E+00	3.5040E-06
beale	2	0	10	0.01	2.1500E-15	0.0000E+00
bigbank	1773	814	**IL**			
biggs3	3	0	9	0.01	1.6495E-14	0.0000E+00
biggs5	5	0	23	0.04	4.7953E-15	0.0000E+00
biggs6	6	0	18	0.03	3.0637E-01	0.0000E+00
biggsb1	1000	0	13	0.042	1.5000E-02	0.0000E+00
biggsc4	4	7	11	0.05	-2.4375E+01	0.0000E+00
blockqp1	2005	1001	5	2.41	-9.9650E+02	4.1922E-10
blockqp2	2005	1001	5	2.67	-9.9610E+02	4.1744E-11
blockqp3	2005	1001	13	28.73	-4.9750E+02	3.4333E-08
blockqp4	2005	1001	4	3.87	-4.9810E+02	2.2778E-09
blockqp5	2005	1001	12	30.66	-4.9750E+02	1.8376E-09
bloweya	2002	1002	5	201.7	-8.1236E-06	2.5290E-07
bloweyb	2002	1002	6	143.44	-7.1916E-07	2.2635E-06
bloweyc	2002	1002	5	197.13	-3.2158E-05	3.4326E-07
booth	2	0	2		0.0000E+00	3.2515E-08
box2	2	0	7	0	1.1773E-15	0.0000E+00
box3	3	0	9	0.01	3.9353E-16	0.0000E+00
bqp1var	1	0	3	0	7.6720E-10	0.0000E+00
bqpgabim	46	0	6	0.05	-3.7903E-05	0.0000E+00
bqpgasim	50	0	7	0.02	-5.5195E-05	0.0000E+00
brainpc0	6905	6900	19	1270.56	4.9549E-02	1.6984E-04

Continued on Next Page...

Problem Name	n	m	Outer Iters	Time (secs)	Objective Value	Constraint Violation
brainpc1	13805	13800	19	2144.31	2.3037E-05	4.4625E-06
brainpc2	6905	6900	32	>	3.5027E-05	5.1581E-05
brainpc3	6905	6900	12	956.25	6.2155E-05	7.9544E+00
brainpc4	6905	6900	24	1971.29	9.7382E-05	3.3037E-06
brainpc5	6905	6900	21	698.27	4.6502E-05	1.8553E-05
brainpc6	6905	6900	22	887.82	6.3153E-05	8.2338E-06
brainpc7	6905	6900	22	959.87	6.1039E-05	7.6023E-06
brainpc8	6905	6900	17	753.5	4.6995E-05	1.4310E-05
brainpc9	6905	6900	12	775	8.8475E-05	2.1009E-05
bratu1d	1001	0	11	0.25	-8.5189E+00	0.0000E+00
bratu2d	4900	0	4	2.37	0.0000E+00	5.5633E-06
bratu2dt	4900	0	15	41.14	0.0000E+00	3.6253E-04
bratu3d	3375	0	4	315.43	0.0000E+00	3.0628E-06
britgas	450	360	37	4.99	5.5931E-11	4.0586E-05
brkmcc	2	0	4	0	1.6904E-01	0.0000E+00
brownal	10	0	8	0.01	2.6840E-14	0.0000E+00
brownbs	2	0	**IL**			
brownden	4	0	15	0.02	8.5822E+04	0.0000E+00
broydn3d	10000	0	5	2.28	0.0000E+00	2.4513E-08
broydn7d	1000	0	55	2.69	4.9651E+02	0.0000E+00
broydnbd	5000	0	6	9.59	0.0000E+00	5.8820E-08
brybnd	5000	0	9	2.04	2.4179E-14	0.0000E+00
bt1	2	1	**IL**			
bt10	2	2	7	0	-1.0000E+00	5.5384E-09
bt11	5	3	8	0	8.2489E-01	1.4943E-09
bt12	5	3	4	0	6.1881E+00	5.1484E-06
bt13	5	1	24	0.11	1.0219E-09	1.9817E-08
bt2	3	1	12	0.01	3.2568E-02	2.3894E-09
bt3	5	3	5	0	4.0930E+00	2.9543E-10
bt4	3	2	33	0.07	-4.5511E+01	7.7304E-09
bt5	3	2	6	0	9.6172E+02	4.7201E-07
bt6	5	2	10	0.01	2.7704E-01	1.5312E-12
bt7	5	3	36	0.05	3.0650E+02	6.7376E-09
bt8	5	2	12	0.01	1.0000E+00	1.0395E-07
bt9	4	2	13	0.01	-1.0000E+00	7.0566E-09
byrdsphr	3	2	13	0.01	-4.6833E+00	2.8167E-06
camel6	2	0	8	0	2.1043E+00	0.0000E+00
cant500vr	5	1	14	0.01	1.3400E+00	5.0532E-08
catena	32	11	26	0.07	-2.3078E+04	7.8329E-09
catenary	496	166	128	3.33	-3.4840E+05	1.8580E-09
cb2	3	3	7	0	1.9522E+00	0.0000E+00
cb3	3	3	8	0.01	2.0000E+00	0.0000E+00
cbratu2d	882	0	2	0.06	0.0000E+00	6.9503E-06
cbratu3d	1024	0	2	0.85	0.0000E+00	1.6974E-05

Continued on Next Page...

Problem Name	n	m	Outer Iters	Time (secs)	Objective Value	Constraint Violation
chaconn1	3	3	5	0	1.9522E+00	4.6297E-08
chaconn2	3	3	5	0	2.0000E+00	0.0000E+00
chainwoo	1000	0	180	11.07	4.1205E+02	0.0000E+00
chandheq	100	0	10	1.76	0.0000E+00	9.5384E-05
chebyqad	50	0	149	39.61	6.5632E-03	0.0000E+00
chemrcta	2500	2499	**time**			
chemrctb	1000	999	**IL**			
chenhark	1000	0	5	0.24	-2.0000E+00	0.0000E+00
chnrosnb	50	0	46	0.08	2.8266E-15	0.0000E+00
cliff	2	0	28	0.03	1.9979E-01	0.0000E+00
clnlbeam	1499	1000	5	0.49	3.5000E+02	2.9782E-06
clplatea	4970	0	4	0.82	-1.2588E-02	0.0000E+00
clplateb	4970	0	5	1.09	-6.9882E+00	0.0000E+00
clplatec	4970	0	2	0.33	-5.0207E-03	0.0000E+00
cluster	2	0	7	0	0.0000E+00	8.3960E-05
concon	15	11	**IL**			
congigmz	3	5	5	0	2.8000E+01	7.7004E-07
coolhans	9	0	29	0.13	0.0000E+00	3.7176E-05
core1	65	50	**IL**			
core2	157	122	**IL**			
corkscrw	8997	7000	**IL**			
coshfun	61	20	24	0.21	-2.3630E+01	0.0000E+00
cosine	10000	0	55	57.94	-9.9743E+03	0.0000E+00
cragglvy	5000	0	15	1.28	1.6882E+03	0.0000E+00
cresc100	6	200	**IL**			
cresc132	6	2654	**time**			
cresc4	6	8	**IL**			
cresc50	6	100	**IL**			
csfi1	5	4	29	0.06	-4.9075E+01	1.7255E-08
csfi2	5	4	**IL**			
cube	2	0	27	0.03	9.0615E-14	0.0000E+00
curly10	10000	0	74	43.82	-1.0031E+06	0.0000E+00
curly20	10000	0	**IL**			
curly30	10000	0	**IL**			
cvxbqp1	10000	0	3	4.4	2.2502E+06	0.0000E+00
cvxqp1	1000	500	126	19.29	1.0875E+06	2.1971E-06
cvxqp2	10000	2500	54	121.75	8.1842E+07	1.0186E-05
cvxqp3	10000	7500	**IL**			
dallasl	837	598	192	7.17	-2.0260E+05	1.9499E-06
dallasm	164	119	63	0.48	-4.8198E+04	3.2196E-07
dallass	44	29	45	0.17	-3.2393E+04	8.4356E-08
deconvb	51	0	62	1.62	5.4130E-07	0.0000E+00
deconvc	51	1	39	0.46	2.5695E-03	7.2831E-14
deconvu	51	0	280	2.52	2.7187E-03	0.0000E+00

Continued on Next Page...

Problem Name	n	m	Outer Iters	Time (secs)	Objective Value	Constraint Violation
degenlpa	20	14	9	0.03	2.2159E+00	1.8912E-04
degenlpb	20	15	21	0.14	-3.0731E+01	6.9877E-09
demymalo	3	3	6	0	-3.0000E+00	0.0000E+00
denschna	2	0	7	0	3.7124E-17	0.0000E+00
denschnb	2	0	8	0	7.4867E-20	0.0000E+00
denschnc	2	0	11	0.01	1.9884E-18	0.0000E+00
denschnd	3	0	29	0.03	1.0082E-07	0.0000E+00
denschne	3	0	12	0.02	1.0000E+00	0.0000E+00
denschnf	2	0	7	0	6.6823E-18	0.0000E+00
dipigri	7	4	7	0.01	6.8063E+02	5.5387E-07
disc2	28	23	418	3.94	1.5625E+00	2.7634E-05
discs	33	66	104	3.14	1.2000E+01	1.6869E-09
dittert	327	264	55	7.89	-1.9976E+00	1.4761E-06
dixchlng	10	5	**IL**			
dixchlnv	100	50	**IL**			
dixmaana	3000	0	6	0.28	1.0000E+00	0.0000E+00
dixmaanb	3000	0	34	4.13	1.0000E+00	0.0000E+00
dixmaanc	3000	0	57	9.88	1.0000E+00	0.0000E+00
dixmaand	3000	0	62	10.35	1.0000E+00	0.0000E+00
dixmaane	3000	0	34	3.47	1.0000E+00	0.0000E+00
dixmaanf	3000	0	87	13.07	1.0000E+00	0.0000E+00
dixmaang	3000	0	131	21.69	1.0000E+00	0.0000E+00
dixmaanh	3000	0	174	36.1	1.0000E+00	0.0000E+00
dixmaani	3000	0	36	3.37	1.0000E+00	0.0000E+00
dixmaanj	3000	0	199	37.97	1.0054E+00	0.0000E+00
dixmaank	3000	0	263	50.78	1.0178E+00	0.0000E+00
dixmaanl	3000	0	300	64.02	1.0589E+00	0.0000E+00
dixon3dq	10	0	3	0	8.7925E-17	0.0000E+00
djtl	2	0	**IL**			
dnieper	57	24	30	0.13	1.8744E+04	2.1286E-05
dqdrtic	5000	0	3	0.19	4.4043E-12	0.0000E+00
dqrtic	5000	0	268	17.77	7.8770E-06	0.0000E+00
drcav1lq	10000	0	**time**			
drcav2lq	10000	0	**time**			
drcav3lq	10000	0	38	>	1.3134E-01	0.0000E+00
drcavty1	10000	0	**time**			
drcavty2	10000	0	19	>	1.5444E-02	0.0000E+00
drcavty3	10000	0	38	>	1.3134E-01	0.0000E+00
dtoc1l	14985	9990	7	6.49	1.2534E+02	1.8410E-05
dtoc1na	1485	990	7	1.59	1.2702E+01	2.4671E-06
dtoc1nb	1485	990	6	1.39	1.5938E+01	7.9264E-07
dtoc1nc	1485	990	185	88.54	2.5041E+01	2.8728E-06
dtoc1nd	735	490	**IL**			
dtoc2	5994	3996	11	11.08	5.0993E-01	6.7170E-06

Continued on Next Page...

Problem Name	n	m	Outer Iters	Time (secs)	Objective Value	Constraint Violation
dtoc3	14996	9997	2	0.9	2.3526E+02	6.4115E-05
dtoc4	14996	9997	7	>	2.8748E+00	3.1463E-05
dtoc5	9998	4999	5	1.81	1.5351E+00	8.3195E-06
dtoc6	10000	5000	23	18.79	1.3485E+05	8.7696E-06
dual1	85	1	5	0.01	3.5013E-02	3.3307E-16
dual2	96	1	4	0.08	3.3734E-02	1.1102E-15
dual3	111	1	5	0.18	1.3323E-15	1.5470E-06
dual4	75	1	13	0.21	7.4609E-01	3.5527E-15
dualc1	9	215	26	0.07	6.1553E+03	3.4118E-09
dualc2	7	229	21	0.05	3.5513E+03	1.0161E-09
dualc5	8	278	18	0.02	4.2723E+02	3.8949E-10
dualc8	8	503	25	0.07	1.8309E+04	1.6686E-10
edensch	2000	0	8	0.25	1.2003E+04	0.0000E+00
eg1	3	0	8	0	-1.4293E+00	0.0000E+00
eg2	1000	0	7	0.1	-9.9895E+02	0.0000E+00
eg3	101	200	6	0.4	6.8404E-02	6.8981E-11
eigena	110	0	24	0.79	3.1708E-09	0.0000E+00
eigena2	110	55	16	0.15	9.5057E-17	4.1311E-09
eigenaco	110	55	10	0.48	1.2901E-16	1.3097E-07
eigenals	110	0	171	5.6	1.9238E-02	0.0000E+00
eigenb	110	0	74	1.3	5.0480E-02	0.0000E+00
eigenb2	110	55	**IL**			
eigenbco	110	55	131	5.35	5.8675E-02	3.2077E-06
eigenbls	110	0	107	2.5	1.1898E-09	0.0000E+00
eigenc2	462	231	**IL**			
eigencco	30	15	17	0.06	6.3105E-01	1.3400E-07
eigmaxa	101	101	23	0.59	-1.0000E+01	7.3812E-08
eigmaxb	101	101	8	0.05	-9.6743E-04	1.7598E-07
eigmaxc	22	22	13	0.04	-3.0000E+00	2.1752E-08
eigmina	101	101	17	0.28	1.0000E+00	4.5609E-07
eigminb	101	101	18	0.15	9.6743E-04	5.8830E-08
eigminc	22	22	23	0.08	1.0000E+00	1.1523E-08
engval1	5000	0	9	0.69	5.5487E+03	0.0000E+00
engval2	3	0	16	0.01	1.6632E-11	0.0000E+00
errinros	50	0	35	0.09	4.0404E+01	0.0000E+00
expfit	2	0	9	0.01	2.4051E-01	0.0000E+00
expfita	5	21	19	0.27	1.1366E-03	0.0000E+00
expfitb	5	101	14	0.15	5.0194E-03	0.0000E+00
expfitc	5	501	18	10.88	2.3303E-02	0.0000E+00
explin	120	0	123	2.17	-7.2341E+05	0.0000E+00
explin2	120	0	25	0.28	-7.2411E+05	0.0000E+00
expquad	120	0	**IL**			
extrasim	2	1	2	0	1.0000E+00	1.3568E-10
extrosnb	10	0	1	0	0.0000E+00	0.0000E+00

Continued on Next Page...

Problem Name	n	m	Outer Iters	Time (secs)	Objective Value	Constraint Violation
fccu	19	8	5	0	1.1149E+01	1.2748E-08
fletcbv2	0	0	3	0	-5.1401E-01	0.0000E+00
fletcbv3	10000	0	**IL**			
fletchbv	100	0	**mem**			
fletchcr	100	0	7	0.01	3.0877E-14	0.0000E+00
fletcher	4	4	23	0.08	1.9525E+01	8.6564E-11
flosp2hh	650	0	**IL**			
flosp2hl	650	0	11	1.73	3.8871E+01	0.0000E+00
flosp2hm	650	0	**IL**			
flosp2th	650	0	**IL**			
flosp2tl	650	0	6	0.91	1.0000E+01	0.0000E+00
flosp2tm	650	0	**IL**			
fminsrf2	15625	0	91	6.99	1.0177E+00	0.0000E+00
fminsurf	1024	0	97	118.578	1.0002E+00	0.0000E+00
freuroth	5000	0	9	0.93	6.0816E+05	0.0000E+00
gausselm	1495	3690	43	1403.91	-9.9984E-01	1.2005E-05
genhs28	10	8	4	0	9.2717E-01	1.2990E-09
genhumps	5	0	121	0.53	8.5891E-16	0.0000E+00
genrose	500	0	5	0.03	4.9496E+02	0.0000E+00
gigomez1	3	3	8	0.01	-3.0000E+00	0.0000E+00
g500bert	1000	1	26	0.42	4.8203E+02	3.2045E-12
goffin	51	50	4	0.04	6.0084E-09	0.0000E+00
gottfr	2	0	5	0	0.0000E+00	1.3687E-05
gouldqp2	600	349	4	0.18	1.8800E-04	1.9851E-13
gouldqp3	699	349	5	0.16	2.0652E+00	4.9248E-13
gpp	250	498	21	5	1.4401E+04	1.2708E-07
gridneta	8964	6724	4	2.64	3.0498E+02	2.0379E-06
gridnetb	13284	6724	4	3.1	1.4332E+02	3.2137E-05
gridnetc	7564	3844	4	2.65	1.6187E+02	5.7075E-07
gridnetd	3945	2644	7	2.9	5.6644E+02	2.1748E-06
gridnete	7565	3844	5	3.73	2.0655E+02	1.4409E-05
gridnetf	7565	3844	13	15.7	2.4211E+02	6.5690E-06
gridnetg	44	34	5	0.01	7.3317E+01	6.2644E-08
gridneth	61	36	5	0.01	3.9626E+01	3.9742E-09
gridneti	61	36	6	0.02	4.0247E+01	3.8133E-09
grouping	100	125	2	0.01	1.3850E+01	1.0622E-19
growth	3	0	369	0.62	1.0040E+00	0.0000E+00
growthls	3	0	329	0.48	1.0040E+00	0.0000E+00
gulf	3	0	23	0.04	9.5913E-13	0.0000E+00
hadamals	90	0	15	0.13	7.6813E+02	0.0000E+00
hadamard	65	256	3	0.06	1.0000E+00	8.3858E-09
hager1	10000	5000	2	0.64	8.8080E-01	5.4752E-08
hager2	10000	5000	2	1.23	4.3208E-01	1.7982E-06
hager3	10000	5000	2	1.46	1.4096E-01	3.7085E-07

Continued on Next Page…

Problem Name	n	m	Outer Iters	Time (secs)	Objective Value	Constraint Violation
hager4	10000	5000	5	11.71	2.7940E+00	5.6359E-07
haifam	85	150	18	0.96	-4.5000E+01	8.8799E-06
haifas	7	9	8	0.01	-4.5000E-01	2.8451E-08
hairy	2	0	36	0.21	2.0000E+01	0.0000E+00
haldmads	6	42	9	0.04	3.3292E-02	2.8025E-10
hanging	288	180	25	0.75	-6.2018E+02	9.4783E-08
harkerp2	100	0	10	0.18	-5.0000E-01	0.0000E+00
hart6	6	0	7	0.02	-3.3229E+00	0.0000E+00
hatflda	4	0	24	0.06	5.0178E-16	0.0000E+00
hatfldb	4	0	22	0.06	5.5728E-03	0.0000E+00
hatfldc	4	0	5	0	5.0200E-17	0.0000E+00
hatfldd	3	0	19	0.02	6.6151E-08	0.0000E+00
hatflde	3	0	24	0.03	4.4344E-07	0.0000E+00
hatfldf	3	0	**IL**			
hatfldg	25	0	21	0.12	0.0000E+00	5.2621E-06
hatfldh	4	7	11	0.05	-2.4375E+01	0.0000E+00
heart6	6	0	38	0.11	0.0000E+00	3.6397E-05
heart6ls	6	0	**IL**			
heart8	8	0	**IL**			
heart8ls	8	0	233	1.85	1.0561E+00	0.0000E+00
helix	3	0	16	0.02	3.5284E-24	0.0000E+00
h500berta	10	0	3	0	6.1634E-10	0.0000E+00
h500bertb	50	0	3	0.01	6.1743E-18	0.0000E+00
himmelba	2	0	0	0	0.0000E+00	0.0000E+00
himmelbb	2	0	20	0.02	4.3513E-10	0.0000E+00
himmelbc	2	0	6	0	0.0000E+00	4.3074E-08
himmelbd	2	0	**IL**			
himmelbe	3	0	0	0	0.0000E+00	0.0000E+00
himmelbf	4	0	91	0.14	3.1857E+02	0.0000E+00
himmelbg	2	0	6	0	2.9302E-17	0.0000E+00
himmelbh	2	0	5	0	-1.0000E+00	0.0000E+00
himmelbi	100	12	17	0.07	-1.7550E+03	0.0000E+00
himmelbj	43	14	**IL**			
himmelbk	24	14	9	0.08	5.1814E-02	9.7298E-09
himmelp1	2	0	10	0.01	-5.1738E+01	0.0000E+00
himmelp2	2	1	9	0.01	-6.2054E+01	0.0000E+00
himmelp3	2	2	7	0.01	-5.9013E+01	0.0000E+00
himmelp4	2	3	6	0.01	-5.9013E+01	0.0000E+00
himmelp5	2	3	9	0.03	-5.9013E+01	0.0000E+00
himmelp6	2	4	2	0.02	-5.9013E+01	0.0000E+00
hong	4	1	8	0.01	1.3473E+00	9.8752E-11
hs001	2	0	27	0.03	3.5943E-15	0.0000E+00
hs002	2	0	10	0.04	4.9412E+00	0.0000E+00
hs003	2	0	3	0	1.4567E-11	0.0000E+00

Continued on Next Page…

Problem Name	n	m	Outer Iters	Time (secs)	Objective Value	Constraint Violation
hs004	2	0	3	0	2.6667E+00	0.0000E+00
hs005	2	0	7	0.01	-1.9132E+00	0.0000E+00
hs006	2	1	4	0	1.0838E-19	1.5073E-10
hs007	2	1	15	0.02	-1.7321E+00	4.0442E-10
hs008	2	2	5	0	-1.0000E+00	6.8276E-05
hs009	2	1	3	0	-5.0000E-01	2.1885E-12
hs010	2	1	13	0.01	-1.0000E+00	1.6882E-08
hs011	2	1	6	0	-8.4985E+00	0.0000E+00
hs012	2	1	11	0.01	-3.0000E+01	0.0000E+00
hs013	2	1	**IL**			
hs014	2	2	6	0	1.3935E+00	1.5687E-09
hs015	2	2	17	0.03	3.0650E+02	1.4319E-09
hs016	2	2	8	0.05	2.3145E+01	0.0000E+00
hs017	2	2	11	0.01	1.0000E+00	0.0000E+00
hs018	2	2	7	0.01	5.0000E+00	0.0000E+00
hs019	2	2	50	0.09	-6.9618E+03	0.0000E+00
hs020	2	3	6	0.01	4.0199E+01	3.7943E-10
hs021	2	1	2	>	-9.9960E+01	0.0000E+00
hs022	2	2	5	0	1.0000E+00	0.0000E+00
hs023	2	5	7	0.01	2.0000E+00	0.0000E+00
hs024	2	2	3	0	-1.0000E+00	0.0000E+00
hs025	3	0	1	0	3.2835E+01	0.0000E+00
hs026	3	1	15	0.02	4.1531E-10	1.1465E-05
hs027	3	1	23	0.03	4.0000E-02	2.8755E-14
hs028	3	1	2	0	1.2354E-13	2.4968E-10
hs029	3	1	13	0.12	-2.2627E+01	8.4792E-08
hs030	3	1	8	0.01	1.0000E+00	1.0400E-10
hs031	3	1	6	0.01	6.0000E+00	0.0000E+00
hs032	3	2	20	0.08	1.0000E+00	4.6592E-10
hs033	3	2	14	0.1	-4.5858E+00	0.0000E+00
hs034	3	2	8	0	-8.3403E-01	8.6519E-08
hs035	3	1	2	0	1.1111E-01	0.0000E+00
hs036	3	1	6	0.04	-3.3000E+03	0.0000E+00
hs037	3	1	19	0.04	-3.4560E+03	0.0000E+00
hs038	4	0	39	0.06	1.4400E-12	0.0000E+00
hs039	4	2	13	0.01	-1.0000E+00	7.0566E-09
hs040	4	3	5	0	-2.5000E-01	7.6080E-12
hs041	4	1	6	0	1.9259E+00	9.4233E-09
hs042	3	1	6	0	1.3858E+01	1.1527E-11
hs043	4	3	8	0.01	-4.4000E+01	0.0000E+00
hs044	4	6	3	0.01	-1.5000E+01	0.0000E+00
hs045	5	0	1	0	2.0000E+00	0.0000E+00
hs046	5	2	16	0.02	5.3101E-10	5.6864E-06
hs047	5	3	18	0.06	3.1877E-11	1.5131E-07

Continued on Next Page…

Problem Name	n	m	Outer Iters	Time (secs)	Objective Value	Constraint Violation
hs048	5	2	3	0	1.0595E-21	2.5629E-09
hs049	5	2	17	0.02	1.3887E-09	1.5595E-10
hs050	5	3	9	0.01	8.2921E-14	1.6403E-09
hs051	5	3	3	0	9.8501E-19	7.2742E-13
hs052	5	3	2	0	5.3266E+00	3.9869E-08
hs053	5	3	4	0	4.0930E+00	9.8729E-09
hs054	6	1	4	0	1.9286E-01	1.1181E-09
hs055	6	6	3	0.02	6.6667E+00	5.1319E-12
hs056	7	4	34	0.12	-7.8115E-11	1.9993E-05
hs057	2	1	3	0	3.0648E-02	0.0000E+00
hs059	2	3	26	0.28	-7.8028E+00	0.0000E+00
hs060	3	1	7	0.01	3.2568E-02	1.1961E-07
hs061	3	2	9	0.01	-1.4365E+02	7.3316E-09
hs062	3	1	27	0.03	-2.6273E+04	9.5196E-09
hs063	3	2	50	0.09	9.6172E+02	8.5869E-10
hs064	3	1	21	0.03	6.2998E+03	3.5624E-10
hs065	3	1	9	0.02	9.5353E-01	1.7404E-10
hs066	3	2	7	0	5.1816E-01	7.4648E-10
hs067	8	21	312	0.99	-1.1620E+03	2.0365E-07
hs070	4	1	21	0.1	8.9232E-03	0.0000E+00
hs071	4	2	7	0.01	1.7014E+01	2.8945E-10
hs072	4	2	27	0.04	7.2768E+02	9.7447E-09
hs073	4	3	5	0.01	2.9894E+01	1.4294E-09
hs074	4	4	86	0.15	5.1265E+03	4.9795E-11
hs075	4	4	81	0.16	5.1744E+03	1.0690E-08
hs076	4	3	3	0	-4.6818E+00	0.0000E+00
hs077	5	2	10	0.01	2.4151E-01	6.8001E-10
hs078	5	3	7	0	-2.9197E+00	2.3050E-09
hs079	5	3	5	0	7.8777E-02	1.1930E-08
hs080	5	3	8	0.01	5.3950E-02	8.3699E-07
hs081	5	3	33	0.07	5.3950E-02	3.4577E-11
hs083	5	3	6	0	-3.0666E+04	4.8677E-09
hs084	5	3	**IL**			
hs085	5	38	40	0.31	-1.9052E+00	0.0000E+00
hs086	5	6	5	0	-3.2349E+01	0.0000E+00
hs087	11	6	49	0.09	8.8276E+03	4.0567E-10
hs088	2	1	19	0.05	1.3627E+00	3.3729E-11
hs089	3	1	19	0.08	1.3627E+00	3.3606E-11
hs090	4	1	123	0.93	1.3627E+00	9.4641E-13
hs091	5	1	49	0.45	1.3627E+00	0.0000E+00
hs092	6	1	79	0.69	1.3627E+00	0.0000E+00
hs093	6	2	139	0.3	1.3514E+02	1.6520E-07
hs095	6	4	3	0	1.5620E-02	0.0000E+00
hs096	6	4	3	0	1.5620E-02	0.0000E+00

Continued on Next Page…

Problem Name	n	m	Outer Iters	Time (secs)	Objective Value	Constraint Violation
hs097	6	4	**IL**			
hs098	6	4	13	0.02	3.1358E+00	0.0000E+00
hs099	19	14	**IL**			
hs100	7	4	7	0.01	6.8063E+02	5.5387E-07
hs100lnp	7	2	10	0.01	6.8063E+02	2.8110E-09
hs100mod	7	4	8	0.01	6.7875E+02	0.0000E+00
hs101	7	6	**IL**			
hs102	7	6	21	0.06	9.1188E+02	8.6831E-09
hs103	7	6	21	0.05	5.4367E+02	0.0000E+00
hs104	8	6	10	0.01	3.9512E+00	0.0000E+00
hs105	8	0	20	0.28	1.1363E+03	0.0000E+00
hs106	8	6	**IL**			
hs107	9	6	28	0.05	5.0550E+03	2.1564E-09
hs108	9	13	40	0.22	-8.6603E-01	1.6542E-10
hs109	9	10	97	0.3	5.3269E+03	1.9168E-10
hs110	10	0	6	0	-4.5778E+01	0.0000E+00
hs111	10	3	20	0.08	-4.5151E+01	2.7296E-08
hs111lnp	10	3	20	0.08	-4.5151E+01	2.7296E-08
hs112	10	3	14	0.02	-4.7761E+01	2.8775E-10
hs113	10	8	7	0.01	2.4306E+01	1.1292E-08
hs114	10	11	239	0.83	-1.7688E+03	2.9446E-06
hs116	13	15	**IL**			
hs117	15	5	8	0.03	3.2349E+01	0.0000E+00
hs118	15	17	3	0	6.6482E+02	0.0000E+00
hs119	16	8	10	0.02	2.4490E+02	1.7398E-10
hs21mod	7	1	13	0.02	-9.5960E+01	0.0000E+00
hs268	5	5	3	0	2.8787E-06	0.0000E+00
hs35mod	2	1	2	0	2.5000E-01	0.0000E+00
hs3mod	2	0	3	0	2.1777E-08	0.0000E+00
hs44new	4	5	2	0	-3.0000E+00	0.0000E+00
hs99exp	28	21	**IL**			
hubfit	2	1	3	0	1.6894E-02	0.0000E+00
hues-mod	10000	2	**IL**			
huestis	10000	2	**IL**			
humps	2	0	152	0.81	1.6124E+01	0.0000E+00
hvycrash	201	150	**IL**			
hypcir	2	0	5	0	0.0000E+00	6.9871E-08
indef	1000	0	**IL**			
integreq	100	0	3	0.22	0.0000E+00	9.6668E-06
jensmp	2	0	335	1.62	1.2436E+02	0.0000E+00
kissing	127	903	**IL**			
kiwcresc	3	2	13	0.01	1.2761E-08	0.0000E+00
kowosb	4	0	17	0.06	3.0751E-04	0.0000E+00
ksip	20	1000	3	0.57	5.7580E-01	0.0000E+00

Continued on Next Page…

Problem Name	n	m	Outer Iters	Time (secs)	Objective Value	Constraint Violation
lakes	90	78	**IL**			
launch	25	29	**IL**			
lch	600	1	22	1.42	-4.2877E+00	1.8419E-09
lewispol	6	9	5	0	1.1268E+00	9.0002E-05
liarwhd	10000	0	14	8.57	1.8493E-17	0.0000E+00
linspanh	72	32	2	0	-7.7000E+01	8.2577E-13
liswet1	10002	10000	9	8.53	2.5012E+01	3.7982E-04
liswet10	10002	10000	12	11.39	2.5001E+01	2.8227E-04
liswet11	10002	10000	36	58.66	5.1062E+01	2.7673E-06
liswet12	10002	10000	**IL**			
liswet2	10002	10000	5	5.56	2.5000E+01	9.3796E-06
liswet3	10002	10000	6	13.54	2.5000E+01	1.0279E-08
liswet4	10002	10000	6	9.86	2.5000E+01	4.1978E-07
liswet5	10002	10000	6	12.27	2.5000E+01	1.4777E-07
liswet6	10002	10000	5	6.56	2.5000E+01	5.0767E-07
liswet7	10002	10000	31	48.01	3.9077E+01	4.2711E-04
liswet8	10002	10000	**IL**			
liswet9	10002	10000	**IL**			
lminsurf	15129	0	17	37.45	9.0041E+00	0.0000E+00
loadbal	31	31	9	0.03	4.5285E-01	2.5075E-11
loghairy	2	0	35	0.24	6.5524E+00	0.0000E+00
logros	2	0	81	0.5	3.3129E-13	0.0000E+00
lootsma	3	2	14	0.09	1.4142E+00	0.0000E+00
lotschd	12	7	5	0.01	2.3984E+03	1.6458E-08
lsnnodoc	5	4	9	0.01	1.2311E+02	8.7438E-09
lsqfit	2	1	3	0	3.3787E-02	0.0000E+00
madsen	3	6	12	0.04	6.1643E-01	0.0000E+00
madsschj	81	158	**IL**			
makela1	3	2	10	0.01	-1.4142E+00	0.0000E+00
makela2	3	3	15	0.02	7.2000E+00	0.0000E+00
makela3	21	20	30	0.1	-2.8106E-10	2.9936E-07
makela4	21	40	3	0.01	2.3033E-10	0.0000E+00
mancino	100	0	168	18.21	2.4400E-12	0.0000E+00
manne	1094	730	6	1.28	-9.7425E-01	0.0000E+00
maratos	2	1	7	0	-1.0000E+00	2.0749E-10
maratosb	2	0	400	1.26	-1.0000E+00	0.0000E+00
matrix2	6	2	14	0.03	1.5699E-08	0.0000E+00
maxlika	713	0	20	0.28	1.1363E+03	0.0000E+00
mccormck	50000	0	7	>	-4.5662E+04	0.0000E+00
mconcon	15	11	**IL**			
mdhole	2	0	20	0.02	9.4726E-11	0.0000E+00
methanb8	31	0	9	0.02	8.7640E-07	0.0000E+00
methanl8	31	0	126	0.33	4.6035E-04	0.0000E+00
mexhat	2	0	6	0	-4.0100E-02	0.0000E+00

Continued on Next Page...

Problem Name	n	m	Outer Iters	Time (secs)	Objective Value	Constraint Violation
meyer3	3	0	**IL**			
mifflin1	3	2	7	0.01	-1.0000E+00	7.9343E-10
mifflin2	3	2	14	0.01	-1.0000E+00	0.0000E+00
minc44	303	262	7	1.53	2.5774E-03	1.5026E-07
minmaxbd	5	20	19	0.05	1.1571E+02	0.0000E+00
minmaxrb	3	4	12	0.01	6.8315E-06	0.0000E+00
minperm	1113	1227	5	948.94	3.6288E-04	2.9800E-10
minsurf	36	0	5	0.01	1.0000E+00	0.0000E+00
mistake	9	13	149	0.71	-1.0000E+00	3.3631E-08
model	60	32	21	0.08	5.7422E+03	3.3692E-08
morebv	5000	0	1	0.07	1.0395E-11	0.0000E+00
mosarqp1	2500	700	4	0.88	-9.5288E+02	0.0000E+00
mosarqp2	900	600	9	3.17	-1.5975E+03	0.0000E+00
msqrta	1024	0	6	>	0.0000E+00	1.3623E-05
msqrtals	1024	0	**mem**			
msqrtb	1024	0	**time**			
msqrtbls	1024	0	**mem**			
mwright	5	3	9	0.01	2.4979E+01	1.2124E-08
nasty	2	0	**IL**			
ncvxbqp1	10000	0	**mem**			
ncvxbqp2	10000	0	**mem**			
ncvxbqp3	10000	0	**mem**			
ncvxqp1	1000	500	**IL**			
ncvxqp2	1000	500	185	>	-5.7785E+07	9.4635E-07
ncvxqp3	1000	500	**IL**			
ncvxqp4	1000	250	73	225.67	-9.4013E+07	9.7292E-07
ncvxqp5	1000	250	119	394.6	-6.6376E+07	1.0523E-07
ncvxqp6	1000	250	424	1206.3	-3.4620E+07	1.1313E-06
ncvxqp7	1000	750	109	>	-4.3420E+07	1.6346E-06
ncvxqp8	1000	750	**IL**			
ncvxqp9	1000	750	**IL**			
ngone	97	1273	33	>	-6.0910E-01	1.9124E-08
noncvxu2	1000	0	193	625.53	2.3362E+03	0.0000E+00
noncvxun	1000	0	67	1.22	2.3168E+03	0.0000E+00
nondia	9999	0	18	2.246	1.2022E-13	0.0000E+00
nondquar	10000	0	14	6.54	7.0004E-06	0.0000E+00
nonmsqrt	9	0	26	0.06	1.6384E+00	0.0000E+00
nonscomp	10000	0	84	89.9	1.3404E-05	0.0000E+00
obstclal	100	0	3	0	1.3979E+00	0.0000E+00
obstclbl	100	0	5	0.01	2.8750E+00	0.0000E+00
obstclbu	100	0	5	0.01	2.8750E+00	0.0000E+00
odfits	10	6	29	0.05	-2.3800E+03	2.0509E-09
oet1	3	1002	2	0.08	5.3833E-01	0.0000E+00
oet2	3	1002	6	0.49	8.7160E-02	9.0218E-10

Continued on Next Page...

Problem Name	n	m	Outer Iters	Time (secs)	Objective Value	Constraint Violation
oet3	4	1002	2	0.09	4.5113E-03	0.0000E+00
oet7	7	1002	6	0.64	8.7160E-02	7.7538E-10
optcdeg2	1198	799	8	0.64	2.2959E+02	1.5082E-06
optcdeg3	1198	799	16	1.55	4.6181E+01	1.6231E-06
optcntrl	28	20	2	0	5.5000E+02	2.6636E-08
optctrl3	118	80	24	0.15	2.0480E+03	1.4347E-07
optctrl6	118	80	24	0.14	2.0480E+03	1.4347E+00
optmass	66	55	34	0.26	-5.8684E-05	1.5400E-07
optprloc	30	29	20	0.1	-1.6420E+01	0.0000E+00
orthrdm2	4003	2000	7	2.87	1.5553E+02	2.0624E-07
orthrds2	203	100	**IL**			
orthrega	517	256	**IL**			
orthregb	27	6	8	0.01	2.6599E-17	8.5020E-05
orthregc	10005	5000	**IL**			
orthregd	10003	5000	8	17.36	1.5239E+03	1.2952E-07
orthrege	36	20	**IL**			
orthrgdm	10003	5000	**time**			
orthrgds	10003	5000	**mem**			
osbornea	5	0	41	0.06	5.4670E-05	0.0000E+00
osborneb	11	0	54	0.22	3.1334E-01	0.0000E+00
oslbqp	8	0	12	0.02	6.2500E+00	0.0000E+00
palmer1	4	0	73	0.17	1.1755E+04	0.0000E+00
palmer1a	6	0	32	0.06	8.9884E-02	0.0000E+00
palmer1b	4	0	18	0.04	3.4474E+00	0.0000E+00
palmer1c	8	0	**IL**			
palmer1d	7	0	**IL**			
palmer1e	8	0	94	0.17	6.3040E-02	0.0000E+00
palmer2	4	0	23	0.06	4.5811E+03	0.0000E+00
palmer2a	6	0	55	0.13	1.7161E-02	0.0000E+00
palmer2b	4	0	23	0.06	6.2339E-01	0.0000E+00
palmer2c	8	0	**IL**			
palmer2e	8	0	61	0.09	2.6274E-02	0.0000E+00
palmer3	4	0	12	0.04	2.4170E+03	0.0000E+00
palmer3a	6	0	8	0.01	4.5559E+00	0.0000E+00
palmer3b	4	0	10	0.02	4.2276E+00	0.0000E+00
palmer3c	8	0	41	0.1	2.1453E-01	0.0000E+00
palmer3e	8	0	46	0.08	1.3838E-01	0.0000E+00
palmer4	4	0	10	0.03	2.4240E+03	0.0000E+00
palmer4a	6	0	8	0.01	6.8903E+00	0.0000E+00
palmer4b	4	0	11	0.02	6.8351E+00	0.0000E+00
palmer4c	8	0	43	0.1	4.0865E-01	0.0000E+00
palmer4e	8	0	77	0.33	1.4803E-04	0.0000E+00
palmer5a	8	0	10	0.01	2.1281E+00	0.0000E+00
palmer5b	9	0	236	0.43	1.5158E-02	0.0000E+00

Continued on Next Page…

Problem Name	n	m	Outer Iters	Time (secs)	Objective Value	Constraint Violation
palmer5c	6	0	6	0	2.1281E+00	0.0000E+00
palmer5d	4	0	12	0.02	8.7339E+01	0.0000E+00
palmer5e	8	0	483	0.77	2.9508E-02	0.0000E+00
palmer6a	6	0	110	0.27	5.5949E-02	0.0000E+00
palmer6c	8	0	165	0.27	1.4492E-01	0.0000E+00
palmer6e	8	0	42	0.07	2.2766E-04	0.0000E+00
palmer7a	6	0	**IL**			
palmer7c	8	0	97	0.25	4.3613E+00	0.0000E+00
palmer7e	8	0	**IL**			
palmer8a	6	0	35	0.06	7.4010E-02	0.0000E+00
palmer8c	8	0	149	0.3	5.9929E-01	0.0000E+00
palmer8e	8	0	54	0.1	6.3393E-03	0.0000E+00
penalty1	1000	0	68	333.78	9.8917E-03	0.0000E+00
penalty2	100	0	21	0.19	9.7096E+04	0.0000E+00
pentagon	6	12	7	0.18	1.4513E-04	0.0000E+00
pentdi	1000	0	5	0.17	-7.5000E-01	0.0000E+00
pfit1	3	0	319	0.54	1.0719E-12	0.0000E+00
pfit1ls	3	0	319	0.56	1.0719E-12	0.0000E+00
pfit2	3	0	137	0.22	5.8591E-08	0.0000E+00
pfit2ls	3	0	137	0.22	5.8591E-08	0.0000E+00
pfit3	3	0	**err**			
pfit3ls	3	0	**err**			
pfit4	3	0	179	0.35	2.9810E-11	0.0000E+00
pfit4ls	3	0	179	0.37	2.9810E-11	0.0000E+00
polak1	3	2	6	0.01	2.7183E+00	6.1571E-08
polak2	11	2	38	0.09	5.4598E+01	7.4741E-06
polak3	12	10	20	0.05	5.9330E+00	9.5425E-09
polak4	3	3	32	0.08	3.5461E-09	0.0000E+00
polak5	3	2	68	0.13	5.0000E+01	5.9342E-05
polak6	5	4	82	0.2	-4.4000E+01	0.0000E+00
porous1	4900	0	13	52.57	0.0000E+00	5.2399E-06
porous2	4900	0	9	39.36	0.0000E+00	8.2168E-06
portfl1	12	1	3	0	2.0486E-02	0.0000E+00
portfl2	12	1	3	0	2.9689E-02	0.0000E+00
portfl3	12	1	3	0	3.2750E-02	5.5511E-16
portfl4	12	1	3	0	2.6307E-02	3.1086E-15
portfl6	12	1	3	0	2.5792E-02	4.4409E-16
powell20	1000	1000	28	1.87	5.2146E+07	4.3038E-07
powellbs	2	0	40	0.1	0.0000E+00	6.0736E-05
powellsq	2	0	40	0.06	0.0000E+00	5.6474E-06
power	1000	0	6	0.08	4.0488E-11	0.0000E+00
probpenl	500	0	13	8.84	3.9900E-07	0.0000E+00
prodpl0	60	29	9	0.03	6.0919E+01	2.1684E-09
prodpl1	60	29	9	0.03	5.3037E+01	7.0154E-10

Continued on Next Page…

Problem Name	n	m	Outer Iters	Time (secs)	Objective Value	Constraint Violation
pspdoc	4	0	8	0.01	2.4142E+00	0.0000E+00
pt	2	501	2	0.02	1.7840E-01	0.0000E+00
qpcboei1	372	288	39	1.68	1.4434E+07	1.9537E-08
qpcboei2	143	125	73	1.16	8.2937E+06	6.4791E-07
qpcstair	385	356	42	3.32	6.2044E+06	4.0680E-07
qpnboei1	372	288	**IL**			
qpnboei2	143	125	58	8.31	1.7698E+06	9.6227E-05
qpnstair	385	356	34	6.39	5.1460E+06	5.4678E-07
qr3d	155	0	**IL**			
qr3dbd	155	0	60	1.59	1.2587E-05	0.0000E+00
qr3dls	155	0	404	51.99	4.6560E-01	0.0000E+00
qrtquad	120	0	68	0.3	-3.6481E+06	0.0000E+00
quartc	10000	0	516	84.15	2.1567E-04	0.0000E+00
qudlin	12	0	7	0.02	-7.2000E+03	0.0000E+00
reading1	10001	5000	7	1185.52	-1.7534E-02	2.7092E-05
reading2	15001	10000	4	24.3	-1.8864E-07	7.6504E-06
reading3	202	102	**IL**			
recipe	3	0	0	0	0.0000E+00	0.0000E+00
res	18	2	1	0	0.0000E+00	1.0270E-15
rk23	17	11	8	0.01	8.3333E-02	1.5765E-06
robot	7	2	13	0.01	1.2628E+01	4.7030E-10
rosenbr	2	0	24	0.02	3.0337E-13	0.0000E+00
rosenmmx	5	4	40	0.08	-4.4000E+01	0.0000E+00
s332	2	1	10	0.08	2.9924E+01	0.0000E+00
s365mod	7	5	**IL**			
s368	100	0	1	0.13	0.0000E+00	0.0000E+00
sawpath	589	782	**IL**			
scon1dls	1000	0	176	4.47	2.2515E-01	0.0000E+00
scosine	10000	0	**mem**			
scurly10	10000	0	**IL**			
scurly20	10000	0	**IL**			
semicon1	1000	0	**err**			
semicon2	1000	0	**IL**			
sensors	1000	0	36	435.24	-2.1069E+05	0.0000E+00
sim2bqp	2	0	4	0	2.7622E-12	0.0000E+00
simbqp	2	0	4	0	1.5842E-07	0.0000E+00
simpllpa	2	2	2	0	1.0000E+00	0.0000E+00
simpllpb	2	3	2	0	1.1000E+00	0.0000E+00
sineali	20	0	29	0.07	-1.8734E+03	0.0000E+00
sineval	2	0	43	0.05	5.3140E-22	0.0000E+00
sinquad	10000	0	38	82.16	6.2197E-10	0.0000E+00
sinrosnb	1000	999	1	0.01	-9.9901E+04	0.0000E+00
sipow1	2	10000	2	1.82	-9.9975E-01	0.0000E+00
sipow1m	2	10000	2	1.93	-1.0000E+00	0.0000E+00

Continued on Next Page…

Problem Name	n	m	Outer Iters	Time (secs)	Objective Value	Constraint Violation
sipow2	2	5000	2	0.64	-9.9985E-01	0.0000E+00
sipow2m	2	5000	2	0.58	-9.9980E-01	0.0000E+00
sipow4	4	10000	2	2.62	2.7283E-01	5.4671E-07
sisser	2	0	15	0.01	4.1034E-10	0.0000E+00
smbank	117	64	**IL**			
smmpsf	720	263	112	5.95	1.0518E+06	7.5073E-07
snake	2	2	79	0.36	4.2793E-05	0.0000E+00
sosqp1	20000	20000	0	0	0.0000E+00	0.0000E+00
sosqp2	20000	10001	5	>	-4.9987E+03	2.9961E-06
spanhyd	72	32	58	0.39	2.3974E+02	6.8633E-08
spiral	3	2	49	0.09	-6.1121E-10	1.2227E-09
sreadin3	10000	5000	3	934.36	-1.9443E-05	2.2486E-05
srosenbr	10000	0	24	3.57	1.6322E-09	0.0000E+00
sseblin	192	72	391	2.72	1.6171E+07	6.7717E-08
ssebnln	192	96	**IL**			
ssnlbeam	31	20	**IL**			
stancmin	3	2	6	0	4.2500E+00	0.0000E+00
static3	434	96	**IL**			
steenbra	432	108	91	29.36	1.6958E+04	5.5879E-08
steenbrb	468	108	**IL**			
steenbrc	540	126	**IL**			
steenbrd	468	108	**IL**			
steenbre	540	126	**IL**			
steenbrf	468	108	**IL**			
steenbrg	540	126	**IL**			
supersim	2	2	4	0	6.6667E-01	1.0193E-08
svanberg	5000	5000	10	7.05	8.3614E+03	2.8800E-06
swopf	82	91	7	0.05	6.7860E-02	1.6890E-07
synthes1	6	6	5	0	7.5928E-01	3.5612E-11
tame	2	1	2	0	0.0000E+00	1.5321E-14
tfi2	3	10000	2	3.11	6.4904E-01	0.0000E+00
tointqor	50	0	3	0	1.1755E+03	0.0000E+00
trainf	20000	10002	17	-1000	3.4877E+00	6.7600E-06
trainh	20000	10002	**time**			
tridia	10000	0	5	1.59	1.3282E-17	0.0000E+00
trimloss	142	72	15	0.14	9.0600E+00	1.7094E-09
try-b	2	1	9	0.01	7.9286E-20	7.1412E-10
twirism1	343	313	**IL**			
twobars	2	2	7	0	1.5087E+00	9.4641E-09
ubh1	17997	12000	124	>	1.1456E+01	5.8959E-05
ubh5	19997	14000	78	>	9.0923E+01	6.6515E-05
vanderm1	100	99	**err**			
vanderm2	100	99	**IL**			
vanderm3	100	99	**IL**			

Continued on Next Page…

Problem Name	n	m	Outer Iters	Time (secs)	Objective Value	Constraint Violation
vanderm4	9	8	**IL**			
vardim	100	0	**IL**			
watson	31	0	15	0.04	1.8427E-08	0.0000E+00
weeds	0	0	17	0.02	9.2054E+03	0.0000E+00
womflet	3	3	53	0.12	4.8637E+00	0.0000E+00
woods	10000	0	**mem**			
yao	2000	1999	24	4.69	1.9732E+02	4.1344E-06
yfit	3	0	**IL**			
yfitu	3	0	**IL**			
zangw5002	2	0	2	0.00	-1.8200E+01	0.0000E+00
zangw5003	3	0	6	0	0.0000E+00	6.8907E-07
zecevic2	2	2	2	0	-4.1250E+00	0.0000E+00
zecevic3	2	2	7	0.01	9.7309E+01	1.1216E-10
zecevic4	2	2	6	0	7.5575E+00	0.0000E+00
zigzag	58	50	218	1.86	5.0492E+00	4.4475E-07
zy2	3	1	15	0.15	2.0000E+00	0.0000E+00

Table A.2: Shows `hopdmSQP`'s success on whole of CUTE set, including problem sizes, iteration count, time taken, objective found and final constraint violation.

Table A.3: Problems for which `hopdm` fails to find an optimal solution.

Problem Name	Behaviour demonstrated
aljazzaf	Gets very close to optimal point, but doesnt terminate.
allinitc	Gets very close to optimal point, but doesnt terminate
artif	α gets very small, but constraints are not satisfied. Constraints only: they include arctan.
aug2dqp	Gets very close to optimal point, but doesnt terminate
avion2	Gets very close to optimal point, but doesnt terminate
bigbank	Is still diverging after 500 iterations. Could it be unbounded?
brownbs	Huge objective function (of order $1.0e^{+11}$). Decreasing gradually at each iteration, but (3.3a) is of order $1.0e^{+6}$ throughout.
bt1	Converges to a point where constraint violation = 1. $\alpha \to 0$.
chemrcta	Runs out of time.
chemrctb	A feasible point is not found. $\alpha \to 0$.
concon	Gets very close to optimal point, but doesnt terminate
core1	`hopdm` error code 2 is returned at every QP iteration. Increasing ρ does not help. As direction given from `hopdm` is never reliable, steepest descent is used, but it is not effective.

Continued on Next Page…

Problem Name	Behaviour demonstrated
core2	hopdm error code 2 is returned at every QP iteration. Increasing ρ does not help. As direction given from hopdm is never reliable, steepest descent is used, but it is not effective.
corkscrw	Initially, QP approximation is primal infeasible. Eventually, algorithm converges to nonfeasible point, $\alpha \to 0$ at every iteration.
cresc100	Direction found is never descent. When diagonal matrix is added to $H\!L$, a direction is found in which a full step can be taken, but this doesn't progress to an optimal point.
cresc132	Runs out of time.
cresc4	Direction found is never descent. It is still not descent after diagonal matrix is added to $H\!L$. Steepest descent is used, but this is not a suitable direction either as $\alpha \to 0$.
cresc50	Direction found is never descent. When diagonal matrix is added to $H\!L$, a direction is found in which a full step can be taken, but this doesn't progress to an optimal point.
csfi2	No obvious reason for this problem's failure to converge. hopdm terminates normally each time and a full step size is always taken.
curly20	Gets very close to optimal point, but doesnt terminate, although full step is always taken.
curly30	Gets very close to optimal point, but doesnt terminate, although full step is always taken.
cvxqp3	Converges to a nonoptimal point.
dixchlng	A feasible point is never found. α is small.
dixchlnv	A feasible point is never found. $\alpha \to 0$.
djtl	(3.3a) is of order $1.0e^{+11}$ throughout.
drcav1lq	Runs out of time after converging to an infeasible point.
drcav2lq	Runs out of time after converging to an infeasible point.
drcavty1	Runs out of time with (3.3a) nearly satisfied.
dtoc1nd	A feasible point is never found.
eigenb2	A feasible point is never found.
eigenc2	A feasible point is never found.
expquad	Gets very close to optimal point, but doesnt terminate
fletcbv3	Is still diverging after 500 iterations. Could it be unbounded?
fletchbv	Is still diverging after 478 iterations when program runs out of memory.
flosp2hh	This is an unconstrained problem. Although (3.3a) is sufficiently satisfied, the objective is still decreasing and so algorithm does not terminate.
flosp2hm	Unbounded primal at each QP iteration, despite l_1 penalty parameter ρ being increased to its maximum ($1.0e^{+16}$). Consequentially, a steepest descent direction is always used and this is not a successful direction, so $\alpha \to 0$.

Continued on Next Page…

Problem Name	Behaviour demonstrated
flosp2th	Diagonal term added to $H\!L$ at each iteration. Full step then taken in direction given by hopdm, but little progress is made.
flosp2tm	Diagonal term added to $H\!L$ at each iteration. Full step then taken in direction given by hopdm, but little progress is made.
hatfldh	Objective is constant. Feasible point is never found, and $\alpha \to 0$.
heart6ls	Diagonal term added to $H\!L$ at each iteration. Full step then taken in direction given by hopdm, but little progress is made.
heart8	Objective is constant. Partial steps made at each iteration, but feasible point is never found.
himmelbd	Objective is constant. Feasible point is never found as $\alpha \to 0$.
himmelbj	Constraints and KKT conditions (3.3) are satisfied after just 3 iterations. However, objective function is still decreasing, so algorithm does not terminate. After another 20 iterations, nonmonotonicity is always allowed and algorithm takes step-sizes of 1 in a direction which doesn't make an improvement on the current point.
hs013	(3.3a) never met. $\alpha \to 0$.
hs084	Gets very close to optimal point, but doesnt terminate. Possibly because final objective value is of order $1.0e^{+6}$ and so small fluctuations are larger than what is allowed for algorithm termination.
hs097	Takes full step, but objective is still decreasing when iteration limit is reached.
hs099	Takes full steps, but objective is still decreasing when iteration limit is reached.
hs101	Converges to optimum point on several occasions. However, (3.3a) is not satisfied with desired accuracy and next iteration allows a nonmonotone step which moves far away from the optimum.
hs106	Converges to a non-KKT point.
hs116	Partial steps taken at each iteration. Feasible point never found.
hs99exp	Feasible point is never found.
hues-mod	Progress is made at every iteration, but iteration limit is reached.
huestis	Converges to infeasible point.
hvycrash	Feasible point is never found.
indef	Unconstrained problem. Objective is still decreasing when iteration limit is reached.
kissing	Progress is made at every iteration, but iteration limit is reached.
lakes	Feasible point not found. $\alpha \to 0$.
launch	Feasible point not found. $\alpha \to 0$.
liswet12	Feasible point is not found.

Continued on Next Page...

Problem Name	Behaviour demonstrated
liswet8	Feasible point is not found. At each iteration, full step is allowed after allowing nonmonotonicity. Therefore, improvement is not made.
liswet9	Feasible point is not found.
madsschj	Unconstrained problem. Objective still decreasing when iteration limit is reached.
mconcon	Converged to optimal point, but first (3.3a) is never satisfied.
meyer3	Unconstrained problem. Objective still decreasing when iteration limit is reached.
msqrtals	Runs out of memory.
msqrtb	Runs out of time.
msqrtbls	Runs out of memory.
nasty	Unconstrained problem. Consistently far away from a KKT point.
ncvxbqp1	Gets close to optimal point but doesn't terminate. Runs out of memory.
ncvxbqp2	Gets close to optimal point but doesn't terminate. Runs out of memory.
ncvxbqp3	Gets close to optimal point but doesn't terminate. Runs out of memory.
ncvxqp1	Gets close to optimal point but doesn't terminate.
ncvxqp3	Gets close to optimal point but doesn't terminate.
ncvxqp8	Gets close to optimal point but doesn't terminate.
ncvxqp9	Gets close to optimal point but doesn't terminate.
orthrds2	Feasible point is never found.
orthrega	Feasible point is never found.
orthregc	Feasible point is never found.
orthrege	Gets close to optimal point but doesn't terminate.
orthrgdm	Runs out of time after converging to infeasible point. $\alpha \to 0$.
orthrgds	Memory error.
palmer1c	Allowing nonmonotonicity means that algorithm fluctuates around a nonoptimal point.
palmer1d	Unconstrained problem. Objective still decreasing when iteration limit is reached.
palmer2c	Unconstrained problem. Objective still decreasing when iteration limit is reached.
palmer7a	Unconstrained problem. Objective decreases slightly at each iteration and is still decreasing when iteration limit is reached.
palmer7e	Converged to optimal point, but (3.3a) is never satisfied.
qpnboei1	QP approximation is extremely difficult to solve and a QP solution is rarely found, even after regularization. Instead, the steepest descent direction is proposed and as $\alpha \to 0$ in this direction, no progress is made.

Continued on Next Page...

Problem Name	Behaviour demonstrated
qr3d	Unconstrained problem. Objective still decreasing gradually when iteration limit is reached.
reading3	Gets close to optimal point but doesn't terminate.
s365mod	Never reaches a feasible point. $\alpha \to 0$.
sawpath	Never reaches a feasible point. $\alpha \to 0$.
scosine	Runs out of memory after converging to nonoptimal point. $\alpha \to 0$.
scurly10	Still diverging (slowly) after 500 iterations.
scurly20	Unconstrained problem. No progress being made in objective, which is of order $1.0e^{+14}$ throughout iteration sequence.
semicon1	Feasible point is never found. $\alpha \to 0$. Error in implementing steepest descent method.
semicon2	Feasible point is never found. $\alpha \to 0$
smbank	Still diverging after 500 iterations.
ssebnln	(3.3a) not met.
ssnlbeam	Strong regularization required at each iteration. (3.3a) never met.
static3	Still diverging after 500 iterations. Could it be unbounded?
steenbrb	Strong regularization required at each iteration. (3.3a) never met.
steenbrc	Strong regularization required at each iteration. (3.3a) never met.
steenbrd	Strong regularization required at each iteration. (3.3a) never met.
steenbre	Strong regularization required at each iteration. (3.3a) never met.
steenbrf	Strong regularization required at each iteration. (3.3a) never met.
steenbrg	Strong regularization required at each iteration. (3.3a) never met.
trainh	Runs out of time.
twirism1	Converges to point where (3.3a) is not met.
vanderm1	Error in the implementation of steepest descent method.
vanderm2	Converges to infeasible point. $\alpha \to 0$.
vanderm3	Does not find feasible point.
vanderm4	Converges to infeasible point. $\alpha \to 0$.
vardim	Gets close to optimal point but doesn't terminate.
woods	Runs out of memory after onverging to point which does not satisfy (3.3a)
yfit	Gets close to optimal point but doesn't terminate.
yfitu	Gets close to optimal point but doesn't terminate.

Table A.3: Problems for which hopdm fails to find an optimal solution.

Appendix B
Ampl models

B.1 Mountain Pass

```
#define parameters
param p     = 200;     #no. of discretization steps
param x10   = -1.5;   #starting position on mountain
param x20   = -0.6;
param x1p   =  0.0;   #ending position on mountain
param x2p   =  0.8;

#define variables
var x1         {0..p};
var x2         {0..p};
var ux1        {0..p} <= 2, >=-2;
var ux2        {0..p} <= 2, >=-2;
var height;
var z     {i in 0..p}  = (4 - 2.1*x1[i]^2 + x1[i]^4/3)*x1[i]^2
                            + x1[i]*x2[i] + 4*(x2[i]^2-1)*x2[i]^2;

#write model
minimize maxheight: height;
s.t. x1diff{i in 0..p-1}: x1[i+1] =  x1[i] + (1/p)*ux1[i];
s.t. x2diff{i in 0..p-1}: x2[i+1] =  x2[i] + (1/p)*ux2[i];
s.t. Height{i in 0..p}  : z[i] <= height;
s.t. x1start            :   x1[0] = x10;
s.t. x1end              :   x1[p] = x1p;
s.t. x2start            :   x2[0] = x20;
s.t. x2end              :   x2[p] = x2p;
```

```
#provide sensible starting point
let {j in 0..p} x1[j] := x10 + (j/p)*(x1p-x10);
let {j in 0..p} x2[j] := x20 + (j/p)*(x2p-x20);
```

B.2 Bicycle

```
#define parameters
param p    = 50;  #no. of discretization steps
param x10  = 0;   #starting position of bike
param x20  = 0;
param x1p  = 0;   #final position of bike
param x2p  = 0;
param pi   = 3.141592;

#define variables
var x1    {0..p}; #position
var x2    {0..p};
var theta {0..p} <=2*pi, >=-2*pi; #angle of travel

#write model
maximize Tan: sum{i in 0..p}sum{i in 0..p}cos((i*pi/p)+theta[i]);
s.t. Cx1{i in 0..p-1}: x1[i+1] = x1[i] + (pi/p)*cos(theta[i]);
s.t. Cx2{i in 0..p-1}: x2[i+1] = x2[i] + (pi/p)*sin(theta[i]);
s.t. x1start         :   x1[0] = x10;
s.t. x2start         :   x2[0] = x20;
s.t. x1end           :   x1[p] = x1p;
s.t. x2end           :   x2[p] = x2p;
```

B.3 Sailing

```
#define parameters
param p   = 40;   #no. of discretization steps
param x10  = 0;   #starting position of boat
param x20  = 0;
param x1p  = 100; #final position of boat
param x2p  = 100;

#define variables
```

```
var x1      {0..p} ; #position
var x2      {0..p} ;
var theta   {0..p} >= -2.617, <=2.617;
var absthta {i in 0..p} = sqrt(theta[i]^2);
var z       {i in 0..p} =  -0.3675*absthta[i]^5 + 1.0479*theta[i]^4
                          + 0.9402*absthta[i]^3 - 4.7994*theta[i]^2
                          + 3.0336*absthta[i] + 4.8401;
var tF >= 0;         #total time of motion

#write model
minimize Ttime: tF;
s.t. Cx1{i in 0..p-1}: x1[i+1] = x1[i] + (tF/p)*z[i]*cos(theta[i]);
s.t. Cx2{i in 0..p-1}: x2[i+1] = x2[i] + (tF/p)*z[i]*sin(theta[i]);
s.t. x1start         :   x1[0] = x10;
s.t. x2start         :   x2[0] = x20;
s.t. x1end           :   x1[p] = x1p;
s.t. x2end           :   x2[p] = x2p;
s.t. time            :      tF>= 0;
```

B.4 Golf

B.4.1 1-dimensional Euler discretization

This is the ampl model for an Euler discretization of the problem described in section C.2.1.

```
#define parameters
param p   = 25;   #no. of discretization steps
param x0  = 0;    #starting position of ball
param xp  = 20;   #position of hole
param mu  = 0.07; #friction coefficient
param g   = 9.8;  #gravitational coefficient
param m   = 0.01; #mass of ball

#define variables
var x     {0..p};
var vx    {0..p};
var speed {i in 0..p} = sqrt(vx[i]^2);
var dirx  {i in 0..p} = vx[i]/speed[i];
```

```
# + or- 1, indicates direction, used to calculate friction
var tF >= 0;      #total time of motion

#write model
minimize finalspeed: vx[p]^2;
s.t. Cx{i in 0..p-1} : x[i+1] =  x[i] + (tF/p)*vx[i];
s.t. Cv{i in 0..p-1} :vx[i+1] = vx[i] - (tF/p)*mu*g;
s.t. xstart          :   x[0] = x0;
s.t. xend            :   x[p] = xp;

#give some initial starting points
let tF    := 15;
let vx[0] := 30;
```

B.4.2 2-dimensional Trapezoidal discretization

This is the ampl model for a Trapezoidal discretization of the problem described in section C.2.3.

```
#define parameters
param p    = 25;   #no. of discretization steps
param x10  = 0;    #starting position of ball
param x1p  = 20;   #position of hole
param mu   = 0.07; #friction coefficient
param g    = 9.8;  #gravitational coefficient
param m    = 0.01; #mass of ball

#define variables
var x1     {0..p};
var x3     {i in 0..p} = 2*x1[i]/5;
var vx1    {0..p};
var vx3    {0..p};
var speed  {i in 0..p} = sqrt(vx1[i]^2 + vx3[i]^2);
param dzdx  = 0.4;
param Nx3   = 1/(sqrt(1 + dzdx^2));
param Nx1   = -dzdx*Nx3;
var dirx1  {i in 0..p} = vx1[i]/speed[i];
var dirx3  {i in 0..p} = vx3[i]/speed[i];
var tF >= 0;      #total time of motion
```

```
#write model
minimize finalspeed: vx1[p]^2 + vx3[p]^2;
s.t. Cx1{i in 0..p-1} :
            x1[i+1] =  x1[i] + (tF/(2*p))*(vx1[i] + vx1[i+1]);
s.t. Cx3{i in 0..p-1} :
            x3[i+1] =  x3[i] + (tF/(2*p))*(vx3[i] + vx3[i+1]);
s.t.Cvx1{i in 0..p-1} :vx1[i+1] = vx1[i] +
(tF/(2*p))*g*Nx3*(2*Nx1-mu*(vx1[i]/speed[i]+vx1[i+1]/speed[i+1]));
s.t. xstart           :   x1[0] = x10;
s.t. xend             :   x1[p] = x1p;

#give some initial starting points
let tF     := 3;
let vx1[0] := 12;
let vx1[p] := 0.1;
```

B.4.3 3-dimensional Runge-Kutta discretization

This is the ampl model for a Runge-Kutta discretization of the problem described in section C.2.7.

```
#define parameters
param p = 25;      #no. of discretization steps
param x10  =  1;   #starting position of ball
param x20  =  2;
param x1p  =  1;   #position of hole
param x2p  = -2;
param mu   = 0.07; #friction coefficient
param g    = 9.8;  #gravitational coefficient
param m    = 0.01; #mass of ball

#define variables
var x1    {0..p};
var kx11  {0..p-1};
var kx12  {0..p-1};
var kx13  {0..p-1};
var kx14  {0..p-1};
var x2    {0..p};
var kx21  {0..p-1};
var kx22  {0..p-1};
var kx23  {0..p-1};
var kx24  {0..p-1};
var x3    {i in 0..p} = -0.3*atan(x2[i]) + 0.05*(x1[i] + x2[i]);
var kx31  {0..p-1};
var kx32  {0..p-1};
var kx33  {0..p-1};
var kx34  {0..p-1};
param dzdx              = 0.05;
var dzdy0 {i in 0..p}   = -0.3/(1 + x2[i]^2) + 0.05;
var dzdy1 {i in 0..p-1} = -0.3/(1 + (x2[i] + kx21[i]/2)^2) + 0.05;
var dzdy2 {i in 0..p-1} = -0.3/(1 + (x2[i] + kx22[i]/2)^2) + 0.05;
var dzdy3 {i in 0..p-1} = -0.3/(1 + (x2[i] + kx23[i])^2) + 0.05;
var ax1   {i in 0..p};
var ax2   {i in 0..p};
var ax3   {i in 0..p};
var Nmag0 {i in 0..p}   =
     (g-ax1[i]*dzdx - ax2[i]*dzdy0[i] + ax3[i])/sqrt(1 + dzdx^2);
var Nmag1 {i in 0..p-1} =
     (g-ax1[i]*dzdx - ax2[i]*dzdy1[i] + ax3[i])/sqrt(1 + dzdx^2);
var Nmag2 {i in 0..p-1} =
     (g-ax1[i]*dzdx - ax2[i]*dzdy2[i] + ax3[i])/sqrt(1 + dzdx^2);
var Nmag3 {i in 0..p-1} =
     (g-ax1[i]*dzdx - ax2[i]*dzdy3[i] + ax3[i])/sqrt(1 + dzdx^2);
var Nx30  {i in 0..p}   =
     (g-ax1[i]*dzdx - ax2[i]*dzdy0[i] + ax3[i])/(1 + dzdx^2);
var Nx31  {i in 0..p-1} =
     (g-ax1[i]*dzdx - ax2[i]*dzdy1[i] + ax3[i])/(1 + dzdx^2);
var Nx32  {i in 0..p-1} =
     (g-ax1[i]*dzdx - ax2[i]*dzdy2[i] + ax3[i])/(1 + dzdx^2);
var Nx33  {i in 0..p-1} =
     (g-ax1[i]*dzdx - ax2[i]*dzdy3[i] + ax3[i])/(1 + dzdx^2);
var Nx10  {i in 0..p}   = -dzdx*Nx30[i];
var Nx11  {i in 0..p-1} = -dzdx*Nx31[i];
var Nx12  {i in 0..p-1} = -dzdx*Nx32[i];
var Nx13  {i in 0..p-1} = -dzdx*Nx33[i];
var Nx20  {i in 0..p}   = -dzdy0[i]*Nx30[i];
var Nx21  {i in 0..p-1} = -dzdy1[i]*Nx31[i];
var Nx22  {i in 0..p-1} = -dzdy2[i]*Nx32[i];
var Nx23  {i in 0..p-1} = -dzdy3[i]*Nx33[i];
var vx1   {0..p};
var kvx11 {0..p-1};
var kvx12 {0..p-1};
var kvx13 {0..p-1};
var kvx14 {0..p-1};
var vx2   {0..p};
var kvx21 {0..p-1};
var kvx22 {0..p-1};
var kvx23 {0..p-1};
var kvx24 {0..p-1};
var vx3   {0..p};
var kvx31 {0..p-1};
var kvx32 {0..p-1};
var kvx33 {0..p-1};
var kvx34 {0..p-1};
var speed0 {i in 0..p}   = sqrt(vx1[i]^2 + + vx2[i]^2 + vx3[i]^2);
var speed1 {i in 0..p-1} = sqrt((vx1[i] + kvx11[i]/2)^2 +
               (vx2[i] + kvx21[i]/2)^2 + (vx3[i] + kvx31[i]/2)^2);
var speed2 {i in 0..p-1} = sqrt((vx1[i] + kvx12[i]/2)^2 +
               (vx2[i] + kvx22[i]/2)^2 + (vx3[i] + kvx32[i]/2)^2);
var speed3 {i in 0..p-1} = sqrt((vx1[i] + kvx13[i])^2   +
               (vx2[i] + kvx23[i])^2   + (vx3[i] + kvx33[i])^2);
var tF >= 0;      #total time of motion

var dirx10 {i in 0..p-1} = vx1[i]/speed0[i];
var dirx11 {i in 0..p-1} =(vx1[i] + kvx11[i]/2)/speed1[i];
var dirx12 {i in 0..p-1} =(vx1[i] + kvx12[i]/2)/speed2[i];
var dirx13 {i in 0..p-1} =(vx1[i] + kvx13[i])/speed3[i];
var dirx20 {i in 0..p-1} = vx2[i]/speed0[i];
var dirx21 {i in 0..p-1} =(vx2[i] + kvx21[i]/2)/speed1[i];
var dirx22 {i in 0..p-1} =(vx2[i] + kvy2[i]/2)/speed2[i];
var dirx23 {i in 0..p-1} =(vx2[i] + kvy3[i])/speed3[i];
var dirx30 {i in 0..p-1} = vx3[i]/speed0[i];
var dirx31 {i in 0..p-1} =(vx3[i] + kvx31[i]/2)/speed1[i];
var dirx32 {i in 0..p-1} =(vx3[i] + kvx32[i]/2)/speed2[i];
var dirx33 {i in 0..p-1} =(vx3[i] + kvx33[i])/speed3[i];

#write model
minimize finalspeed: vx1[p]^2 + vx2[p]^2 + vx3[p]^2;
s.t. Cx1k1{i in 0..p-1} :  kx11[i] = (tF/n)* vx1[i];
s.t. Cx1k2{i in 0..p-1} :  kx12[i] = (tF/n)*(vx1[i] + kvx11[i]/2);
s.t. Cx1k3{i in 0..p-1} :  kx13[i] = (tF/n)*(vx1[i] + kvx12[i]/2);
s.t. Cx1k4{i in 0..p-1} :  kx14[i] = (tF/n)*(vx1[i] + kvx13[i]);
s.t. Cx2k1{i in 0..p-1} :  kx21[i] = (tF/n)* vx2[i];
s.t. Cx2k2{i in 0..p-1} :  kx22[i] = (tF/n)*(vx2[i] + kvx21[i]/2);
s.t. Cx2k3{i in 0..p-1} :  kx23[i] = (tF/n)*(vx2[i] + kvx22[i]/2);
s.t. Cx2k4{i in 0..p-1} :  kx24[i] = (tF/n)*(vx2[i] + kvx23[i]);
s.t. Cx3k1{i in 0..p-1} :  kx31[i] = (tF/n)* vx3[i];
s.t. Cx3k2{i in 0..p-1} :  kx32[i] = (tF/n)*(vx3[i] + kvx31[i]/2);
s.t. Cx3k3{i in 0..p-1} :  kx33[i] = (tF/n)*(vx3[i] + kvx32[i]/2);
s.t. Cx3k4{i in 0..p-1} :  kx34[i] = (tF/n)*(vx3[i] + kvx33[i]);
s.t.Cvx1k1{i in 0..p-1} : kvx11[i] = (tF/n)*(vx1[i] + Nx10[i] -
                                          mu*Nmag0[i]*dirx10[i]);
s.t.Cvx1k2{i in 0..p-1} : kvx12[i] = (tF/n)*(vx1[i] + Nx11[i] -
                                          mu*Nmag1[i]*dirx11[i]);
s.t.Cvx1k3{i in 0..p-1} : kvx13[i] = (tF/n)*(vx1[i] + Nx12[i] -
                                          mu*Nmag2[i]*dirx12[i]);
s.t.Cvx1k4{i in 0..p-1} : kvx14[i] = (tF/n)*(vx1[i] + Nx13[i] -
                                          mu*Nmag3[i]*dirx13[i]);
s.t.Cvx2k1{i in 0..p-1} : kvx21[i] = (tF/n)*(vx2[i] + Nx20[i] -
                                          mu*Nmag0[i]*dirx20[i]);
s.t.Cvx2k2{i in 0..p-1} : kvx22[i] = (tF/n)*(vx2[i] + Nx21[i] -
                                          mu*Nmag1[i]*dirx21[i]);
s.t.Cvx2k3{i in 0..p-1} : kvx23[i] = (tF/n)*(vx2[i] + Nx22[i] -
                                          mu*Nmag2[i]*dirx22[i]);
s.t.Cvx2k4{i in 0..p-1} : kvx24[i] = (tF/n)*(vx2[i] + Nx23[i] -
                                          mu*Nmag3[i]*dirx23[i]);
s.t.Cvx3k1{i in 0..p-1} : kvx31[i] = (tF/n)*(vx3[i] + Nx30[i] -
                                          mu*Nmag0[i]*dirx30[i]-g);
s.t.Cvx3k2{i in 0..p-1} : kvx32[i] = (tF/n)*(vx3[i] + Nx31[i] -
                                          mu*Nmag1[i]*dirx31[i]-g);
s.t.Cvx3k3{i in 0..p-1} : kvx33[i] = (tF/n)*(vx3[i] + Nx32[i] -
                                          mu*Nmag2[i]*dirx32[i]-g);
```

```
s.t.Cvx3k4{i in 0..p-1} : kvx34[i] = (tF/n)*(vx3[i] + Nx33[i] -
                                          mu*Nmag3[i]*dirx33[i]-g);
s.t.  Cx1{i in 0..p-1}  : x1[i+1] =  x1[i] +
               ( kx11[i] +  2*kx12[i] +  2*kx13[i] +  kx14[i])/6;
s.t.  Cx2{i in 0..p-1}  : x2[i+1] =  x2[i] +
               ( kx21[i] +  2*kx22[i] +  2*kx23[i] +  kx24[i])/6;
s.t.  Cx3{i in 0..p-1}  : x3[i+1] =  x3[i] +
               ( kx31[i] +  2*kx32[i] +  2*kx33[i] +  kx34[i])/6;
s.t. Cvx1{i in 0..p-1}  : vx1[i+1] = vx1[i] +
               (kvx11[i] + 2*kvx12[i] + 2*kvx13[i] + kvx14[i])/6;
s.t. Cvx2{i in 0..p-1}  : vx2[i+1] = vx2[i] +
               (kvx21[i] + 2*kvx22[i] + 2*kvx23[i] + kvx24[i])/6;
s.t. Cvx3{i in 0..p-1}  : vx3[i+1] = vx3[i] +
               (kvx31[i] + 2*kvx32[i] + 2*kvx33[i] + kvx34[i])/6;
s.t. x1start            :   x1[0] = x10;
s.t. x1end              :   x1[p] = x1p;
s.t. x2start            :   x2[0] = x20;
s.t. x2end              :   x2[p] = x2p;

let tF    := 5;
let vx1[0] := -2;
let vx1[n] := 0.1;
```

Appendix C

Golf

In Chapter 7 we introduced three Runge-Kutta schemes which can be used to approximate optimal control problems by nonlinear programming problems. The errors associated with each of these methods was discussed and we proposed that investigating the trade-off between accuracy and solution time would be an interesting work. In this appendix, we introduce a further optimal control problem and use it to begin these comparisons.

C.1 The problem

The problem of determining the optimal speed and direction with which to hit a golf ball from a position on the green so that it reaches the hole with a small enough velocity to drop into it has been discussed by Alessandrini [1], whose work was later corrected by Vanderbei [75].

We give the green a standard coordinate system ($\mathbf{x} = (x_1, x_2, x_3)$ where x_1, x_2 represent the horizontal components of direction and x_3 represents the vertical component) and define velocity and acceleration with the same system ($\mathbf{v} = (v_{x_1}, v_{x_2}, v_{x_3})$ and $\mathbf{acc} = (acc_{x_1}, acc_{x_2}, acc_{x_3})$). Then the problem formulation is

$$\begin{array}{ll} \min & \sqrt{v_{x_1}^2(t_F) + v_{x_2}^2(t_F) + v_{x_3}^2(t_F)} \\ \text{s.t.} & \dot{\mathbf{x}}(t) \;=\; \mathbf{v}(t) \\ & \dot{\mathbf{v}}(t) \;=\; \mathbf{acc}(t), \end{array}$$

where t_F is the time taken for the ball to reach the hole, $\mathbf{x}(t)$, $\mathbf{v}(t)$, $\mathbf{acc}(t)$ are functions of time and $\mathbf{acc}(t)$ is given by Newton's equation

$$\text{Force} = \text{Mass} \times \text{Acceleration}.$$

That is, the acceleration is determined by the forces acting on the golf ball (see Figure C.1). Using the notation $(N_{x_1}(t), N_{x_2}(t), N_{x_3}(t))$ for the normal force, $(Fr_{x_1}(t), Fr_{x_2}(t), Fr_{x_3}(t))$ for the frictional force and $(0, 0, -mg)$ for the force

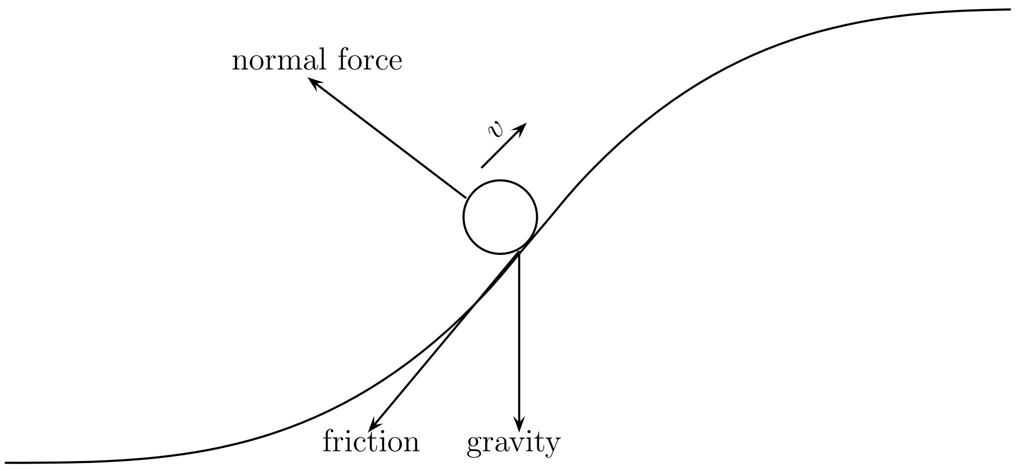


Figure C.1: Showing the forces present on a golf ball on the green.

due to gravity, where m is the mass of the golf ball and g is the gravitational constant, the problem can also be written:

$$\begin{array}{lrcl}
\min & \multicolumn{3}{l}{\sqrt{v_{x_1}^2(t_F) + v_{x_2}^2(t_F) + v_{x_3}^2(t_F)}} \\
\text{s.t.} & \dot{x}_1(t) & = & v_{x_1}(t) \\
& \dot{x}_2(t) & = & v_{x_2}(t) \\
& \dot{x}_3(t) & = & v_{x_3}(t) \\
& m\dot{v}_{x_1}(t) & = & N_{x_1}(t) + Fr_{x_1}(t) \\
& m\dot{v}_{x_2}(t) & = & N_{x_2}(t) + Fr_{x_2}(t) \\
& m\dot{v}_{x_1}(t) & = & N_{x_3}(t) + Fr_{x_3}(t) - mg.
\end{array}$$

The control variables are the components of the initial velocity given to the ball ($v_{x_1}(0)$, $v_{x_2}(0)$, $v_{x_3}(0)$).

C.2 Problem instances solved

Vanderbei [75] gives `ampl` models for Euler and Trapezoidal discretizations of the golf problem where the surface of the green is an approximation of the 18th hole of the 2000 PGA championship.

In this work, we begin more simply, first considering problems in 1 and 2 dimensions with varying curvature in the surface of the green. For each problem we have written `ampl` models which use Euler (7.1), Trapezoidal (7.2) and Runge-Kutta (7.3) discretization schemes. We use `hopdmSQP` to attempt to solve each of these models with 5, 25, 125, 250, 500 and 1000 integration steps. For each case, the starting point given to the solver was chosen judiciously, often as an interpolation of the solution of a model with fewer integration steps. A selection of the `ampl` models used are shown in Appendix B.4. The formulae which are used to calculate normal and frictional forces can be found in [1, 10, 75].

In the following sections we will describe the surface of the green for each problem considered. We will then record the time taken for each model to be solved, along with the number of outer iterations required. We also record the objective value at a solution, the time taken for the ball to traverse the green to the hole and the values of the control variables, $v_{x_1}(0), v_{x_2}(0)$. (Units of measurement are metres, metres/second and seconds.) We then use this data to comment on the relative merits of models based on different discretization schemes.

Finally, we will draw together conclusions taken from the comments made on the individual problems.

C.2.1 1D line

We begin with the simplest of green shapes. The ball is restricted to the straight 1-dimensional line between $x_1 = 0$ and $x_1 = 20$.

Results from solving discretized models of this problem are shown in Table C.1. Every time a problem was successfully solved the final speed approximated

Discr. Scheme	No. Steps	Time Taken	No. Iters	Final Speed	t_F	$v_{x_1}(0)$
E	5	0.61	23	$2.30E-15$	6.97071	4.78191
E	25	0.01	4	$4.82E-09$	7.48775	5.13660
E	125	0.06	5	$7.14E-09$	7.60567	5.21749
E	250	0.06	3	$8.87E-08$	7.62081	5.22788
E	500	0.15	3	$2.53E-08$	7.62841	5.23309
E	1000	1.07	3	$2.25E-07$	7.63222	5.23570
T	5	0.03	13	$8.72E-10$	7.63604	5.23832
T	25	0.06	15	$3.99E-09$	7.63604	5.23832
T	125	0.22	13	$1.17E-10$	7.63604	5.23832
T	250	0.32	10	$1.39E-08$	7.66658	5.23820
T	500	0.84	11	$4.08E-09$	7.66669	5.23820
T	1000	0.97	5	$8.32E-09$	7.63604	5.23832
RK	5	0.02	7	$7.03E-09$	7.69231	5.26735
RK	25	0.24	5	$3.74E-10$	7.73993	5.23823
RK	125	0.28	4	$6.25E-09$	7.63604	5.23832
RK	250	1.57	6	$1.07E-08$	7.63604	5.23832
RK	500	3.69	7	$3.75E-09$	7.64113	5.23832
RK	1000	10.51	6	$8.81E-10$	7.63604	5.23832

Table C.1: Results from using `hopdmSQP` to solve Euler, Trapezoidal and Runge-Kutta approximations of the golf problem when the green surface is a 1-dimensional line.

0, so little can be told about the accuracy of the solution by considering whether improvement is made in the objective.

However, we can see that the optimal solutions to each model display similarities. The solution to Trapezoidal models with 5, 25, 125 and 1000 integration steps and to Runge-Kutta models with 125, 250 and 1000 integration steps are $t_F = 7.63604$, $v_{x_1}(0) = 5.23832$. Also, t_F and $v_{x_1}(0)$ found using Euler discretizations both increase towards these values with each refinement of the discretization.

C.2.2 2D flat plane

Our next green is a level plane. The ball is placed at $x_1 = 0$, $x_2 = 10$ and is hit towards the hole at $x_1 = 20$, $x_2 = 0$. The optimal trajectory found is along the straight line connecting these two points and the results of solving models of this problem are shown in Table C.2.

Discr. Scheme	No. Steps	Time Taken	No. Iters	Final Speed	t_F	$v_{x_1}(0)$	$v_{x_2}(0)$
E	5	0.00	4	$5.38E-09$	7.37063	4.52245	-2.26123
E	25	0.02	5	$4.28E-08$	7.91733	4.69610	-2.42895
E	125	0.13	4	$1.01E-09$	8.04202	4.93440	-2.46720
E	250	0.21	3	$8.02E-07$	8.05802	4.94422	-2.47211
E	500	0.47	3	$7.11E-07$	8.06606	4.94915	-2.47458
E	1000	1.21	3	$9.20E-06$	8.07008	4.95162	-2.47581
T	5	solution not found					
T	25	0.32	24	$7.28E-06$	8.07411	4.95410	-2.47705
T	125	72.20	27	$5.77E-04$	8.07328	4.95410	-2.47705
T	250	solution not found					
T	500	solution not found					
T	1000	26.78	18	$5.98E-03$	8.08215	4.95409	-2.47705
RK	5	0.06	9	$2.85E-01$	6.61884	5.30979	-2.65490
RK	25	4.03	7	$1.28E-02$	7.93598	4.97338	-2.48699
RK	125	2.96	7	$1.28E-02$	7.91400	4.97333	-2.48666
RK	250	solution not found					
RK	500	solution not found					
RK	1000	solution not found					

Table C.2: Results from using `hopdmSQP` to solve Euler, Trapezoidal and Runge-Kutta approximations of the golf problem when the green surface is a 2-dimensional flat plane.

Firstly, it can be seen that the final speed increases as the number of integration steps used in Euler discretizations increases and is greater still when found using Trapezoidal or Runge-Kutta discretization schemes. This is not unexpected. In fact, as less accurate discretizations are also less constrained, it is likely that solving these models would find solutions with better objective values.

It is interesting to note that, with the exception of the solution to the Euler discretization model with 25 integration steps, the initial velocity given to the ball propels it directly towards the hole ($v_{x_1}(0) \approx -2 \times v_{x_2}(0)$).

The solutions found by the different discretization schemes differ slightly, but from the data, we can suggest that the optimal trajectory begins with an initial velocity of $v_{x_1}(0) \in (4.9, 5)$, $v_{x_2}(0) \in (-2.5, -2.4)$ and takes approximately 8 seconds.

C.2.3 2D tilted line

The next green to be considered is another 2-dimensional green. In this problem instance, one of the dimensions is horizontal and the other is vertical (x_1 and x_3, say). The green is a straight line, tilted such that $x_3 = \frac{2}{5}x_1$. The ball starts at $x_1 = 0$ and is hit towards the hole at $x_1 = 20$. The results found when solving this problem with different discretizations are shown in Table C.3.

Discr. Scheme	No. Steps	Time Taken	No. Iters	Final Speed	t_F	$v_{x_1}(0)$
E	5	0.07	15	$1.45E-10$	2.89739	11.50462
E	25	0.45	32	$2.96E-06$	3.11229	12.35795
E	125	10.72	15	$7.57E-07$	3.22296	12.55270
E	250	5.10	11	$2.04E-07$	3.16760	12.57760
E	500	8.87	9	$1.15E-07$	3.17076	12.59010
E	1000	13.70	8	$8.99E-08$	3.17234	12.59640
T[1]	5	0.20	47	$1.72E-06$	3.26616	12.56770
T[1]	25	1.98	217	$2.17E-06$	3.19624	12.60190
T[1]	125	0.79	8	$3.51E-06$	3.17781	12.60260
T	250	5.48	18	$3.53E-07$	3.17407	12.60270
T	500	4.37	11	$2.93E-07$	3.17390	12.60270
T	1000	15.35	11	$1.50E-07$	3.17411	12.60270
RK	5	solution not found				
RK	25	solution not found				
RK	125	solution not found				
RK	250	solution not found				
RK	500	solution not found				
RK	1000	solution not found				

Table C.3: Results from using `hopdmSQP` to solve Euler, Trapezoidal and Runge-Kutta approximations of the golf problem when the green surface is a 2-dimensional tilted line.

We were unable to find a starting point from which any of the models which use a Runge-Kutta discretization scheme were able to converge to an optimal

[1]KKT condition (3.3a) satisfied to accuracy less than requested.

point. This, and the fact that the solver converges to a point which is not as accurate as initially requested when solving the Trapezoidal models with 5, 25 and 125 integration steps, reflects that these models, which would theoretically give more accurate solutions, are also more difficult to solve than the models which use an Euler discretization scheme.

However, when the models which use a Trapezoidal discretization scheme can be solved to the requested accuracy, t_F is always 3.174 and $v_{x_1}(0)$ is always 12.603, values which the solutions of models which use an Euler discretization scheme tend towards as the number of integration steps increases.

C.2.4 2D curve

We now consider another 2-dimensional green with one horizontal and one vertical dimension. In this problem instance we add curvature to the shape of the green, aiming to hit the ball along the curves provided by a damped sin wave,

$$x_3 = \frac{4sin(2x_1)}{x_1}.$$

This green is shown in Figure C.2, which also shows the start point of the ball

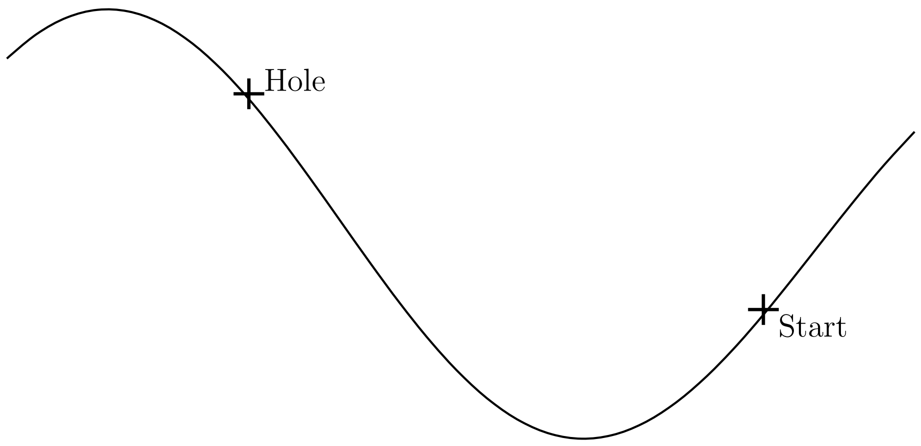


Figure C.2: Showing the start and finish points of a golf ball's trajectory over a 2-dimensional green shaped as a damped sin curve.

($x_1 = 15.5$) and the position of the hole ($x_1 = 13.8$). This is the first problem for which we are required to use the corrected formulation of Vanderbei [75]. In his work on finding optimal golf trajectories he found that Alessandrini's formulation [1] was only valid for problems where the green surface is planar. In order to adapt the formulation for problems where the green surface is curved, he notices that the normal force is not constant on a curved surface, as Alessandrini proposed.

See [75] for more details of how the normal force can be adjusted to take this into account.

The results found when solving discretized models of this problem are shown in Table C.4.

Discr. Scheme	No. Steps	Time Taken	No. Iters	Final Speed	t_F	$v_{x_1}(0)$
E	5	0.05	19	$1.19E+00$	0.95736	-0.73091
E	25	3.33	91	$3.18E-11$	0.90599	-2.24495
E	125	0.15	5	$4.66E-09$	0.88788	-2.63845
E	250	0.40	5	$7.01E-11$	0.88610	-2.69005
E	500	1.14	5	$2.53E-08$	0.88523	-2.71609
E	1000	2.84	5	$4.54E-09$	0.88482	-2.72918
T	5	0.16	34	$2.43E-09$	0.84529	-2.45797
T	25	0.73	25	$4.41E-04$	0.77707	-2.79796
T	125	solution not found				
T	250	solution not found				
T	500	solution not found				
T	1000	solution not found				
RK	5	solution not found				
RK	25	solution not found				
RK	125	solution not found				
RK	250	solution not found				
RK	500	solution not found				
RK	1000	solution not found				

Table C.4: Results from using `hopdmSQP` to solve Euler, Trapezoidal and Runge-Kutta approximations of the golf problem when the green surface is a 2-dimensional curve.

We were again unable to find starting points from which solutions to the Runge-Kutta models of this problem could be found and were also unable to find starting points from which Trapezoidal models with more than 25 discretization steps could converge.

In fact, we only have two reasonable solutions to models which use a Trapezoidal discretization scheme and only the first of these (5 integration steps) is similar to the solutions of models which use an Euler discretization scheme. The second (25 integration steps) has a final speed which is significantly higher than the final speeds found by using all but the first of the Euler discretizations.

So, taking the solutions from models which use the Euler discretization scheme as our guide, we conjecture that an optimal solution to this problem is $t_F \approx 0.88$, $v_{x_1}(0) \approx -2.73$.

C.2.5 3D tilted plane

Now we are ready to consider 3-dimensional greens. We start with a planar green, which enables us to use Alessandrini's simpler model formulation. The green we consider is $x_3 = \frac{x_1}{4} + \frac{x_2}{6}$. The ball is placed at $x_1 = x_2 = 1$ and hit towards the hole at $x_1 = x_2 = 10$. An optimal trajectory found is shown in Figure C.3 (page 133). The ball is hit towards a point above the hole and as it slows down it falls to the hole.

Numerical results found when solving discretized models of the problem are shown in Table C.5. If we study the data closely we can clearly see disadvantages

Discr. Scheme	No. Steps	Time Taken	No. Iters	Final Speed	t_F	$v_{x_1}(0)$	$v_{x_2}(0)$
E	5	0.04	14	$6.99E-01$	2.54224	6.29010	5.50250
E	25	0.07	7	$5.62E-01$	2.76276	6.83272	5.82559
E	125	2.92	12	$5.30E-01$	2.81460	6.95580	5.90364
E	250	59.30	19	$5.26E-01$	2.82123	6.97154	5.91378
E[2]	500	165.11	25	$1.31E+00$	3.35671	7.13092	5.91810
E	1000	209.54	11	$5.23E-01$	2.82623	6.98340	5.92143
T	5	0.01	6	$5.07E-01$	2.83560	6.96891	5.92291
T	25	0.04	5	$5.22E-01$	2.83150	6.98719	5.92348
T	125	0.27	4	$5.21E-01$	2.83157	6.98793	5.92344
T	250	0.64	4	$5.21E-01$	2.83158	6.98795	5.92343
T	500	1.65	4	$5.21E-01$	2.83158	6.98796	5.92343
T	1000	4.28	4	$5.21E-01$	2.83158	6.98796	5.92343
RK	5	0.27	20	$5.33E-01$	2.82102	6.98716	5.92532
RK	25	0.73	7	$5.23E-01$	2.82792	6.98736	5.92399
RK	125	7.76	7	$5.22E-01$	2.82791	6.92567	5.92399
RK[2]	250	343.84	17	$5.22E-01$	2.82791	6.95651	5.92399
RK	500	721.03	8	$6.26E-01$	2.96327	6.99815	5.91087
RK	1000	solution not found					

Table C.5: Results from using `hopdmSQP` to solve Euler, Trapezoidal and Runge-Kutta approximations of the golf problem when the green surface is a 3-dimensional tilted plane.

of both the Euler and Runge-Kutta discretization schemes:

The mean value found for t_F value is 2.84619. If we assume that this mean value is close to that of an optimal trajectory then it is interesting to consider the solutions which are farthest from it. Only 4 of the values for t_F found by solving our discretized models differ from the mean by more than 0.05. These values are:

- those found by the first two Euler discretizations. This is expected, as the

[2]KKT condition (3.3a) satisfied to accuracy less than requested.

error analysis of Runge-Kutta schemes in Chapter 7 predicts that these discretization schemes would find the least accurate trajectories.

- an anomalous result found by an Euler discretization with 500 integration steps, which converges before the requested accuracy has been achieved. This highlights the fact that it is not possible to rely entirely on the error analysis of Chapter 7 when using an NLP solver which does not solve the problem exactly. That is, the NLP solver has potential for introducing further errors than those inherent in the discretization schemes themselves.
- that found by a Runge-Kutta discretization with 500 integration steps. It takes over 12 minutes for this solution to be found, which is more than twice as long as the solution time for any other model. This, added to our inability to find suitable starting points for Runge-Kutta models of 2D greens, leads to the observation that Runge-Kutta schemes, despite having the potential for finding more accurate trajectories than Euler discretization schemes, are less suited to the NLP solution technique because of the form that their NLP models take. A large number of additional variables must be introduced to represent the sub-intervals at each integration step.

The trends in t_F noted here are repeated in $v_{x_1}(0)$ and $v_{x_2}(0)$ which are conjectured to have approximate values of 6.99 and 5.92 respectively in an optimal solution.

C.2.6 3D bowl shape

We now consider a curved 3-dimensional green. This green is a gentle bowl shape given by the equation

$$x_3 = \frac{(x_1 - 10)^2}{125} + \frac{(x_2 - 5)^2}{125} - 1.$$

The ball is placed on one side of the dip in the green ($x_1 = 0$, $x_2 = 0$) and is to be hit towards the hole, on the other side of the dip ($x_1 = 20$, $x_2 = 0$), as shown in Figure C.4 (page 133). An optimal trajectory, also shown in Figure C.4, curves around the side of the "bowl". Numerical results found when solving this problem are shown in Table C.6.

As with previous green shapes, we were unable to find solutions to many of the Trapezoidal and Runge-Kutta models of this problem, further confirmation that these more accurate models are more difficult for our NLP algorithm to solve.

Also of note is that the final speed of the ball is much smaller when found by solving models which use the Trapezoidal discretization scheme than that

Discr. Scheme	No. Steps	Time Taken	No. Iters	Final Speed	t_F	$v_{x_1}(0)$	$v_{x_2}(0)$
E	5	0.21	15	$3.70E+00$	4.48338	3.87882	-2.17623
E	25	9.45	109	$2.52E+00$	4.72299	4.77061	-2.91586
E	125	0.37	4	$2.32E+00$	4.77247	5.06979	-3.04781
E	250	1.82	4	$2.28E+00$	4.78245	5.13453	-3.07294
E	500	3.90	8	$2.28E+00$	4.78182	5.13044	-3.07138
E	1000	3.10	4	$2.28E+00$	4.78338	5.14069	-3.07528
T	5	0.01	22	$8.02E-08$	5.99623	4.92089	-3.80787
T	25	0.97	24	$2.54E-06$	4.57167	5.31378	-2.73992
T	125	2.14	22	$8.51E-03$	5.59234	5.01320	-4.20357
T	250	solution not found					
T	500	solution not found					
T	1000	solution not found					
RK	5	solution not found					
RK	25	42.49	51	$1.92E+01$	3.54693	-0.14865	-1.13071
RK	125	solution not found					
RK	250	solution not found					
RK	500	solution not found					
RK	1000	solution not found					

Table C.6: Results from using `hopdmSQP` to solve Euler, Trapezoidal and Runge-Kutta approximations of the golf problem when the green surface is a 3-dimensional bowl shape.

found when solving models which use the Euler discretization scheme. However, there is no consistency to the solutions found when solving models based on the Trapezoidal discretization scheme, whilst the solutions found by solving models based on the Euler discretization scheme appear to tend towards the values $t_F \approx 4.8$, $v_{x_1}(0) \approx 5.1$, $v_{x_2}(0) \approx -3.1$.

C.2.7 3D ramp

Finally, we consider the green shape which was tackled by Vanderbei [75]. This is a ramp shaped green, given by the equation

$$x_3 = -0.3\tan^{-1}(x_2) + 0.05(x_1 + x_2)$$

and shown with an optimal trajectory in Figure C.5 (page 134). The numerical results are shown in Table C.7.

We are again unable to find starting points for which several of the Runge-Kutta and Trapezoidal models of the problem can converge to a solution. Also as previously, the solutions to models which use an Euler discretization appear to tend towards an optimal solution as the number of integration steps increases.

Discr. Scheme	No. Steps	Time Taken	No. Iters	Final Speed	t_F	$v_{x_1}(0)$	$v_{x_2}(0)$
E	5	0.60	11	$6.82E-01$	2.57579	0.52547	-2.63985
E	25	0.08	5	$6.67E-01$	2.98528	0.72548	-0.31283
E	125	0.53	5	$6.64E-01$	3.07780	0.77243	-3.27833
E	250	0.91	5	$6.65E-01$	3.08980	0.77855	-3.29869
E	500	1.81	5	$6.65E-01$	3.09585	0.78163	-3.30899
E	1000	3.83	5	$6.65E-01$	3.09888	0.78317	-3.31416
T	5	0.09	17	$3.04E-09$	5.00455	0.92140	$-$[3]
T	25	9.62	116	$1.65E+00$	1.43353	0.38855	-4.24225
T	125	16.49	14	$1.10E-03$	3.02320	0.70362	-3.88182
T[4]	250	201.98	132	$2.20E-01$	2.89665	0.69002	-3.88219
T	500	solution not found					
T	1000	solution not found					
RK[5]	5	2.42	103	$8.85E-04$	5.39880	-0.15212	-0.483080
RK	25	129.91	72	$7.90E-01$	4.33160	0.56695	-1.61588
RK	125	69.25	10	$8.56E-01$	4.07336	0.58740	-1.62957
RK	250	solution not found					
RK	500	solution not found					
RK	1000	solution not found					

Table C.7: Results from using `hopdmSQP` to solve Euler, Trapezoidal and Runge-Kutta approximations of the golf problem when the green surface is a 3-dimensional ramp, taken from the 18th hole at the 2000 PGA championship.

In this case, the solution tended towards is $t_F \approx 3.1$, $v_{x_1}(0) \approx 0.8$, $v_{x_2}(0) \approx -3.3$. The solutions found using Trapezoidal and Runge-Kutta discretizations differ from this, but are more erratic, no pattern can be seen.

C.3 Comments and conclusions

The aim of this appendix was to begin work on comparing the different discretization schemes. We expected that the Runge-Kutta scheme would have the highest accuracy, as it is a fourth order method, and were interested in whether this increase in accuracy would cause the model solution time to increase.

However, the increase in accuracy expected by the Runge-Kutta scheme is only evident in the first (and simplest) problem considered here and even in that problem instance, the Trapezoidal method (second order) appears to be as accurate as the Runge-Kutta method. In other problem instances, finding

[3]this value was not displayed correctly by `ampl`.

[4]constraint violation greater than requested and KKT condition (3.3a) satisfied to accuracy less than requested.

[5]KKT condition (3.3a) satisfied to accuracy less than requested.

any solution using Runge-Kutta schemes proved to be very difficult and heavily dependent on the choice of starting point. The Trapezoidal discretization scheme also proved unreliable. Only using the Euler discretization scheme (first order) were we able to find solutions in every instance and for every choice of number of integration steps. As expected, these solutions appear to increase in accuracy as the number of integration steps increase.

We would like to mention some possible reasons for the Runge-Kutta scheme being less successful than we expected it to be:

- Runge-Kutta discretization schemes include many more variables and constraints than the lower-order methods. Also, as they ask for higher accuracy, they are more difficult to solve.

- The error analysis in Chapter 7 cannot be relied on because `hopdmSQP` only requires 6 decimal place accuracy.

- Vanderbei [75] comments that a problem which is reported to be infeasible is more likely to be infeasible due to poor model formulation than to a bad algorithm. It is not impossible that the Runge-Kutta models which we created for these problems contain unnoticed errors.

  There is a certain degree of freedom involved in choosing which variables to include in the model and in choosing how to represent the constraints. We experimented with various forms of each model before choosing the forms used in the analysis above, but there may be ways in which the models can be formulated that would be better suited to the NLP solution technique.

- Vanderbei [75] also notes that both `Loqo` [71] and SNOPT [36] are sensitive to the starting point chosen. When solving each of the models we found that this was also the case for `hopdmSQP`.

  In general, we found a solution to a model with a small number of integration steps and interpolated this solution to provide a good starting point for model with more integration steps. Observations suggest that this method is not suitable for the Runge-Kutta variables which correspond to the sub-interval steps (i.e. $\mathbf{kx1}(t), \mathbf{kx2}(t), \mathbf{kvx1}(t) \ldots$).

Our preliminary analysis suggests that the Euler discretization scheme is the most robust method for use in solving OCPs as NLP problems.

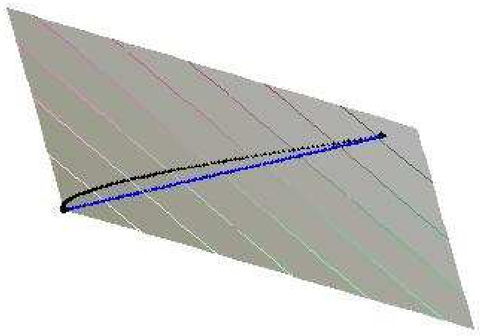


Figure C.3: Showing an optimal golf trajectory found on a 3D planar green. An Euler discretization with 125 integration steps is used to find this solution. The black line shows the optimal trajectory, whilst the blue line shows a reference straight line trajectory between the starting point and the hole. The optimal trajectory runs from right to left.

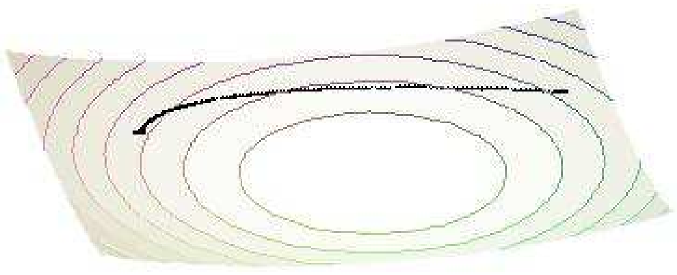


Figure C.4: Showing an optimal golf trajectory found on a 3D bowl shaped green. An Euler discretization with 125 integration steps is used to find this solution. The optimal trajectory runs from right to left.

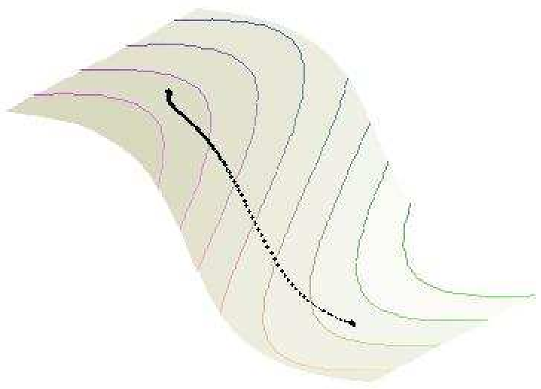


Figure C.5: Showing an optimal golf trajectory found on a 3D ramp based on the 18th hole at the 2000 PGA championship. An Euler discretization with 125 integration steps is used to find this solution. The optimal trajectory runs from bottom to top.

Appendix D

Sailing

In Chapters 6 and 7, we introduced the sailing problem discussed in Bryson & Ho [15]. We concentrated on the simplest instance of the problem, describing Hennessey *et al.*'s use of Hamiltonian theory and calculus of variations [46] and using `hopdmSQP` to find solutions to NLP approximations.

In this appendix we discuss the variety of problems whose solution is an optimal route for a sailing boat, mentioning some of the difficulties encountered when trying to write nonlinear models for more involved versions of the problem.

We then describe the optimization technique of dynamic programming, referring to Vanderbei's method of finding optimal sailing routes [73]. We show our alternative adaptation of the technique (which is discussed in greater detail in Buchanan & Stern [16]) and conclude with some preliminary results based on an implementation of this adaptation.

D.1 Variations of the problem

D.1.1 Wind fields

In the problem discussed in Chapters 6 and 7, we considered the case in which the wind was constant in both strength and direction. However, it is more realistic for the strength and direction of the wind to vary, both by time and by position.

In [47], Hennessey & Kumar worked on the problem of finding an optimal sailing route in the Apostle Islands in Lake Superior. Hamiltonians are used to find an optimal sailing route through these islands, given the artificial assumption that wind speed is constant. Figure D.1 shows an approximate representation of a possible wind field for that area which varies with position but not with time.

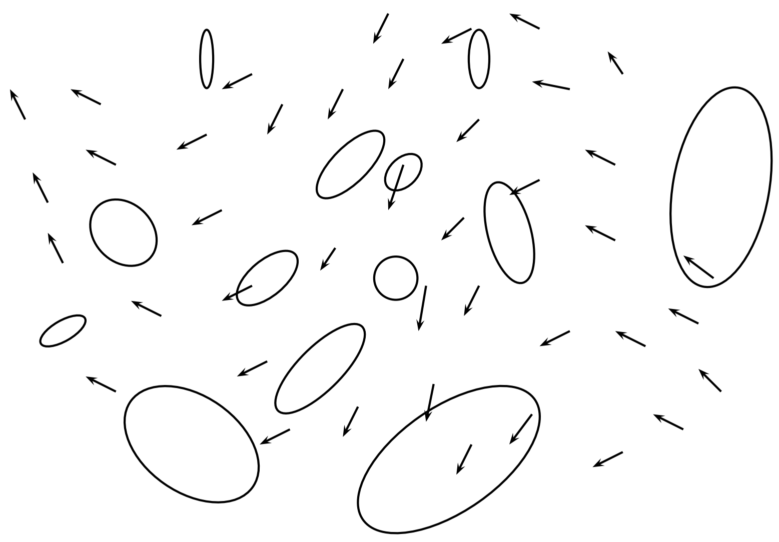


Figure D.1: An approximate representation of a spacial wind field in the Apostle Islands, shown in detail in [47]. In this representation, ellipses show the approximate position of islands and arrows show mean wind directions.

D.1.2 Water movement

In our basic instance of the sailing problem, we considered only the case of sailing on still water. However, in reality, water is often not still. For example, rivers flow towards the sea; and seas and oceans are subject to tidal flows. As an example, tidal currents around the Isle of Wight in South England, taken from the Reeds Oki Channel Almanac [27], are shown in Figures D.2 and D.3.

When trying to determine suitable nonlinear models for the sailing problem with wind fields which vary spatially like the one shown in Figure D.1 or with water currents which vary with time and position like those shown in Figures D.2 and D.3, we encountered difficulties. Our initial assumptions about the standard nonlinear programming problem (1.1) included that the constraint functions $c_i(x)$ are continuous and second-order differentiable. Therefore, in order to include wind and water variations in the nonlinear program, we need to be able to write them as continuous functions of time and position. This may be very difficult.

D.1.3 Shape of the water

In the simple formulation of Chapters 6 and 7, the optimal route sought was from a starting point to a finish point, across a stretch of water with no boundaries or islands. However, areas of water may be bordered by land or contain islands. Figures D.1, D.2 and D.3 all show how the route of a sailing boat can be restricted by the shape of the water. The irregular shapes that the water can take should be carefully considered when trying to define the feasible region in the nonlinear

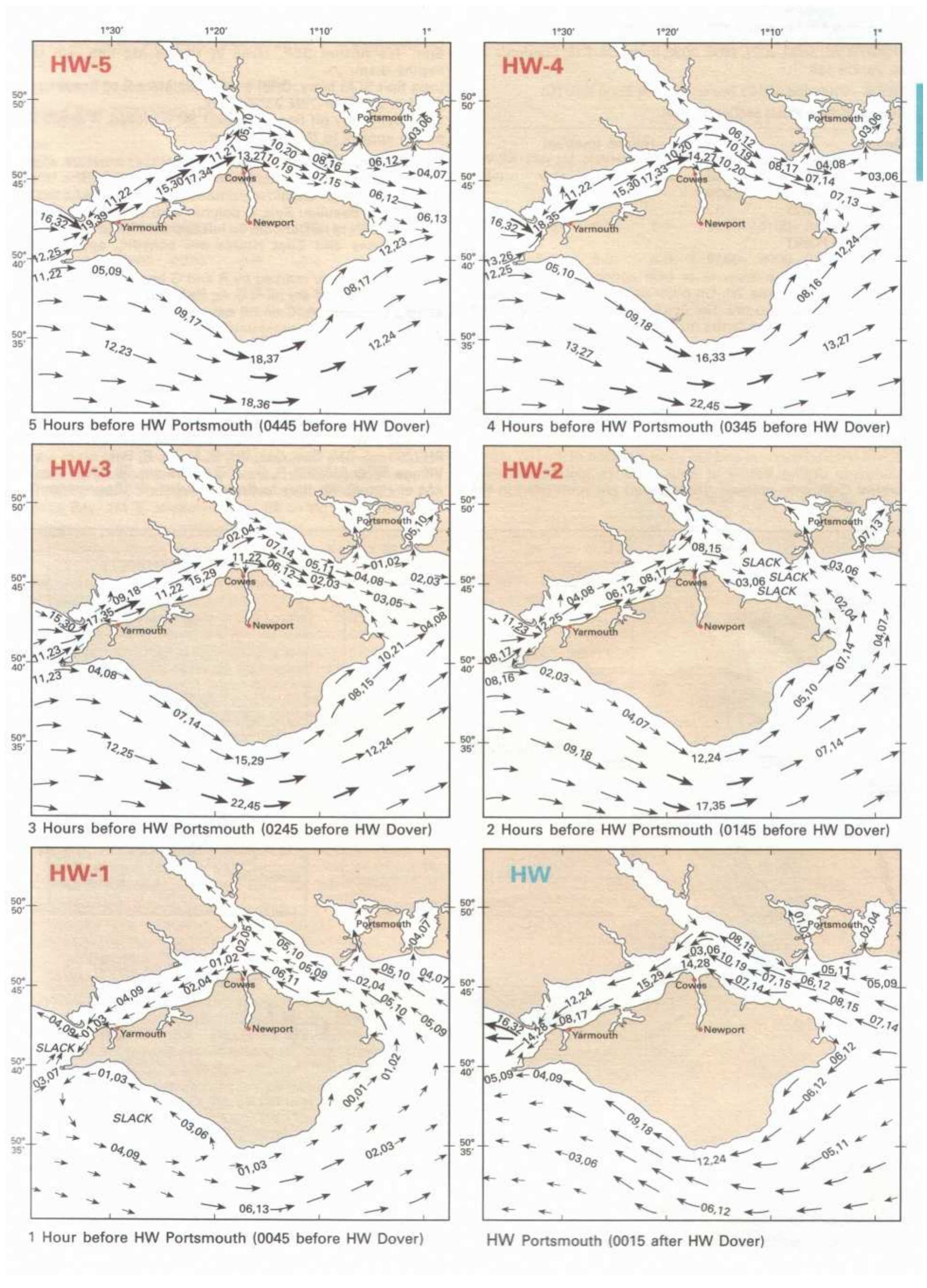
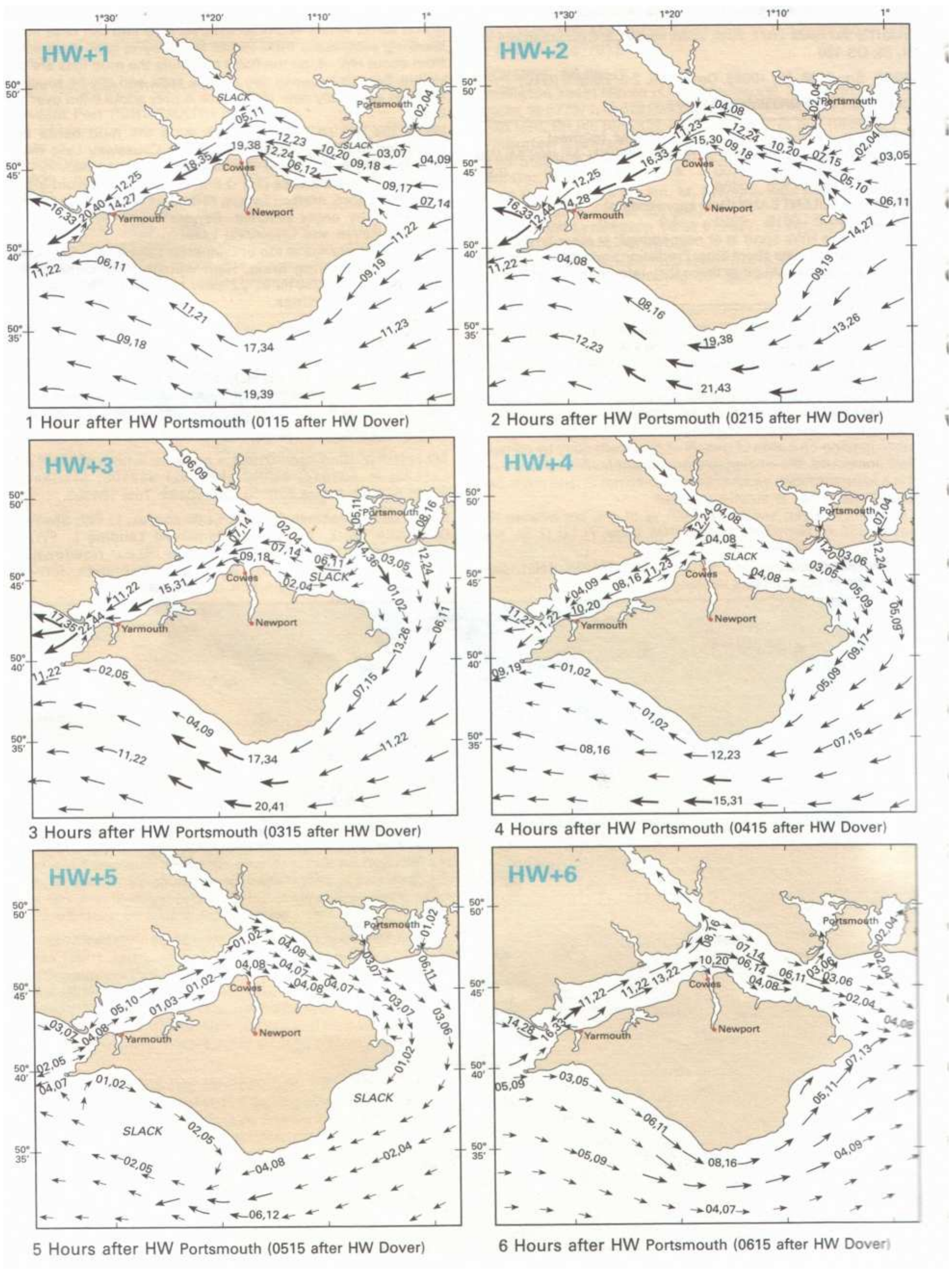

HW-5
5 Hours before HW Portsmouth (0445 before HW Dover)
HW-4
4 Hours before HW Portsmouth (0345 before HW Dover)
HW-3
3 Hours before HW Portsmouth (0245 before HW Dover)
HW-2
2 Hours before HW Portsmouth (0145 before HW Dover)
HW-1
1 Hour before HW Portsmouth (0045 before HW Dover)
HW
HW Portsmouth (0015 after HW Dover)
Portsmouth
Cowes
Yarmouth
Newport
SLACK

Figure D.2: Tidal currents near the Isle of Wight in the hours before high tide. The two numbers on each arrow represent the mean speed of the current (in knots) during neap tides and spring tides at the time and place indicated. This figure is copied from the Reeds Oki Channel Almanac [27]



HW+1
HW+2
HW+3
HW+4
HW+5
HW+6
Portsmouth
Cowes
Newport
Yarmouth
SLACK
1 Hour after HW Portsmouth (0115 after HW Dover)
2 Hours after HW Portsmouth (0215 after HW Dover)
3 Hours after HW Portsmouth (0315 after HW Dover)
4 Hours after HW Portsmouth (0415 after HW Dover)
5 Hours after HW Portsmouth (0515 after HW Dover)
6 Hours after HW Portsmouth (0615 after HW Dover)
1°30'
1°20'
1°10'
1°
50° 50'
50° 45'
50° 40'
50° 35'

Figure D.3: Tidal currents near the Isle of Wight in the hours after high tide. The two numbers on each arrow represent the mean speed of the current (in knots) during neap tides and spring tides at the time and place indicated. This figure is copied from the Reeds Oki Channel Almanac [27]

program. Every island and stretch of coastline has to be approximated by a continuous, second-order differentiable function.

Also, the sailing boat may not be constrained to specific start and finish points. For example, problems can be formulated in which the sailing boat begins at a starting line and aims for a circular target zone. Start and finish areas of these types are straightforward to represent in a nonlinear programming model.

D.1.4 Boat type

Finally, the type of boat being sailed has an effect on the problem formulation. Figure 6.2 shows an example of a "wind polar", demonstrating how the speed of a boat varies with the angle between the sailing direction and the direction of the wind. Every different type of boat has its own distinct wind polar.

We noted in Chapter 7 that optimal routes for the simple problem of sailing from one point to another in a constant wind can be expressed as two straight line segments, but that, often, the NLP solver finds a multi-segmented route. At each point where one segment ends and another begins, the boat crosses over the wind. This is called *tacking*. In practice, crossing from one side of the wind to the other takes time, as the sails must be moved across the boat, which is often slowed down by the process. It would, therefore, be appropriate to include a tacking penalty in the nonlinear model so that two-segment routes with a single tack become the only optimal solutions. This tacking penalty should differ with the type of boat and also with the experience of the sailors. We have not found a way to represent this penalty in the formulation of the nonlinear model.

D.2 Dynamic programming

Another optimization technique which can be used to solve the sailing problem is *dynamic programming*. Here, we first describe the technique, before showing ways in which it can be used to solve our problems.

Dynamic programming was invented by Bellman [4]. It is a technique which is applicable to a wide range of problems, the most common of which is to find the shortest, or lowest cost path between two points. The features of a problem which can be solved using dynamic programming are summarized in Hillier & Liebermann [49] as follows:

1. An optimal solution is sought.

2. The problem can be divided into stages.

3. Each stage has a number of states.

4. At each stage, a decision is made which transforms the current set of states into new states.
5. At each stage, the decision made is independent of previous decisions. (This is the *principle of optimality* for dynamic programming.)
6. The solution procedure begins with determination of the optimal decision at the last stage of the problem. This is usually a trivial calculation.
7. It is then possible to define a recursive relationship which identifies the optimal solution for the n^{th} stage given the solution of the $n^{th} + 1$.

Points 6 and 7 can often be reversed without losing any of the benefits of the technique. That is, the solution procedure can begin with determination of the optimal solution at the first stage of the problem. The recursive relationship then identifies the optimal solution of the $n^{th} + 1$ stage given the solution to the n^{th}.

In the next two sections we will show two ways in which dynamic programming can be adapted to find approximate optimal routes for a sailing boat. First, we will describe a simplified version of an algorithm by Vanderbei [73] and then we will introduce our preliminary work on a different variation of dynamic programming for the sailing problem.

D.2.1 Vanderbei's adaptation

Vanderbei [73] breaks the boat's route into stages by placing a grid over the water and allowing the boat to travel between adjacent gridpoints at each stage. The states are defined as the current stage, the boat's position and a record of whether the boat is currently sailing to the left or to the right of the wind. The direction of the wind is determined by the states. From each gridpoint, the boat is able to sail to an adjacent gridpoint in up to 7 directions. (See Figure D.4.) It is not possible to sail directly into the wind.

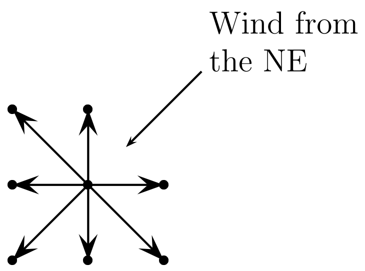


Figure D.4: Showing the 7 directions a boat can sail in in Vanderbei's formulation.

Characteristics of the specific problem which Vanderbei addresses in [73] are as follows:

- The wind speed is kept constant, but its direction is changeable with time. The wind direction for the next stage of the journey, which is uniform across the area of water considered, is decided by user input probabilities. It may remain the same as during the previous stage, or come from 45° to the left or right of the old wind.

- The water is assumed to be still.

- The feasible region is a square lake, with no islands. The optimal route sought is from a starting point to a finish point.

- No wind polar is given in terms of boat speed relative to the wind, but the time the boat takes to travel from one gridpoint to the next is defined with respect to the angle which the boat makes with the wind:

Angle with wind	Time (minutes)
0°	1
±45°	2
±90°	3
±135°	4
180°	N/A

  A penalty of three minutes is added every time the boat tacks.

In order to apply a dynamic programming technique to this program, a recursive relationship must be determined. This is done by defining $F(s, x_1, x_2, R/L)$ to be the minimum time in which gridpoint (x_1, x_2) can be reached at stage s, arriving there sailing on either the right (R) or left (L) side of the wind. Here we've simplified Vanderbei's work by removing the dependence on probabilities. Instead, we assume that we know the direction of the wind at this stage.

This recursion is initialized at the starting point:

$$\begin{aligned} F(0,1,1,L) &= 0 \\ F(0,1,1,R) &= 0 \end{aligned} \tag{D.1}$$

and continues

$$\begin{aligned} F(s+1,x_1,x_2,R) &= \min_{R/L,y_1,y_2}{}^{2} \big(F(s,y_1,y_2,R/L)+t_{yxs}\big), \\ F(s+1,x_1,x_2,L) &= \min_{R/L,y_1,y_2}{}^{3} \big(F(s,y_1,y_2,R/L)+t_{yxs}\big), \end{aligned} \tag{D.2}$$

[2]such that a route to the right hand side of the current wind exists between $\mathbf{y}$ and $\mathbf{x}$.
[3]such that a route to the left hand side of the current wind exists between $\mathbf{y}$ and $\mathbf{x}$.

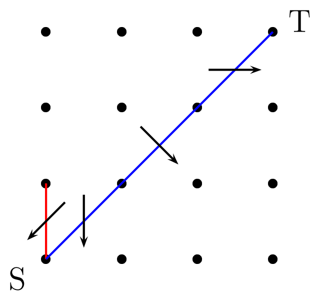
where t_{yxs} is the time taken to travel from gridpoint $\mathbf{y}$ to gridpoint $\mathbf{x}$ subject to the wind conditions of stage s. t_{yxs} includes a tacking penalty if it is incurred.

We considered the problem with the wind directions at each stage chosen to be those given in Table D.1.

Stage	Wind Direction
0	North East
1	North East
2	North
3	North West
4	West

Table D.1: An example of wind directions at stages of a dynamic programming model of the sailing problem.

Using the dynamic programming recursion given by (D.1), (D.2), we solved this problem for routes between $x_1 = x_2 = 1$ (S) and $x_1 = x_2 = 4$ (T) and found six optimal routes, each taking 18 minutes. These routes are shown in Figures D.5–D.10, with stages sailed to the right of the wind shown in red and



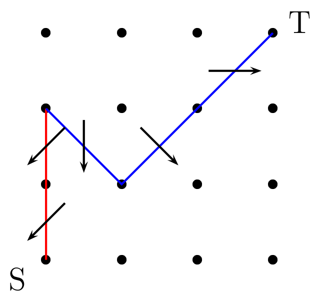
$$(1,1,R) \xrightarrow[4]{} (1,2,R) \xrightarrow[2+3]{} (1,1,L) \xrightarrow[4]{} (2,2,L) \xrightarrow[3]{} (3,3,L) \xrightarrow[2]{} (4,4,L)$$

Figure D.5: Optimal route 1 for sailing problem found by dynamic programming.

stages sailed to the left of the wind shown in blue. Arrows are used to show wind directions and the times taken for each stage are shown beneath the Figures.

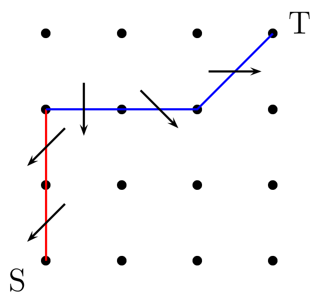
Inaccuracies introduced by this method of approximating the problem include:

- The time taken to travel between horizontally and vertically adjacent gridpoints is equated to the time taken to travel between diagonally adjacent gridpoints. This biases the optimization process towards choosing diagonal segments, as greater distance is travelled in the same time.



$$(1,1,R) \xrightarrow[4]{} (1,2,R) \xrightarrow[4]{} (1,3,R) \xrightarrow[2+3]{} (2,2,L) \xrightarrow[3]{} (3,3,L) \xrightarrow[2]{} (4,4,L)$$

Figure D.6: Optimal route 2 for sailing problem found by dynamic programming.



$$(1,1,R) \xrightarrow[4]{} (1,2,R) \xrightarrow[4]{} (1,3,R) \xrightarrow[3+3]{} (2,3,L) \xrightarrow[2]{} (3,3,L) \xrightarrow[2]{} (4,4,L)$$

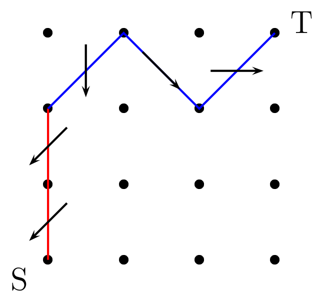
Figure D.7: Optimal route 3 for sailing problem found by dynamic programming.

- Wind directions only vary at gridpoints, which are not regularly spaced out in time. Each wind's duration depends on the direction the boat travels in.

D.2.2 Our adaptation

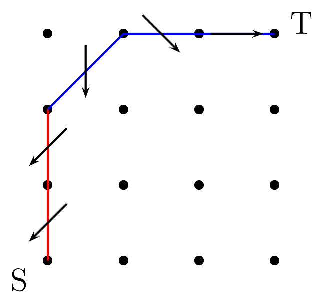
In our adaptation we break the boat's route into stages by allowing it to sail for one minute at each stage. The states are defined as the current time, the boat's position and a record of whether the boat is sailing to the left or right of the wind. The speed and direction of both wind and water are determined by the states. At each stage, the number of directions in which the boat can sail is determined by the accuracy with which its wind polar has been specified. It is not possible to sail directly into the wind.

We call each set of states (time, position, side of wind) an instance of a boat and record the history of the boat at each instance so that an instance also represents a path from the starting point to the current position. Instead of



$$(1,1,R) \underset{4}{\rightarrow} (1,2,R) \underset{4}{\rightarrow} (1,3,R) \underset{4+3}{\rightarrow} (2,4,L) \underset{1}{\rightarrow} (3,3,L) \underset{2}{\rightarrow} (4,4,L)$$

Figure D.8: Optimal route 4 for sailing problem found by dynamic programming.



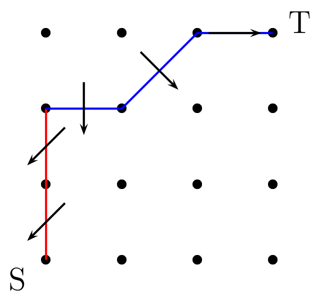
$$(1,1,R) \underset{4}{\rightarrow} (1,2,R) \underset{4}{\rightarrow} (1,3,R) \underset{4+3}{\rightarrow} (2,4,L) \underset{2}{\rightarrow} (3,4,L) \underset{1}{\rightarrow} (4,4,L)$$

Figure D.9: Optimal route 5 for sailing problem found by dynamic programming.

limiting the boat instances to specific waypoints, we simply reject any instance which is "too" close to another at the same stage. We have experimented with different methods of defining one instance's proximity to another. The basic algorithm used is as follows:

Algorithm D.1.

```
Define valid region.
Define finish region.
Set up starting instances.
(There will be at least two, one to the right and one to the left of the wind.)
While finish region is not reached
    For every instance at current stage
        Apply movement due to water flow at current position and time.
        For every possible direction with respect to wind
            Apply movement due to wind at current position and time.
```



$$(1,1,R) \xrightarrow[4]{} (1,2,R) \xrightarrow[4]{} (1,3,R) \xrightarrow[3+3]{} (2,3,L) \xrightarrow[3]{} (3,4,L) \xrightarrow[1]{} (4,4,L)$$

Figure D.10: Optimal route 6 for sailing problem found by dynamic programming.

```
            If boat crosses over wind
                Apply a tacking penalty.
            If moved boat is in valid region
                Add to list of possible new instances.
        End for
    End for
    Empty list of instances at current stage.
    For every instance in list of possible new instances
        If instance is in finish region
            Compile a list of instances which have reached the finish.
        Else If instance is too close to any previous instance in list
            Reject instance.
        Else
            Add instance to new list of instances at current stage.
    End for
```

We can deduce that the first instance found in the finish region represents an optimal route, as the stages correspond to time. If we were to define a DP recurrence similar to (D.1), (D.2) then the value of $F(s, x_1, x_2, R/L)$ would be s for all values of s, x_1, x_2, R/L. However, our adaptation is not a true dynamic programming recursion, as the possible routes are not all considered.

D.3 Numerical results

Our work on modelling the sailing problem as a dynamic programming model is in preliminary stages, but we hope that it will be adaptable to deal with any water shape or flow and with any wind field. For comparison and demonstration purposes we have considered some problems with constant water flow and a variety of simple wind fields of constant strength. The algorithm has been written in `java`.

We considered two wind polars.

1. **Boat 1**: A wind polar which approximates the times which Vanderbei's sailing boat takes to travel in the 7 possible directions available to it. See Table D.2 and Figure D.11. We included the possibility of the boat remaining stationary by giving it a speed of 0 when sailing directly into the wind.

Angle with wind	Speed
$0°$	1
$\pm 45°$	0.5
$\pm 90°$	0.333
$\pm 135°$	0.25
$180°$	0

Table D.2: Data for a wind polar which approximates Vanderbei's sailing times (Boat 1).

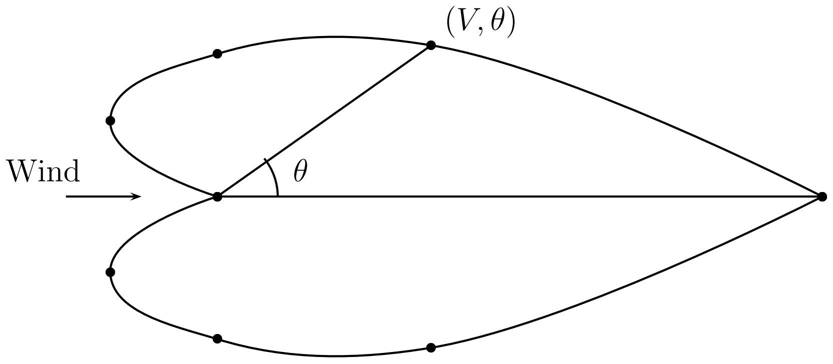


Figure D.11: Approximate wind polar for Boat 1, showing speed (V) and angle between boat and wind (θ) in polar coordinates.

2. **Boat 2**: A wind polar which approximates that found by Hennessey *et al.* [46] by practical experimentation with a C&C yacht on Lake Superior. See Table D.3 and Figure D.12.

Both boats have a fastest speed of 1 unit/minute, but vary in that Boat 1 sails at this speed when sailing with the wind, but Boat 2 moves fastest when it is sailing to one side of the wind. For each boat, the tacking penalty imposed halves the distance which a boat travels in a stage when it tacks. In all the figures in the following section, segments of the route which are sailed to the right of the wind are shown in red and segments sailed to the left of the wind are shown in blue.

Angle with wind	Speed
0°	0.65
±30°	0.93
±40°	1
±60°	0.88
±90°	0.66
±120°	0.35
180°	0

Table D.3: Data for a wind polar which approximates that found by Hennessey *et al.* through practical experimentation (Boat 2).

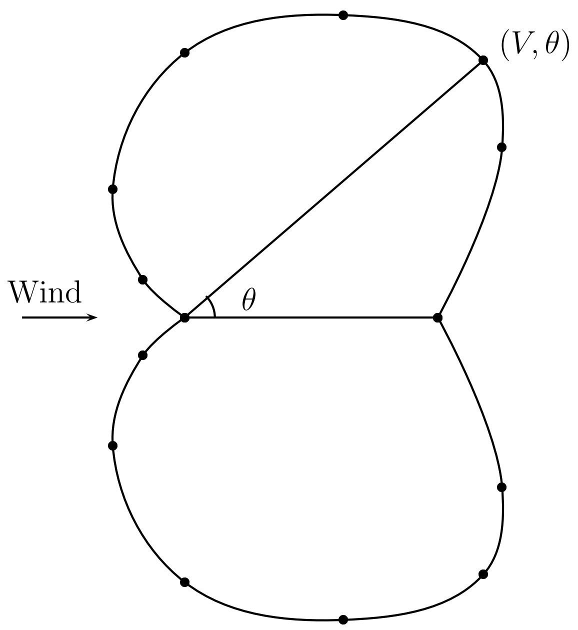


Figure D.12: Wind polar for Boat 2, showing speed (V) and angle between boat and wind (θ) in polar coordinates.

D.3.1 Problems solved

D.3.1.1 Sailing with the wind

We consider the problem of sailing a boat from $x_1 = 0$, $x_2 = 3.5$ (S) to a finish line at $x_1 = 40$ (T) along a river which is just 7 units wide. There is a constant following wind (from the West). A typical optimal route for each boat considered is shown in Figure D.13. Boat 1, whose fastest speed is attained when sailing with the wind, sails down the centre of the river for 12 minutes. Boat 2, whose fastest speed is attained when sailing to one side of the wind, tacks several times, sailing from one side of the river to the other. It takes 17 minutes to reach the finish line.

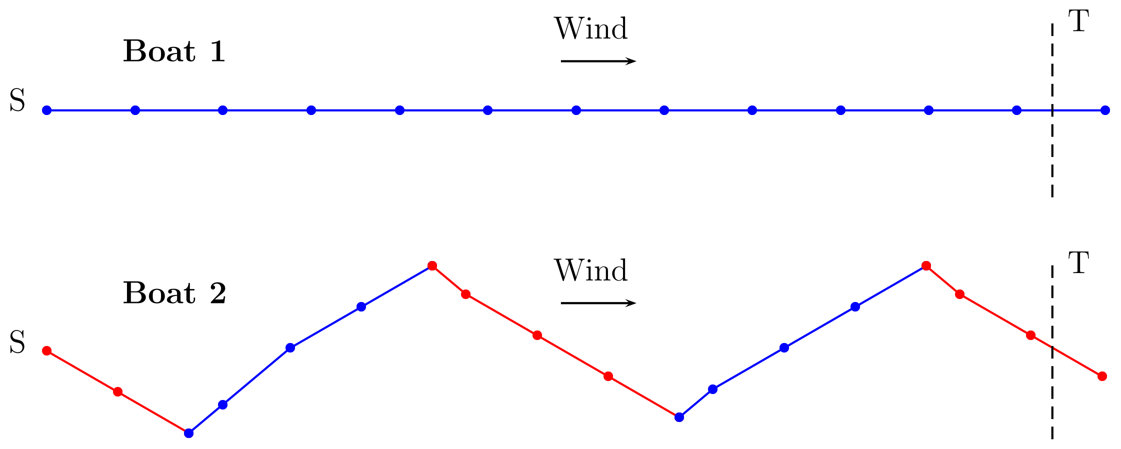


Figure D.13: Optimal routes for Boats 1 & 2 sailing with the wind.

D.3.1.2 Sailing in a wind which varies discretely

Next, we have approximated the problem solved by Vanderbei when the wind directions are chosen to be those given in Table D.1. We take a larger lake than the one in Vanderbei's study and work with the problem of finding the optimal route between the starting point $x_1 = x_2 = 0$ (S) and the finishing point $x_1 = x_2 = 20$ (T). As the boat in Vanderbei's formulation takes between 1 and 7 minutes to traverse a single stage, we consider that this increase in size is appropriate. Similarly, we claim that each wind lasts for four minutes. (Four stages in our formulation). After 20 minutes, the wind continues to come from the West.

A typical optimal route sailed by Boat 1 is shown in Figure D.14 and is

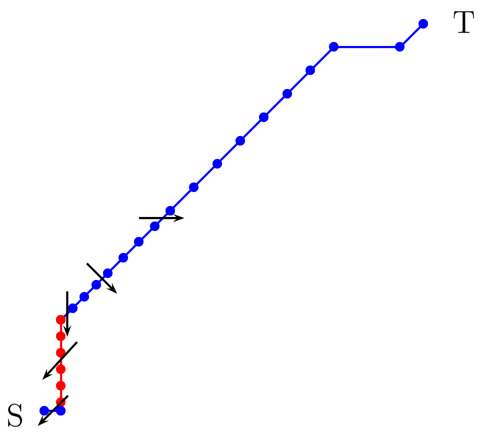


Figure D.14: An optimal route found for Boat 1 sailing in variable wind. The arrows show the wind direction each time it changes.

similar to solutions found using Vanderbei's algorithm (Figures D.6–D.10). The boat begins its journey immediately and travels North until the wind turns away from the North East, allowing the boat to travel diagonally across the lake. The last stages of the journey make use of a following Westerly wind.

The optimal route for Boat 2 involves it staying stationary for the first 8 minutes, when the wind is blowing from the North East. When the wind changes, the optimal route leads across the centre of the lake. A typical route is shown in Figure D.15. The entire route is sailed to the left of the wind.

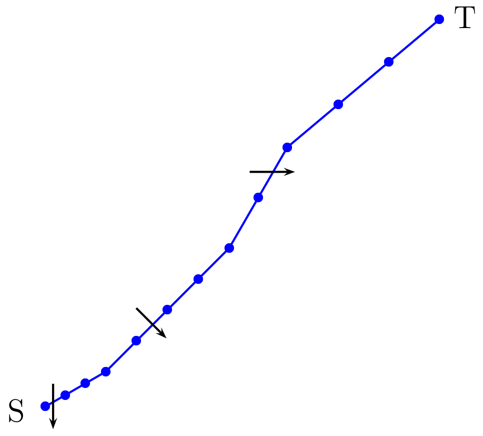


Figure D.15: A typical optimal route found for Boat 2 sailing in wind which varies discretely. The arrows show the wind direction each time it changes.

Boat 1 takes 25 minutes to complete an optimal route and Boat 2, despite waiting for 8 minutes, takes just 21 minutes.

D.3.1.3 Sailing in a wind which varies continuously

Finally, we consider a problem where the wind direction varies continuously. We are looking for an optimal sailing route between $x_1 = 0$, $x_2 = 10$ (S) and $x_1 = 40$ (T) on a river of width 20 units. The wind direction, shown in Figure D.16, varies with x_1.

The typical optimal routes found when sailing with Boat 1 and Boat 2 are shown in Figures D.17 and D.18, respectively. Boat 1 takes 44 minutes to travel from the starting point to the finish line, whilst Boat 2 takes just 28 minutes. However, although the times taken to complete the optimal routes differ significantly, the directions travelled in are not dissimilar. Both boats travel in a South-Easterly direction in the first half of the journey, when the winds are from the North and in a North-Easterly direction in the second half of the journey when the winds are from the South.

There is much more interesting work that could be carried out into the problem of finding optimal routes for sailing boats. For example, we have not considered here the cases where there are islands in the water such as those in the study

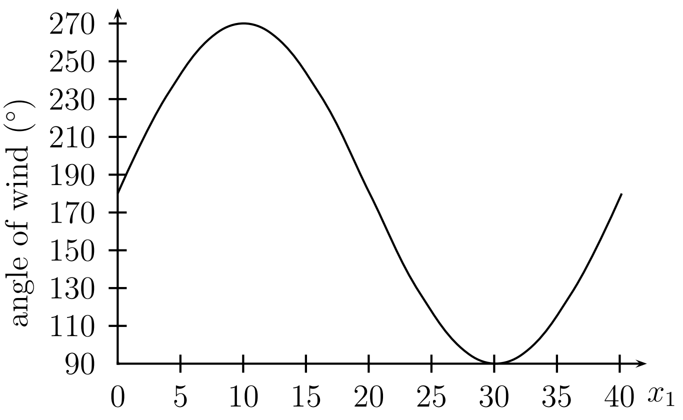


Figure D.16: Continuously varying wind field. This plot shows how the angle that the wind makes with the horizontal changes as x_1 varies.

by Hennessey & Kumar [47] (Figure D.1) or those where the water is not still (Figures D.2 and D.3).

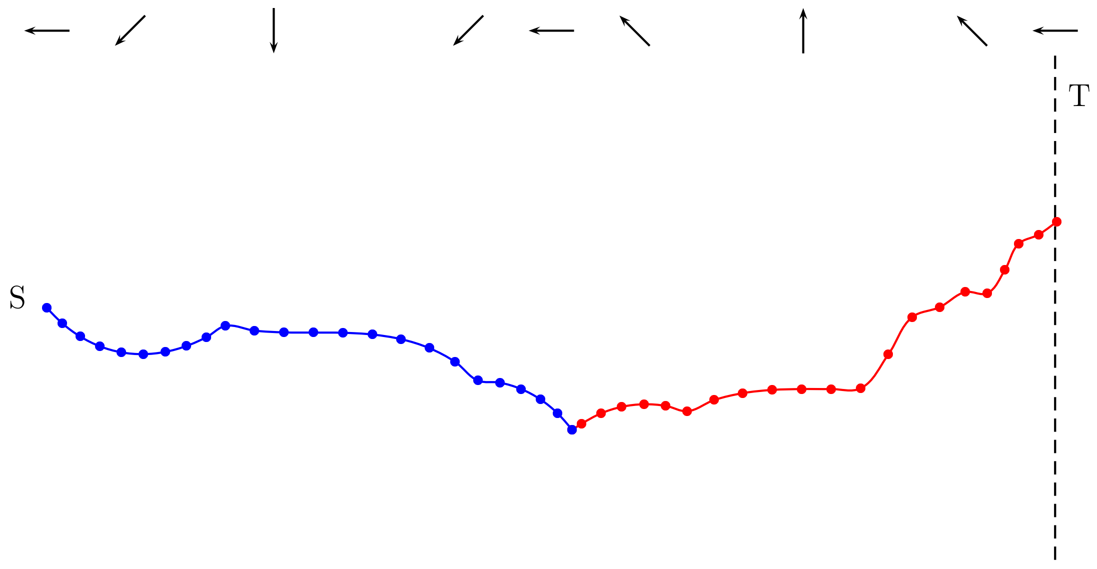


Figure D.17: Optimal route for Boat 1 sailing in wind which varies continuously. The arrows show wind directions with respect to x_1.

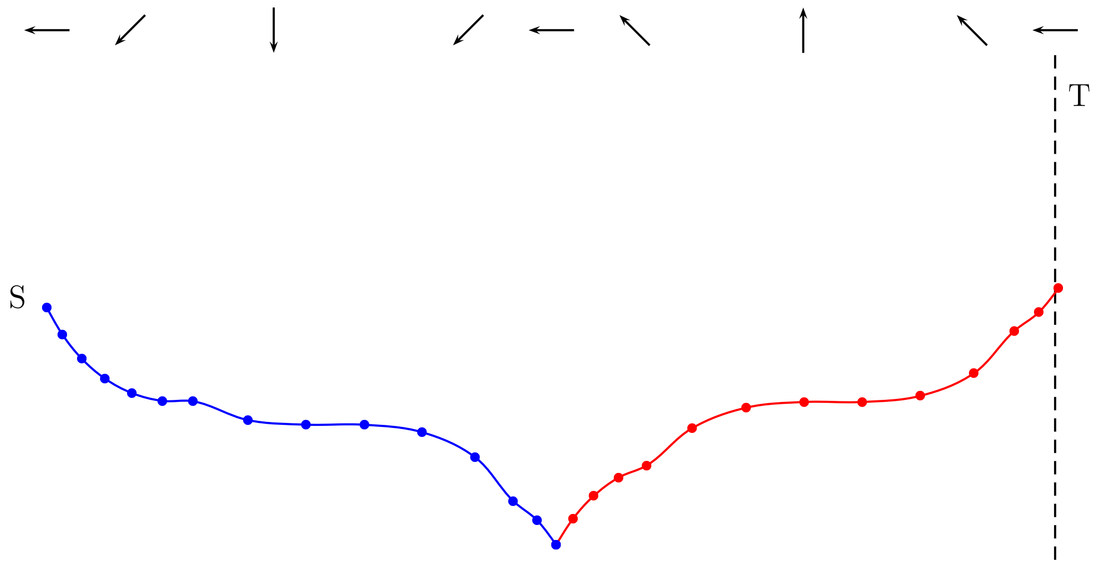


Figure D.18: Optimal route for Boat 2 sailing in wind which varies continuously. The arrows show wind directions with respect to x_1.

Bibliography

[1] Stephen Alessandrini; *A motivational example for the numerical solution of two-point boundary value problems*; SIAM Review 37 (1995) pp. 423–427.

[2] E. D. Andersen, J. Gondzio, C. Meszaros & X. Xu; *Implementation of interior point methods for large-scale linear programming*; in *Interior Point Methods of Mathematical Programming*, Tamas Terlaky, editor; Kluwer Academic Publishers (1996) pp. 189–252.

[3] A. M. Arthurs; *Calculus of Variations*; Routledge Kegan Paul Ltd (1975).

[4] Richard Bellman; *Dynamic Programming*; Princeton University Press (1957).

[5] H. Y. Benson & D. F. Shanno; *An exact primal-dual penalty method approach to warmstarting interior-point methods for linear programming*; Computational Optimization and Applications, to appear (2006).

[6] H. Y. Benson & D. F. Shanno; *Interior point methods for non-convex nonlinear programming: regularization and warmstarts*; Computational Optimization and Applications, to appear (2007).

[7] H. Y. Benson, D. F. Shanno & R. J. Vanderbei; *A comparative study of large-scale nonlinear optimization algorithms*; Technical Report ORFE 01-04; Department of Operations Research and Financial Engineering, Princeton University (2001).

[8] Hande Benson.
URL `http://orfe.princeton.edu/~rvdb/ampl/nlmodels/cute/`

[9] Dimitri Bertsekas; *Nonlinear Programming*; Athena Scientific (1999).

[10] John T. Betts; *Practical Methods for Optimal Control Using Nonlinear Programming*; SIAM (2001).

[11] John T. Betts; *Planning a trip to the moon? ... and back?* (2006); 3rd International Workshop on Astrodynamics Tools and Techniques, ESA, Noordwijk.

[12] P. T. Boggs, A. J. Kearsley & J. W. Tolle; *Practical algorithm for general large scale nonlinear optimization problems*; SIAM Journal on Optimization 9 (1999) pp. 755–778.

[13] P. T. Boggs & J. W. Tolle; *Sequential quadratic programming*; Acta Numerica 4 (1995) pp. 1–51.

[14] I. Bongartz, A. R. Conn, N. I. M. Gould & Ph. L. Toint; *CUTE: Constrained and Unconstrained Testing Environment* (1993).

[15] A. E. Bryson Jr & Y. C. Ho; *Applied Optimal Control*; Hemisphere, New York (1975).

[16] C. R. Buchanan & D. A. Stern; *Optimization on the high seas*; isquared magazine 1 (2007) pp. 9–13.

[17] R. A. Byrd, M. E. Hribar & J. Nocedal; *An interior point algorithm for large-scale nonlinear programming*; SIAM Journal on Optimization 9 (1999) pp. 877–900.

[18] M. Colombo & J. Gondzio; *Further development of multiple centrality correctors*; Computational Optimization and Applications, to appear (2006).

[19] M. Colombo, J. Gondzio & A. Grothey; *A warm-start approach for large-scale stochastic linear programs*; Technical Report MS-2006-04; School of Mathematics, The University of Edinburgh (2007).

[20] A. R. Conn, N. I. M. Gould & Ph. L. Toint; *A globally convergent augmented Lagrangian algorithm for optimization with general constraints and simple bounds*; SIAM Journal on Numerical Analysis 28 (1991) pp. 545–572.

[21] A. R. Conn, N. I. M. Gould & Ph. L. Toint; Lancelot*: A* fortran *package for large-scale nonlinear optimization. (Release A)*; number 17 in Springer Series in Computational Mathematics; Springer Verlag, Heidelberg (1992b).

[22] R. Courant; *Variational methods for the solution of problems of equilibrium and vibrations*; Bulletin of the American Mathematical Society 49 (1943) pp. 1–23.

[23] *Benchmarks comparing **LOQO** with* SNOPT *and* NITRO *on the **CUTE** set and Schittkowski test set.*
URL `http://www.princeton.edu/~rvdb/cute_table.pdf`

[24] E. D. Dolan & J. J. Moré; *Benchmarking optimization software with COPS*; Technical Report ANL/MCS-246; Argonne National Laboratory (2000).

[25] J. Dussault; *Numerical stability and efficiency of penalty algorithms*; SIAM Journal on Numerical Analysis 32 (1995) pp. 296–317.

[26] A. S. El-Bakry, R. A. Tapia, T. Tsuchiya & Y. Zhang; *On the formulation and theory of the Newton interior-point method for nonlinear programming*; Journal of Optimization Theory and Applications 89 (1996) pp. 507–541.

[27] N. Featherstone & P. Lambie, editors; *Reeds Oki Channel Almanac*; Adlard Coles Nautical (2004).

[28] A. V. Fiacco & G. P. McCormick; *Nonlinear Programming: Sequential Unconstrained Minimization Techniques*; John Wiley & Sons (1968).

[29] R. Fletcher & S. Leyffer; *Nonlinear programming without a penalty function*; Mathematical Programming 91 (2002) pp. 239–269.

[30] Roger Fletcher; *Practical Methods of Optimization*; second edition; John Wiley & Sons (1987).

[31] A. Forsgren & P. E. Gill; *Primal-dual interior methods for nonconvex nonlinear programming*; SIAM Journal on Optimization 8 (1998) pp. 1132–1152.

[32] R. Fourer, D. M. Gay & B. W. Kernighan; *AMPL: A modelling language for Mathematical Programming*; Duxbury Press/Wadsworth (1993).

[33] K. R. Frisch; *The logarithmic potential method for convex programming* (1955); Institute of Economics, University of Oslo.

[34] David M. Gay; *Hooking your solver to AMPL*; Technical Report 97-4-06; Computing Sciences Research Center, Bell Laboratories (1997).

[35] P. E. Gill, W. Murray & M. H. Wright; *Practical Optimization*; Academic Press (1981).

[36] P.E. Gill, W. Murray & M. A. Saunders; *SNOPT: An SQP algorithm for large-scale constrained optimization*; SIAM Journal on Optimization 12 (2002) pp. 979–1006.

[37] J. Gondzio & A. Grothey; *Reoptimization with the primal-dual interior point method*; SIAM Journal on Optimization 13 (2003) pp. 842–864.

[38] Jacek Gondzio; *Practical large scale optimization lectures*; University of Edinburgh.
URL `http://student.maths.ed.ac.uk/displaycourse.html`

[39] Jacek Gondzio; *HOPDM: A fast LP solver based on a primal-dual interior point method*; European Journal of Operational Research 85 (1995) pp. 221–225.

[40] Jacek Gondzio; *Multiple centrality corrections in a primal-dual method for linear programming*; Computational Optimization and Applications 6 (1996) pp. 137–156.

[41] Jacek Gondzio; *Warm-start of the primal-dual method applied in the cutting plane scheme*; Mathematical Programming 83 (1998) pp. 125–143.

[42] Nick I. M. Gould; *On the accurate determination of search directions for simple, differentiable penalty functions*; IMA Journal of Numerical Analysis 6 (1986) pp. 357–372.

[43] L. Grippo, F. Lampariello & S. Lucidi; *A nonmonotone linesearch technique for Newton's method*; SIAM Journal on Numerical Analysis 23 (1986) pp. 707–716.

[44] S. P. Han & O. L. Mangasarian; *Exact penalty functions in nonlinear programming*; Mathematical Programming 17 (1979) pp. 251–269.

[45] *A catalogue of subroutines (HSL 2000)* (2002); Harwell Subroutine Library, AEA Technology, Harwell.

[46] M. P. Hennessey, J. A. Jalkio, C. S. Greene & C. M. Sullivan; *Optimal routing of a sailboat in steady winds* (2006); School of Engineering and Center for Applied Mathematics, University of St. Thomas.

[47] M. P. Hennessey & S. Kumar; *Integrated graphical game and simulation-type problem-based learning in kinematics*; International Journal of Mechanical Engineering Education 34 (2006) pp. 220–232.

[48] M. R. Hestenes; *Multiplier and gradient methods*; Journal of Optimization Theory and Applications 4 (1969) pp. 303–320.

[49] F. S. Hillier & G. J. Liebermann; *Introduction to Operations Research*; seventh edition; Mc-Graw Hill Higher Education (2001).

[50] W. Hock & K. Schittkowski; *Test examples for nonlinear programming codes*; number 187 in Lecture Notes in Economics and Mathematical Systems; Springer Verlag, Heidelberg (1981).

[51] N. Karmarkar; *A new polynomial-time algorithm for linear programming*; Combinatorics 4 (1984) pp. 373–395.

[52] D. E. Kirk; *Optimal Control Theory*; Prentice Hall (1970).

[53] I. J. Lustig, R. E. Marsten & D. F. Shanno; *Interior point methods for linear programming: Computational state of the art*; ORSA Journal on Computing 6 (1994) pp. 1–14.

[54] N. Maratos; *Exact penalty function algorithms for finite dimensional and control optimization problems*; Ph.D. thesis; University of London (1978).

[55] S. Mehrotra; *On the implementation of a primal-dual interior point method*; SIAM Journal on Optimization 2 (1992) pp. 575–601.

[56] H. Mittelmann; *Benchmarks for optimization software.*
URL `http://plato.la.asu.edu/bench.html`

[57] J. L. Morales, J. Nocedal, R. A. Waltz, G. Liu & J. P. Goux; *Assessing the potential of interior methods for nonlinear optimization*; Technical report; Evanston IL (2001).

[58] J. Moré & T. Munson; *Computing mountain passes and transition states*; Mathematical Programming 100 (2004) pp. 151–182.

[59] W. Murray & M. H. Wright; *Line search procedures for the logarithmic barrier function*; SIAM Journal on Optimization 4 (1994) pp. 229–246.

[60] B. A. Murtagh & M. A. Saunders; *MINOS 5.5 user's guide*; Technical Report SOL 83-20R; Systems Optimization Laboratory, Stanford University (1998).

[61] S. G. Nash & A. Sofer; *A barrier method for large-scale constrained optimization*; ORSA Journal on computing 5 (1993) pp. 40–53.

[62] J. Nocedal & S. Wright; *Numerical Optimization*; Springer (1999).

[63] Geert Jan Olsder; *Bicycle routing for maximum suntan*; SIAM Review 45 (2003) pp. 345–358.

[64] I. Percival & D. Richards; *Introduction to Dynamics*; Cambridge University Press (1982).

[65] L. S. Pontryagin, V. G. Boltyanskii, R. V. Gamkrelidze & E. F. Mishchenko; *The Mathematical Theory of Optimal Processes*; John Wiley & Sons, New York (1962).

[66] Mike Powell; *A method for nonlinear constraints in minimization problems*; in *Optimization*, Roger Fletcher, editor; Academic Press, London, New York (1969) pp. 283–298.

[67] Mike Powell; *Convergence properties of algorithms for nonlinear optimization*; SIAM Review 28 (1986) pp. 487–500.

[68] Johannes Reitzenstein; *Designing and computing mountain-pass trajectories via methods of optimal control with applications in chemistry*; Ph.D. thesis; University of Bayreuth (2005).

[69] A. L. Tits, A. Wächter, S. Bakhtiari, T. J. Urban & C. T. Lawrence; *A primal-dual interior point method for nonlinear programming with strong global and local convergence properties*; SIAM Journal on Optimization 14 (2003) pp. 173–199.

[70] Kaoru Tone; *Revisions of constraint approximations in the successive QP method for nonlinear programming problems*; Mathematical Programming 26 (1983) pp. 144–152.

[71] R. J. Vanderbei & D. F. Shanno; *An interior-point algorithm for nonconvex nonlinear programming*; Computational Optimization and Applications 13 (1999) pp. 231–252.

[72] Robert J. Vanderbei.
URL `http://orfe.princeton.edu/~rvdb/ampl/nlmodels`

[73] Robert J. Vanderbei; *Optimal sailing strategies, statistics and operations research program* (1996); University of Princeton.
URL `http://www.sor.princeton.edu/~rvdb/sail/sail.html`

[74] Robert J. Vanderbei; *Loqo: An interior point code for quadratic programming*; Optimization Methods and Software 11 (1999) pp. 451–484.

[75] Robert J. Vanderbei; *A case study in trajectory optimization: putting on an uneven green*; SIAG/OPT Views & News 12 (2001) pp. 6–14.

[76] A. Wächter & L. T. Biegler; *Failure of global convergence for a class of interior point methods for nonlinear programming*; Mathematical Programming 88 (2000) pp. 565–574.

[77] A. Wächter & L. T. Biegler; *On the implementation of an interior point filter line-search algorithm for large-scale nonlinear programming*; Mathematical Programming 106 (2006) pp. 25–37.

[78] R. A. Waltz, J. L. Morales, J. Nocedal & D. Orban; *An interior algorithm for nonlinear optimization that combines line search and trust region steps*; Mathematical Programming 2006 (2006) pp. 391–408.

[79] Stephen Wright; *Primal Dual Interior Point Methods*; SIAM Publications (1997).

[80] H. Yamashita, H. Yabe & T. Tanabe; *A globally and superlinearly convergent primal-dual interior point trust region method for large scale constrained optimization*; Mathematical Programming 102 (2005) pp. 111–151.

[81] Hiroshi Yamashita; *A globally convergent primal-dual interior point method for constrained optimization*; Optimization Methods and Software 10 (1998) pp. 443–469.

[82] E. A. Yildirim & S. J. Wright; *Warm-start strategies in interior point methods for linear programming*; SIAM Journal on Optimization 12 (2002) pp. 782–870.

[83] E. Zermelo; *Über das Navigationsproblem bei Ruhender oder veränderlicher Windverteilung*; Zeitschrift fur Angewandte Mathematik und Mechanik 11 (1931) pp. 114–124.