Iowa Research Online
The University of Iowa's Institutional Repository

University of Iowa
Iowa Research Online

Theses and Dissertations

Spring 2017

Design of a large-scale constrained optimization algorithm and its application to digital human simulation

John Corbett Nicholson
University of Iowa

Copyright © 2017 John Corbett Nicholson

This dissertation is available at Iowa Research Online: https://ir.uiowa.edu/etd/5583

Recommended Citation
Nicholson, John Corbett. "Design of a large-scale constrained optimization algorithm and its application to digital human simulation." PhD (Doctor of Philosophy) thesis, University of Iowa, 2017.
https://doi.org/10.17077/etd.a72hyjg2.

Follow this and additional works at: https://ir.uiowa.edu/etd

Part of the Civil and Environmental Engineering Commons

DESIGN OF A LARGE-SCALE CONSTRAINED OPTIMIZATION ALGORITHM AND ITS APPLICATION TO DIGITAL HUMAN SIMULATION

by

John Corbett Nicholson

A thesis submitted in partial fulfillment
of the requirements for the Doctor of Philosophy
degree in Civil and Environmental Engineering
in the Graduate College of
The University of Iowa

May 2017

Thesis Supervisor: Professor Jasbir S. Arora
Professor Karim Abdel-Malek

Copyright by

JOHN CORBETT NICHOLSON

2017

All Rights Reserved

Graduate College
The University of Iowa
Iowa City, Iowa

CERTIFICATE OF APPROVAL

PH.D. THESIS

This is to certify that the Ph.D. thesis of

John Corbett Nicholson

has been approved by the Examining Committee
for the thesis requirement for the Doctor of Philosophy
degree in Civil and Environmental Engineering at the May 2017 graduation.

Thesis Committee: ___________________________________
Jasbir S. Arora, Thesis Supervisor

____
Karim Abdel-Malek, Thesis Supervisor

____
Salam Rahmatalla

____
Asghar Bhatti

____
Rajan Bhatt

To my teachers and mentors.

Our greatest weakness lies in giving up. The most certain way to succeed is always to try just one more time.

Thomas A. Edison

ACKNOWLEDGMENTS

I am extremely grateful to Professor Jasbir S. Arora, Professor Karim Abdel-Malek, Professor Salam Rahmatalla, Professor Asghar Bhatti, and Dr. Rajan Bhatt for their direct support of this work. As members of my dissertation committee and experts in the fields of optimization and digital human simulation, they have been crucial in helping me achieve the objectives of this research. I am very thankful for the time they have taken to review my dissertation and provide suggestions to improve it. Their efforts add a great deal to this research and challenge me to think about my research more critically.

Special thanks are given to my academic advisor Professor Jasbir S. Arora for sharing his expertise and passion for optimization with me. I am forever indebted to him for the immensely valuable knowledge of optimization, and numerical methods in general, he has bestowed upon me. Additionally, I am immensely grateful for his steadfast commitment to this work, through setbacks and successes, and to me.

Special thanks are also given to Professor Karim Abdel-Malek and the Virtual Soldier Research Program in general for motivating and supporting, both financially and technically, this research. It has been an exciting and inspiring experience to be a part of a team that continuously pushes the limits of what software can achieve.

ABSTRACT

A new optimization algorithm and general purpose software package, which can efficiently solve large-scale constrained non-linear optimization problems and leverage parallel computing, is designed and studied. The new algorithm, referred to herein as LASO or LArge Scale Optimizer, combines the best features of various algorithms to create a computationally efficient algorithm with strong convergence properties. Numerous algorithms were implemented and tested in its creation. Bound-constrained, step-size, and constrained algorithms have been designed that push the state-of-the-art. Along the way, five novel discoveries have been made: (1) a more efficient and robust method for obtaining second order Lagrange multiplier updates in Augmented Lagrangian algorithms, (2) a method for directly identifying the active constraint set at each iteration, (3) a simplified formulation of the penalty parameter sub-problem, (4) an efficient backtracking line-search procedure, (5) a novel hybrid line-search trust-region step-size calculation method. The broader impact of these contributions is that, for the first time, an Augmented Lagrangian algorithm is made to be competitive with state-of-the-art Sequential Quadratic Programming and Interior Point algorithms.

The present work concludes by showing the applicability of the LASO algorithm to simulate one step of digital human walking and to accelerate the optimization process using parallel computing.

PUBLIC ABSTRACT

Numerical optimization is a fundamental and necessary step in the simulation of many real-world processes in the sciences, physics, engineering, and economics. Unfortunately, however, the sequential nature of the optimization process often acts as one of the greatest bottlenecks to achieving real-time simulation in these fields. One such example, which is considered here, is the problem of digital human simulation. Therefore, a new optimization algorithm and associated software package, which can efficiently solve large-scale problems and leverage parallel computing, is needed. The result of this research is a new optimization algorithm and general purpose software package that push the state-of-the-art in the field of optimization.

TABLE OF CONTENTS

LIST OF TABLES x

LIST OF FIGURES xi

LIST OF SYMBOLS xii

LIST OF ABBREVIATIONS xiv

CHAPTER I INTRODUCTION 1

1.1 Introductory Remarks 1
1.2 Review of Literature 2
1.3 Objective of Research 8
1.4 Scope of Thesis 8

CHAPTER II DESIGN OF ALGORITHM FOR BOUND-CONSTRAINED OPTIMIZATION PROBLEMS 10

2.1 The Bound-Constrained Minimization 10
2.1.1 Basic Algorithm 11
2.1.1.1 Active/Inactive Variables Procedure 12
2.2 Steepest Descent 13
2.2.1 Bound-Constrained Steepest Descent Algorithm 13
2.2.2 Steepest Descent Numerical Results 14
2.3 L-BFGS 15
2.3.1 Bound-Constrained L-BFGS Algorithm 17
2.3.1.1 L-BFGS Two-Loop Recursion 19
2.3.2 L-BFGS Numerical Results 20
2.4 Conjugate Gradient 21
2.4.1 Bound-Constrained Conjugate Gradient Algorithm 22
2.4.1.1 Traditional Conjugate Gradient 23
2.4.1.2 Preconditioned Conjugate Descent 23
2.4.1.3 Hybrid Conjugate Gradient 24
2.4.2 Conjugate Gradient Numerical Results 24

CHAPTER III DESIGN OF ALGORITHM FOR CALCULATING STEP SIZE 26

3.1 The Armijo’s Rule Line Search 26
3.1.1 Flow Diagram 26
3.1.2 Basic Algorithm 28
3.1.3 Modified Step Size Procedure 28
3.1.3.1 Modified Step Size Procedure Algorithm 30
3.1.4 Armijo’s Rule Numerical Results 31
3.2 Efficent Backtracking Algorithm 32
3.2.1 Standard Backtracking 35
3.2.2 Quadratic Interpolation 35
3.2.2.1 Quadratic Interpolation Algorithm 1 35

3.2.2.2 Quadratic Interpolation Algorithm 2 35
3.2.3 Cubic Interpolation 36
3.2.3.1 Cubic Interpolation Algorithm 1 36
3.2.3.2 Cubic Interpolation Algorithm 2 37
3.2.4 Quagratic Regression..37
3.2.5 Efficient Backtracking Armijo's Rule Numerical Results 38
3.3 Incorporation of a Trust Region 39
3.3.1 Literature Review 40
3.3.2 A Novel Hybrid Line Search / Trust Region Approach 41
3.3.2.1 Liu's Rule for Updating the Trust Region Radius 43
3.3.3 Hybrid Line Search / Trust Region Numerical Results.................44

CHAPTER IV DESIGN OF ALGORITHM FOR CONSTRAINED

OPTIMIZATION PROBLEMS...46

4.1 Augmented Lagrangian 46
4.1.1 Basic Algorithm...47
4.1.2 Flow Diagram for Basic Algorithm...50
4.1.3 Limitations of the Basic Algorithm...52
4.1.4 Literature Review 52
4.1.5 Overcoming Limitations of the Basic Algorithm..........................58
4.1.6 Novel Algorithm..59
4.1.7 Flow Diagram for Novel Algorithm 73
4.1.8 Novel Algorithm Numerical Results 76

CHAPTER V PARALLELIZATION...78

5.1 Literature Review 78
5.2 Parallel L-BFGS Two-Loop Recursion 82
5.3 Parallelization on the Central Processing Unit (CPU)..............................83
5.4 Parallelization on the Graphics Processor Unit (GPU).............................85

CHAPTER VI DIGITAL HUMAN WALKING PROBLEM...87

6.1 Design Variables...87
6.2 Objective Function..89
6.3 Constraints 91
6.3.1 Joint Limits...91
6.3.2 Torque Limits 91
6.3.3 Ground Penetration...92
6.3.4 Dynamic Balance...92
6.3.5 Arm-Leg Coupling 93
6.3.6 Self-Avoidance 93
6.3.7 Symmetry Conditions..93
6.3.8 Ground Clearance 94
6.3.9 Initial and Final Foot Contacting Position.......................................94
6.4 Numerical Results...94

CHAPTER VII CONCLUSIONS AND AREAS FOR FURTHER RESEARCH 96

7.1 Conclusions....96
7.2 Areas for Further Research....97
7.2.1 A Holistic Strategy for Parallelization 97
7.2.2 Nonmonotone Line Search Rule 99
7.2.2.1 Literature Review....100
7.2.2.2 A Novel Simulated Annealing Like Rule 101

REFERENCES 103

APPENDIX A: LASO USER GUIDE....108

LIST OF TABLES

Table

2.1 Steepest Descent Numerical Results 15

2.2 L-BFGS Numerical Results 21

2.3 Traditional, Preconditioned, and Hybrid Conjugate Gradient Performance 25

3.1 Standard Armijo's Rule and Modified Step Size Procedure Performance 32

3.2 Standard Backtracking and Efficient Backtracking Performance 39

3.3 Efficient Backtracking and Hybrid Method Performance 45

4.1 LASO and SNOPT Performance on Hock-Schittkowski Test Problems 77

5.1 CPU Parallelized LASO Performance on Extended Rosenbrock Function 84

5.2 GPU Parallelized LASO Performance on Extended Rosenbrock Function 86

6.1 LASO and SNOPT Walking Problem Performance 95

LIST OF FIGURES

Figure

3.1 Basic Armijo’s Rule Line Search Flow Diagram..27

3.2 Modified Step Size Procedure Flow Diagram...29

3.3 Efficient Backtracking Step Size Algorithm Flow Diagram...................................34

3.4 Hybrid Line Search / Trust Region Flow Diagram..42

4.1 Basic Augmented Lagrangian Iteration Flow Diagram...51

4.2 Novel Augmented Lagrangian Algorithm Flow Diagram.......................................75

6.1 55-DOF Digital Human Model from Xiang et al. (2009)..89

7.1 A Holistic Parallelization Scheme for Multi-Core Workstations............................99

7.2 Simulated Annealing Like Nonmonotone Rule Flow Diagram.............................102

LIST OF SYMBOLS

n	Number of design variables
m	Number of general linear and nonlinear constraints
$\boldsymbol{x}$	Design variables
$f(\boldsymbol{x})$	Objective function
$\nabla f(\boldsymbol{x})$	Gradient of the objective function
$\mathcal{L}$	Lagrangian function
$\nabla\mathcal{L}$	Gradient of the Lagrangian function
Φ	Augmented Lagrangian function
$\nabla\Phi$	Gradient of the augmented Lagrangian function
$\phi(\alpha)$	Line search function
$\boldsymbol{c}(\boldsymbol{x})$	General linear and nonlinear constraints
$\boldsymbol{h}(\boldsymbol{x})$	Equality constraints to be satisfied
$\boldsymbol{g}(\boldsymbol{x})$	Inequality constraints to be satisfied
$\nabla\boldsymbol{c}$ or $\boldsymbol{J_c}$	Jacobian of the general linear and nonlinear constraints
$\nabla\boldsymbol{h}$ or $\boldsymbol{J_h}$	Equality constraint Jacobian
$\nabla\boldsymbol{g}$ or $\boldsymbol{J_g}$	Inequality constraint Jacobian
$\boldsymbol{\mu}$	Equality constraint Lagrange multiplier vector
$\boldsymbol{\lambda}$	Inequality constraint Lagrange multiplier vector
$\boldsymbol{l}$	Lower bounds on the design variables and general constraints
$\boldsymbol{u}$	Upper bounds on the design variables and general constraints
ε_o	Optimality tolerance
ε_f	Feasibility tolerance
$\overline{K}$	Maximum constraint violation

LIST OF SYMBOLS (CONTINUED)

$\|\|\mathbf{a}\|\|$	2-norm of a vector **a**
$\mathbf{d}$	Search direction vector
$\mathbf{a} \cdot \mathbf{b}$	Dot product of vectors **a** and **b**
$\mathbf{H}$	Inverse Hessian approximation
$\mathbf{B}$	Hessian approximation
α	Step size
k	Iteration counter

LIST OF ABBREVIATIONS

LASO	LArge Scale Optimizer
FSP	Foot Support Polygon
GPU	Graphics Processor Unit
DOF	Degrees Of Freedom
SQP	Sequential Quadratic Programming
QP	Quadratic Programming
AL	Augmented Lagrangian
SD	Steepest Descent
SAR	Standard Armijo’s Rule
L-BFGS	Limited Memory BFGS
T-CG	Traditional Conjugate Gradient
P-CD	Preconditioned Conjugate Gradient
H-CG	Hybrid Conjugate Gradient
QI	Quadratic Interpolation
SBT	Standard Backtracking
EBT	Efficient Backtracking
TR	Trust Region
KKT	Karush-Kuhn-Tucker conditions
ZMP	Zero Moment Point

CHAPTER I

INTRODUCTION

1.1 Introductory Remarks

Researchers at The University of Iowa's Virtual Soldier Research (VSR) Program formulate the problem of simulating human motion as an optimization problem. As human motion simulation tasks increase in duration and complexity, this optimization problem grows quickly in size and computational expense. A new optimization procedure and general purpose software package, which can efficiently solve this high-dimensional tightly-constrained problem and take advantage of parallelization wherever possible, is needed.

In the above-mentioned approach to simulating human motion, dynamic effort of a spatial digital human model is typically taken as the objective to be minimized and the various physical and kinematical requirements of the model are imposed as constraints. The spatial digital human model developed at The University of Iowa consists of 55 Degrees of Freedom (DOF) of which 6 are virtual DOFs that represent global translation and rotation and 49 are physical joint angle DOFs that represent local joint rotations. Time histories of these 55 DOFs make up the design variables. Thus, there are an infinite number of design variables. These design functions are represented by cubic B-splines to transform the problem to a finite dimensional one. The control points of each B-spline are the final design variables that are finite in number. If the contact forces between the body and the external environment are also treated as unknown variables, the number of design variables is increased. Furthermore, as the duration and complexity of tasks increases, greater numbers of constraints to represent the increased physical and kinematical requirements will be needed as well.

Even seemingly simple tasks such as walking and lifting a box are large problems that take minutes to solve. For instance, Xiang (2009) found that simulating one step of

human walking requires 330 design variables (55 DOFs with 6 control points each) and 1036 nonlinear constraints (158 of which are active at the optimal solution) and takes 512 CPU seconds on a Pentium(R) 4, 3.46 GHz computer. Likewise, Xiang (2008) found that lifting a 10-lb box from a lower shelf to a higher shelf requires 220 design variables (55 DOFs with 4 control points each) and 420 nonlinear constraints and takes 200 CPU seconds on a Pentium(R) 4, 3.46 GHz computer. As longer duration and more complex tasks, such as climbing a wall or running an obstacle course, are developed, achieving real time digital human simulation will become increasingly difficult and specialized tools will be needed.

While existing optimization software packages for solving such large-scale nonlinear constrained optimization problems are quite efficient and robust, they offer few options for parallelization and will most likely be less efficient for longer duration and more complex tasks that require greater numbers of design variables and constraints.

A thorough review of existing large-scale nonlinear constrained optimization software packages is now presented to show the current state-of-the-art in the field, conceptualize the best algorithmic structure for the problem at hand, and identify important assumptions relevant to the present work. It is important to note, however, that the literature review for this work is not limited to the next section. Rather, the literature has been reviewed, as needed, throughout this document on specific topic areas.

1.2 Review of Literature

Currently, some of the most efficient and robust software packages for solving large-scale nonlinear constrained optimization problems are LOQO, KNITRO, SNOPT, LANCELOT, and MINOS.

Vanderbei's (1999) LOQO package implements an interior point method and, as such, formulates and solves a primal-dual linear system to generate each design / Lagrange multiplier iterate towards the solution. Requiring the solution of only one

linear system at each iteration, ranks LOQO among the most computationally efficient of the algorithms. LOQO adds slack variables to inequalities to convert them to equalities. This facilitates the formulation of the primal-dual linear system solved by the algorithm, but does increase problem dimensionality substantially when many inequality constraints are present. Instead of explicitly enforcing positivity of the slack variables, LOQO implicitly enforces positivity by incorporating a logarithmic barrier function, which is in terms of the slack variables, into the objective. This removes the need to add slack variable constraints to the problem, however, it does add complicating logarithmic terms to the objective. Rather than combine the objective function and constraint violations into a single penalty objective function to be minimized, LOQO performs updates to the primal and dual (i.e. design and Lagrange multiplier) variables and accepts the new point if either the barrier objective function or infeasibility decreases, similar to filter methods. Thus, LOQO can make progress towards optimality and/or feasibility at every iteration. LOQO uses an Armijo rule line search to ensure global convergence.

Like LOQO, Byrd, Hribar, and Nocedal's (1999) KNITRO package implements an interior point method with a barrier objective function and treats all constraints as equalities. Unlike LOQO, however, KNITRO uses a Sequential Quadratic Programming (SQP) method to solve the subproblem that generates each design / Lagrange multiplier iterate towards the solution, instead of formulating and solving a primal-dual linear system. Specifically, the SQP method is applied to the subproblem to obtain updated design variables and an explicit second-order Lagrange multiplier update formula, which is in terms of the design variables, is applied to obtain updated Lagrange multiplier variables, in a so called two-phase approach. This approach is more computationally expensive than directly solving one linear system, but has the benefit of being able to handle nonlinearities in the problem better. KNITRO uses a trust region approach to ensure global convergence instead of a line search.

Gill, Murray, and Saunders' (2002) SNOPT package implements an active-set SQP method and, as such, solves a linearly constrained Quadratic Programming (QP) subproblem at each iteration. If the linearized versions of the original problem constraints are inconsistent then the SQP subproblem is infeasible and SNOPT switches into elastic mode. Elastic mode ensures consistency of the linearized constraints by adding two slack variables to each linearized equality constraint and one slack variable to each linearized inequality constraint. Unlike both LOQO and KNITRO, which use second order information about the problem directly, SNOPT uses a limited-memory quasi-Newton approach to maintain an approximation of the Hessian. This facilitates the solution of problems where second order information is not readily available. Like LOQO, SNOPT uses a line search to ensure global convergence.

Conn, Gould, and Toint's (1992) LANCELOT package implements an augmented Lagrangian technique and, therefore, combines the objective function and constraint violations into a single augmented Lagrangian objective function. This augmented Lagrangian objective function is then minimized to get an approximate solution of the design variables. After an approximate solution has been obtained, the Lagrange multiplier variable for each constraint is updated, along with the penalty parameter, and the process is repeated. LANCELOT efficiently handles constraints on the design variable bounds by enforcing them directly, using a bound-constrained algorithm, instead of treating them as additional constraints. LANCELOT converts all inequality constraints to equalities by introducing slack variables and uses a trust region approach to ensure global convergence.

Murtagh and Saunders' (2003) MINOS package implements a Linearly Constrained Lagrangian (LCL) approach that minimizes an augmented Lagrangian function subject to linearizations of the constraints. MINOS uses the same smooth exact augmented Lagrangian as LANCELOT, which is essentially a combination of a Lagrangian and a quadratic penalty function. However, MINOS replaces the augmented

Lagrangian's constraint terms with the difference between each constraint's actual value and the value of its linearization at the current design. Unlike LANCELOT, which avoids a factorization of the constraint Jacobian matrix by performing updates to the Lagrange multiplier variables at the end of each bound-constrained minimization, MINOS obtains new estimates of the Lagrange multiplier variables at every step towards the solution. This results in superior progress towards feasibility, especially in problems with large numbers of constraints, but becomes very computationally expensive on large degree of freedom problems. Both LANCELOT and MINOS solve their bound-constrained and linearly-constrained subproblems, respectively, using quasi-Newton approximations of the Hessian. However, MINOS is based on an approach that uses a reduced approximation of the Hessian whereas LANCELOT's approach maintains a full approximation of the Hessian, making MINOS, once again, better suited for relatively few degree of freedom problems and LANCELOT better suited for larger degree of freedom problems.

Benson, Shanno, and Vanderbei (2002) studied the performance of LOQO, KNITRO, and SNOPT on a large set of large-scale test problems of various types. For generally constrained (i.e. both equality and inequality constrained) nonlinear problems they found that, in general, KNITRO was the most robust (i.e. it solved the greatest number of problems using the solver's default settings before a maximum time was exceeded) and that LOQO was the fastest in terms of CPU time. Both KNITRO and LOQO were substantially faster than SNOPT in terms of CPU time, which is to be expected since SNOPT requires the solution of a linearly constrained subproblem at each step and KNITRO and LOQO essentially only require solution of a linear system of equations. LOQO and KNITRO also use the actual Hessian matrix in computations, thus requiring second order information about the problem, whereas SNOPT uses an approximation of the Hessian only. It is unclear from Benson, Shanno, and Vanderbei's work, however, which software package performed best in terms of function and gradient

evaluations, a critical aspect in solving problems with expensive function and gradient evaluations. Also, many of the SNOPT runs were stopped prematurely because the maximum time limit was exceeded. So, it is unclear which software package truly leads in robustness.

Dolan and More (2001) studied the performance of MINOS, SNOPT, LANCELOT, and LOQO on a set of large-scale constrained optimal control and parameter estimation problems. They note that software packages like LANCELOT and LOQO that require second-order problem information typically converge in fewer iterations, but each iteration is more expensive because of the cost of calculating second-order information. They also note how obtaining second-order information may not even be possible in some cases. In general, in terms of performance profiles, they found LOQO, SNOPT, and MINOS to be competitive with each other and LANCELOT to be less competitive. This is to be expected since LOQO, SNOPT, and MINOS make steps towards optimality and feasibility at every step whereas LANCELOT repeatedly approximates the optimal solution before taking a step towards feasibility. Specifically, LOQO was the fastest solver, in terms of CPU time, and SNOPT was the most robust, solving over 90% of the test problems.

Bongartz, Conn, Gould, Saunders, and Toint (1997) studied the performance of the LANCELOT and MINOS software packages on a set of over 900 constrained and unconstrained large-scale nonlinear optimization problems. In general, they found that LANCELOT was more efficient in terms of function and gradient evaluations, but that MINOS was more computationally efficient, except when there are many degrees of freedom. These results make sense since MINOS's reduced Hessian approximation strategy requires less computation, but may hinder convergence on large degree of freedom problems. Also, in Bongartz, Conn, Gould, Saunders, and Toint's work, LANCELOT's default configuration has been used, which uses second-order problem information directly instead of maintaining an approximation of the Hessian. Lastly,

MINOS is found to be more reliable on linear programming problems and LANCELOT is found to be somewhat more reliable on problems with nonlinear constraints. This results from MINOS's linearly constrained Lagrangian approach.

Nocedal and Wright (2006) note that, at present, interior point methods (e.g. LOQO and KNITRO) and active-set methods (e.g. SNOPT) appear to be the most promising and augmented Lagrangian methods (e.g. MINOS and LANCELOT) appear to be less efficient. However, rather than pick a single algorithmic framework to move forward with, the present work attempts to combine favorable aspects of multiple of the abovementioned algorithms to further the state-of-the-art in the field. Specifically, based on the findings of previous researchers, it is hypothesized that the following guiding principles will yield the most efficient and robust algorithm: (i) similar to MINOS and LANCELOT constraints will be incorporated into the objective via a smooth exact augmented Lagrangian, which preserves the convergence properties of the underlying unconstrained algorithm, to avoid the addition of dimensionality increasing slack variables and complicating logarithmic terms to the objective; (ii) similar to LANCELOT and SNOPT a bound-constrained formulation will be used to keep the number of constraints as small as possible; (iii) similar to LOQO, KNITRO, MINOS, and SNOPT, updates to both the design and Lagrange multiplier variables will occur at every iteration to support rapid progress towards optimality and feasibility; (iv) similar to LOQO and KNITRO, second-order updates to the Lagrange multiplier variables, which require the solution of a linear system of equations only, will be preferred over first-order updates, like LANCELOT, or obtaining estimates of the multipliers directly by solving a computationally expensive linearly constrained subproblem, like SNOPT; (v) similar to SNOPT, inequality constraints will be handled directly via an active set strategy, instead of converting them to equality constraints by introducing dimensionality increasing slack variables; (vi) unlike SNOPT and MINOS, constraints will not be linearized, but instead used in their original form to avoid infeasible subproblems, which require the addition of

more dimensionality increasing slack variables, and efficiency hindering numerical issues, such as the Maratos effect; (vii) similar to LOQO, KNITRO, SNOPT, and LANCELOT a full Hessian will be maintained to handle large degree of freedom problems efficiently; (viii) similar to SNOPT, MINOS, and LANCELOT (using non-default options), a limited-memory BFGS approximation of the Hessian will be used to facilitate the solution of problems where second-order information is not readily available and to limit memory usage on high-dimensional problems; (ix) similar to LOQO, SNOPT, and MINOS, line search based methods will be preferred over computationally expensive trust region approaches and proper scaling of the Hessian will be used to encourage unit step size acceptance, thus keeping the number of function evaluations to a minimum; (x) unlike all of the abovementioned software packages, parallelization will be used to the maximum extent possible to improve efficiency.

1.3 Objective of Research

The primary objectives of this research are two-fold: 1) study various algorithms and develop a general purpose algorithm for constrained optimization, which efficiently solves the digital human simulation problem formulated by researchers at The University of Iowa's Virtual Soldier Research Program, and integrate this software into the Santos digital human simulation software, and 2) develop and implement a parallelization strategy that further improves the efficiency of the optimization solution process and software package just described.

1.4 Scope of Thesis

This thesis studies the various aspects necessary to develop algorithms and a general purpose large-scale nonlinear constrained optimization software package. Furthermore, it offers five novel contributions that extend the current state-of-the-art in the field and studies the numerical performance of these contributions. Specifically, a novel hybrid line search / trust region approach, a new efficient backtracking line search

procedure, a more efficient and robust method for obtaining second order Lagrange multiplier updates in augmented Lagrangian algorithms, a method for directly identifying the active constraint set at each iteration, and a simplified formulation of the penalty parameter sub-problem are offered and their numerical performance is studied on a set of test problems from the literature.

Chapter II takes a closer look at bound-constrained algorithm design and studies the numerical performance of the underlying unconstrained search direction techniques, which determine the efficiency and convergence properties of a bound-constrained algorithm. Chapter III examines the step size calculation sub-problem that, in addition to ensuring global convergence, has a significant impact on an algorithm's convergence and computational efficiency. Here, a new efficient backtracking algorithm and a novel hybrid line search / trust region approach are proposed and their numerical performance is studied. Chapter IV studies the design of constrained algorithms and proposes modifications to the standard augmented Lagrangian formulation that incorporate favorable aspects of SQP and interior point methods. Specifically, a bound-constrained augmented Lagrangian method with continuous multiplier updates, which requires the solution of only two linear systems per iterate towards the solution, is proposed and its numerical performance studied. In chapter V, a parallel L-BFGS two-loop recursion is proposed and implemented on the Central Processing Unit (CPU) and Graphics Processing Unit (GPU) to better understand which offers the greater promise for real time optimization. In chapter VI, the algorithm developed in chapters II through V is applied to the tightly-constrained high-dimensional highly-nonlinear predictive dynamics digital human walking problem and its performance on this problem is compared to that of SNOPT. Finally, in chapter VII some conclusions obtained from the results of chapters II through VI are made and areas for further research are identified.

CHAPTER II

DESIGN OF ALGORITHM FOR BOUND-CONSTRAINED OPTIMIZATION PROBLEMS

2.1 The Bound-Constrained Minimization

Bound-constrained algorithms are preferred over constrained algorithms for high-dimensional problems with only simple bounds on the design variables. They're preferred not only for their efficient handling of bound constraints, but also for their ability to quickly identify and essentially remove active variables from the problem early on. Specifically, by enforcing the KKT optimality conditions for the bound constraints explicitly, instead of defining a Lagrangian function that enforces them implicitly, bound-constrained methods are able to solve for bound constraint Lagrange multiplier variables exactly at each iteration and avoid the addition of complicating penalty terms to the objective. Additionally, by identifying the set of active variables early on and handling them separately, these methods allow the search direction calculation and step size identification steps to occur in a lower-dimensional space, which has various numerical benefits.

The bound-constrained algorithm presented here is based on Schwartz and Polak's (1997) approach. Schwartz and Polak's (1997) approach has been proven to identify the active set in a finite number of iterations and to preserve the rate of convergence of the underlying unconstrained search direction. It has also been designed such that it can be used with any unconstrained search direction calculation technique. Section 2.1.1 presents the basic algorithm developed by Schwartz and Polak (1997) and clarified by Arora (2012). Similar to these works, the basic algorithm presented includes the following general steps: initialization of algorithmic parameters, identification of the inactive/active sets and Lagrange multiplier variable values, check for convergence, setting of the active set search direction to that of steepest descent, calculation of an

inactive set search direction satisfying a set of conditions, identification of a step size satisfying an Armijo-like rule, updating the design using a projection operator. Unlike these works, a slightly different set of conditions, i.e. eqns. 2.5 and 2.6, have been used to test the acceptability of the inactive set search direction. This change was made after numerical testing showed that for some poorly scaled problems, the original conditions proposed by Schwartz and Polak (1997) were rejecting some search directions that actually satisfied the original intent of the conditions. Therefore, conditions that are less compact, but more robust to poorly scaled problems have been used.

2.1.1 Basic Algorithm

The following bound-constrained algorithm applies to optimization problems with lower and upper bounds on the design variables defined as:

minimize $f(\mathbf{x})$

subject to $l_i \le x_i \le u_i \quad i = 1, n$

Step 1: Set k = 0, ε_a = 10e-8, ε_c = 10e-6, $\rho \in (0,1]$, $\sigma_1 \in (0,1)$, and $\sigma_2 \in (1, \infty)$, where k is the minor iteration counter, ε_a is the constraint activity parameter, ε_c is the convergence parameter, ρ is a factor between 0 and 1, and σ_1 and σ_2 are algorithmic parameters.

Step 2: Ensure all design variables are on or within their bounds by applying the following projection operator on each element of $\mathbf{x}^{(k)}$ such that $x_i^{(k)} = P_i\left(x_i^{(k)}\right)$:

$$P_i(z) = \begin{cases} l_i & if\ z \le l_i \\ z & if\ l_i < z < u_i \\ u_i & if\ z \ge u_i \end{cases} ; \quad i = 1, n \tag{2.1}$$

Step 3: Calculate the gradient vector at the current design $\mathbf{x}^{(k)}$:

$$\boldsymbol{\nabla} \boldsymbol{f}^{(k)} = \boldsymbol{\nabla} \boldsymbol{f}\left(\mathbf{x}^{(k)}\right) \tag{2.2}$$

Step 4: Identify the active- and inactive-sets A_k and I_k of the design variables, respectively; Calculate the Lagrange multipliers for the lower- and upper-bound

constraints and the active elements of the search direction vector $\mathbf{d}^{(k)}$ using the active/inactive variables procedure shown in section 2.1.1.1.

Step 5: Check if either of the following convergence criteria are satisfied. If satisfied, stop. Otherwise, continue.

$$\left\|\nabla \boldsymbol{f}_{I_k}^{(k)}\right\| \leq \varepsilon_c\{max(1, \|\mathbf{x}^{(k)}\|)\} \tag{2.3}$$

$$\left\|\nabla \boldsymbol{f}_{I_k}^{(k)}\right\| \leq \varepsilon_c\{max(1, |f(\mathbf{x}^{(k)})|)\} \tag{2.4}$$

Step 6: Calculate the components of the search direction vector corresponding to the inactive set (i.e., $\mathbf{d}_{I_k}^{(k)}$) using any unconstrained method. Ensure that the search direction obtained is a direction of descent by checking the following conditions. If either of the following conditions is not satisfied, restart the algorithm from the current design by taking the search direction as that of steepest descent and by throwing out any historical information used in calculating $\mathbf{d}_{I_k}^{(k)}$:

$$\sigma_1\left\|\nabla \boldsymbol{f}_{I_k}^{(k)}\right\| \leq \left\|\mathbf{d}_{I_k}^{(k)}\right\| \leq \sigma_2\left\|\nabla \boldsymbol{f}_{I_k}^{(k)}\right\| \tag{2.5}$$

$$-\frac{\left(\mathbf{d}_{I_k}^{(k)} \cdot \nabla \boldsymbol{f}_{I_k}^{(k)}\right)}{\left\|\mathbf{d}_{I_k}^{(k)}\right\|\left\|\nabla \boldsymbol{f}_{I_k}^{(k)}\right\|} \geq \frac{\sigma_1}{\sigma_2} \tag{2.6}$$

Step 7: Find a step size $\alpha^* > 0$ that satisfies the following Armijo-like rule:

$$f(\alpha) \leq f(0) + \rho\left[\alpha\left(\nabla \boldsymbol{f}_{I_k}^{(k)} \cdot \mathbf{d}_{I_k}^{(k)}\right) + \nabla \boldsymbol{f}_{A_k}^{(k)} \cdot \{\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}\}_{A_k}\right] \tag{2.7}$$

Note: at trial steps the design should be updated using the projection operator defined in step 2 such that $\mathbf{x}^{(k+1)} = \boldsymbol{P}(\mathbf{x}^{(k)} + \alpha\mathbf{d}^{(k)})$. Update the design by setting $\mathbf{x}^{(k+1)} = \boldsymbol{P}(\mathbf{x}^{(k)} + \alpha^*\mathbf{d}^{(k)})$. Calculate $\nabla \boldsymbol{f}^{(k+1)}$.

Step 8: Set $k = k + 1$, $\mathbf{x}^{(k)} = \mathbf{x}^{(k+1)}$, $\nabla \boldsymbol{f}^{(k)} = \nabla \boldsymbol{f}^{(k+1)}$, and go to Step 4.

2.1.1.1 Active/Inactive Variables Procedure

The following procedure may be used to identify the active- and inactive-sets A_k and I_k of the design variables, respectively; Calculate the Lagrange multipliers for the

lower- and upper-bound constraints and the active elements of the search direction vector $\mathbf{d}^{(k)}$ such that the KKT optimality conditions are satisfied:

if $l_i \leq x_i^{(k)} \leq l_i + \varepsilon(\mathbf{x}^{(k)})$ and $\nabla f_i^{(k)} > 0$ (i.e., lower-bound is active)

then $i \in A_k,\ u_{Li} = \nabla f_i^{(k)} \geq 0,\ u_{Ui} = 0,\ d_i^{(k)} = -\nabla f_i^{(k)}$

else if $u_i - \varepsilon(\mathbf{x}^{(k)}) \leq x_i^{(k)} \leq u_i$ and $\nabla f_i^{(k)} < 0$ (i.e., upper-bound is active)

then $i \in A_k,\ u_{Li} = 0,\ u_{Ui} = -\nabla f_i^{(k)} \geq 0,\ d_i^{(k)} = -\nabla f_i^{(k)}$

else $l_i + \varepsilon(\mathbf{x}^{(k)}) < x_i^{(k)} < u_i - \varepsilon(\mathbf{x}^{(k)})$ (i.e., neither bound is active)

then $i \in I_k,\ u_{Li} = 0,$ and $u_{Ui} = 0$

where $\varepsilon(\mathbf{x}^{(k)}) = min\{\varepsilon_a,\ \|\boldsymbol{w}(\mathbf{x}^{(k)})\|\}$

and $w_i(x_i^{(k)}) = \begin{cases} max\left\{-\nabla f_i^{(k)},\ \left(l_i - x_i^{(k)}\right)\right\} & if\ \nabla f_i^{(k)} > 0 \\ max\left\{\ \nabla f_i^{(k)},\ \left(x_i^{(k)} - u_i\right)\right\} & if\ \nabla f_i^{(k)} < 0 \\ 0 & if\ \nabla f_i^{(k)} = 0 \end{cases}$

2.2 Steepest Descent

The steepest descent or simple gradient descent method, which has origins dating back to the work of Cauchy in 1847, calculates the search direction as the direction in which the objective function decreases most rapidly, at least in the immediate area around current point. The steepest descent approach integrated into a bound-constrained algorithm is shown in section 2.2.1.

2.2.1 Bound-Constrained Steepest Descent Algorithm

The following bound-constrained algorithm applies to optimization problems with lower and upper bounds on the design variables defined as:

minimize $f(\mathbf{x})$

subject to $l_i \leq x_i \leq u_i \quad i = 1, n$

Step 1: Set k = 0, ε_a = 10e-8, ε_c = 10e-6, ρ = 0.2, σ_1 = 0.2, and $\sigma_2 = 10$, where k is the minor iteration counter, ε_a is the constraint activity parameter, ε_c is the convergence parameter, ρ is a factor between 0 and 1, and σ_1 and σ_2 are algorithmic parameters.

Step 2: Ensure all design variables are on or within their bounds by applying the projection operator, i.e. eqn. 2.1, on each element of $\mathbf{x}^{(k)}$ such that $x_i^{(k)} = P_i\left(x_i^{(k)}\right)$.

Step 3: Calculate the gradient vector at the current design $\mathbf{x}^{(k)}$ as in eqn. 2.2.

Step 4: Identify the active- and inactive-sets A_k and I_k of the design variables, respectively; Calculate the Lagrange multipliers for the lower- and upper-bound constraints and the active elements of the search direction vector $\mathbf{d}^{(k)}$ using the active/inactive variables procedure shown in section 2.1.1.1.

Step 5: Check if either of the convergence criteria, i.e. eqn. 2.3 or eqn. 2.4, are satisfied. If satisfied, stop. Otherwise, continue.

Step 6: Calculate the components of the search direction vector corresponding to the inactive set (i.e., $\mathbf{d}_{I_k}^{(k)}$) by setting $\mathbf{d}_{I_k}^{(k)} = -\boldsymbol{\nabla f}_{I_k}^{(k)}$ (i.e., the steepest descent direction).

Ensure that the search direction obtained is a direction of descent by checking the conditions of eqns. 2.5 and 2.6.

Step 7: Find a step size $\alpha^* > 0$ that satisfies the Armijo-like rule of eqn. 2.7. Note: at trial steps the design should be updated using the projection operator defined in step 2 such that $\mathbf{x}^{(k+1)} = \boldsymbol{P}(\mathbf{x}^{(k)} + \alpha\mathbf{d}^{(k)})$. Update the design by setting $\mathbf{x}^{(k+1)} = \boldsymbol{P}(\mathbf{x}^{(k)} + \alpha^*\mathbf{d}^{(k)})$. Calculate $\boldsymbol{\nabla f}^{(k+1)}$.

Step 8: Set $k = k + 1$, $\mathbf{x}^{(k)} = \mathbf{x}^{(k+1)}$, $\boldsymbol{\nabla f}^{(k)} = \boldsymbol{\nabla f}^{(k+1)}$, and go to Step 4.

2.2.2 Steepest Descent Numerical Results

Table 2.1 shows the performance of the Steepest Descent (SD) bound-constrained algorithm using a Standard Armijo's Rule (SAR) linesearch, which involves finding a stepsize to satisfy the Armijo-like rule of equation 2.7 and checking a curvature condition

to ensure that the stepsize has not become too small, on a set of five bound-constrained test problems from the Hock-Schittkowski (2009) collection. Details of the Standard Armijo's Rule linesearch may be found in the next chapter. As expected, the Steepest Descent bound-constrained algorithm converged rapidly during early iterations, resulting in large decreases in the objective early on, and slowly during later iterations. While the algorithm performed reasonably well on three of the five test problems, it performed unacceptably slow on two of the test problems, requiring thousands of iterations and function evaluations.

Table 2.1 Steepest Descent Numerical Results.

			SAR
			SD
No.	Problem	n	Iter (F)
1	Hock-Schittkowski Problem 1	2	18,291 (257,000)
2	Hock-Schittkowski Problem 2	2	4,132 (62,543)
3	Hock-Schittkowski Problem 4	2	1 (20)
4	Hock-Schittkowski Problem 5	2	25 (93)
5	Hock-Schittkowski Problem 110	10	26 (120)

2.3 L-BFGS

Dennis and Schnabel (1996), Fletcher (2000), Nocedal and Wright (2006), and many others agree that BFGS quasi-Newton methods are among the most efficient and robust methods available for solving unconstrained optimization problems. In the case of high-dimensional problems, where it becomes unrealistic to store full BFGS matrices in a computer's high speed memory, limited memory BFGS algorithms have proven to be effective tools. Perry (1977) and Shanno (1978) were the first to study limited memory methods and they have since been studied by many others. Findings from some of these works are discussed here.

Limited memory BFGS approaches typically differ in their choice of BFGS updating strategy and initial BFGS matrix. Morales (2002) studied the numerical performance of two of the most efficient and robust limited memory approaches on a large set of both low- and high-dimensional unconstrained test problems. The two approaches chosen by Morales were Nocedal's (1980) L-BFGS method and Gill, Murray, and Saunders' (2002) limited memory approach as implemented in the SNOPT software package. The two approaches differ in that the L-BFGS method uses a continuous updating strategy for all problems and SNOPT maintains full BFGS matrices for small problems and uses a restart strategy for large problems. The L-BFGS method outperformed SNOPT, in terms of function and gradient evaluations, by a small margin on the low-dimensional test problems and by a significant margin on the high-dimensional test problems. Additionally, Liu and Nocedal (1989) compared the performance of Nocedal's L-BFGS method to Griewank and Toint's (1982) partitioned quasi-Newton method and found the L-BFGS method to be superior for large problems with non-sparse Hessian matrices and for problems where information on the seperability of the objective function is unknown a priori. Therefore, since the present work is concerned with designing a general purpose and efficient algorithm and optimization software package, the L-BFGS approach proposed by Nocedal (1980) is studied further here.

Nocedal's (1980) L-BFGS approach works by generating BFGS quasi-Newton matrices using vectors $\mathbf{s}^{(k)} = \mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}$ and $\mathbf{y}^{(k)} = \boldsymbol{\nabla f}^{(k+1)} - \boldsymbol{\nabla f}^{(k)}$ from the previous m iterations. Nocedal's procedure for generating these matrices and efficiently performing the necessary matrix vector product is presented in section 2.3.1.1. One requirement of the L-BFGS approach is that $\mathbf{s}^{(k)} \cdot \mathbf{y}^{(k)}$ be greater than 0 for all $\mathbf{s}^{(k)}$ and $\mathbf{y}^{(k)}$ vectors used to update BFGS matrices. This ensures the positive definiteness of BFGS matrices generated using the approach. However, in practical implementations, it is desirable that $\mathbf{s}^{(k)} \cdot \mathbf{y}^{(k)}$ remain greater than a small positive number. Here, we use the

requirement on $\mathbf{s}^{(k)} \cdot \mathbf{y}^{(k)}$ proposed by Morales (2002), which is similar to SNOPT's requirement on $\mathbf{s}^{(k)} \cdot \mathbf{y}^{(k)}$. Morales' requirement is satisfied so long as the stepsize obtained during linesearch satisfies Wolf's curvature condition, which is presented in the next chapter. Also, in the case of constrained algorithms where linesearches may be performed on merit functions and satisfaction of Wolf's curvature condition can't be guaranteed, Morales' requirement becomes a useful check. This is discussed more in Chapter IV. Nocedal's L-BFGS approach integrated into a bound-constrained algorithm is shown in section 2.3.1. Per the recommendations of Schwartz and Polak (1997), algorithmic parameters σ_1 and σ_2 have been chosen as 0.0002 and $\sqrt{1000}\times 10^3$, respectively. Nocedal's numerical testing suggested that performance improved as the number of corrections, m, stored increased. Liu and Nocedal (1989) observed that the number of function evaluations decreased, in general, as the number of corrections stored increased from 15 to 40, but that the decrease was not dramatic. Morales (2002) achieved his impressive results by taking the number of corrections stored as 20. Therefore, the number of corrections stored is taken as 20 in algorithm 2.3.1 as well. In line with the recommendations of Morales (2002) and others, factors ρ and β are taken as 0.0001 and 0.9, respectively.

2.3.1 Bound-Constrained L-BFGS Algorithm

The following bound-constrained algorithm applies to optimization problems with lower and upper bounds on the design variables defined as:

minimize $f(\mathbf{x})$

subject to $l_i \leq x_i \leq u_i \quad i = 1, n$

Step 1: Set k = 0, $H_0^{(k)}$ = 1, ε_a = 10e-8, ε_c = 10e-6, $\rho = 0.0001$, $\beta = 0.9$, $m = 20$, σ_1 = 0.0002, and $\sigma_2 = \sqrt{1000}\times 10^3$, where k is the minor iteration counter, $H_0^{(k)}$ is an initial approximation of the elements of the diagonal *inverse Hessian* matrix for the kth iteration, ε_a is the constraint activity parameter, ε_c is the convergence parameter, ρ is a

factor between 0 and 1, β is a factor between ρ and 1, m is a nonnegative integer specifying the number of L-BFGS correction vectors to store, and σ_1 and σ_2 are algorithmic parameters.

Step 2: Ensure all design variables are on or within their bounds by applying the projection operator, i.e. eqn. 2.1, on each element of $\mathbf{x}^{(k)}$ such that $x_i^{(k)} = P_i\left(x_i^{(k)}\right)$.

Step 3: Calculate the gradient vector at the current design $\mathbf{x}^{(k)}$ as in eqn. 2.2.

Step 4: Identify the active- and inactive-sets A_k and I_k of the design variables, respectively; Calculate the Lagrange multipliers for the lower- and upper-bound constraints and the active elements of the search direction vector $\mathbf{d}^{(k)}$ using the active/inactive variables procedure shown in section 2.1.1.1.

Step 5: Check if either of the convergence criteria, i.e. eqn. 2.3 or eqn. 2.4, are satisfied. If satisfied, stop. Otherwise, continue.

Step 6: Calculate the components of the search direction vector corresponding to the inactive set (i.e., $\mathbf{d}_{I_k}^{(k)}$). If $k < 1$ set $\mathbf{d}_{I_k}^{(k)} = -\boldsymbol{\nabla f}_{I_k}^{(k)}$ (i.e., the steepest descent direction). Otherwise, calculate $\mathbf{d}_{I_k}^{(k)} = -\mathbf{H}_{I_k}^{(k)}\boldsymbol{\nabla f}_{I_k}^{(k)}$ (where $\mathbf{H}_{I_k}^{(k)}$ contains the elements of the *inverse Hessian* approximation of the objective corresponding to the inactive set) using the L-BFGS two-loop recursion given in section 2.3.1.1.

Ensure that the search direction obtained is a direction of descent by checking the conditions of eqns. 2.5 and 2.6. If either of the conditions is not satisfied, restart the algorithm from the current design by taking the search direction as that of steepest descent and by throwing out historical information used in calculating $\mathbf{d}_{I_k}^{(k)}$.

Step 7: Find a step size $\alpha^* > 0$ that satisfies the Armijo-like rule of eqn. 2.7. Note: at trial steps the design should be updated using the projection operator defined in step 2 such that $\mathbf{x}^{(k+1)} = \boldsymbol{P}(\mathbf{x}^{(k)} + \alpha\mathbf{d}^{(k)})$. Update the design by setting $\mathbf{x}^{(k+1)} = \boldsymbol{P}(\mathbf{x}^{(k)} + \alpha^*\mathbf{d}^{(k)})$. Calculate $\boldsymbol{\nabla f}^{(k+1)}$.

Step 8: Set $\mathbf{s}^{(k)} = \mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}$ and $\mathbf{y}^{(k)} = \boldsymbol{\nabla f}^{(k+1)} - \boldsymbol{\nabla f}^{(k)}$. Note: For $k > m$, $\mathbf{s}^{(k-m)}$ and $\mathbf{y}^{(k-m)}$ are discarded and $\mathbf{s}^{(k)}$ and $\mathbf{y}^{(k)}$ are stored in their place, respectively.

Also, if storing $\mathbf{s}$ and $\mathbf{y}$ vectors in $m \times n$ matrices $\mathbf{S}$ and $\mathbf{Y}$ the correction location corresponding to the kth iteration, $j(k)$, can be calculated using the formula:

$$j(k) = k - m \times int\left(\frac{k}{m}\right) \tag{2.8}$$

where int() truncates k/m to an integer value that represents the number of times m has been traversed. Last, ensure that the inverse Hessian approximation of the objective will maintain positive definiteness by checking that $\mathbf{s}^{(k)} \cdot \mathbf{y}^{(k)} \geq \alpha^*(\beta - 1)\boldsymbol{\nabla f}^{(k)} \cdot \mathbf{d}^{(k)}$. If this condition is not satisfied, discard the current $\mathbf{s}^{(k)}$ and $\mathbf{y}^{(k)}$ and do not use them when updating the inverse Hessian approximation of the objective during subsequent iterations.

Step 9: Set $k = k + 1$, $\mathbf{x}^{(k)} = \mathbf{x}^{(k+1)}$, $\boldsymbol{\nabla f}^{(k)} = \boldsymbol{\nabla f}^{(k+1)}$, and go to Step 4.

2.3.1.1 L-BFGS Two-Loop Recursion

The following two-loop recursion proposed by Nocedal (1980) and clarified by Sun and Yuan (2006) offers a computationally efficient means of computing $\mathbf{H}_{I_k}^{(k)}\boldsymbol{\nabla f}_{I_k}^{(k)}$. Per the findings and recommendations of Liu and Nocedal (1989), the dynamic scaling factor $H_0^{(k)}$ is used. Per the findings of Schwartz and Polak (1997), $H_0^{(k)}$ is restricted such that $10^{-3} \leq H_0^{(k)} \leq 10^3$.

$$\mathbf{q} = \boldsymbol{\nabla f}_{I_k}^{(k)}$$

$$\text{LIMIT} = \begin{cases} 0 & if\ k < m \\ k - m & if\ k \geq m \end{cases}$$

for $i = k - 1 : \ -1 : \ i \geq \text{LIMIT}$

set j to the correction location for the ith iteration (i.e. $j(i)$ in step 8).

store $\rho_i = \dfrac{1}{\mathbf{s}_{I_k}^{(j)} \cdot \mathbf{y}_{I_k}^{(j)}}$

store $\alpha_i = \rho_i\left(\mathbf{s}_{I_k}^{(j)} \cdot \mathbf{q}\right)$

$\mathbf{q} = \mathbf{q} - \alpha_i \mathbf{y}_{I_k}^{(j)}$

end

$$H_0^{(k)} = \frac{\mathbf{s}_{I_k}^{(k-1)} \cdot \mathbf{y}_{I_k}^{(k-1)}}{\mathbf{y}_{I_k}^{(k-1)} \cdot \mathbf{y}_{I_k}^{(k-1)}}$$

restrict $H_0^{(k)}$ such that $10^{-3} \le H_0^{(k)} \le 10^3$

$\mathbf{r} = H_0^{(k)}\mathbf{q}$

for $i = \text{LIMIT} : +1 : i \le k-1$

set j to the correction location for the i$^{\text{th}}$ iteration (i.e. $j(i)$ in step 8).

$\beta = \rho_i\left(\mathbf{y}_{I_k}^{(j)} \cdot \mathbf{r}\right)$

$\mathbf{r} = \mathbf{r} + \mathbf{s}_{I_k}^{(j)}(\alpha_i - \beta)$

end

$\mathbf{d}_{I_k}^{(k)} = -\mathbf{r}$

where $\mathbf{r} = \mathbf{H}_{I_k}^{(k)} \nabla \boldsymbol{f}_{I_k}^{(k)}$.

2.3.2 L-BFGS Numerical Results

Table 2.2 compares the performance of the Steepest Descent and L-BFGS bound-constrained algorithms using the Standard Armijo's Rule linesearch on the bound-constrained test problems. As expected, the L-BFGS bound-constrained algorithm outperformed the Steepest Descent bound-constrained algorithm by a substantial margin on all five test problems.

Table 2.2 L-BFGS Numerical Results.

No.	Problem	n	SAR SD Iter (F)	SAR L-BFGS Iter (F)
1	Hock-Schittkowski Problem 1	2	18,291 (257,000)	18 (67)
2	Hock-Schittkowski Problem 2	2	4,132 (62,543)	13 (63)
3	Hock-Schittkowski Problem 4	2	1 (20)	1 (4)
4	Hock-Schittkowski Problem 5	2	25 (93)	7 (17)
5	Hock-Schittkowski Problem 110	10	26 (120)	7 (18)

2.4 Conjugate Gradient

Conjugate gradient methods continue to be a popular option for high-dimensional problems because of their simplicity, small storage requirements, and strong convergence properties. One of the most popular implementations of the method is that proposed by Fletcher and Reeves (1964). In order for such conjugate gradient methods to be competitive with BFGS methods, however, innovative new strategies are required. Some researchers have proposed variable storage or preconditioned conjugate gradient methods that attempt to scale the conjugate gradient direction such that fewer linesearch iterations are required. These methods are effective at reducing the number of objective function evaluations, but do little to reduce the number of unconstrained iterations to convergence. Other researchers have proposed hybrid strategies that attempt to combine favorable properties of various conjugate gradient techniques such that convergence is improved. These methods are effective at reducing the number of unconstrained iterations to convergence, but do relatively little to reduce the number of objective function evaluations. In section 2.4.1, the traditional Fletcher and Reeves (1964) conjugate gradient method, a preconditioned version of Fletcher's (1987) conjugate descent method, and a hybrid approach proposed by Zhou, Zhu, Fan, and Qing (2011) are integrated into a bound-constrained algorithm. In section 2.4.2, the numerical performance of the three methods is studied.

2.4.1 Bound-Constrained Conjugate Gradient Algorithm

The following bound-constrained algorithm applies to optimization problems with lower and upper bounds on the design variables defined as:

minimize $f(\mathbf{x})$

subject to $l_i \leq x_i \leq u_i \quad i = 1, n$

Step 1: Set k = 0, $H_0^{(k)}$ = 1, ε_a = 10e-8, ε_c = 10e-6, $\rho = 0.0001$, $\beta = 0.9$, $m = 20$, σ_1 = 0.2, and $\sigma_2 = 10$, where k is the minor iteration counter, $H_0^{(k)}$ is an initial approximation of the elements of the diagonal *inverse Hessian* matrix for the kth iteration, ε_a is the constraint activity parameter, ε_c is the convergence parameter, ρ is a factor between 0 and 1, β is a factor between ρ and 1, m is a nonnegative integer specifying the number of L-BFGS correction vectors to store, and σ_1 and σ_2 are algorithmic parameters.

Step 2: Ensure all design variables are on or within their bounds by applying the projection operator, i.e. eqn. 2.1, on each element of $\mathbf{x}^{(k)}$ such that $x_i^{(k)} = P_i\left(x_i^{(k)}\right)$.

Step 3: Calculate the gradient vector at the current design $\mathbf{x}^{(k)}$ as in eqn. 2.2.

Step 4: Identify the active- and inactive-sets A_k and I_k of the design variables, respectively; Calculate the Lagrange multipliers for the lower- and upper-bound constraints and the active elements of the search direction vector $\mathbf{d}^{(k)}$ using the active/inactive variables procedure shown in section 2.1.1.1.

Step 5: Check if either of the convergence criteria, i.e. eqn. 2.3 or eqn. 2.4, are satisfied. If satisfied, stop. Otherwise, continue.

Step 6: Calculate the components of the search direction vector corresponding to the inactive set (i.e., $\mathbf{d}_{I_k}^{(k)}$). If $k < 1$ set $\mathbf{d}_{I_k}^{(k)} = -\boldsymbol{\nabla f}_{I_k}^{(k)}$ (i.e., the steepest descent direction). Otherwise, calculate $\mathbf{d}_{I_k}^{(k)}$ using any of the conjugate gradient directions defined in sections 2.4.1.1 to 2.4.1.3.

Ensure that the search direction obtained is a direction of descent by checking the conditions of eqns. 2.5 and 2.6. If either of the conditions is not satisfied, restart the

algorithm from the current design by taking the search direction as that of steepest descent and by throwing out historical information used in calculating $\mathbf{d}_{I_k}^{(k)}$.

Step 7: Find a step size $\alpha^* > 0$ that satisfies the Armijo-like rule of eqn. 2.7. Note: at trial steps the design should be updated using the projection operator defined in step 2 such that $\mathbf{x}^{(k+1)} = \boldsymbol{P}(\mathbf{x}^{(k)} + \alpha\mathbf{d}^{(k)})$. Update the design by setting $\mathbf{x}^{(k+1)} = \boldsymbol{P}(\mathbf{x}^{(k)} + \alpha^*\mathbf{d}^{(k)})$. Calculate $\boldsymbol{\nabla f}^{(k+1)}$.

Step 8: Set $\mathbf{s}^{(k)} = \mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}$ and $\mathbf{y}^{(k)} = \boldsymbol{\nabla f}^{(k+1)} - \boldsymbol{\nabla f}^{(k)}$. Note: For $k > m$, $\mathbf{s}^{(k-m)}$ and $\mathbf{y}^{(k-m)}$ are discarded and $\mathbf{s}^{(k)}$ and $\mathbf{y}^{(k)}$ are stored in their place, respectively. Also, if storing **s** and **y** vectors in $m \times n$ matrices **S** and **Y** the correction location corresponding to the kth iteration, $j(k)$, can be calculated using eqn. 2.8. Last, ensure that the inverse Hessian approximation of the objective will maintain positive definiteness by checking that $\mathbf{s}^{(k)} \cdot \mathbf{y}^{(k)} \geq \alpha^*(\beta - 1)\boldsymbol{\nabla f}^{(k)} \cdot \mathbf{d}^{(k)}$. If this condition is not satisfied, discard the current $\mathbf{s}^{(k)}$ and $\mathbf{y}^{(k)}$ and do not use them when updating the inverse Hessian approximation of the objective during subsequent iterations.

Step 9: Set $k = k + 1$, $\mathbf{x}^{(k)} = \mathbf{x}^{(k+1)}$, $\boldsymbol{\nabla f}^{(k)} = \boldsymbol{\nabla f}^{(k+1)}$, and go to Step 4.

2.4.1.1 Traditional Conjugate Gradient

The traditional Fletcher and Reeves (1964) conjugate gradient method calculates the search direction as follows:

$$\boldsymbol{d}_k = -\boldsymbol{g}_k + \beta_k^{FR}\boldsymbol{d}_{k-1} \tag{2.9}$$

where

$$\beta_k^{FR} = \frac{\boldsymbol{g}_k^T\boldsymbol{g}_k}{\boldsymbol{g}_{k-1}^T\boldsymbol{g}_{k-1}} \tag{2.10}$$

2.4.1.2 Preconditioned Conjugate Descent

The preconditioned version of Fletcher's (1987) conjugate descent method calculates the search direction as follows:

$$\boldsymbol{d}_k = -\mathbf{H}^{\mathrm{LBFGS}}\boldsymbol{g}_k + \bar{\beta}_k^{CD}\boldsymbol{d}_{k-1} \tag{2.11}$$

where

$$\bar{\beta}_k^{CD} = -\frac{\boldsymbol{g}_k^T \mathbf{H}^{\mathrm{LBFGS}} \boldsymbol{g}_k}{\boldsymbol{d}_{k-1}^T \boldsymbol{g}_{k-1}} \tag{2.12}$$

2.4.1.3 Hybrid Conjugate Gradient

The hybrid approach proposed by Zhou, Zhu, Fan, and Qing (2011) calculates the search direction as follows:

$$\boldsymbol{d}_k = -\boldsymbol{g}_k + \beta_k^H \boldsymbol{d}_{k-1} \tag{2.13}$$

where

$$\beta_k^H = max\left\{0, min\{\beta_k^{LS}, \beta_k^{CD}\}\right\} \tag{2.14}$$

and

$$\beta_k^{LS} = -\frac{\boldsymbol{g}_k^T \boldsymbol{y}_{k-1}}{\boldsymbol{d}_{k-1}^T \boldsymbol{g}_{k-1}} \tag{2.15}$$

$$\beta_k^{CD} = -\frac{\boldsymbol{g}_k^T \boldsymbol{g}_k}{\boldsymbol{d}_{k-1}^T \boldsymbol{g}_{k-1}} \tag{2.16}$$

2.4.2 Conjugate Gradient Numerical Results

Table 2.3 compares the performance of the Steepest Descent, L-BFGS, Traditional Conjugate Gradient (T-CG), Preconditioned Conjugate Descent (P-CD), and Hybrid Conjugate Gradient (H-CG) bound-constrained algorithms using the Standard Armijo's Rule linesearch on the bound-constrained test problems. The L-BFGS bound-constrained algorithm outperformed all other methods on four of the five test problems and the P-CD bound-constrained algorithm outperformed all other methods on one of the five test problems. Of the conjugate gradient methods, the P-CD method performed best overall with the T-CG method taking second and the H-CG method coming in third. It is interesting to note, however, that for four of the five problems the H-CG method converged in the same or fewer iterations than the T-CG. Furthermore, upon further study, if a more exact linesearch technique is used then the H-CG method outperforms

the T-CG method both in terms of iterations and function evaluations. Since the L-BFGS method still outperforms the other methods, it is used throughout the rest of this work.

Table 2.3 Traditional, Preconditioned, and Hybrid Conjugate Gradient Performance.

			SAR SD	SAR L-BFGS	SAR T-CG	SAR P-CD	SAR H-CG
No.	Problem	n	Iter (F)	Iter (F)	Iter (F)	Iter (F)	Iter (F)
1	H-S Problem 1	2	18,291 (257,000)	18 (67)	538 (5,205)	>>	1,524 (14,799)
2	H-S Problem 2	2	4,132 (62,543)	13 (63)	37 (374)	17 (50)	33 (317)
3	H-S Problem 4	2	1 (20)	1 (4)	1 (20)	1 (20)	1 (20)
4	H-S Problem 5	2	25 (93)	7 (17)	22 (90)	16 (71)	17 (162)
5	H-S Problem 110	10	26 (120)	7 (18)	13 (142)	10 (56)	13 (142)

CHAPTER III

DESIGN OF ALGORITHM FOR CALCULATING STEP SIZE

3.1 The Armijo's Rule Line Search

Inexact line search techniques that satisfy global convergence requirements are preferable to exact line search techniques when efficiency is of the essence. One of the most widely used inexact line search rules is that proposed by Armijo (1966). Armijo's rule guarantees that the step size is not too large by requiring a sufficient decrease in the cost function. This so-called sufficient decrease condition, however, has the drawback that it may accept arbitrarily small step sizes that keep the algorithm from making reasonable progress towards the minimum. Wolfe (1969) and (1971) overcame this drawback by introducing a so-called curvature condition that guarantees a step size is not too small by requiring that the slope at the step size be greater than the slope at a step size of 0 by some factor. Nocedal and Wright (2006) modified Wolfe's curvature condition by requiring that the slope at the step size be negative. They then combined this modified curvature condition with Armijo's sufficient decrease condition to form what is known as the strong Wolfe conditions, which are used in the basic Armijo's rule line search algorithm presented in this section.

3.1.1 Flow Diagram

The overall flow of the basic Armijo's rule line search algorithm is shown in Figure 3-1 and involves: selecting an initial step size α_k, increasing α_k repeatedly by a factor η until the sufficient decrease condition is violated or decreasing α_k repeatedly by a factor η until either the sufficient decrease condition is satisfied or the modified curvature condition is violated, selecting the largest step size satisfying the strong Wolfe conditions.

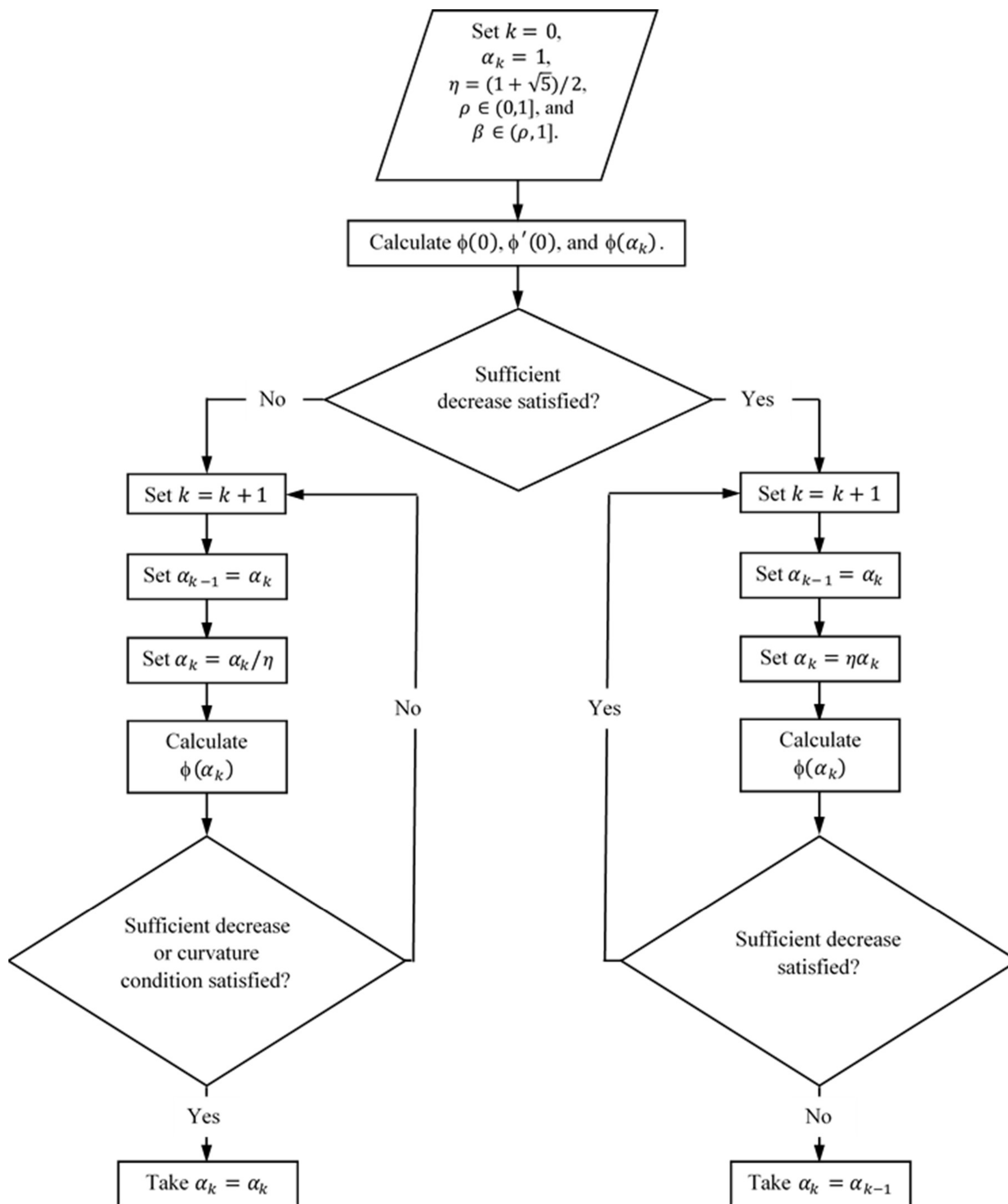

Figure 3.1 Basic Armijo’s Rule Line Search Flow Diagram.

3.1.2 Basic Algorithm

Here, the basic Armijo's rule line search algorithm is presented. It applies to the step size calculation sub-problem of finding α_k to

minimize $\phi(\alpha) = f(\mathbf{x} + \alpha\mathbf{d})$

Step 1: Set $k = 0$, $\alpha_k = 1$, $\rho \in (0,1]$, $\beta \in (\rho, 1]$, and $\eta = \left(1 + \sqrt{5}\right)/2$, where k is the line search iteration counter, α_k is the trial step size at the current iteration, ρ is a factor between 0 and 1, β is a factor between ρ and 1, and η is a factor greater than 1 used to decrease the current step size.

Step 2: Calculate the line search function and its inactive-set slope at the current design (i.e. $\phi(0)$ and $\phi'(0) = \boldsymbol{\nabla f} \cdot \mathbf{d}$, respectively).

Step 3: Calculate $\phi(\alpha_k)$ (i.e., the line search function at the trial design). Check if the sufficient decrease conditions (i.e. eqn 3.1) is satisfied. If satisfied, α_k is considered *not too large* and the step size is repeatedly increased by the factor η until the sufficient decrease condition is violated. If not satisfied, α_k is considered *too large* and the step size is repeatedly decreased by the factor η until the sufficient decrease condition or the curvature condition (i.e. eqn 3.2) is satisfied. Take the final step size α_k as the largest step size satisfying the strong Wolfe conditions (i.e. eqns 3.1 and 3.2).

$$\phi(\alpha_k) \leq \phi(0) + \alpha_k[\rho\, \phi'(0)] \;\; and \;\; [\phi(\alpha_k) < \phi(\alpha_{k-1}) \;\; if \;\; k > 0] \tag{3.1}$$

$$|\phi'(\alpha_k)| \leq \beta|\phi'(0)| \;\; and \;\; \phi'(\alpha_k) < 0 \tag{3.2}$$

3.1.3 Modified Step Size Procedure

In practice, it is common to apply some type of interpolation scheme to try to improve upon the step size identified by Armijo's rule, α_k. Here, a modified step size procedure is presented that uses a quadratic interpolation outer iteration to try to improve upon the step size. The overall procedure is shown in Figure 3-2 and involves: locating the interval of uncertainty using Armijo's rule, performing quadratic interpolation, selecting the appropriate step size. The algorithm is described in detail in section 3.1.3.1.

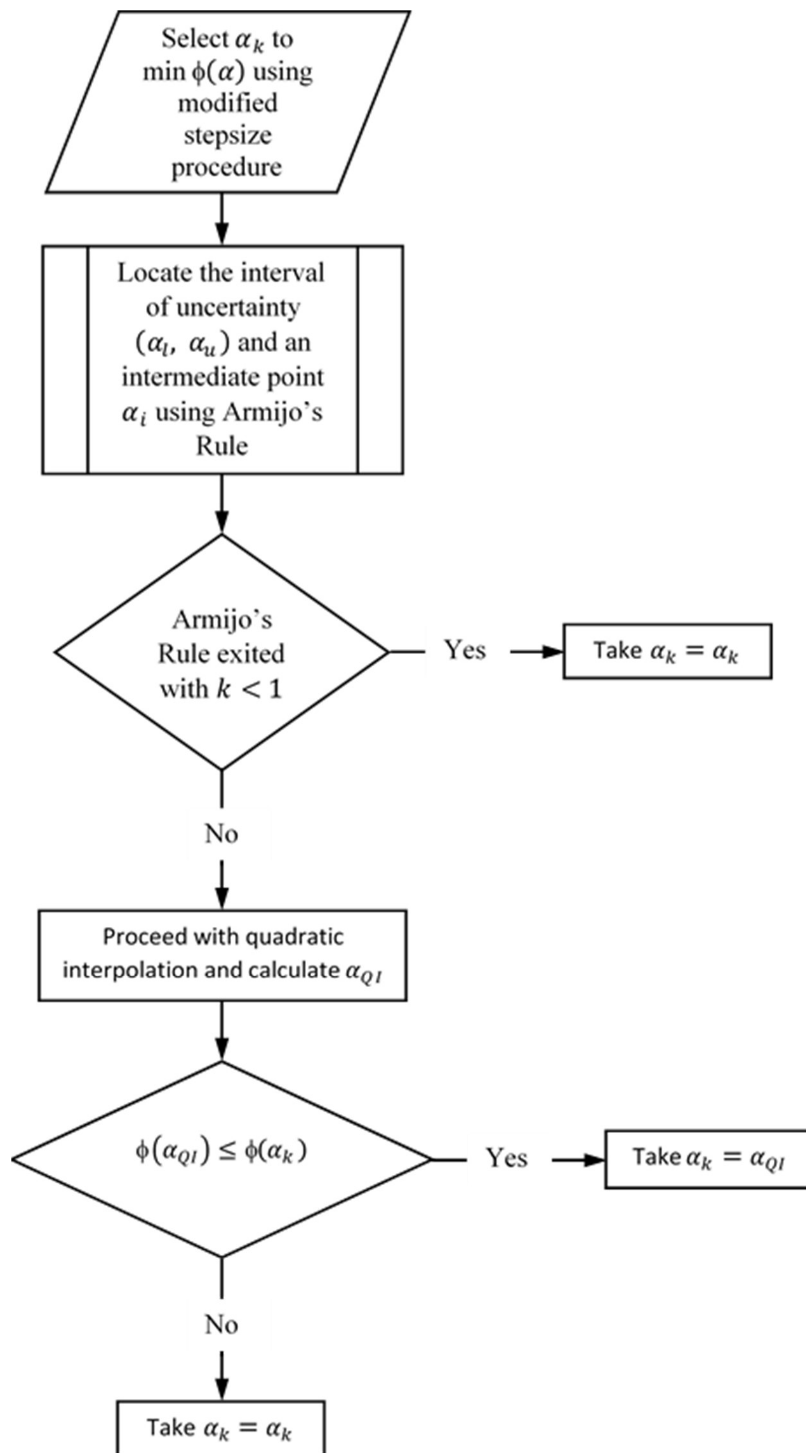

Figure 3.2 Modified Step Size Procedure Flow Diagram.

3.1.3.1 Modified Step Size Procedure Algorithm

The following modified step size procedure constructs a quadratic curve $q(\alpha)$ that approximates $\phi(\alpha)$, finds the minimum point of this curve $\bar{\alpha}$, identifies the quadratic interpolation step size α_{QI}, and selects α_k to

minimize $\phi(\alpha) = f(\mathbf{x} + \alpha\mathbf{d})$

Step 1: Locate the interval of uncertainty (α_l, α_u) and an intermediate point α_i using the Armijo's Rule described previously. Let $\phi(\alpha_l)$, $\phi(\alpha_i)$, and $\phi(\alpha_u)$ be the values of $\phi(\alpha)$ at α_l, α_i, and α_u, respectively. Note that if Armijo's Rule exited before completing a full iteration (i.e. before locating the interval of uncertainty), then quadratic interpolation is skipped and the step size is taken as the one obtained by Armijo's Rule.

Step 2: Calculate the coefficients a_0, a_1, and a_2 in the quadratic function $q(\alpha) = a_0 + a_1\alpha + a_2\alpha^2$ as follows. Also, solve the necessary condition $dq/d\alpha = 0$ to get $\bar{\alpha}$, verify the sufficiency condition $d^2q/d\alpha^2 > 0$, and evaluate $\phi(\bar{\alpha})$.

$$a_2 = \frac{1}{(\alpha_u - \alpha_i)}\left[\frac{\phi(\alpha_u) - \phi(\alpha_l)}{(\alpha_u - \alpha_l)} - \frac{\phi(\alpha_i) - \phi(\alpha_l)}{(\alpha_i - \alpha_l)}\right] \tag{3.3}$$

$$a_1 = \frac{\phi(\alpha_i) - \phi(\alpha_l)}{(\alpha_i - \alpha_l)} - a_2(\alpha_l + \alpha_i) \tag{3.4}$$

$$a_0 = \phi(\alpha_l) - a_1\alpha_l - a_2\alpha_l^2 \tag{3.5}$$

$$\frac{dq}{d\alpha} = 0 \quad \Rightarrow \quad a_1 + 2a_2\alpha = 0 \quad \Rightarrow \quad \bar{\alpha} = -\frac{a_1}{2a_2}; \quad if \quad \frac{d^2q}{d\alpha^2} = 2a_2 > 0 \tag{3.6}$$

Step 3a: If $\alpha_i < \bar{\alpha}$, continue with this step. Otherwise, go to Step 3b. If $\phi(\alpha_i) < \phi(\bar{\alpha})$, then the new limits of the reduced interval of uncertainty are set to $\alpha_l' = \alpha_l$, $\alpha_i' = \alpha_i$, and $\alpha_u' = \bar{\alpha}$ since $\alpha_l \leq \alpha_k \leq \bar{\alpha}$. Otherwise, set the new limits of the reduced interval of uncertainty to $\alpha_l' = \alpha_i$, $\alpha_i' = \bar{\alpha}$, and $\alpha_u' = \alpha_u$ since $\alpha_i \leq \alpha_k \leq \alpha_u$. Go to Step 4.

Step 3b: If $\alpha_i \geq \bar{\alpha}$, continue with this step. Otherwise, go to Step 4. If $\phi(\alpha_i) < \phi(\bar{\alpha})$, then the new limits of the reduced interval of uncertainty are set to $\alpha_l' = \bar{\alpha}$, $\alpha_i' = \alpha_i$, and $\alpha_u' = \alpha_u$ since $\bar{\alpha} \leq \alpha_k \leq \alpha_u$. Otherwise, set the new limits of the reduced interval of uncertainty to $\alpha_l' = \alpha_l$, $\alpha_i' = \bar{\alpha}$, and $\alpha_u' = \alpha_i$ since $\alpha_l \leq \alpha_k \leq \alpha_i$.

Step 4: Take $\alpha_{QI} = \alpha_i'$.

Step 5: If $\phi(\alpha_{QI}) < \phi(\alpha_k)$, then take $\alpha_k = \alpha_{QI}$ and stop. Otherwise, take α_k as the step size and stop.

3.1.4 Armijo's Rule Numerical Results

Table 3.1 compares the performance of the standard Armijo's Rule (SAR) and the modified step size procedure (SAR+QI), which completes a quadratic interpolation iteration after the standard Armijo's Rule has run, on fifteen of the unconstrained test problems from the Hock-Schittkowski collection. As expected, the more accurate step size obtained by the modified procedure allows for convergence in fewer L-BFGS iterations and enables convergence of poorly scaled problems that require a more accurate step. While all problems converged in fewer L-BFGS iterations, only half of the problems converged with fewer calls to the objective function. In other words, the improved convergence of the modified procedure was not always enough to offset the additional function evaluation incurred by the quadratic interpolation iteration. Studying the detailed output from these problems, however, showed that the majority of them were cases where the L-BFGS search direction was well scaled and all the quadratic interpolation iteration was doing was calculating a step size very close to the expected value of one. This motivates the backtracking approach presented in the next section where the step size is only allowed to decrease from its expected value of one.

Table 3.1 Standard Armijo's Rule and Modified Step Size Procedure Performance.

No.	Problem	n	SAR L-BFGS Iter (F)	SAR+QI L-BFGS Iter (F)
1	Helical valley function	3	-	-
2	Biggs EXP6 function	6	30 (114)	28 (132)
3	Gaussian function	3	5 (22)	3 (21)
4	Powell badly scaled function	2	-	133 (5829)
5	Box three-dimensional function	3	20 (93)	19 (108)
6	Variably dimensioned function	6	8 (74)	4 (47)
7	Watson function	9	70 (295)	39 (261)
8	Penalty function I	8	44 (151)	26 (162)
9	Penalty function II	3	47 (165)	38 (174)
10	Brown and Dennis function	4	26 (444)	15 (193)
11	Trigonometric function	20	43 (114)	41 (155)
12	Extended Rosenbrock function	14	77 (197)	70 (247)
13	Extended Powell singular function	16	43 (182)	32 (163)
14	Beale function	2	14 (43)	11 (48)
15	Wood function	4	13 (61)	9 (62)

3.2 Efficient Backtracking Algorithm

A different approach to the basic Armijo's rule line search presented in section 3.1 is what is known as a backtracking algorithm. In general, a backtracking algorithm starts with a larger step size and repeatedly decreases the step size until either the sufficient decrease condition or the modified curvature condition is satisfied. Backtracking algorithms are typically used in Newton or quasi-Newton unconstrained algorithms. Since these methods create a quadratic model with exact minimizer one, a starting step size of one is expected to be optimal for the majority of iterations. Here, we present a novel efficient backtracking algorithm that makes use of standard backtracking, quadratic interpolation, cubic interpolation, and quadratic regression wherever possible without incurring additional function evaluations. Figure 3-3 shows the overall flow of

the algorithm and sections 3.2.1 through 3.2.4 describe the specific backtracking techniques.

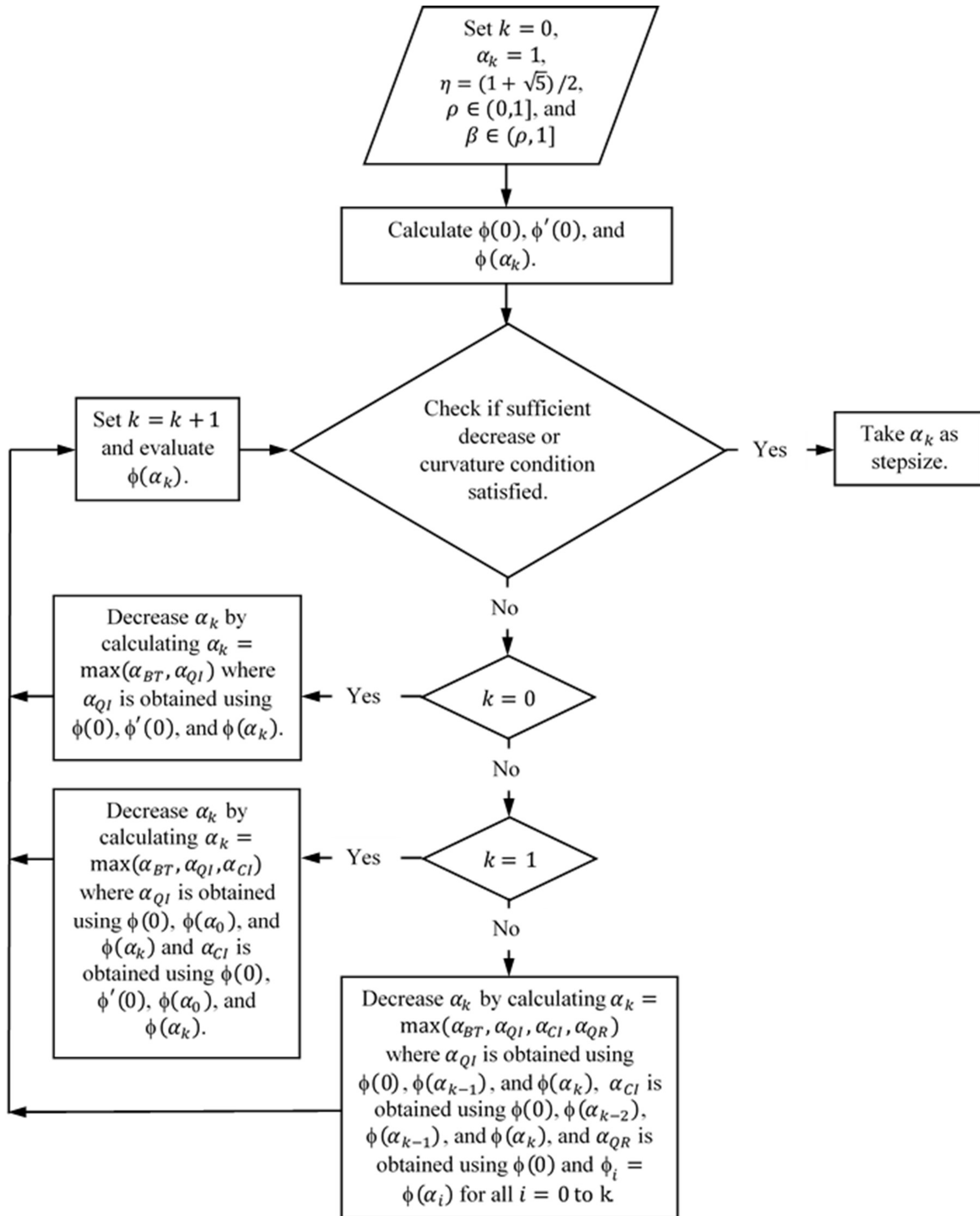

Figure 3.3 Efficient Backtracking Step Size Algorithm Flow Diagram.

3.2.1 Standard Backtracking

Standard backtracking to get α_{BT} is performed as follows:

$$\alpha_{BT} = \frac{\alpha_k}{\eta^2} \tag{3.7}$$

3.2.2 Quadratic Interpolation

3.2.2.1 Quadratic Interpolation Algorithm 1

Quadratic interpolation using $\phi(0)$, $\phi'(0)$, and $\phi(\alpha_k)$ to get α_{QI} is performed by: calculating the coefficients a_0, a_1, and a_2 of the quadratic function $q(\alpha) = a_0 + a_1\alpha + a_2\alpha^2$, solving the necessary condition $dq/d\alpha = 0$ to get α_{QI}, verifying the sufficiency condition $d^2q/d\alpha^2 > 0$.

$$a_0 = \phi(0) \tag{3.8}$$

$$a_1 = \phi'(0) \tag{3.9}$$

$$a_2 = \frac{\phi(\alpha_k) - a_0 - a_1\alpha_k}{a_k^2} \tag{3.10}$$

$$\frac{dq}{d\alpha} = 0 \quad \Rightarrow \quad a_1 + 2a_2\alpha = 0 \quad \Rightarrow \quad \alpha_{QI} = -\frac{a_1}{2a_2} \tag{3.11}$$

$$if \quad \frac{d^2q}{d\alpha^2} = 2a_2 > 0 \quad and \quad 0 < \alpha_{QI} < \alpha_k \quad then\ keep \quad \alpha_{QI} \quad else\ set \quad \alpha_{QI} = 0$$

3.2.2.2 Quadratic Interpolation Algorithm 2

Quadratic interpolation using $\phi(0)$, $\phi(\alpha_{k-1})$, and $\phi(\alpha_k)$ to get α_{QI} is performed by: calculating the coefficients a_0, a_1, and a_2 of the quadratic function $q(\alpha) = a_0 + a_1\alpha + a_2\alpha^2$, solving the necessary condition $dq/d\alpha = 0$ to get α_{QI}, verifying the sufficiency condition $d^2q/d\alpha^2 > 0$.

$$a_2 = \frac{1}{(\alpha_{k-1} - \alpha_k)} \left[\frac{\phi(\alpha_{k-1}) - \phi(0)}{(\alpha_{k-1})} - \frac{\phi(\alpha_k) - \phi(0)}{(\alpha_k)} \right] \tag{3.12}$$

$$a_1 = \frac{\phi(\alpha_k) - \phi(0)}{(\alpha_k)} - a_2(\alpha_k) \tag{3.13}$$

$$a_0 = \phi(0) \tag{3.14}$$

$$\frac{dq}{d\alpha} = 0 \quad \Rightarrow \quad a_1 + 2a_2\alpha = 0 \quad \Rightarrow \quad \alpha_{QI} = -\frac{a_1}{2a_2} \tag{3.15}$$

$$if \quad \frac{d^2q}{d\alpha^2} = 2a_2 > 0 \quad and \quad 0 < \alpha_{QI} < \alpha_k \quad then\ keep \quad \alpha_{QI} \quad else\ set \quad \alpha_{QI} = 0$$

3.2.3 Cubic Interpolation

3.2.3.1 Cubic Interpolation Algorithm 1

Cubic interpolation using $\phi(0)$, $\phi'(0)$, $\phi(\alpha_0)$, and $\phi(\alpha_k)$ to get α_{CI} is performed by: calculating the coefficients a_0, a_1, a_2, and a_3 of the cubic function $c(\alpha) = a_0 + a_1\alpha + a_2\alpha^2 + a_3\alpha^3$, solving the necessary condition $dc/d\alpha = 0$ to get α_{CI}, verifying the sufficiency condition $d^2c/d\alpha^2 > 0$.

$$a_0 = \phi(0) \tag{3.16}$$

$$a_1 = \phi'(0) \tag{3.17}$$

$$a_2 = \frac{\alpha_0\alpha_k^3\, a_1 + \alpha_k^3(\, a_0 - \phi(\alpha_0)) - \alpha_0^3(\, a_0 + \alpha_k\, a_1 - \phi(\alpha_k))}{\alpha_0^2(\alpha_0 - \alpha_k)\alpha_k^2} \tag{3.18}$$

$$a_3 = \frac{-\alpha_0\alpha_k^2\, a_1 + \alpha_k^2(-\, a_0 + \phi(\alpha_0)) + \alpha_0^2(\, a_0 + \alpha_k\, a_1 - \phi(\alpha_k))}{\alpha_0^2(\alpha_0 - \alpha_k)\alpha_k^2} \tag{3.19}$$

$$\frac{dc}{d\alpha} = 0 \quad \Rightarrow \quad a_1 + 2a_2\alpha + 3a_3\alpha^2 = 0 \quad \Rightarrow \quad \alpha_{CI1,CI} \quad = \frac{-a_2 \pm \sqrt{a_2^2 - 3a_1a_3}}{3a_3} \tag{3.20}$$

$$\frac{d^2c}{d\alpha^2} > 0 \quad \Rightarrow \quad 2a_2 + 6a_3\alpha > 0 \tag{3.21}$$

$$if \quad a_2^2 - 3a_1a_3 \geq 0 \quad and \quad 0 < \alpha_{CI1} < \alpha_k \quad and \quad 2a_2 + 6a_3\alpha_{CI1} > 0 \quad then\ keep \quad \alpha_{CI1}$$
$$else\ set \quad \alpha_{CI1} = 0$$

$$if \quad a_2^2 - 3a_1a_3 \geq 0 \quad and \quad 0 < \alpha_{CI2} < \alpha_k \quad and \quad 2a_2 + 6a_3\alpha_{CI2} > 0 \quad then\ keep \quad \alpha_{CI}$$
$$else\ set \quad \alpha_{CI2} = 0$$

$$\alpha_{CI} = \max(\alpha_{CI1}, \alpha_{CI}\ ) \tag{3.22}$$

3.2.3.2 Cubic Interpolation Algorithm 2

Cubic interpolation using $\phi(0)$, $\phi(\alpha_{k-2})$, $\phi(\alpha_{k-1})$, and $\phi(\alpha_k)$ to get α_{CI} is performed by: solving four equations for the four unknown coefficients a_0, a_1, a_2, and a_3 of the cubic function $c(\alpha) = a_0 + a_1\alpha + a_2\alpha^2 + a_3\alpha^3$, solving the necessary condition $dc/d\alpha = 0$ to get α_{CI}, verifying the sufficiency condition $d^2c/d\alpha^2 > 0$.

$$\phi(0) = a_0 \tag{3.23}$$

$$\phi(\alpha_{k-2}) = a_0 + a_1\alpha_{k-2} + a_2{\alpha_{k-2}}^2 + a_3{\alpha_{k-2}}^3 \tag{3.24}$$

$$\phi(\alpha_{k-1}) = a_0 + a_1\alpha_{k-1} + a_2{\alpha_{k-1}}^2 + a_3{\alpha_{k-1}}^3 \tag{3.25}$$

$$\phi(\alpha_k) = a_0 + a_1\alpha_k + a_2{\alpha_k}^2 + a_3{\alpha_k}^3 \tag{3.26}$$

$$\frac{dc}{d\alpha} = 0 \quad \Rightarrow \quad a_1 + 2a_2\alpha + 3a_3\alpha^2 = 0 \quad \Rightarrow \quad \alpha_{CI1,CI} = \frac{-a_2 \pm \sqrt{a_2^2 - 3a_1a_3}}{3a_3} \tag{3.27}$$

$$\frac{d^2c}{d\alpha^2} > 0 \quad \Rightarrow \quad 2a_2 + 6a_3\alpha > 0 \tag{3.28}$$

$$if \quad a_2^2 - 3a_1a_3 \geq 0 \quad and \quad 0 < \alpha_{CI1} < \alpha_k \quad and \quad 2a_2 + 6a_3\alpha_{CI1} > 0 \quad then\ keep \quad \alpha_{CI}$$

$$else\ set \quad \alpha_{CI1} = 0$$

$$if \quad a_2^2 - 3a_1a_3 \geq 0 \quad and \quad 0 < \alpha_{CI2} < \alpha_k \quad and \quad 2a_2 + 6a_3\alpha_{CI2} > 0 \quad then\ keep \quad \alpha_{CI}$$

$$else\ set \quad \alpha_{CI2} = 0$$

$$\alpha_{CI} = \max(\alpha_{CI1}, \alpha_{CI2}) \tag{3.29}$$

3.2.4 Quadratic Regression

Quadratic regression using $\phi(0)$ and $\phi_i = \phi(\alpha_i)$ for all $i = 0$ to k to get α_{QR} is performed by: solving a linear system for the coefficients a_0, a_1, and a_2 of the quadratic function $q(\alpha) = a_0 + a_1\alpha + a_2\alpha^2$, solving the necessary condition $dq/d\alpha = 0$ to get α_{QR}, verifying the sufficiency condition $d^2q/d\alpha^2 > 0$.

$$l = 1 + k \tag{3.30}$$

$$\begin{bmatrix} l & \sum_{j=0}^{l} \alpha_j & \sum_{j=0}^{l} \alpha_j^2 \\ \sum_{j=0}^{l} \alpha_j & \sum_{j=0}^{l} \alpha_j^2 & \sum_{j=0}^{l} \alpha_j^3 \\ \sum_{j=0}^{l} \alpha_j^2 & \sum_{j=0}^{l} \alpha_j^3 & \sum_{j=0}^{l} \alpha_j^4 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \end{bmatrix} = \begin{bmatrix} \sum_{j=0}^{l} \phi_j \\ \sum_{j=0}^{l} \alpha_j \phi_j \\ \sum_{j=0}^{l} \alpha_j^2 \phi_j \end{bmatrix} \quad (3.31)$$

$$\frac{dq}{d\alpha} = 0 \quad \Rightarrow \quad a_1 + 2a_2\alpha = 0 \quad \Rightarrow \quad \alpha_{QR} = -\frac{a_1}{2a_2} \quad (3.32)$$

$$if \quad \frac{d^2q}{d\alpha^2} = 2a_2 > 0 \quad and \quad 0 < \alpha_{QR} < \alpha_k \quad then\ keep \quad \alpha_{QR} \quad else\ set \quad \alpha_{QR} = 0$$

3.2.5 Efficient Backtracking Armijo's Rule Numerical Results

Table 3.2 repeats the results of the previous section and compares performance to the modified step size procedure with backtracking only, which we refer to here as standard backtracking (SBT+QI), and the new efficient backtracking procedure (EBT+QI) just presented. Both backtracking procedures take more iterations to converge, but require significantly fewer function evaluations than the standard and modified Armijo's Rule. The new efficient backtracking procedure outperforms the standard backtracking procedure, in terms of function evaluations, roughly three-quarters of the time by a good margin and barely underperforms the standard procedure the other quarter of the time.

Table 3.2 Standard Backtracking and Efficient Backtracking Performance.

No.	Problem	n	SAR L-BFGS Iter (F)	SAR+QI L-BFGS Iter (F)	SBT+QI L-BFGS Iter (F)	EBT+QI L-BFGS Iter (F)
1	Helical valley function	3	-	-	-	50 (143)
2	Biggs EXP6 function	6	30 (114)	28 (132)	41 (54)	40 (45)
3	Gaussian function	3	5 (22)	3 (21)	4 (10)	6 (8)
4	Powell badly scaled function	2	-	133 (5829)	323 (5174)	91 (909)
5	Box three-dimensional function	3	20 (93)	19 (108)	30 (57)	37 (46)
6	Variably dimensioned function	6	8 (74)	4 (47)	6 (44)	8 (30)
7	Watson function	9	70 (295)	39 (261)	79 (103)	56 (66)
8	Penalty function I	8	44 (151)	26 (162)	51 (82)	66 (85)
9	Penalty function II	3	47 (165)	38 (174)	16 (24)	15 (19)
10	Brown and Dennis function	4	26 (444)	15 (193)	15 (165)	20 (205)
11	Trigonometric function	20	43 (114)	41 (155)	49 (58)	48 (59)
12	Extended Rosenbrock function	14	77 (197)	70 (247)	90 (126)	98 (131)
13	Extended Powell singular function	16	43 (182)	32 (163)	70 (86)	49 (51)
14	Beale function	2	14 (43)	11 (48)	15 (26)	16 (18)
15	Wood function	4	13 (61)	9 (62)	22 (39)	23 (26)

3.3 Incorporation of a Trust Region

As we have seen, line search based methods first calculate a search direction and then determine a step to take in that direction using an iterative scheme. Alternatively, trust region based methods place an upper bound on the step, based on the performance of previous steps, and then solve a sub-problem to ascertain a search direction / step combination or trial step. Trust region based methods are very efficient in terms of function evaluations since they typically only require a single function evaluation per unconstrained iteration. However, when a trust region trial step is rejected, the trust region sub-problem must be repeatedly solved until a trial step is accepted, which can become computationally expensive. Line search based methods explicitly calculate a search direction, making them computationally efficient, but are often inefficient in terms

of function evaluations since they typically require multiple iterations to identify an acceptable step if the initial step is rejected.

3.3.1 Literature Review

A good deal of research has been done to study how line search and trust region concepts can be combined to develop more efficient step size calculation methods. Most work has centered on incorporating line search concepts into trust region methods. For instance, Nocedal and Yuan (1998) suggested that a backtracking line search be performed, in lieu of re-solving the sub-problem, when a trust region trial step resulted in an increase in the objective function. In their method, backtracking is performed from the failed point along the direction of the failed trial step using a truncated quadratic interpolation scheme. Numerical results for this approach were promising with the new combined technique outperforming the pure line search and pure trust region techniques. Similarly, Yu, Wang, and Yu (2004) proposed an algorithm that switches to a line search when the trust region trial step is rejected. Their approach differed from that of Nocedal and Yuan in that they focused on the specific case of equality constrained problems, added a correction step to the trust region step instead of performing standard backtracking, and employed a nonmonotone line search technique. Likewise, Yuan, Meng, and Wei (2009) as well as Ou (2011) proposed trust region based methods that use backtracking line searches when the trial step is unsuccessful.

All of the approaches discussed thus far use the ratio of actual to predicted reduction in the objective function, along with some conditional logic, to expand or contract the trust region radius between unconstrained iterations. In general, the trust region radius is: expanded if the ratio is close to 1 (i.e. agreement between actual and predicted reduction is good) and the accepted step was taken as the full trust region radius in the previous iteration, contracted if the ratio is close to 0 (i.e. agreement between actual and predicted reduction is bad), left the same if the ratio is roughly midway

between 0 and 1 (i.e. agreement is neither good nor bad). This standard approach to updating the trust region radius can be improved upon by using so called self-adaptive trust region methods that use problem information at each iteration to automatically adjust the trust region radius. Quite a few researchers have proposed such self-adaptive methods including Zhang, Zhang, and Liao (2002), Hei (2003), Fu and Sun (2005), Shi and Guo (2008), Sang and Sun (2011), Cui and Wu (2011), and most recently Liu (2013). Liu (2013) still uses the ratio of actual to predicted reduction. However, the actual update of the trust region radius is completed using just about all of the available problem information: the ratio of the actual to predicted reduction at the current and previous iteration, the Hessian approximation at the current and next iteration, the search direction at the current iteration, the gradient at the current iteration. Numerical results show that Liu's approach outperforms both the standard trust region updating scheme and Zhang, Zhang, and Liao's (2002) self-adaptive scheme both in terms of total iterations and function evaluations.

3.3.2 A Novel Hybrid Line Search / Trust Region Approach

Here, a hybrid line search / trust region approach is presented. Unlike previous hybrid approaches, which incorporated line search concepts into trust region algorithms, the approach presented here incorporates trust region concepts into a line search algorithm. The overall procedure is shown in Figure 3-4 and involves: setting the initial step size to Liu's self-adaptive trust region radius for the next bound constrained iteration, checking the strong Wolfe conditions and, if not satisfied, performing a backtracking line search. Liu's self-adaptive method for updating the trust region radius for the next bound constrained iteration is described in detail in section 3.3.2.1.

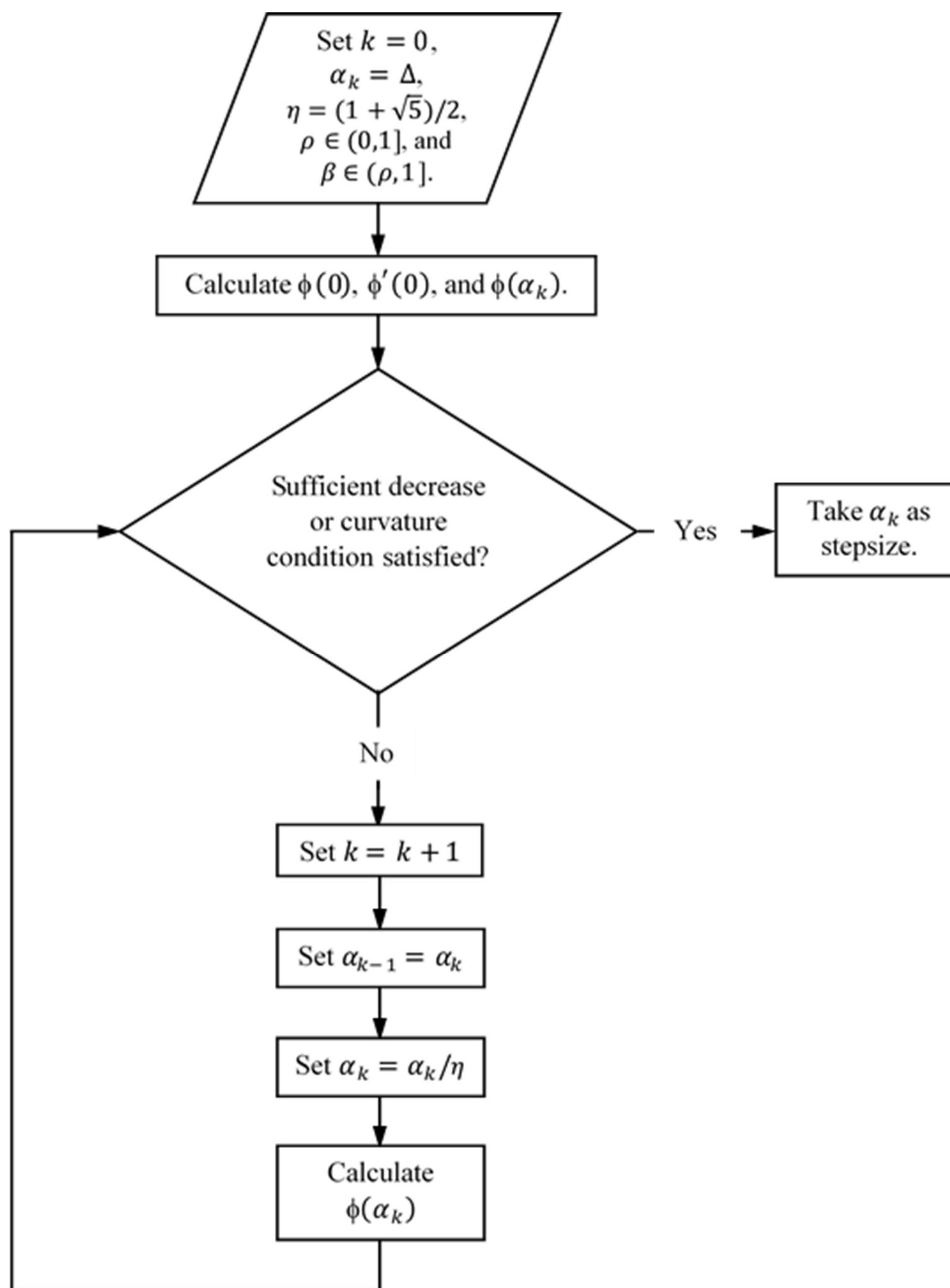

Figure 3.4 Hybrid Line Search / Trust Region Flow Diagram.

3.3.2.1 Liu's Rule for Updating the Trust Region Radius

The following rule may be used to update the trust region radius for the next bound constrained iteration: calculate the change in design, calculate the ratio of the actual to predicted reduction to assess how well the quadratic model is representing the underlying function, implement Liu's self-adaptive rule.

$$\Delta\mathbf{x} = \mathbf{x}^{(k+1)} - \mathbf{x}^{(k)} \tag{3.33}$$

$$m_k(\mathbf{0}) = \Phi\left(\mathbf{x}^{(k)}\right) \tag{3.34}$$

$$m_k(\Delta\mathbf{x}) = \Phi\left(\mathbf{x}^{(k)}\right) + \left(\nabla\Phi^{(k)}\right)^T \Delta\mathbf{x} + \frac{1}{2}\Delta\mathbf{x}^T\mathbf{B}^{(k)}\Delta\mathbf{x} \tag{3.35}$$

$$r_k = \frac{\Phi\left(\mathbf{x}^{(k)}\right) - \Phi\left(\mathbf{x}^{(k+1)}\right)}{m_k(\mathbf{0}) - m_k(\Delta\mathbf{x})} \quad \text{(i.e. ratio of actual to predicted reduction)} \tag{3.36}$$

$$\mathbf{B}^{(k+1)} = \begin{cases} \mathbf{B}^{(k)} & if\ r_k < 0.1 \\ \mathbf{B}^{(k+1)} & otherwise \end{cases} \tag{3.37}$$

$$\hat{r}_k = \begin{cases} r_k & k = 0 \\ 0.9r_k + (1 - 0.9)r_{k-1} & k \geq 0 \end{cases} \tag{3.38}$$

$$\mu_{k+1} = \begin{cases} 2\mu_k & if\ \hat{r}_k > 0.9 \\ 0.5\mu_k & if\ \hat{r}_k < 0.1 \\ \mu_k & otherwise \end{cases} \tag{3.39}$$

$$\Delta_{k+1} = \min\left\{ \mu_{k+1} \frac{\|\boldsymbol{d}_k\|^2}{\boldsymbol{d}_k^T\left[0.9\mathbf{B}^{(k+1)} + (1-0.9)\mathbf{B}^{(k)}\right]\boldsymbol{d}_k} \left\|\nabla\Phi^{(k+1)}\right\|, 1\right\} \tag{3.40}$$

where $\mathbf{B}^{(k)}$ is the Hessian approximation corresponding to the current search direction technique. For example, $\mathbf{B}^{(k)}$ would be a zero matrix when the steepest descent direction is taken as the search direction and would be the L-BFGS approximation of the Hessian if the L-BFGS direction was taken as the search direction. In order to avoid storing the full L-BFGS approximation of the Hessian, the product $\Delta\mathbf{x}^T\mathbf{B}^{(k)}\Delta\mathbf{x}$ may be calculated directly as follows:

$$B_0^{(k)} = \frac{1}{H_0^{(k)}} \tag{3.41}$$

for $i = \text{LIMIT} : +1 : i \leq k - 1$

$$p(i) = i - m \times int\left(\frac{i}{m}\right) \tag{3.42}$$

$$\mathbf{b}_p = \frac{\mathbf{y}^{(\boldsymbol{p})}}{\left(\mathbf{y}^{(\boldsymbol{p})} \cdot \mathbf{s}^{(\boldsymbol{p})}\right)^{1/2}} \tag{3.43}$$

$$\mathbf{a}_p = B_0^{(k)} \mathbf{s}^{(\boldsymbol{p})} \tag{3.44}$$

for $j = \text{LIMIT} : +1 : j < i$

$$q(j) = j - m \times int\left(\frac{j}{m}\right) \tag{3.45}$$

$$\mathbf{a}_p = \mathbf{a}_p + \left[\left(\mathbf{b}_q \cdot \mathbf{s}^{(\boldsymbol{p})}\right)\mathbf{b}_q - \left(\mathbf{a}_q \cdot \mathbf{s}^{(\boldsymbol{p})}\right)\mathbf{a}_q\right] \tag{3.46}$$

end

$$\mathbf{a}_p = \frac{\mathbf{a}_p}{\left(\mathbf{s}^{(\boldsymbol{p})} \cdot \mathbf{a}_p\right)^{1/2}} \tag{3.47}$$

end

$$\Delta\mathbf{x}^T \mathbf{B}^{(k)} \Delta\mathbf{x} = \Delta\mathbf{x}^T B_0^{(k)} \mathbf{I}\, \Delta\mathbf{x} + \sum_{i=0}^{m} [\Delta\mathbf{x}^T \mathbf{b}_i \mathbf{b}_i^T \Delta\mathbf{x} - \Delta\mathbf{x}^T \mathbf{a}_i \mathbf{a}_i^T \Delta\mathbf{x}] \tag{3.48}$$

where m is the total number of corrections being stored.

3.3.3 Hybrid Line Search / Trust Region Numerical Results

Table 3.3 repeats the results of the previous section for the efficient backtracking procedure, presents new results incorporating the hybrid line search / trust region approach just described, and compares performance of the resulting L-BFGS algorithm (H+EBT+QI) with that of the BFGS hybrid trust region / line search algorithm (TR+BT) presented by Nocedal and Yuan (1998). The new hybrid line search / trust region approach outperformed the efficient backtracking procedure by a significant margin on two problems and by a lesser margin on one problem. It underperformed the efficient backtracking procedure by a significant margin on one problem and by a lesser margin on two problems. The balance of the problems had no change in performance. The problems that the hybrid method yielded the greatest improvement on were those where the L-BFGS algorithm consistently failed to produce a search direction scaled for a step

size of one and backtracking had to be performed. Conversely, the problems that the hybrid method yielded worse performance on were those where the L-BFGS algorithm switched back and forth frequently between producing a well scaled and poorly scaled search direction. Overall, the new hybrid line search / trust region algorithm performed comparably to Nocedal and Yuan's hybrid trust region / line search algorithm despite their use of a full BFGS approximation to the inverse Hessian.

Table 3.3 Efficient Backtracking and Hybrid Method Performance.

			EBT+QI L-BFGS	H+ EBT+QI L-BFGS	TR+BT BFGS
No.	Problem	n	Iter (F)	Iter (F)	Iter (F)
1	Helical valley function	3	50 (143)	50 (143)	24 (26)
2	Biggs EXP6 function	6	40 (45)	40 (45)	35 (36)
3	Gaussian function	3	6 (8)	6 (8)	5 (6)
4	Powell badly scaled function	2	91 (909)	197 (1863)	175 (212)
5	Box three-dimensional function	3	37 (46)	37 (46)	30 (31)
6	Variably dimensioned function	6	8 (30)	9 (31)	17 (17)
7	Watson function	9	56 (66)	56 (66)	66 (70)
8	Penalty function I	8	66 (85)	43 (48)	70 (82)
9	Penalty function II	3	15 (19)	15 (19)	12 (13)
10	Brown and Dennis function	4	20 (205)	22 (157)	24 (31)
11	Trigonometric function	20	48 (59)	47 (57)	46 (51)
12	Extended Rosenbrock function	14	98 (131)	102 (132)	112 (138)
13	Extended Powell singular function	16	49 (51)	49 (51)	76 (87)
14	Beale function	2	16 (18)	16 (18)	16 (16)
15	Wood function	4	23 (26)	23 (26)	67 (79)

CHAPTER IV

DESIGN OF ALGORITHM FOR CONSTRAINED OPTIMIZATION PROBLEMS

4.1 Augmented Lagrangian

Augmented Lagrangian or multiplier methods solve constrained optimization problems by solving a sequence of essentially unconstrained problems. Specifically, the methods form an augmented Lagrangian function, which adds a penalty term to the traditional Lagrangian for each violated constraint, and minimize this function to obtain new estimates of the design variables. Then, the Lagrange multipliers and penalty parameter are updated, as needed, creating a new augmented Lagrangian function to be minimized. This process is repeated until some optimality and feasibility convergence criteria are reached.

Augmented Lagrangian methods generally offer the following benefits: (1) good conditioning / numerical stability, since they do not require the penalty parameter to go to infinity; (2) converge to a local minimum from any starting point, i.e. globally convergent; (3) converge faster than penalty or barrier function methods; (4) remove inactive constraints from the problem automatically; (5) perform *updates* to the Lagrange multiplier variables, instead of solving a computationally expensive linearly-constrained quadratic-programming sub-problem at every iteration; (6) efficiently handle high-dimensional problems with many simple bound constraints; (7) convergence towards optimality matches that of the underlying bound-constrained algorithm, which can be quite strong. Augmented Lagrangian methods do, however, have a few notable drawbacks: (1) solution of an unconstrained sub-problem at every iteration, before updating the Lagrange multiplier variables, may result in slow progress towards feasibility; (2) using traditional Lagrange multiplier updating limits convergence towards feasibility to being linear; (3) selecting too small of a penalty parameter may result in

non-convexity; (4) selecting too large of a penalty parameter may result in slow progress towards optimality or even numerical instability. In section 4.1.5, these challenges are studied in more depth and strategies to overcome them are identified. However, first, a closer look is taken at the basic augmented Lagrangian algorithm.

4.1.1 Basic Algorithm

The following augmented Lagrangian algorithm applies to generally constrained optimization problems defined as:

minimize $f(\boldsymbol{x})$

subject to $l_i \leq c_i(\boldsymbol{x}) \leq u_i \quad i = 1, m$

Reformatting of Constraints: To simplify implementation, it is helpful to convert both lower and upper bound constraints to the form $\boldsymbol{h}(\boldsymbol{x}) = \boldsymbol{0}$ and $\boldsymbol{g}(\boldsymbol{x}) \leq \boldsymbol{0}$ and consider the constraint on each bound individually. This can be achieved, without incurring additional constraint function or gradient evaluations, by redefining each constraint and its gradient as follows:

$$h_i(\boldsymbol{x}) \equiv c_i(\boldsymbol{x}) - \tfrac{1}{2}(u_i + l_i) = 0; \quad \nabla h_i(\boldsymbol{x}) \equiv \nabla c_i(\boldsymbol{x}); \quad i \in E \tag{4.1}$$

$$g_{Ui}(\boldsymbol{x}) \equiv c_i(\boldsymbol{x}) - u_i \leq 0; \quad \nabla g_{Ui}(\boldsymbol{x}) \equiv \nabla c_i(\boldsymbol{x}); \quad i \in I \tag{4.2}$$

$$g_{Li}(\boldsymbol{x}) \equiv l_i - c_i(\boldsymbol{x}) \leq 0; \quad \nabla g_{Li}(\boldsymbol{x}) \equiv -\nabla c_i(\boldsymbol{x}); \quad i \in I \tag{4.3}$$

where $E = \{\, i \mid |u_i - l_i| \leq \delta;\ i = 1, m \,\}$ and $I = \{\, i \mid |u_i - l_i| > \delta;\ i = 1, m \,\}$

Step 1: Set $k = 0$, $K = 10^{20}$; estimate $\boldsymbol{x}^{(0)}$, $\boldsymbol{\mu}^{(0)}$, $\boldsymbol{\lambda}_{\boldsymbol{U}}^{(0)} \geq \boldsymbol{0}$, $\boldsymbol{\lambda}_{\boldsymbol{L}}^{(0)} \geq \boldsymbol{0}$, $r > 0$; set $\alpha > 1$, $\beta > 1$, $\delta > 0$, where k is the iteration counter, K is the maximum constraint violation parameter used in determining whether the constraint violation improved, $\boldsymbol{x}^{(0)}$ is the vector containing the design variable values at $k = 0$, $\boldsymbol{\mu}^{(0)}$ is the vector containing the equality constraint Lagrange multiplier values at $k = 0$, $\boldsymbol{\lambda}_{\boldsymbol{U}}^{(0)}$ is the vector containing the upper bound inequality constraint Lagrange multiplier values at $k = 0$, $\boldsymbol{\lambda}_{\boldsymbol{L}}^{(0)}$ is the vector containing the lower bound inequality constraint Lagrange multiplier values at $k = 0$, r is

the penalty parameter, α is the factor used to test if the constraint violation has improved sufficiently, β is the factor used to increase the current penalty parameter value, δ is the constraint violation tolerance.

Step 2: Set $k = k + 1$

Step 3: Obtain $\boldsymbol{x}^{(k)}$ by performing bound-constrained minimization on the augmented Lagrangian function, Φ, with gradient, ∇Φ, given in equations 4.4 and 4.5, respectively, keeping the current r, $\boldsymbol{\mu}^{(k)}$, $\boldsymbol{\lambda}_{\boldsymbol{U}}^{(k)}$, and $\boldsymbol{\lambda}_{\boldsymbol{L}}^{(k)}$ fixed:

$$\begin{aligned}
&\Phi(\boldsymbol{x}, r, \boldsymbol{\mu}, \boldsymbol{\lambda}_{\boldsymbol{U}}, \boldsymbol{\lambda}_{\boldsymbol{L}}) \\
&= f(x) + \sum_{i\in E}\left[\mu_i h_i(x) + \tfrac{1}{2} r h_i^2(x)\right] \\
&+ \sum_{i\in I}\begin{cases} \lambda_{Ui} g_{Ui}(x) + \frac{1}{2} r g_{Ui}^2(x) & if\ g_{Ui} + \frac{\lambda_{Ui}}{r} \geq 0 \\ -\frac{1}{2r}\lambda_{Ui}^2 & if\ g_{Ui} + \frac{\lambda_{Ui}}{r} < 0 \end{cases} \\
&+ \sum_{i\in I}\begin{cases} \lambda_{Li} g_{Li}(x) + \frac{1}{2} r g_{Li}^2(x) & if\ g_{Li} + \frac{\lambda_{Li}}{r} \geq 0 \\ -\frac{1}{2r}\lambda_{Li}^2 & if\ g_{Li} + \frac{\lambda_{Li}}{r} < 0 \end{cases}
\end{aligned} \tag{4.4}$$

which has the gradient:

$$\begin{aligned}
&\nabla\Phi(\boldsymbol{x}, r, \boldsymbol{\mu}, \boldsymbol{\lambda}_{\boldsymbol{U}}, \boldsymbol{\lambda}_{\boldsymbol{L}}) \\
&= \nabla f(x) + \sum_{i\in E}\left[\mu_i \nabla h_i(x) + r h_i(x) \nabla h_i(x)\right] \\
&+ \sum_{i\in I}\begin{cases} \lambda_{Ui} \nabla g_{Ui}(x) + r g_{Ui}(x) \nabla g_{Ui}(x) & if\ g_{Ui} + \frac{\lambda_{Ui}}{r} \geq 0 \\ 0 & if\ g_{Ui} + \frac{\lambda_{Ui}}{r} < 0 \end{cases} \\
&+ \sum_{i\in I}\begin{cases} \lambda_{Li} \nabla g_{Li}(x) + r g_{Li}(x) \nabla g_{Li}(x) & if\ g_{Li} + \frac{\lambda_{Li}}{r} \geq 0 \\ 0 & if\ g_{Li} + \frac{\lambda_{Li}}{r} < 0 \end{cases}
\end{aligned} \tag{4.5}$$

Step 4: Using current constraint function values, $\boldsymbol{h}(\boldsymbol{x}^{(k)})$, $\boldsymbol{g}_{\boldsymbol{U}}(\boldsymbol{x}^{(k)})$, and $\boldsymbol{g}_{\boldsymbol{L}}(\boldsymbol{x}^{(k)})$, determine the current constraint violation parameter, $\bar{K}$, as follows:

$$\bar{K} = max\left\{|h_i|, i \in E;\ \left|max\left(g_{Ui}, -\lambda_{Ui}^{(k)}/r_k\right)\right|, i \in I;\ \left|max\left(g_{Li}, -\lambda_{Li}^{(k)}/r_k\right)\right|, i \in I\right\} \tag{4.6}$$

Step 5: Check if constraint violation tolerance is satisfied (i.e. $\bar{K} \leq \delta$). If satisfied, stop. Otherwise, continue.

Step 6: Check if $\bar{K} \geq K$. If satisfied (i.e. the constraint violation has not improved), increase the penalty parameter by the factor β such that $r_{k+1} = \beta r_k$ and go to step 2. Otherwise, continue.

Step 7: Update the Lagrange multipliers as follows:

$$\mu_i^{(k+1)} = \mu_i^{(k)} + r_k h_i\left(x^{(k)}\right); \;\; i \in E \tag{4.7}$$

$$\lambda_{Ui}^{(k+1)} = \lambda_{Ui}^{(k)} + r_k \max\left[g_{Ui}\left(x^{(k)}\right), -\frac{\lambda_{Ui}^{(k)}}{r_k}\right]; \; i \in I \tag{4.8}$$

$$\lambda_{Li}^{(k+1)} = \lambda_{Li}^{(k)} + r_k \max\left[g_{Li}\left(x^{(k)}\right), -\frac{\lambda_{Li}^{(k)}}{r_k}\right]; \; i \in I \tag{4.9}$$

Before continuing, ensure that if $g_{Ui}\left(\boldsymbol{x}^{(k)}\right) \geq 0$ then $\lambda_{Li}^{(k+1)} = 0$ and if $g_{Li}\left(\boldsymbol{x}^{(k)}\right) \geq 0$ then $\lambda_{Ui}^{(k+1)} = 0$.

Step 8: Check if $\bar{K} \leq K/\alpha$. If satisfied (i.e. the constraint violation has improved by the factor α), set $K = \bar{K}$ and go to step 2. Otherwise, continue.

Step 9: Set $r_{k+1} = \beta r_k$ and $K = \bar{K}$ and go to step 2.

4.1.2 Flow Diagram for Basic Algorithm

The overall flow of the basic augmented Lagrangian iteration is shown in Figure 4.1 and involves: initializing algorithmic parameters; performing bound constrained minimization on an augmented Lagrangian function; calculating the current constraint violation at the updated design; checking for convergence; increasing the penalty parameter and re-minimizing if constraint violation did not improve; updating the Lagrange multipliers if the constraint violation did improve; setting the maximum constraint violation parameter to the current constraint violation parameter and re-minimizing if the constraint violation improved by a factor alpha; increasing the penalty parameter and setting the maximum constraint violation parameter to the current constraint violation parameter and re-minimizing if the constraint violation did not improve by a factor alpha.

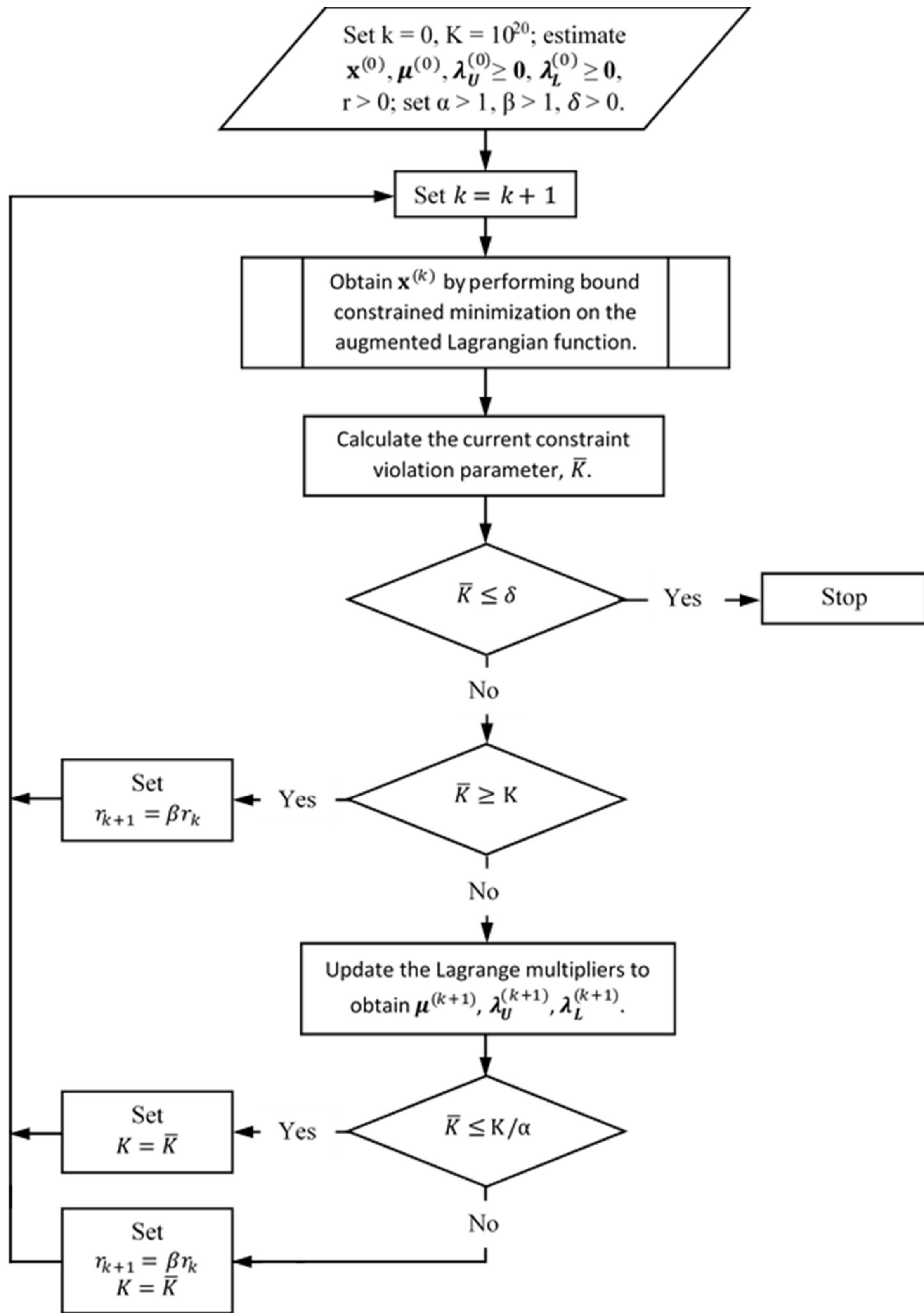

Figure 4.1 Basic Augmented Lagrangian Iteration Flow Diagram.

4.1.3 Limitations of the Basic Algorithm

The basic augmented Lagrangian algorithm permits convergence, in terms of the design variables, at a rate consistent with the underlying bound-constrained algorithm. However, convergence of the constraint or Lagrange multiplier variables is only as good as the linear, or steepest ascent, Lagrange multiplier updates given in formulas 4.7 to 4.9. Furthermore, updates to the Lagrange multiplier variables only occur after nearly exact bound-constrained minimization, further slowing progress towards feasibility. Ideally, Lagrange multiplier variables should converge at the same rate as the design variables, so that feasibility and optimality can be achieved in as few iterations as possible. In fact, this is one of the primary reasons for the recent popularity of Sequential Quadratic Programming (SQP) and Interior Point (IP) methods for solving constrained problems, as they typically simultaneously update the Lagrange multiplier and design variable values. Another limitation of the basic augmented Lagrangian algorithm, is its heavy reliance on the penalty parameter. Selecting too small of a penalty parameter may lead to a non-convex bound-constrained sub-problem. Selecting too large of a penalty parameter may lead to a numerically unstable bound-constrained sub-problem. In the sections that follow, however, it is shown how continuous Lagrange multiplier update strategies and ideas from SQP and IP methods can be combined to create an augmented Lagrangian algorithm competitive with SQP and IP methods.

4.1.4 Literature Review

One of the most important aspects of designing an efficient and robust augmented Lagrangian algorithm is the selection of an augmented Lagrangian function. Many augmented Lagrangian functions have been proposed. In general, however, they fall into two categories, non-smooth and smooth. Nocedal and Wright (2006) note that the most popular of the non-smooth functions is the exact L1 penalty function. Its exactness property makes its performance less dependent on the penalty parameter update strategy,

since its minimizer will be the same as the minimizer of the original, non-penalized, problem for any positive value of the penalty parameter. Since the L1 penalty function uses the L1 norm to penalize constraint violations, it naturally treats all constraint violations as close to equally as possible when deciding where to move to improve feasibility. The L1 norm's nearly ideal ability to capture constraint sensitivity, however, comes at the cost of non-smoothness at the constraint boundaries, which can hinder progress towards optimality if a gradient based method is being used to perform the unconstrained minimizations. Getting around non-smoothness of the L1 penalty function involves: developing a quadratic model of the L1 penalty function; introducing two slack variables for every linearized equality constraint and one slack variable for every linearized inequality constraint to promote smoothness; solving a linearly constrained quadratic programming sub-problem, similar to that of an SQP method running in elastic mode, to handle inconsistent constraint linearizations. This approach has been implemented successfully in practice. However, ideally an augmented Lagrangian function should not increase the dimensionality of the problem and should use a problem's objective and constraint functions directly, instead of quadratic and linear approximations of them. Furthermore, solution of a computationally expensive linearly constrained quadratic programming sub-problem, in order to ascertain a search direction, defeats the purpose of using an augmented Lagrangian approach over an SQP approach.

To avoid these complications, smooth exact augmented Lagrangian functions, notably the penalty Lagrangian developed by Hestenes (1969) and Powell (1969) have been used. Similar to the L1 penalty function, Hestenes and Powell's penalty Lagrangian's exactness property makes its performance less dependent on the penalty parameter update strategy used. Since their penalty Lagrangian uses the square of the constraint violation in its penalty terms, smoothness at the constraint boundary is ensured and the expensive and potentially problematic solution procedure required by the L1 penalty function is avoided. The only potential drawback of their penalty Lagrangian, is

that the square of the constraint violations in the penalty terms may encourage satisfaction of constraints with quickly increasing violations first, rather than seeking satisfaction of all constraints at a relatively equal rate like the L1 penalty function. However, since practical use of Hestenes and Powell's penalty Lagrangian does not require linearization of the constraints, progress towards feasibility may be competitive with that of the L1 penalty function.

Rockafellar (1973) was the first to extend Hestenes and Powell's penalty Lagrangian for the case of inequality constraints. Rockafellar's penalty Lagrangian is impressive in that it automatically adds and removes constraints that are violated and satisfied, respectively, from the problem. However, it has the drawback of not offering a convenient choice of penalty parameter and of having discontinuities in the second derivative, which may adversely affect line search iterations. Gill, Murray, Saunders, and Wright (1986) noted these drawbacks while researching which augmented Lagrangian function to use as a merit function, to assess trial line search steps with in SNOPT's SQP implementation. They circumnavigated these drawbacks by explicitly incorporating slack variables into Hestenes and Powell's penalty Lagrangian, which essentially converted all constraints to equality constraints for line search purposes. Furthermore, they avoided increasing problem dimensionality by deriving explicit procedures for calculating starting slack variable values and slack variable search direction values that were compatible with the solution to their QP subproblem.

As noted in section 4.1, a crucial aspect of achieving rapid convergence towards feasibility with augmented Lagrangian algorithms is the choice of Lagrange multiplier update formula and the frequency with which those updates occur. Arora, Chahande, and Paeng (1991) reviewed various augmented Lagrangian or multiplier methods and their applicability to engineering design problems. They also provided an interesting review of Lagrange multiplier update procedures and an introduction to continuous multiplier update methods. Arora et al. note, for continuous multiplier update methods, the

importance of design variable and Lagrange multiplier variable update procedures being compatible, i.e. a quasi-Newton design variable update used with a quasi-Newton Lagrange multiplier variable update. Compatibility between updates to the primal or design variables and dual or Lagrange multiplier variables allows for exact unconstrained minimization in terms of the design variables to be bypassed and the Lagrange multiplier variables to be updated continuously with the design variables, similar to an SQP method. In fact, Arora et al. show that one iteration of a quasi-Newton unconstrained algorithm, followed by a compatible update of the Lagrange multipliers, corresponds to one iteration of an SQP method. Specifically, they show that if the Lagrange multiplier variables are updated using the solution of the SQP method's QP sub-problem at each unconstrained iteration, then the multiplier method iterations are equivalent to the SQP method iterations.

Nocedal and Wright (2006) study the solution of local SQP methods' QP sub-problems in depth, before incorporating line search or trust region techniques to develop global SQP methods. They note how local SQP methods' QP sub-problems may be derived simply by applying Newton's method to the KKT optimality conditions for the constrained problem. This clearly shows the extension of unconstrained Newton / quasi-Newton methods to constrained problems. Similar to the case of unconstrained problems, the solution of the resulting Newton-KKT system is a Newton step in the design and Lagrange multiplier variables, which may also be viewed as search directions in the design and Lagrange multiplier variables, respectively, for line search purposes. Also of interest, they derive local SQP methods' QP sub-problems in the form of a quadratic program and perform some algebraic manipulations that suggest a strategy for updating the Lagrange multiplier and design variables in sequence instead of simultaneously, which is useful for continuous multiplier update methods. Arora et al. (1991) recognized this as well and derived an explicit formula for calculating the design variable search

direction in terms of Lagrange multiplier variable values updated using a compatible update formula or method.

Tapia (1977) studied various Lagrange multiplier update formulas and proposed a diagonalized quasi-newton multiplier method for equality constrained problems that used a new Lagrange multiplier update formula, which was consistent with quasi-newton design variable updates, to continuously update the Lagrange multipliers with every update to the design variables. The general steps of Tapia's method include: setting initial values of the design variables, Lagrange multiplier variables, penalty parameter, and inverse Hessian approximation; using their new Lagrange multiplier update formula to update the Lagrange multipliers; performing a Newton step to update the design variables; updating the Lagrange multipliers again for the new design; updating the penalty parameter; updating the inverse Hessian approximation using updated Lagrange multiplier values. However, Tapia notes that the penalty parameter need not be updated at every iteration and that the second Lagrange multiplier update need not be performed if descent is enforced, i.e. a decrease in the augmented Lagrangian is ensured via some type of line search on both the Lagrange multiplier and design variable updates. Tapia extended his diagonalized multiplier method to the case of the general equality-inequality constrained problem by introducing slack variables to the inequality constraints and converting them to equality constraints.

At the same time Tapia (1977) was developing his diagonalized multiplier method, Han (1977) was developing his dual variable metric (i.e. quasi-Newton) algorithm for constrained optimization. While Han proposed updating equality constraint Lagrange multiplier variables by solving a system of linear equations instead of evaluating an explicit update formula like Tapia, both approaches are essentially the same and generate the same sequence of Lagrange multiplier variable values. Where Han and Tapia differ, however, is in their treatment of the general equality-inequality constrained

problem. Instead of converting inequalities to equalities, Han proposes a separate quadratic programming sub-problem to handle inequality constraints directly.

Mayne and Polak (1980) also handled updating both equality and inequality constraint Lagrange multiplier variables directly and were the first to propose a combined sub-problem that updated them simultaneously. Mayne and Polak's approach uses the KKT conditions for the generally constrained problem to develop an objective function that, when minimized, provides estimates of the Lagrange multiplier variable values at the current candidate minimum point. This approach is general in that it is not dependent on the specific search direction technique and augmented Lagrangian function used like Tapia and Han's methods, which are specifically designed for quasi-newton search direction techniques and the penalty Lagrangian developed by Hestenes (1969) and Powell (1969). However, it suffers from the drawback of requiring complicating penalty terms in the proposed objective function to ensure continuity.

The final challenge identified in section 4.1 to achieving rapid convergence with augmented Lagrangian algorithms, was devising a robust penalty parameter update strategy. The relationship between the Lagrange multiplier variables and the penalty parameter was studied in depth by Mayne and Polak (1982). Specifically, they studied properties of the penalty parameter necessary for global convergence and proposed an automatic rule for selecting and updating the penalty parameter that ensures these properties are satisfied. They note that if a given primal and dual step is a KKT triple for the quadratic program then the primal search direction is a direction of descent for the augmented Lagrangian if the inverse Hessian approximation is positive definite and the penalty parameter is larger than the sum of the absolute values of the Lagrange multiplier variables. Taking this into account and recognizing that current estimates of the Lagrange multiplier variables may not be accurate at a point far from the solution, their proposed formula for estimating the penalty parameter sums the absolute values of the Lagrange multiplier variables and then increases that value by ten-percent to ensure that

the current estimate of the penalty parameter is large enough. Their automatic procedure for selecting and updating the penalty parameter works by: starting with an initial penalty parameter value; calculating the penalty parameter value predicted by their formula; keeping the penalty parameter value unchanged if it is already larger than that predicted by the formula; taking the penalty parameter as the maximum of 10 times its current value or the value predicted by the formula if it is smaller than that predicted by the formula; performing unconstrained minimization in order to move to the next design point. This process is repeated as necessary until convergence is reached. It ensures that the penalty parameter is never smaller than the conditions necessary to ensure global convergence and keeps the penalty parameter from becoming too large.

While Mayne and Polak's (1982) penalty parameter update strategy is based on conditions required for global convergence and is applicable to multiple augmented Lagrangian functions, it is still heuristic in nature. Gill, Murray, Saunders, and Wright (1986) noted that, for their SQP method's augmented Lagrangian merit function, the penalty parameter update need not be heuristic. In fact, they noted that a current value of the penalty parameter could be calculated directly using a condition required for global convergence and the fact that rapid convergence is strongly tied to keeping the penalty parameter as small as possible. Specifically, their method calculates the current value of the penalty parameter, before each line search, such that the projected gradient of the merit function, i.e. line search function slope, will satisfy a condition required for global convergence.

4.1.5 Overcoming Limitations of the Basic Algorithm

Here, ideas from L-BFGS bound-constrained algorithms, augmented Lagrangian algorithms, and SQP algorithms are combined and extended to create a novel generally constrained augmented Lagrangian algorithm.

The augmented Lagrangian merit function proposed by Gill, Murray, Saunders, and Wright is taken as the augmented Lagrangian to be minimized, over Rockafellar's traditional augmented Lagrangian, for its continuous second derivatives and convenient choice of penalty parameter. While Gill, Murray, Saunders, and Wright's augmented Lagrangian merit function does introduce somewhat complicating slack variables, it is shown how their search direction can be calculated explicitly, to avoid increasing problem dimensionality.

Since the augmented Lagrangian function chosen converts general constraints to equalities, a similar method to Tapia and Han's may be used for updating the equality constraint Lagrange multiplier variable values. However, instead of evaluating an explicit update formula or solving a system of linear equations, a simple unconstrained QP sub-problem is proposed. After the updated Lagrange multiplier variable values have been identified, Arora's technique for calculating a search direction consistent with the updated Lagrange multiplier variable values may be applied and a line search along the design variable, Lagrange multiplier, and slack variable search directions performed.

4.1.6 Novel Algorithm

The following novel augmented Lagrangian algorithm applies to generally constrained optimization problems of the form:

minimize $f(\boldsymbol{x})$ (4.10a)

subject to $\boldsymbol{l} \leq \begin{pmatrix} \boldsymbol{x} \\ \boldsymbol{c}(\boldsymbol{x}) \end{pmatrix} \leq \boldsymbol{u}$ (4.10b)

where $f(\boldsymbol{x})$ represents the objective function, $\boldsymbol{x}$ represents the vector of design variables, $\boldsymbol{c}(\boldsymbol{x})$ represents the vector of linear and nonlinear general constraints, $\boldsymbol{l}$ represents the vector of lower bounds on $\boldsymbol{x}$ and $\boldsymbol{c}(\boldsymbol{x})$, and $\boldsymbol{u}$ represents the vector of upper bounds on $\boldsymbol{x}$ and $\boldsymbol{c}(\boldsymbol{x})$. Note that equality constraints may be specified by setting the lower and upper bounds equal for a constraint.

Reformatting of constraints: To simplify implementation, it is helpful to define transformed vectors of constraints of the form $\boldsymbol{h}(\boldsymbol{x}) = \boldsymbol{0}$ and $\boldsymbol{g}(\boldsymbol{x}) \leq \boldsymbol{0}$. Note that evaluation of $\boldsymbol{h}(\boldsymbol{x})$ and $\boldsymbol{g}(\boldsymbol{x})$ need not require more than one evaluation of the vector of constraints $\boldsymbol{c}(\boldsymbol{x})$:

$$\boldsymbol{h}(\boldsymbol{x}) \equiv \boldsymbol{c}_{\boldsymbol{E}}(\boldsymbol{x}) - \frac{1}{2}(\boldsymbol{l}_{\boldsymbol{E}} + \boldsymbol{u}_{\boldsymbol{E}}) = \boldsymbol{0}; \quad \boldsymbol{J}_{\boldsymbol{h}}(\boldsymbol{x}) \equiv \boldsymbol{J}_{\boldsymbol{c}_{\boldsymbol{E}}}(\boldsymbol{x}) \tag{4.11}$$

$$\boldsymbol{g}(\boldsymbol{x}) \equiv \begin{pmatrix} \boldsymbol{c}_{\boldsymbol{U}}(\boldsymbol{x}) - \boldsymbol{u}_{\boldsymbol{U}} \\ \boldsymbol{l}_{\boldsymbol{L}} - \boldsymbol{c}_{\boldsymbol{L}}(\boldsymbol{x}) \end{pmatrix} \leq \boldsymbol{0}; \quad \boldsymbol{J}_{\boldsymbol{g}}(\boldsymbol{x}) \equiv \begin{pmatrix} \boldsymbol{J}_{\boldsymbol{c}_{\boldsymbol{U}}}(\boldsymbol{x}) \\ -\boldsymbol{J}_{\boldsymbol{c}_{\boldsymbol{L}}}(\boldsymbol{x}) \end{pmatrix} \tag{4.12}$$

where

$$\boldsymbol{c}_{\boldsymbol{E}}(\boldsymbol{x}), \boldsymbol{l}_{\boldsymbol{E}}, \boldsymbol{u}_{\boldsymbol{E}} \in \mathbb{R}^{\boldsymbol{i} \times \boldsymbol{1}}; \quad \boldsymbol{J}_{\boldsymbol{c}_{\boldsymbol{E}}}(\boldsymbol{x}) \in \mathbb{R}^{\boldsymbol{i} \times \boldsymbol{n}}; \quad i \in \boldsymbol{E} \tag{4.13}$$

$$\boldsymbol{c}_{\boldsymbol{U}}(\boldsymbol{x}), \boldsymbol{u}_{\boldsymbol{U}} \in \mathbb{R}^{\boldsymbol{i} \times \boldsymbol{1}}; \quad \boldsymbol{J}_{\boldsymbol{c}_{\boldsymbol{U}}}(\boldsymbol{x}) \in \mathbb{R}^{\boldsymbol{i} \times \boldsymbol{n}}; \quad i \in \boldsymbol{U} \tag{4.14}$$

$$\boldsymbol{c}_{\boldsymbol{L}}(\boldsymbol{x}), \boldsymbol{l}_{\boldsymbol{L}} \in \mathbb{R}^{\boldsymbol{i} \times \boldsymbol{1}}; \quad \boldsymbol{J}_{\boldsymbol{c}_{\boldsymbol{L}}}(\boldsymbol{x}) \in \mathbb{R}^{\boldsymbol{i} \times \boldsymbol{n}}; \quad i \in \boldsymbol{L} \tag{4.15}$$

and equality, upper bound inequality, and lower bound inequality constraint sets $\boldsymbol{E}$, $\boldsymbol{U}$, and $\boldsymbol{L}$, respectively, are defined as follows:

$$\boldsymbol{E} = \{i\!: \ |u_i - l_i| \leq \varepsilon_f, \ \ i = n+1, n+m\} \tag{4.16}$$

$$\boldsymbol{U} = \{i\!: \ |u_i - l_i| > \varepsilon_f \ \& \ u_i \neq \infty, \quad i = n+1, n+m\} \tag{4.17}$$

$$\boldsymbol{L} = \{i\!: \ |u_i - l_i| > \varepsilon_f \ \& \ l_i \neq -\infty, \quad i = n+1, n+m\} \tag{4.18}$$

where ε_f represents the constraint feasibility tolerance, n represents the number of design variables $\boldsymbol{x}$, and m represents the number of linear and nonlinear general constraints in $\boldsymbol{c}(\boldsymbol{x})$. Finally, to further simplify implementation, it is helpful to define bookkeeping variables m_E, m_U, and m_L representing the *total* number of *transformed* equality, upper bound inequality, and lower bound inequality constraints, respectively:

$$m_E = length(\boldsymbol{E}) \tag{4.19}$$

$$m_U = length(\boldsymbol{U}) \tag{4.20}$$

$$m_L = length(\boldsymbol{L}) \tag{4.21}$$

Stating the generally constrained optimization problem in terms of the transformed vectors of constraints gives:

$$\text{minimize} \quad f(\boldsymbol{x}) \tag{4.22a}$$

$$\text{subject to} \quad l_i \leq x_i \leq u_i, \;\; i = 1, n \tag{4.22b}$$

$$\boldsymbol{h}(\boldsymbol{x}) = \boldsymbol{0} \tag{4.22c}$$

$$\boldsymbol{g}(\boldsymbol{x}) \leq \boldsymbol{0} \tag{4.22d}$$

where $\boldsymbol{h}(\boldsymbol{x}) \in \mathbb{R}^{m_E \times 1}$ and $\boldsymbol{g}(\boldsymbol{x}) \in \mathbb{R}^{(m_U + m_L) \times 1}$. This is the form used throughout the algorithm that follows. Also, for clarity, transformed equality and transformed inequality constraints are referred to as equality and inequality constraints.

In general, the algorithm that follows consists of 14 steps: (1) Initialize algorithmic parameters, ensure all design variables are on or within their bounds, and evaluate the optimization problem functions; (2) Calculate starting slack variable values, corresponding to the inequality constraints, that minimize the augmented Lagrangian function, subject to non-negativity of the slack variables; (3) Calculate the maximum constraint violation from the equality constraints, inequality constraints, and inequality constraint Lagrange multiplier non-negativity constraints; (4) Calculate the inactive-variable-set and active-constraint-set. The inactive-variable-set consists of variables within their bounds or variables near their bounds that tend to move inward towards the *feasible* region. The active-constraint-set consists of inequality constraints with corresponding slack variables that are near their bounds and tend to move outward towards the *infeasible* region, i.e. negativity. Here, the Lagrange multipliers for the lower and upper bound constraints on the design and slack variables, the active design variable search direction components, the active slack variable search direction components, the inactive inequality constraint Lagrange multipliers, and the inactive inequality constraint Lagrange multiplier search direction components are also set; (5) Calculate the gradient of the Lagrangian function with respect to the design variables and take the maximum

optimality violation as the infinity norm of the components of the Lagrangian gradient corresponding to the inactive-variable-set; (6) Check if the current point is a KKT point, i.e. the maximum constraint violation and maximum optimality violation are small. If so, stop. Otherwise, continue; (7) Calculate second order equality constraint and active inequality constraint Lagrange multiplier search direction components by solving a linear system of equations or by solving an equivalent unconstrained sub-problem, considering inactive design variables only. Either solution method is equivalent to solving the KKT system of equations for an equality constrained problem explicitly. Also, set the Quadratic Programming (QP) Lagrange multipliers; (8) Calculate second order inactive design variable search direction components using the L-BFGS two-loop recursion of section 2.3.1.1, taking the gradient vector to be multiplied by the inverse Hessian approximation as the gradient of the Lagrangian evaluated at the QP Lagrange multipliers. Once again, this is equivalent to solving the KKT system of equations for an equality constrained problem explicitly; (9) Calculate QP slack variables corresponding to the linearized inequality constraints, considering inactive design variables only, and set the inactive slack variable search direction components to the difference between the current slack variable values and the QP slack variable values; (10) Update each constraint's penalty parameter, as necessary, to ensure the directional derivative of the line search function is sufficiently negative; (11) Use the efficient backtracking line search procedure given in section 3.2 to find a step size satisfying Armijo's sufficient decrease rule. Update the design variables, slack variables, and Lagrange multipliers; (12) Evaluate the optimization problem functions for the next iteration; (13) Set the L-BFGS vectors used to approximate the *inverse Hessian* of the *Lagrangian* function; (14) Set all problem variables for the next iteration, increment the iteration counter, and go to step 3.

The Lagrangian function and its gradient with respect to the design variables, which is used in second order search direction calculations and to assess optimality, is given by:

$$L(\boldsymbol{x},\boldsymbol{\mu},\boldsymbol{\lambda}) = f(\boldsymbol{x}) + \boldsymbol{\mu}^T\boldsymbol{h}(\boldsymbol{x}) + \boldsymbol{\lambda}^T(\boldsymbol{g}(\boldsymbol{x}) + \boldsymbol{z}) \tag{4.23}$$

$$\nabla_x L(\boldsymbol{x},\boldsymbol{\mu},\boldsymbol{\lambda}) = \nabla_x f(\boldsymbol{x}) + \boldsymbol{J_h}(\boldsymbol{x})^T\boldsymbol{\mu} + \boldsymbol{J_g}(\boldsymbol{x})^T\boldsymbol{\lambda} \tag{4.24}$$

The augmented Lagrangian function and its gradient with respect to the design variables, slack variables, equality constraint Lagrange multipliers, and inequality constraint Lagrange multipliers is given by:

$$\Phi(\boldsymbol{x},\boldsymbol{z},\boldsymbol{\mu},\boldsymbol{\lambda},\boldsymbol{P_h},\boldsymbol{P_g}) = f(\boldsymbol{x}) + \boldsymbol{\mu}^T\boldsymbol{h}(\boldsymbol{x}) + \frac{1}{2}\boldsymbol{h}(\boldsymbol{x})^T\boldsymbol{P_h}\boldsymbol{h}(\boldsymbol{x}) \tag{4.25}$$

$$+ \boldsymbol{\lambda}^T(\boldsymbol{g}(\boldsymbol{x}) + \boldsymbol{z}) + \frac{1}{2}(\boldsymbol{g}(\boldsymbol{x}) + \boldsymbol{z})^T\boldsymbol{P_g}(\boldsymbol{g}(\boldsymbol{x}) + \boldsymbol{z})$$

$$\nabla_x\Phi(\boldsymbol{x},\boldsymbol{z},\boldsymbol{\mu},\boldsymbol{\lambda},\boldsymbol{P_h},\boldsymbol{P_g}) = \nabla_x f(\boldsymbol{x}) + \boldsymbol{J_h}(\boldsymbol{x})^T\boldsymbol{\mu} + \boldsymbol{J_h}(\boldsymbol{x})^T\boldsymbol{P_h}\boldsymbol{h}(\boldsymbol{x}) \tag{4.26}$$

$$+ \mathbf{J}_{\boldsymbol{g}}(\boldsymbol{x})^T\boldsymbol{\lambda} + \boldsymbol{J_g}(\boldsymbol{x})^T\boldsymbol{P_g}(\boldsymbol{g}(\boldsymbol{x}) + \boldsymbol{z})$$

$$\nabla_z\Phi(\boldsymbol{x},\boldsymbol{z},\boldsymbol{\mu},\boldsymbol{\lambda},\boldsymbol{P_h},\boldsymbol{P_g}) = \boldsymbol{\lambda} + \boldsymbol{P_g}(\boldsymbol{g}(\boldsymbol{x}) + \boldsymbol{z}) \tag{4.27}$$

$$\nabla_{\boldsymbol{\mu}}\Phi(\boldsymbol{x},\boldsymbol{z},\boldsymbol{\mu},\boldsymbol{\lambda},\boldsymbol{P_h},\boldsymbol{P_g}) = \boldsymbol{h}(\boldsymbol{x}) \tag{4.28}$$

$$\nabla_{\boldsymbol{\lambda}}\Phi(\boldsymbol{x},\boldsymbol{z},\boldsymbol{\mu},\boldsymbol{\lambda},\boldsymbol{P_h},\boldsymbol{P_g}) = \boldsymbol{g}(\boldsymbol{x}) + \boldsymbol{z} \tag{4.29}$$

It is used in identifying the inactive-variable-set, identifying the active-constraint-set, and assessing sufficient decrease during line search iterations. It is not used in calculating second order search directions, as is typical in augmented Lagrangian methods, since it depends on a penalty parameter. Dependence on a penalty parameter has been observed to cause numerical instability when the penalty parameter becomes large. Therefore, the Lagrangian function is used in calculating second order search directions and positive definiteness of the inverse Hessian approximation is maintained by placing limits on the L-BFGS correction vectors. Interestingly, when the Lagrangian function is used in

calculating second order search directions, the sequence of iterates is identical, in theory, to that of an SQP method.

Step 1: Set $k = 0$, $\varepsilon_o = 1.1e - 6$, $\varepsilon_f = 1.1e - 6$, $\rho = 0.0001$, $\boldsymbol{p}_{\boldsymbol{h}}^{(k)} = 0.01$, $\boldsymbol{p}_{\boldsymbol{g}}^{(k)} = 0.01$, and $m_c = 20$; estimate $\boldsymbol{x}^{(k)}$, $\boldsymbol{z}^{(k)}$, $\boldsymbol{\mu}^{(k)}$, $\boldsymbol{\lambda}^{(k)}$, $\boldsymbol{\lambda}_{\boldsymbol{l}}^{(k)}$, and $\boldsymbol{\lambda}_{\boldsymbol{u}}^{(k)}$; ensure all $\boldsymbol{x}^{(k)}$ are on or within their bounds by applying the projection operator, i.e. eqn. 2.1, on each element of $\boldsymbol{x}^{(k)}$; evaluate $f^{(k)}$, $\boldsymbol{h}^{(k)}$, $\boldsymbol{g}^{(k)}$, $\nabla f^{(k)}$, $\boldsymbol{J}_{\boldsymbol{h}}^{(k)}$, and $\boldsymbol{J}_{\boldsymbol{g}}^{(k)}$; where k is the iteration counter, ε_o is the optimality tolerance, ε_f is the feasibility tolerance, ρ is a factor between 0 and 1 used by Armijo's rule in determining satisfactory step sizes, $\boldsymbol{p}_{\boldsymbol{h}}^{(k)} \in \mathbb{R}^{m_E \times 1}$ contains the penalty parameter values corresponding to the equality constraints for iteration k, $\boldsymbol{p}_{\boldsymbol{g}}^{(k)} \in \mathbb{R}^{(m_U + m_L) \times 1}$ contains the penalty parameter values corresponding to the inequality constraints for iteration k, m_c is the number of L-BFGS corrections to store, $\boldsymbol{x}^{(k)} \in \mathbb{R}^{n \times 1}$ contains the design variable values for iteration k, $\boldsymbol{z}^{(k)} \in \mathbb{R}^{(m_U + m_L) \times 1}$ contains the slack variable values for iteration k, $\boldsymbol{\mu}^{(k)} \in \mathbb{R}^{m_E \times 1}$ contains the equality constraint Lagrange multiplier values for iteration k, $\boldsymbol{\lambda}^{(k)} \in \mathbb{R}^{(m_U + m_L) \times 1}$ contains the inequality constraint Lagrange multiplier values for iteration k, $\boldsymbol{\lambda}_{\boldsymbol{l}}^{(k)} \in \mathbb{R}^{(n + m_U + m_L) \times 1}$ contains the lower bound constraint Lagrange multiplier values for iteration k for the lower bounds on the design and slack variables, $\boldsymbol{\lambda}_{\boldsymbol{u}}^{(k)} \in \mathbb{R}^{(n + m_U + m_L) \times 1}$ contains the upper bound constraint Lagrange multiplier values for iteration k for the upper bounds on the design and slack variables, $f^{(k)}$ contains the objective function evaluated at $\boldsymbol{x}^{(k)}$, $\boldsymbol{h}^{(k)} \in \mathbb{R}^{m_E \times 1}$ contains the equality constraints evaluated at $\boldsymbol{x}^{(k)}$, $\boldsymbol{g}^{(k)} \in \mathbb{R}^{(m_U + m_L) \times 1}$ contains the inequality constraints evaluated at $\boldsymbol{x}^{(k)}$, $\nabla f^{(k)} \in \mathbb{R}^{n \times 1}$ contains the gradient of the objective function with respect to the design variables evaluated at $\boldsymbol{x}^{(k)}$, $\boldsymbol{J}_{\boldsymbol{h}}^{(k)} \in \mathbb{R}^{m_E \times n}$ contains the Jacobian of the equality constraints evaluated at $\boldsymbol{x}^{(k)}$, and $\boldsymbol{J}_{\boldsymbol{g}}^{(k)} \in \mathbb{R}^{(m_U + m_L) \times n}$ contains the Jacobian of the inequality constraints evaluated at $\boldsymbol{x}^{(k)}$.

Step 2: Using current inequality constraint function values, $\boldsymbol{g}^{(k)}$, inequality constraint Lagrange multiplier values, $\boldsymbol{\lambda}^{(k)}$, and inequality constraint penalty parameter values, $\boldsymbol{p}_{\boldsymbol{g}}^{(k)}$, calculate slack variable values, $\boldsymbol{z}^{(k)}$:

$$z_j = \begin{cases} max\left(0, -g_j\right) & if\ p_{gj} = 0 \\ max\left(0, -g_j - \frac{\lambda_j}{p_{gj}}\right) & otherwise \end{cases}, \quad j = 1, \dots, m_U + m_L \tag{4.30}$$

This equation, for penalty parameter values greater than zero, is equivalent to minimizing the augmented Lagrangian function with respect to the slack variables alone. Thus, equation 4.30 calculates current optimal slack variable values. This equation was first discovered by Gill, Murray, Saunders, and Wright (1986) and has been modified here for the case of less-than-or-equal-to type constraints.

Step 3: Using current transformed constraint function values, $\boldsymbol{h}^{(k)}$ and $\boldsymbol{g}^{(k)}$, and inequality constraint Lagrange multiplier values, $\boldsymbol{\lambda}^{(k)}$, calculate the maximum feasibility violation, $\bar{F}$, as follows:

$$\bar{F} = max\left\{|h_i|,\ \ i = 1, \dots, m_E;\ \max\left(0,\ g_j\right),\ \ j = 1, \dots, m_U + m_L\right\} \tag{4.31}$$

Step 4: Using the procedure below, identify the inactive-variable-set and active-constraint-set, $\boldsymbol{I}^{(k)}$ and $\boldsymbol{A}^{(k)}$, respectively. Set the lower- and upper-bound constraint Lagrange multipliers, $\boldsymbol{\lambda}_{\boldsymbol{l}}^{(k)}$ and $\boldsymbol{\lambda}_{\boldsymbol{u}}^{(k)}$, respectively, for the bound constraints on the design and slack variables. Set the active design and slack variable search direction components such that a step size of one moves a variable, at most, to its bound or to the steepest descent step, whichever is closer. This way, in algorithms where the largest step allowed is one, variables that are close to their bounds can move towards their bounds, but not past their bounds. Set each inactive inequality constraint Lagrange multiplier to zero to enforce the KKT complementary slackness condition for the constraint. Similarly, set the inactive inequality constraint Lagrange multiplier search direction components to zero.

$for\ i = 1, \dots, n$

if $l_i \le x_i^{(k)} \le l_i + \varepsilon_f$ and $\nabla_x \Phi_i^{(k)} > 0$ (i.e., lower-bound is active)

then $i \notin \boldsymbol{I}^{(k)},\ \lambda_{li}^{(k)} = \nabla_x \Phi_i^{(k)},\ \lambda_{ui}^{(k)} = 0$, and $d_{xi}^{(k)} = max\left(-\nabla_x \Phi_i^{(k)},\ l_i - x_i^{(k)}\right)$

else if $u_i - \varepsilon_f \leq x_i^{(k)} \leq u_i$ and $\nabla_x \Phi_i^{(k)} < 0$ (i.e., upper-bound is active)

then $i \notin \boldsymbol{I}^{(k)},\ \lambda_{li}^{(k)} = 0,\ \lambda_{ui}^{(k)} = -\nabla_x \Phi_i^{(k)}$, and $d_{xi}^{(k)} = min\left(-\nabla_x \Phi_i^{(k)},\ u_i - x_i^{(k)}\right)$

else $l_i + \varepsilon_f < x_i^{(k)} < u_i - \varepsilon_f$ (i.e., neither bound is active)

then $i \in \boldsymbol{I}^{(k)},\ \lambda_{li}^{(k)} = 0$, and $\lambda_{ui}^{(k)} = 0$

$for\ j = 1, \ldots, m_U + m_L$

if $0 \leq z_j^{(k)} \leq \varepsilon_f$ and $\nabla_z \Phi_j^{(k)} > 0$ (i.e., inequality constraint is active)

then $j \in \boldsymbol{A}^{(k)},\ \lambda_{l(n+j)}^{(k)} = \nabla_z \Phi_j^{(k)},\ \lambda_{u(n+j)}^{(k)} = 0$, and $d_{zj}^{(k)} = max\left(-\nabla_z \Phi_j^{(k)},\ -z_j^{(k)}\right)$

else $\varepsilon_f < z_j^{(k)} < \infty$ (i.e., inequality constraint is inactive)

then $j \notin \boldsymbol{A}^{(k)},\ \lambda_{l(n+j)}^{(k)} = 0,\ \lambda_{u(n+j)}^{(k)} = 0,\ \lambda_j^{(k)} = 0$, and $\boldsymbol{d}_{\lambda j}^{(k)} = 0$

where

$$\nabla_x \Phi^{(k)} \equiv \nabla_x \Phi\left(\boldsymbol{x}^{(k)}, \boldsymbol{z}^{(k)}, \boldsymbol{\mu}^{(k)}, \boldsymbol{\lambda}^{(k)}, \boldsymbol{P}_{\boldsymbol{h}}^{(k)}, \boldsymbol{P}_{\boldsymbol{g}}^{(k)}\right) \tag{4.32}$$

and

$$\nabla_z \Phi^{(k)} \equiv \nabla_z \Phi\left(\boldsymbol{x}^{(k)}, \boldsymbol{z}^{(k)}, \boldsymbol{\mu}^{(k)}, \boldsymbol{\lambda}^{(k)}, \boldsymbol{P}_{\boldsymbol{h}}^{(k)}, \boldsymbol{P}_{\boldsymbol{g}}^{(k)}\right) \tag{4.33}$$

Note, $\boldsymbol{P}_{\boldsymbol{h}}^{(k)}$ and $\boldsymbol{P}_{\boldsymbol{g}}^{(k)}$ represent diagonal matrices containing the values of $\boldsymbol{p}_{\boldsymbol{h}}^{(k)}$ and $\boldsymbol{p}_{\boldsymbol{g}}^{(k)}$, respectively. Also, in steps 5, 7, 8, and 9 that follow, calculations are carried out in terms of the inactive set of design variables *only*. In step 9, calculations are carried out in terms of the inactive set of constraints *only.*

Step 5: Using current transformed constraint Jacobian values, $\boldsymbol{J}_{\boldsymbol{h}}^{(k)}$ and $\boldsymbol{J}_{\boldsymbol{g}}^{(k)}$, and Lagrange multiplier values, $\boldsymbol{\mu}^{(k)}$ and $\boldsymbol{\lambda}^{(k)}$, calculate the maximum optimality violation, $\bar{O}$, as follows:

$$\bar{O} = \left\|\nabla_x L^{(k)}\right\|_\infty \tag{4.34}$$

where

$$\nabla_x L^{(k)} \equiv \nabla_x L(\boldsymbol{x}^{(k)}, \boldsymbol{\mu}^{(k)}, \boldsymbol{\lambda}^{(k)}) \tag{4.35}$$

Note: the maximum optimality violation of equation 4.34 is calculated considering the inactive set of design variables *only*, since the KKT conditions for active design variables are enforced explicitly in step 4.

Step 6: Check if current point is a KKT point, i.e. check if the following criteria are satisfied:

$$\bar{F} \leq \varepsilon_f \tag{4.36}$$

$$\bar{O} \leq \varepsilon_o \tag{4.37}$$

If current point is a KKT point, stop. Otherwise, continue.

Step 7: If $k < 1$, take $\boldsymbol{H}^{(k)}$ as an identity matrix and solve the following unconstrained sub-problem for equality constraint and active inequality constraint Lagrange multiplier search direction components, $\boldsymbol{d}_{\boldsymbol{\mu\lambda_A}}^{(k)}$:

$$\text{minimize} \quad q = \frac{1}{2} \boldsymbol{d}_{\boldsymbol{\mu\lambda_A}}^{(k)\,T} \mathbf{B}\, \boldsymbol{d}_{\boldsymbol{\mu\lambda_A}}^{(k)} - \boldsymbol{d}_{\boldsymbol{\mu\lambda_A}}^{(k)\,T} \mathbf{b} \tag{4.38}$$

$$\text{which has gradient} \quad \nabla q = \mathbf{B}\, \boldsymbol{d}_{\boldsymbol{\mu\lambda_A}}^{(k)} - \mathbf{b} \tag{4.39}$$

where

$$\mathbf{B} = \begin{pmatrix} \boldsymbol{J}_{\boldsymbol{h}}^{(k)} \\ \boldsymbol{J}_{\boldsymbol{g_A}}^{(k)} \end{pmatrix} \boldsymbol{H}^{(k)} \begin{pmatrix} \boldsymbol{J}_{\boldsymbol{h}}^{(k)} \\ \boldsymbol{J}_{\boldsymbol{g_A}}^{(k)} \end{pmatrix}^T \tag{4.40}$$

$$\mathbf{b} = \begin{pmatrix} \boldsymbol{h}^{(k)} \\ \boldsymbol{g}_{\boldsymbol{A}}^{(k)} \end{pmatrix} - \begin{pmatrix} \boldsymbol{J}_{\boldsymbol{h}}^{(k)} \\ \boldsymbol{J}_{\boldsymbol{g_A}}^{(k)} \end{pmatrix} \boldsymbol{H}^{(k)} \nabla_x L(\boldsymbol{x}^{(k)}, \boldsymbol{\mu}^{(k)}, \boldsymbol{\lambda}^{(k)}) \tag{4.41}$$

$$\boldsymbol{d}_{\boldsymbol{\mu\lambda_A}}^{(k)} = \begin{pmatrix} \boldsymbol{d}_{\boldsymbol{\mu}}^{(k)} \\ \boldsymbol{d}_{\boldsymbol{\lambda_A}}^{(k)} \end{pmatrix} \tag{4.42}$$

Note: the above unconstrained sub-problem solves the positive-definite linear system $\mathbf{B}\, \boldsymbol{d}_{\boldsymbol{\mu\lambda_A}}^{(k)} = \mathbf{b}$ for $\boldsymbol{d}_{\boldsymbol{\mu\lambda_A}}^{(k)}$, which is equivalent to solving the KKT system, for the equality constrained problem, explicitly for a Lagrange multiplier search direction. Tapia (1977) and Han (1977) were the first to propose such a sub-problem. The sub-problem proposed

here, however, is more robust since it does not require the inverse of matrix **B**, which may be quite large and poorly conditioned.

If $k \geq 1$, take $\boldsymbol{H}^{(k)}$ as the L-BFGS approximation of the *inverse Hessian* of the *Lagrangian* function and solve the above unconstrained sub-problem.

Set the Quadratic Programming (QP) Lagrange multipliers:

$$\bar{\boldsymbol{\mu}} = \boldsymbol{\mu}^{(k)} + \boldsymbol{d}_{\boldsymbol{\mu}}^{(k)} \tag{4.43}$$

$$\bar{\boldsymbol{\lambda}} = \boldsymbol{\lambda}^{(k)} + \boldsymbol{d}_{\boldsymbol{\lambda}}^{(k)} \tag{4.44}$$

Note: inactive inequality constraint Lagrange multiplier search direction components were set during step 4. Also, the QP Lagrange multipliers generated by equations 4.43 and 4.44 are equivalent, in theory, to those generated by an SQP algorithm. Lastly, matrix **B** and vector **b** are computed considering inactive design variable components of the constraint Jacobians, inverse Hessian approximation, and Lagrangian gradient *only*.

Step 8: If $k < 1$, set the inactive components of the design variable search direction, $\boldsymbol{d}_{\boldsymbol{x}}^{(k)}$, to that of steepest descent:

$$\boldsymbol{d}_{\boldsymbol{x}}^{(k)} = -\nabla_x L(\boldsymbol{x}^{(k)}, \bar{\boldsymbol{\mu}}, \bar{\boldsymbol{\lambda}}) \tag{4.45}$$

If $k \geq 1$, use the L-BFGS two-loop recursion given in section 2.3.1.1 to calculate the inactive components of the design variable search direction:

$$\boldsymbol{d}_{\boldsymbol{x}}^{(k)} = -\boldsymbol{H}^{(k)} \nabla_x L(\boldsymbol{x}^{(k)}, \bar{\boldsymbol{\mu}}, \bar{\boldsymbol{\lambda}}) \tag{4.46}$$

where $\boldsymbol{H}^{(k)}$ represents the L-BFGS approximation of the *inverse Hessian* of the *Lagrangian* function.

Step 9: Calculate QP slack variables, $\bar{\boldsymbol{z}}$, corresponding to the linearized inequality constraints as follows:

$$\boldsymbol{g}^{(k)} + \boldsymbol{J}_{\boldsymbol{g}}^{(k)} \boldsymbol{d}_{\boldsymbol{x}}^{(k)} \leq \boldsymbol{0} \tag{4.47}$$

$$\boldsymbol{g}^{(k)} + \boldsymbol{J}_{\boldsymbol{g}}^{(k)} \boldsymbol{d}_{\boldsymbol{x}}^{(k)} + \bar{\boldsymbol{z}} = \boldsymbol{0} \tag{4.48}$$

$$\bar{\boldsymbol{z}} = -\boldsymbol{J}_{\boldsymbol{g}}^{(k)} \boldsymbol{d}_{\boldsymbol{x}}^{(k)} - \boldsymbol{g}^{(k)} \tag{4.49}$$

Set the slack variable search direction, $\boldsymbol{d}_{\boldsymbol{z}}^{(k)}$:

$$\boldsymbol{d}_{\boldsymbol{z}}^{(k)} = \bar{\boldsymbol{z}} - \boldsymbol{z}^{(k)} \tag{4.50}$$

Note: QP slack variables are computed considering inactive design variable components of the constraint Jacobian and design variable search direction *only*.

Step 10: Update equality and inequality constraint penalty parameter vectors, $\boldsymbol{p}_{\boldsymbol{h}}^{(k)}$ and $\boldsymbol{p}_{\boldsymbol{g}}^{(k)}$, respectively, using the following procedure.

First, solve a linearly constrained least-squares sub-problem for intermediate equality and inequality constraint penalty parameter vectors, $\widehat{\boldsymbol{p}}_{\boldsymbol{h}}$ and $\widehat{\boldsymbol{p}}_{\boldsymbol{g}}$, respectively:

$$\text{minimize} \quad \|\boldsymbol{p}\|^2 \tag{4.51}$$

$$\text{subject to} \quad \phi'(0,\boldsymbol{p}) = \tfrac{1}{2}\nabla_x L\left(\boldsymbol{x}^{(k)}, \bar{\boldsymbol{\mu}}, \bar{\boldsymbol{\lambda}}\right)^T \boldsymbol{d}_{\boldsymbol{x}}^{(k)} \tag{4.52}$$

$$\boldsymbol{p} \geq \boldsymbol{0} \tag{4.53}$$

where

$$\boldsymbol{p} = \begin{pmatrix} \boldsymbol{p}_{\boldsymbol{h}} \\ \boldsymbol{p}_{\boldsymbol{g}} \end{pmatrix} \tag{4.54}$$

$$\begin{aligned} \phi'(0,\boldsymbol{p}) = {} & \nabla_x \Phi\left(\boldsymbol{x}^{(k)}, \boldsymbol{z}^{(k)}, \boldsymbol{\mu}^{(k)}, \boldsymbol{\lambda}^{(k)}, \boldsymbol{P}_{\boldsymbol{h}}, \boldsymbol{P}_{\boldsymbol{g}}\right)^T \boldsymbol{d}_{\boldsymbol{x}}^{(k)} \\ & + \nabla_{\boldsymbol{z}} \Phi\left(\boldsymbol{x}^{(k)}, \boldsymbol{z}^{(k)}, \boldsymbol{\mu}^{(k)}, \boldsymbol{\lambda}^{(k)}, \boldsymbol{P}_{\boldsymbol{h}}, \boldsymbol{P}_{\boldsymbol{g}}\right)^T \boldsymbol{d}_{\boldsymbol{z}}^{(k)} \\ & + \nabla_{\boldsymbol{\mu}} \Phi\left(\boldsymbol{x}^{(k)}, \boldsymbol{z}^{(k)}, \boldsymbol{\mu}^{(k)}, \boldsymbol{\lambda}^{(k)}, \boldsymbol{P}_{\boldsymbol{h}}, \boldsymbol{P}_{\boldsymbol{g}}\right)^T \boldsymbol{d}_{\boldsymbol{\mu}}^{(k)} \\ & + \nabla_{\boldsymbol{\lambda}} \Phi\left(\boldsymbol{x}^{(k)}, \boldsymbol{z}^{(k)}, \boldsymbol{\mu}^{(k)}, \boldsymbol{\lambda}^{(k)}, \boldsymbol{P}_{\boldsymbol{h}}, \boldsymbol{P}_{\boldsymbol{g}}\right)^T \boldsymbol{d}_{\boldsymbol{\lambda}}^{(k)} \end{aligned} \tag{4.55}$$

represents the concatenated penalty parameter vector and the directional derivative of the line search function. Gill, Murray, and Saunders (2005) were the first to propose this sub-problem, which derives from Gill, Murray, Saunders, and Wright's (1986) previous work. The sub-problem finds the smallest positive penalty parameters that ensure the directional derivative of the line search function satisfies a condition associated with global convergence. Here, it is shown how this sub-problem can be transformed into a simpler bound-constrained sub-problem.

To simplify implementation, it is helpful to note that the above sub-problem is a least-norm problem, i.e. minimize $\|\boldsymbol{p}\|^2$ subject to $\boldsymbol{Cp} = \boldsymbol{c}$, with non-negativity of the design variables, $\boldsymbol{p}$. Boyd (2016) shows that the KKT conditions for least-norm problems are straight forward to write:

$$\begin{pmatrix} 2\boldsymbol{I} & \boldsymbol{C}^T \\ \boldsymbol{C} & \boldsymbol{0} \end{pmatrix} \begin{pmatrix} \boldsymbol{p} \\ \boldsymbol{\mu} \end{pmatrix} = \begin{pmatrix} \boldsymbol{0} \\ \boldsymbol{c} \end{pmatrix} \tag{4.56}$$

where, for the sub-problem above, $\boldsymbol{I}$ is an identity matrix of size $\mathbb{R}^{(m_E+m_U+m_L)\times(m_E+m_U+m_L)}$, $\boldsymbol{C}$ is a matrix of size $\mathbb{R}^{(1)\times(m_E+m_U+m_L)}$, $\boldsymbol{c}$ is a vector of size $\mathbb{R}^{(1)\times(1)}$, $\boldsymbol{\mu}$ is a vector of size $\mathbb{R}^{(1)\times(1)}$, and $\boldsymbol{p}$ is a vector of size $\mathbb{R}^{(m_E+m_U+m_L)\times(1)}$:

$$\boldsymbol{c} = \Big\{ \tfrac{1}{2} \nabla_x L\big(\boldsymbol{x}^{(k)}, \overline{\boldsymbol{\mu}}, \overline{\boldsymbol{\lambda}}\big)^T \boldsymbol{d}_{\boldsymbol{x}}^{(k)} - \nabla_x L\big(\boldsymbol{x}^{(k)}, \boldsymbol{\mu}^{(k)}, \boldsymbol{\lambda}^{(k)}\big)^T \boldsymbol{d}_{\boldsymbol{x}}^{(k)} \tag{4.57}$$

$$- {\boldsymbol{\lambda}^{(k)}}^T \boldsymbol{d}_{\boldsymbol{z}}^{(k)} - {\boldsymbol{h}^{(k)}}^T \boldsymbol{d}_{\boldsymbol{\mu}}^{(k)} - \big(\boldsymbol{g}^{(k)} + \boldsymbol{z}^{(k)}\big)^T \boldsymbol{d}_{\boldsymbol{\lambda}}^{(k)} \Big\}$$

$$\boldsymbol{C} = \left\{ \begin{matrix} \boldsymbol{J}_{\boldsymbol{h}i}^{(k)} \boldsymbol{d}_{\boldsymbol{x}}^{(k)} h_i^{(k)}, \quad i = 1, \dots, m_E; \\ \boldsymbol{J}_{\boldsymbol{g}j}^{(k)} \boldsymbol{d}_{\boldsymbol{x}}^{(k)} \big(g_j^{(k)} + z_j^{(k)}\big) + d_{zj}^{(k)} \big(g_j^{(k)} + z_j^{(k)}\big), \quad j = 1, \dots, m_U + m_L \end{matrix} \right\} \tag{4.58}$$

Enforcing non-negativity of the design variables is straight forward when the KKT system is solved as a bound-constrained sub-problem:

$$\text{minimize} \quad q = \frac{1}{2} \begin{pmatrix} \boldsymbol{p} \\ \boldsymbol{\mu} \end{pmatrix}^T \begin{pmatrix} 2\boldsymbol{I} & \boldsymbol{C}^T \\ \boldsymbol{C} & \boldsymbol{0} \end{pmatrix} \begin{pmatrix} \boldsymbol{p} \\ \boldsymbol{\mu} \end{pmatrix} - \begin{pmatrix} \boldsymbol{p} \\ \boldsymbol{\mu} \end{pmatrix}^T \begin{pmatrix} \boldsymbol{0} \\ \boldsymbol{c} \end{pmatrix} \tag{4.59}$$

$$\text{subject to} \quad \boldsymbol{p} \geq \boldsymbol{0} \tag{4.60}$$

$$\text{which has gradient} \quad \boldsymbol{\nabla q} = \begin{pmatrix} 2\boldsymbol{I} & \boldsymbol{C}^T \\ \boldsymbol{C} & \boldsymbol{0} \end{pmatrix} \begin{pmatrix} \boldsymbol{p} \\ \boldsymbol{\mu} \end{pmatrix} - \begin{pmatrix} \boldsymbol{0} \\ \boldsymbol{c} \end{pmatrix} \tag{4.61}$$

This sub-problem is equivalent to the original sub-problem and can be solved using bound-constrained techniques. Let the solution to this sub-problem be the intermediate penalty parameter vector, $\widehat{\boldsymbol{p}}$, where:

$$\widehat{\boldsymbol{p}} = \begin{pmatrix} \widehat{\boldsymbol{p}}_{\boldsymbol{h}} \\ \widehat{\boldsymbol{p}}_{\boldsymbol{g}} \end{pmatrix} \tag{4.62}$$

Second, update penalty parameter vector, $\boldsymbol{p}^{(k)}$, as follows:

$$p_i^{(k)} = \begin{cases} max\left(\hat{p}_i\,, p_i^{(k)}\right) & if\ p_i^{(k)} < 4\left(\hat{p}_i + \Delta_p\right) \\ max\left(\hat{p}_i\,, \left(p_i^{(k)}\left(\hat{p}_i + \Delta_p\right)\right)^{1/2}\right) & otherwise \end{cases} \tag{4.63}$$

$i = 1, \dots, m_E + m_U + m_L$

Third, re-evaluate $\nabla_x \Phi^{(k)}$, $\nabla_z \Phi^{(k)}$, and $\phi'\left(0, \boldsymbol{p}^{(k)}\right)$ for updated penalty parameter vector, $\boldsymbol{p}^{(k)}$.

Step 11: Initialize $\bar{\alpha} = 1$. Use the efficient backtracking line search procedure given in section 3.2 to find a step size $\bar{\alpha} \in (0,1]$ satisfying Armijo's rule:

$$\phi\left(\bar{\alpha}, \boldsymbol{p}^{(k)}\right) \leq \phi\left(0, \boldsymbol{p}^{(k)}\right) + \rho\, \bar{\alpha}\, \phi'\left(0, \boldsymbol{p}^{(k)}\right) \tag{4.64}$$

where

$$\phi(\alpha, \boldsymbol{p}) = \Phi \begin{pmatrix} \boldsymbol{P}\left(\boldsymbol{x}^{(k)} + \alpha\, \boldsymbol{d}_{\boldsymbol{x}}^{(k)}, \boldsymbol{l}\,, \boldsymbol{u}\right), \\ \boldsymbol{P}\left(\boldsymbol{z}^{(k)} + \alpha\, \boldsymbol{d}_{\boldsymbol{z}}^{(k)}, 0\,, \infty\right), \\ \boldsymbol{\mu}^{(k)} + \alpha\, \boldsymbol{d}_{\boldsymbol{\mu}}^{(k)}, \\ \boldsymbol{P}\left(\boldsymbol{\lambda}^{(k)} + \alpha\, \boldsymbol{d}_{\boldsymbol{\lambda}}^{(k)}, 0\,, \infty\right), \\ \boldsymbol{P}_{\boldsymbol{h}}^{(k)}, \\ \boldsymbol{P}_{\boldsymbol{g}}^{(k)} \end{pmatrix} \tag{4.65}$$

Let

$$\alpha^{(k)} = \bar{\alpha} \tag{4.66}$$

and update the design variables, slack variables, and Lagrange multipliers:

$$\boldsymbol{x}^{(k+1)} = \boldsymbol{P}\left(\boldsymbol{x}^{(k)} + \alpha^{(k)} \boldsymbol{d}_{\boldsymbol{x}}^{(k)}, \boldsymbol{l}\,, \boldsymbol{u}\right) \tag{4.67}$$

$$\boldsymbol{z}^{(k+1)} = \boldsymbol{P}\left(\boldsymbol{z}^{(k)} + \alpha^{(k)} \boldsymbol{d}_{\boldsymbol{z}}^{(k)}, 0\,, \infty\right) \tag{4.68}$$

$$\boldsymbol{\mu}^{(k+1)} = \boldsymbol{\mu}^{(k)} + \alpha^{(k)} \boldsymbol{d}_{\boldsymbol{\mu}}^{(k)} \tag{4.69}$$

$$\boldsymbol{\lambda}^{(k+1)} = \boldsymbol{P}\left(\boldsymbol{\lambda}^{(k)} + \alpha^{(k)} \boldsymbol{d}_{\boldsymbol{\lambda}}^{(k)}, 0\,, \infty\right) \tag{4.70}$$

where $\boldsymbol{P}$ represents the projection operator, i.e. eqn. 2.1.

Note: $\boldsymbol{x}$, $\boldsymbol{z}$, and $\boldsymbol{\lambda}$ should be updated using the projection operator at each line search iteration to ensure they remain within their bounds. Calculating $\phi(0,\boldsymbol{p})$ does not incur additional objective or constraint function evaluations since these are known from the previous iteration. Similarly, calculating $\phi'(0,\boldsymbol{p})$ does not incur additional objective or constraint gradient evaluations since these are known from the previous iteration.

Step 12: Evaluate $f^{(k+1)}$, $\boldsymbol{h}^{(k+1)}$, $\boldsymbol{g}^{(k+1)}$, $\nabla f^{(k+1)}$, $\boldsymbol{J}_{\boldsymbol{h}}^{(k+1)}$, and $\boldsymbol{J}_{\boldsymbol{g}}^{(k+1)}$. Note, $f^{(k+1)}$, $\boldsymbol{h}^{(k+1)}$, and $\boldsymbol{g}^{(k+1)}$ need only be set since they were evaluated at the final line search iteration, i.e. when $\phi(\bar{\alpha})$ was evaluated.

Step 13: Set L-BFGS vectors:

$$\boldsymbol{s}^{(k)} = \boldsymbol{x}^{(k+1)} - \boldsymbol{x}^{(k)} \tag{4.71}$$

$$\boldsymbol{y}^{(k)} = \nabla_x L(\boldsymbol{x}^{(k+1)}, \boldsymbol{\mu}^{(k+1)}, \boldsymbol{\lambda}^{(k+1)}) - \nabla_x L(\boldsymbol{x}^{(k)}, \boldsymbol{\mu}^{(k+1)}, \boldsymbol{\lambda}^{(k+1)}) \tag{4.72}$$

For $k > m$, $\mathbf{s}^{(k-m)}$ and $\mathbf{y}^{(k-m)}$ are discarded and $\mathbf{s}^{(k)}$ and $\mathbf{y}^{(k)}$ are stored in their place, respectively. Also, if storing **s** and **y** vectors in $m_c \times n$ matrices **S** and **Y**, the correction location corresponding to the kth iteration, $j(k)$, can be calculated using eqn. 2.8. Last, to ensure that the L-BFGS approximation of the *inverse Hessian* of the *Lagrangian* function maintains positive definiteness, discard any $\mathbf{s}^{(k)}$ and $\mathbf{y}^{(k)}$ not satisfying:

$$\mathbf{s}^{(k)^{\boldsymbol{T}}} \mathbf{y}^{(k)} > 2.2\times10^{-1} \; \left\|\mathbf{y}^{(k)}\right\|^2 \tag{4.73}$$

Step 14: Set $\boldsymbol{x}^{(k)} = \boldsymbol{x}^{(k+1)}$, $\boldsymbol{z}^{(k)} = \boldsymbol{z}^{(k+1)}$, $\boldsymbol{\mu}^{(k)} = \boldsymbol{\mu}^{(k+1)}$, $\boldsymbol{\lambda}^{(k)} = \boldsymbol{\lambda}^{(k+1)}$, $f^{(k)} = f^{(k+1)}$, $\boldsymbol{h}^{(k)} = \boldsymbol{h}^{(k+1)}$, $\boldsymbol{g}^{(k)} = \boldsymbol{g}^{(k+1)}$, $\nabla f^{(k)} = \nabla f^{(k+1)}$, $\boldsymbol{J}_{\boldsymbol{h}}^{(k)} = \boldsymbol{J}_{\boldsymbol{h}}^{(k+1)}$, $\boldsymbol{J}_{\boldsymbol{g}}^{(k)} = \boldsymbol{J}_{\boldsymbol{g}}^{(k+1)}$, and $k = k + 1$. Go to Step 3.

To summarize, at each iteration, the above algorithm calculates second order design variable, slack variable, and Lagrange multiplier search directions and then determines a distance to step in those directions. The above algorithm is unique in that it decouples the search direction QP sub-problem, typical of SQP methods, into three simple parts using ideas from augmented Lagrangian methods. These three simple parts

need only require the solution of two positive-definite linear systems. One of which can be solved using the efficient and robust L-BFGS two-loop recursion. It is true that Interior Point methods need only require the solution of one linear system to calculate a search direction. However, that linear system is much larger than both linear systems described here combined and requires specialized, more computationally expensive, linear system solvers that can handle indefinite matrices. The above algorithm is also novel in the way it transforms the equality constrained penalty parameter sub-problem into a simpler, yet equivalent, bound constrained sub-problem. Finally, the above algorithm effectively extends bound-constrained optimization concepts to generally constrained algorithms. This allows for the explicit handling of bound-constraints and the removal of active variables from the problem entirely, potentially greatly reducing the dimensionality of the search direction sub-problem. Using bound-constrained concepts to identify active slack variables provided an intuitive and innovative means of identifying the active-constraint-set at each iteration. The result of these innovations is an algorithm that: (1) has convergence properties competitive with its peers, (2) can handle problems with many variables efficiently, (3) can handle problems with many simple bound constraints efficiently, (4) can handle problems with many general constraints efficiently, (5) is more computationally efficient than its peers.

4.1.7 Flow Diagram for Novel Algorithm

The overall flow of the novel augmented Lagrangian algorithm is shown in Figure 4.2 and involves: initializing algorithmic parameters; ensuring all design variables are on or within their bounds; evaluating the problem functions at the current design; calculating initial slack variable values; calculating the maximum feasibility violation; identifying the inactive-variable-set and active-constraint set; calculating the maximum optimality violation; checking if the current point is a KKT point; calculating a set of second order

search directions; updating the penalty parameters as necessary to ensure the directional derivative of the line search function is sufficiently negative; finding a step size satisfying Armijo's sufficient decrease rule; updating the design, slack, and Lagrange multiplier variable values for the next iteration; evaluating the problem functions for the next iteration; updating the L-BFGS correction vectors.

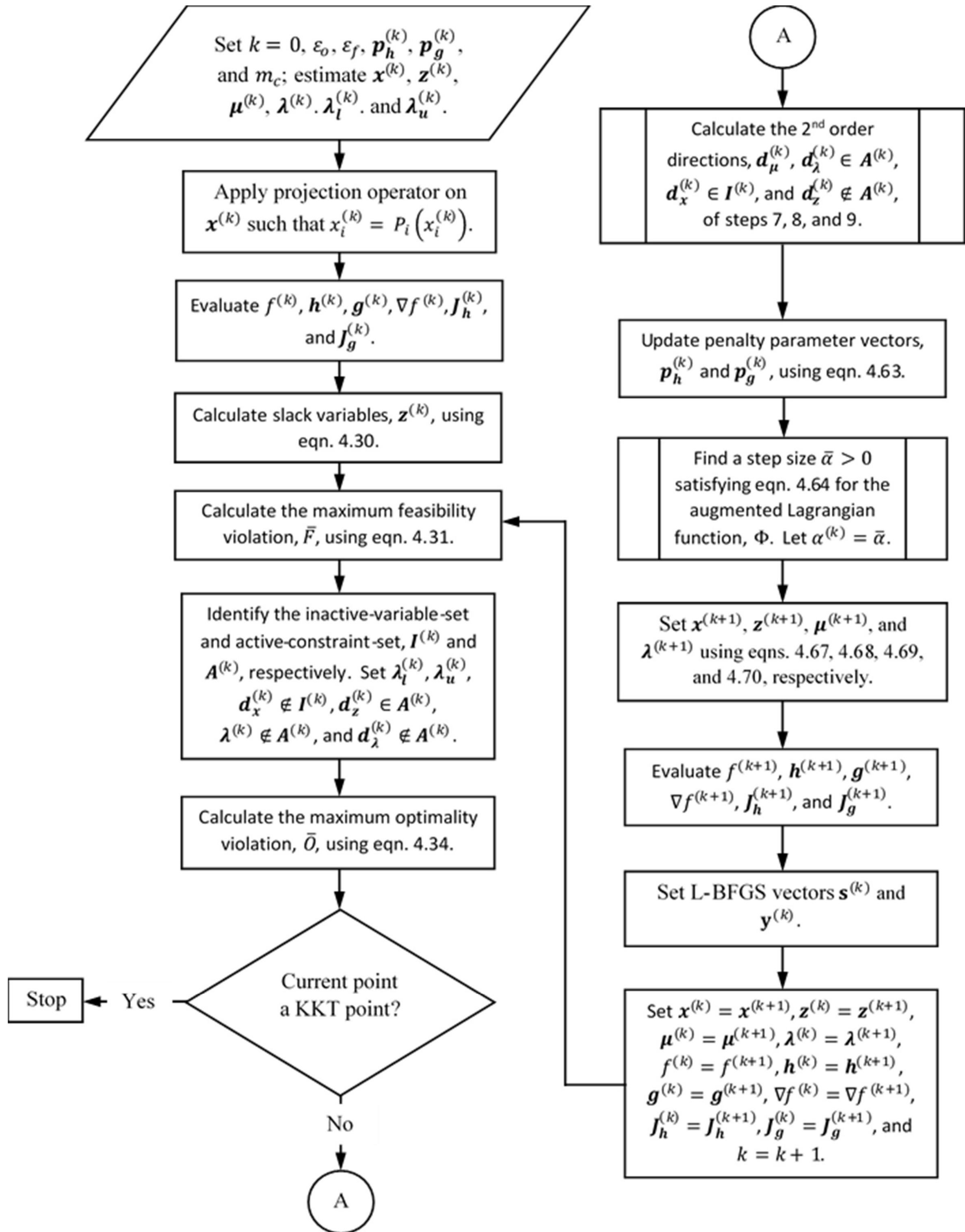

Figure 4.2 Novel Augmented Lagrangian Algorithm Flow Diagram.

4.1.8 Novel Algorithm Numerical Results

Table 4.1 compares the performance of LASO's Augmented Lagrangian (AL) implementation, SNOPT's Sequential Quadratic Programming (SQP) implementation, IPOPT's Interior Point (IP) implementation, LOQO's IP implementation, and LANCELOT's AL implementation on 15 constrained problems from the Hock-Schittkowski test problem collection. The SNOPT results are those obtained by Gill, Murray, and Saunders (2002). The IPOPT and LOQO results are those obtained by Wächter and Biegler (2006). The LANCELOT results are those obtained by Bongartz, Conn, Gould, Saunders, and Toint (1997). In terms of objective function evaluations, LASO outperformed SNOPT on 7 of 15 problems. SNOPT outperformed LASO on 6 of 15 problems. LASO and SNOPT performed the same on 2 of 15 problems. LASO outperformed IPOPT on 6 of 15 problems, IPOPT outperformed LASO on 8 of 15 problems. LASO and IPOPT performed the same on 1 of 15 problems. LASO outperformed LOQO on 9 of 15 problems. LOQO outperformed LASO on 5 of 15 problems. LASO and LOQO performed the same on 1 of 15 problems. LASO outperformed LANCELOT on 11 of 15 problems. LANCELOT outperformed LASO on 4 of 15 problems. In general, LASO performed comparably to or better than SNOPT, IPOPT, LOQO, and LANCELOT, which are all considered to be state-of-the-art implementations of their underlying algorithms. This suggests that AL algorithms can be made to be competitive with state-of-the-art SQP and IP algorithms.

Table 4.1 LASO and SNOPT Performance on Hock-Schittkowski Test Problems.

				LASO	SNOPT	IPOPT	LOQO	LANCELOT
				AL	SQP	IP	IP	AL
No.	Problem	n	ncnln	Iter (F)	Iter (F)	Iter (F)	Iter (F)	Iter (F)
1	HS006	2	1	8 (13)	4 (8)	5 (7)	10 (11)	(57)
2	HS010	2	1	11 (12)	12 (17)	15 (17)	15 (16)	(18)
3	HS015	2	2	5 (10)	2 (5)	17 (22)	30 (31)	(47)
4	HS018	2	2	16 (24)	13 (24)	15 (19)	15 (16)	(92)
5	HS021	2	1	1 (3)	2 (6)	8 (9)	12 (13)	(2)
6	HS022	2	2	5 (6)	0 (3)	6 (7)	9 (10)	(10)
7	HS028	3	1	3 (5)	3 (7)	1 (2)	9 (10)	(4)
8	HS033	3	2	9 (10)	2(6)	13 (16)	11 (12)	(13)
9	HS035	3	1	8 (11)	7 (10)	7 (8)	10 (11)	(7)
10	HS039	4	2	13 (16)	16 (28)	13 (14)	14 (15)	(21)
11	HS040	4	3	7 (8)	5 (9)	3 (4)	8 (9)	(11)
12	HS043	4	3	9 (12)	9 (14)	9 (10)	15 (42)	(23)
13	HS050	5	3	16 (22)	19 (27)	9 (10)	16 (17)	(13)
14	HS055	6	6	2 (3)	0 (3)	8 (9)	10 (11)	(7)
15	HS071	4	2	7 (9)	5 (8)	8 (9)	13 (14)	(16)

CHAPTER V

PARALLELIZATION

Indeed, researchers have parallelized function and gradient evaluations and linear algebra calculations and seen good results. A few researchers have even implemented parallelization during the line search to either identify an acceptable step size more quickly or more accurately. These works are commendable and offer many sound strategies for parallelizing optimization software, but there is still room for improvement on these strategies and for creative new strategies to be developed. Furthermore, a robust and efficient commercial code that takes advantage of parallelization and is designed to be run on a desktop workstation is lacking.

5.1 Literature Review

Eldred and Hart (1998) categorized the opportunities for parallelization in optimization into four main-areas: algorithmic coarse-grained parallelism (e.g. parallelization of numerical finite difference gradients, analytical gradients, and/or line search function evaluations), algorithmic fine-grained parallelism (e.g. parallelization of the linear algebra calculations that occur in various parts of the algorithm), function evaluation coarse-grained parallelism (e.g. parallelization of the individual objective and constraint functions themselves by concurrently calculating their separable parts), function evaluation fine-grained parallelism (e.g. parallelization of the individual steps of a single analysis code that must be run to evaluate objective and constraint functions). They note the limitations of single-level parallelism (i.e. optimizers that take advantage of only one of the opportunities listed above) and provide motivation for the use of multi-level parallelism in the DAKOTA (Design Analysis Kit for Optimization and Terascale Applications) toolkit, which integrates various commercial and in-house codes into more advanced optimization/solution strategies that are well suited for parallel computation on networks of workstations and the Intel TeraFLOPS supercomputer. One important

finding of their analyses, which is relevant to the present work, is that preference should be given to coarse-grained parallelism, whenever possible, before considering fine-grained parallelism.

Eldred and Schimel (1999) built upon the work of Eldred and Hart (1998) and proposed a four-level parallelization scheme that achieved near linear scaling (i.e. speedup of the software increased almost linearly with the number of processors) on massively parallel computers. Their scheme took advantage of algorithmic coarse-grained parallelism through concurrent function evaluations, speculative parallel or gradient based line searches, and a concurrent iterator strategy. It also took advantage of function evaluation coarse-grained parallelism by providing an extended interface that allowed individual function evaluations to be broken into a serial preprocessing portion, concurrent analysis portions, and a serial postprocessing portion. Speculative parallel line searches speculate that the current step size will be successful and proceed with calculating the gradient concurrently with the line search function at the trial step. Gradient-based line searches calculate the gradient concurrently with the line search function as well, but use the gradient information to develop a cubic approximation of the merit function for use in line searches. Numerical results for a test problem with the number of processors equivalent to the number of required gradient and function evaluations showed a 4.80 times speedup of the gradient-based strategy over the serial code and a speedup of 4.12 times for the speculative strategy. In general, the gradient-based method is expected to be most efficient in situations where the number of processors is greater than or equal to the number of required gradient and function evaluations, since it often allows convergence in fewer iterations. The speculative method is expected to be most efficient otherwise, since any gradient and function evaluations exceeding the number of processors can be terminated if the trial step is rejected. A concurrent iterator strategy essentially runs multiple optimizations simultaneously that are controlled by a parallel branch and bound, multi-start local

search, or island-model genetic algorithm optimization method. Eldred and Schimel used a parallel branch and bound method in their concurrent iterator strategy.

Eldred, Hart, Bohnhoff, Romero, Hutchinson, and Salinger (1996) investigated various concurrent iterator or hybrid optimization strategies for use in the DAKOTA toolkit. They developed these strategies in the C++ programming language in order to take advantage of object-oriented software design, which lends itself naturally to parallelization. The hybrid strategies developed combine global Genetic Algorithm (GA) methods, local Nonlinear Programming (NLP) methods, and Coordinate Pattern Search (CPS) methods in two-pass sequences, which take advantage of the strengths of each method to improve robustness and efficiency of the overall optimization process: GA/NLP, GA/CPS, CPS/NLP. The switch between methods occurs when progress slows or when a method reaches its budget on function evaluations. For the problems studied, both GA/NLP and GA/CPS hybrids outperformed the standard NLP and CPS methods, respectively. GA/CPS outperformed GA/NLP in converging to a local minimum. The CPS/NLP hybrid proved to be an efficient and accurate local search technique since it converged in half the time as NLP alone and to a more optimal result than CPS could achieve alone. Areas identified for potential future research included the development of improved algorithm switching criteria and the development of a three-pass GA/CPS/NLP hybrid.

Ghattas and Orozco (1997) address parallelization in the situation that the required analysis is so large that one function evaluation consumes the majority of the available computing resources. They primarily focus on this issue as it relates to shape optimization, but really this issue could be of concern, to varying degrees, in any problem with expensive function evaluations. Their solution to not being able to execute function and gradient evaluations in parallel was to, instead, focus on parallelizing the linear algebra in various parts of the algorithm. The final result of their work was a parallel reduced hessian SQP method for shape optimization.

Nash and Sofer (1989) proposed a block truncated-newton method that took advantage of parallelism by calculating an additional set of gradients at each major iteration and using those gradients to obtain a finite difference approximation of the Hessian. Nash and Sofer (1991) extended their previous work by introducing multiple enhancements to their algorithm. Among these enhancements, was a parallel line search that evaluated a set of trial steps in parallel and took the one that gave the minimal objective function value and satisfied an Armijo type rule. Byrd, Schnabel, and Schultz (1988a) proposed a set of algorithms that take advantage of any extra processors to calculate extra finite difference gradients in carefully chosen directions. These gradients are then used to improve the current approximation of the Hessian. These methods are essentially interpolations between Newton and quasi-Newton methods. The most promising of these algorithms appeared to be one that used BFGS updates followed by a finite difference update of part of the Hessian. This algorithm performed better than the BFGS algorithm alone when one extra processor was available to calculate extra finite difference gradients. Thus, giving it a convergence rate somewhere between super-linear and quadratic. The primary drawback of this method lies in the seeming inability to intuitively identify directions to calculate the extra finite difference gradients in.

Byrd, Schnabel, and Shultz (1988b) offered an improvement to their BFGS / finite difference method, which takes advantage of the situation where gradients are calculated at each trial step in parallel with the line search function and can be used to perform intermediate updates to the BFGS search direction before a successful step has been identified. While this improvement only affects iterations where the initial step is not accepted, a 3-12% reduction in the average number of trial point function evaluations was observed for the test problems considered.

5.2 Parallel L-BFGS 2-Loop Recursion

So far parallelization schemes have, for the most part, either focused on improving computational efficiency or convergence efficiency. The one exception being the DAKOTA toolkit, which incorporates a variety of parallelization strategies, but is primarily designed for use on very large supercomputers where multi-level parallelization is possible. Examples of improving computational efficiency seen so far include: (1) concurrent function evaluations, (2) speculative parallel line searches, (3) extended interfaces for parallelization of individual function evaluations, and (4) linear algebra parallelization. Examples of improving convergence efficiency seen so far include: (1) gradient based line searches, (2) concurrent iterator strategies, (3) hybrid algorithm strategies, and (4) finite difference enhancements to the Hessian.

In chapter 7, a holistic strategy for parallelization, which incorporates multiple of the strategies listed above, is proposed as an area for further research. Here, acceleration of the optimization process itself, not function and gradient evaluations, is the goal. Furthermore, of primary interest, are high-dimensional problems where the linear algebra calculations at each iteration tend to dominate solution times. Therefore, this chapter focuses on studying Central Processing Unit (CPU) versus Graphics Processing Unit (GPU) parallelization of the algorithm's linear algebra operations, to assess which offers the greatest promise for real time optimization. Specifically, a parallel L-BFGS 2-loop recursion is implemented on the CPU and GPU and its performance studied. This part of the algorithm was chosen to be parallelized after profiling of the serial code suggested that 25% of solution time is spent here. The parallel recursion that follows is a modification of the serial recursion of section 2.3.1.1 where high-dimensional dot products and other operations with equals signs prefixed by a colon, i.e. ≔, are carried out in parallel.

$$\mathbf{q} \coloneqq \boldsymbol{\nabla f}_{I_k}^{(k)}$$

$$\text{LIMIT} = \begin{cases} 0 & if\ k < m \\ k - m & if\ k \geq m \end{cases}$$

for $i = k - 1 : -1 : i \geq \text{LIMIT}$

set j to the correction location for the ith iteration.

$$\text{store}\ \rho_i \coloneqq \frac{1}{\mathbf{s}_{I_k}^{(j)} \cdot \mathbf{y}_{I_k}^{(j)}}$$

$$\text{store}\ \alpha_i \coloneqq \rho_i \left(\mathbf{s}_{I_k}^{(j)} \cdot \mathbf{q} \right)$$

$$\mathbf{q} \coloneqq \mathbf{q} - \alpha_i \mathbf{y}_{I_k}^{(j)}$$

end

$$H_0^{(k)} \coloneqq \frac{\mathbf{s}_{I_k}^{(k-1)} \cdot \mathbf{y}_{I_k}^{(k-1)}}{\mathbf{y}_{I_k}^{(k-1)} \cdot \mathbf{y}_{I_k}^{(k-1)}}$$

restrict $H_0^{(k)}$ such that $10^{-3} \leq H_0^{(k)} \leq 10^3$

$$\mathbf{r} \coloneqq H_0^{(k)} \mathbf{q}$$

for $i = \text{LIMIT} : +1 : i \leq k - 1$

set j to the correction location for the ith iteration.

$$\beta \coloneqq \rho_i \left(\mathbf{y}_{I_k}^{(j)} \cdot \mathbf{r} \right)$$

$$\mathbf{r} \coloneqq \mathbf{r} + \mathbf{s}_{I_k}^{(j)} (\alpha_i - \beta)$$

end

5.3 Parallelization on the Central Processing Unit (CPU)

Parallelization on the CPU has been implemented using the Armadillo library built with the OpenBLAS library. Armadillo is a high-quality open-source template-based C++ library for linear algebra and is developed by Conrad Sanderson and Ryan Curtin (2016). OpenBLAS is an optimized open-source implementation of the BLAS (Basic Linear Algebra Subprograms) library and is developed by Zhang Xianyi (2012).

OpenBLAS is multi-threaded and supports up to 16 parallel threads. The number of threads may be specified at run time by setting environment variables or by calling special built-in functions, which OpenBLAS provides. Tests were run using 1, 4, and 8 threads. To conduct each test, the variably dimensioned extended Rosenbrock function was minimized 100 times and the average solution time from those 100 runs was taken as the solution time for that test. To make solution times problem independent, solution time was taken as the total time minus the time spent evaluating problem functions. Tests were run on 100, 500, and 1,000 variable extended Rosenbrock functions. Tests were run on an Intel Xeon E5 1620 v2 3.70GHz CPU, which has 4-cores and 8-threads.

Table 5.1 CPU Parallelized LASO Performance on Extended Rosenbrock Function.

	Solution time [seconds]		
Number of variables	1 thread	4 threads	8 threads
100	0.04914	0.05573	0.05383
500	1.05113	1.06585	1.04524
1,000	4.70435	4.78944	4.6683

For the 100-variable problem, the serial algorithm, i.e. 1-threaded algorithm, outperformed the 4-threaded and 8-threaded algorithms. Though the 8-threaded algorithm did outperform the 4-threaded algorithm. This indicates that the acceleration due to parallelization is not yet large enough to offset the communication overhead incurred sending data to and launching multiple threads. For the 500-variable problem, the 8-threaded algorithm was 0.56% faster than the serial algorithm, but the 4-threaded algorithm was still slower. This suggests that at 500-variables or more, the acceleration due to parallelization on 8 or more threads starts to offset the communication overhead. For the 1,000-variable problem, the 4-threaded algorithm was once again slower than the serial code and the 8-threaded algorithm was 0.77% faster. Overall, margins between the

serial and multi-threaded solution times were small. However, trends in the data above, at least for the 8-threaded algorithm, suggest that margins should increase as problem size increases and the communication to computation ratio decreases. Also, margins should increase as more of the algorithm is parallelized.

5.4 Parallelization on the Graphics Processing Unit (GPU)

During the completion of this research, massively parallel GPUs, which can carry out scientific computations, have substantially decreased in price and now come standard on many desktop workstations. NVIDIA's CUDA parallel computing platform and Application Programming Interface (API) currently leads in popularity. However, OpenCL is quickly gaining in popularity and supports parallelization on both NVIDIA and non-NIVIDIA GPUs as well as a diverse array of CPUs. WebCL, while still nascent, is exciting in that it would allow parallel scientific computation in a web browser. However, currently no mainstream browsers support it. Current generation NVIDIA GPUs can have up to 3,500 CUDA cores. These GPUs can also be networked together. Amazon Web Services rents servers by the hour with up to 16 networked GPUs. These servers can have up to 40,000 CUDA cores.

To assess the potential of GPUs for real time optimization, the parallel L-BFGS 2-loop recursion of section 5.2 was implemented using the cuBLAS or CUDA Basic Linear Algebra Subroutines library, which is described in the cuBLAS CUDA Toolkit Documentation by NVIDIA (2017). Then the testing methodology described in the previous section was carried out in serial on an Intel Xeon E5 1620 v2 3.70GHz CPU and in parallel on a NVIDIA Quadro K2000 GPU with 384 CUDA cores.

Table 5.2 GPU Parallelized LASO Performance on Extended Rosenbrock Function.

	Solution time [seconds]	
Number of variables	1 thread	384 CUDA cores
100	0.04914	2.188
500	1.05113	11.308
1,000	4.70435	25.314

The GPU parallelized algorithm was much slower than the serial algorithm and all the CPU parallelized algorithms. This indicated a very high communication to computation ratio. The likely cause: the communication cost to send 3 n-dimensional vectors to the GPU and return 1 n-dimensional vector from the GPU at every optimization iteration, since data transfer to and from the GPU is slow. However, upon further investigation, the primary cause became apparent: the cost to coordinate work among cores on the GPU. All the operations being parallelized in the algorithm of section 5.2 involve 1-dimensional vectors. The GPU divides these vectors into sections and assigns each core a section to work on. If vectors are small relative to the number of cores on the GPU, then relatively little computation will be assigned to each core and the cost to coordinate the work dominates.

Despite negative test results, the findings above suggest that GPU parallelization may still be an attractive option for achieving real time optimization when: (1) problem functions and all high-dimensional data / linear algebra operations can be stored / computed on the GPU to minimize communication overhead between the CPU and GPU, (2) the problem dimensionality is very high relative to the number of cores on the GPU, say hundreds of thousands for a few hundred core GPU, or (3) a custom kernel, i.e. the program operating on a section of data on each core of the GPU, can be written that combines multiple sequential operations into one to maximize the computation on each core. The third option is most promising for problems that are not massively scaled.

CHAPTER VI

DIGITAL HUMAN WALKING PROBLEM

In order to test the constrained algorithm's applicability to digital human simulation, a digital human walking problem is solved. Specifically, the digital human walking problem formulated by Xiang et al. (2009), which simulates one step of human walking, is solved. As mentioned in the introductory remarks, even this seemingly simple task is a challenging problem requiring a minimum of 330 design variables and 1036 nonlinear constraints. A summary of the design variables, objective function, and constraints implemented by Xiang et al. (2009) is presented in the sections that follow. For details beyond those discussed here, see Xiang et al. (2009). Lastly, numerical results showing LASO's performance on the digital human walking problem are reported and LASO's applicability to digital human simulation is established.

6.1 Design Variables

As mentioned in the introductory remarks, the spatial digital human model developed at The University of Iowa consists of 55 Degrees of Freedom (DOF) of which 6 are virtual DOFs that represent global translation and rotation and 49 are physical joint angle DOFs that represent local joint rotations. The 49 physical joint angle DOFs consist of 12 DOFs at the 4 joints representing the spine, 6 DOFs at the 2 joints representing the hips, 2 DOFs at the 2 joints representing the knees, 4 DOFs at the 2 joints representing the ankles, 2 DOFs at the 2 joints representing the 2 forefeet, 4 DOFs at the 2 joints representing the clavicle, 6 DOFs at the 2 joints representing the shoulders, 4 DOFs at the 2 joints representing the elbows, 4 DOFs at the 2 joints representing the wrists, 3 DOFs at the joint representing the lower neck, and 2 DOFs at the joint representing the upper neck. Time histories of these 55 DOFs make up the design variables. Thus there are an infinite number of design variables. However, the design variables are represented by

cubic B-splines to transform the problem to a finite dimensional one. The control points of each B-spline are the final design variables that are finite in number.

In the human walking simulation problem formulated by Xiang et al. (2009), there are 55 DOFs total, as shown in Figure 5-1 below, that are represented by cubic B-splines having 6 control points each. This makes for 330 design variables in all. However, 17 of the 49 physical joint angle DOFs do not play a significant role and can be frozen at their neutral angles, leaving 228 design variables to be determined by the optimization process. It is interesting to note that freezing DOFs is accomplished by setting the lower and upper bounds on the DOFs, or more specifically the control points, equal to one another. This simple solution avoids redefinition of the skeletal model, which would be a tedious task. The specific joints frozen include the wrist joints, clavicle joints, lower neck joint, and two spine joints. These joints are those enclosed in dashed lines in Figure 5-1 below.

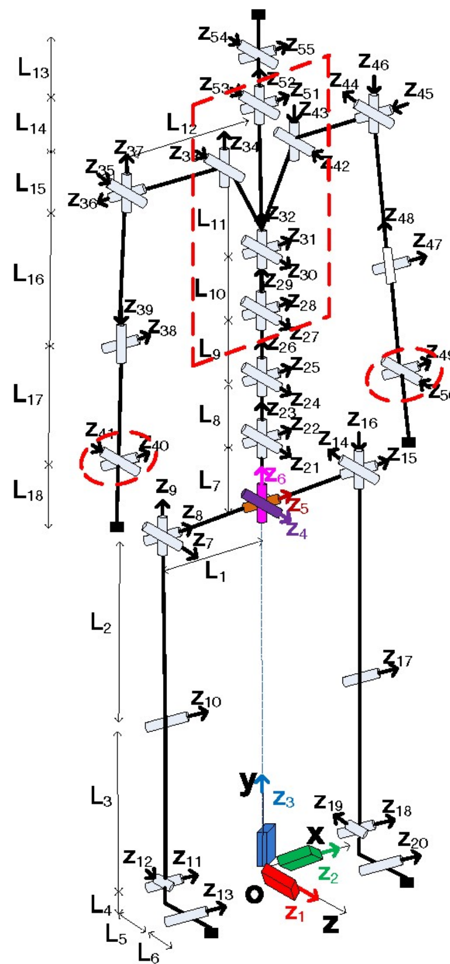

Figure 6.1 55-DOF Digital Human Model from Xiang et al. (2009).

6.2 Objective Function

The objective function chosen by Xiang et al. (2009) is the time integral of the squares of all of the joint torques or, put succinctly, the dynamic effort. This choice of objective assumes that humans move in a way that minimizes energy expended over the entire duration of a motion and guides the optimization process towards values of the design variables that achieve this goal. Different joints have different limits on the maximum joint torque that can be produced in the joint. Therefore, the square of each joint torque is normalized by the square of the maximum joint torque for each joint

before integrating over the duration of a task. This normalization of the joint torques keeps joints with very high maximum joint torques from dominating the objective, which could lead to joints with lower maximum joint torques more or less being ignored.

The joint torques in the objective can be calculated explicitly in terms of the knot vector, i.e. time discretization, and the control points or design variable vector, i.e. joint angle values corresponding with the time discretization. Specifically, joint torques are calculated using a backward recursive dynamics procedure, which depends on the following kinematic state variables for each joint: angular displacement, angular velocity, angular acceleration. These state variables are calculated using a forward recursive kinematics procedure, which depends on the knot vector and control points vector. Additionally, sensitivity or gradient information of the joint torques with respect to the kinematic state variables is calculated using a recursive Lagrangian dynamics formulation. This sensitivity information, in turn, is used along with the chain rule to obtain sensitivity information of the joint torques with respect to the control points, which the optimization process requires. Recursive kinematic and Lagrangian formulations were adopted over the regular Lagrangian formulation for their extremely low $O(n)$ computational costs relative to the regular method's $O(n^4)$ computational cost. Another important benefit of this approach, is that it avoids integration of the equations of motion, which would be a complicated and cumbersome task to perform at every optimization iteration.

The explicit expression for the equations of motion or joint actuation torque resulting from the aforementioned approach includes terms for inertia and Coriolis torque, gravity load torque, external forces torque, and external moments torque. Therefore, in addition to kinematic state variables, the torque expression also depends on the mass and inertia properties of the links connecting the joints. The required anthropometric data needed to calculate these properties was generated using a commercial software package for a skeletal model representing a 50th percentile male.

6.3 Constraints

Optimization constraints are used to enforce the real world limitations that come into play when simulating human walking. These constraints generally fall into one of two categories: time-dependent constraints, time-independent constraints. Time-dependent constraints are imposed throughout the duration of the motion and include: joint limit constraints that ensure joint angles remain within physical ranges of motion, torque limit constraints that ensure joint torques remain within physical limitations on human strength, ground penetration constraints that ensure feet remain stationary when on the ground, dynamic balance constraints that give the digital human a sense of balance, arm-leg coupling constraints that ensure a realistic arm motion, self-avoidance constraints that ensure body parts do not intersect. Time-independent constraints are imposed at specific points in time during the motion and include: symmetry constraints that ensure a smooth and continuous motion, ground clearance constraints that ensure a realistic gait is achieved that clears the ground, initial and final foot contacting position constraints that tell the optimization process what motion to predict.

6.3.1 Joint Limits

These simple bound constraints on the design variables limit joint angles to their physical ranges of motion, thus avoiding hyperextension. They also may be used to freeze a particular joint at a particular angle by setting the lower and upper bounds for the specified joint equal to the specified angle. In Xiang et al. (2009) realistic joint limits for specific joints are obtained from the biomechanics literature and frozen joints are set at their neutral angles.

6.3.2 Torque Limits

Like joint angles, joint torques are also bounded by their physical limits, which are obtained from the biomechanics literature. Unlike joint angles, however, joint torques are nonlinear expressions in terms of the joint angles and, as such, represent

nonlinear constraints when limits are placed on them. These nonlinear constraints cannot be handled explicitly like simple bound constraints and require the use of constrained optimization techniques.

6.3.3 Ground Penetration

Over the course of a single step there are 4 modes in which combinations of the left and right heel, ball, and toe come into contact with the ground: right toe left heel, left heel left ball, left ball left toe, left toe right heal. During each of these contacting conditions, equality constraints are imposed on the points contacting the ground to ensure their heights and velocities remain at 0. Conversely, inequality constraints are imposed on the height of non-contacting points to ensure they remain above 0.

6.3.4 Dynamic Balance

Dynamic balance is achieved by constraining the Zero Moment Point (ZMP) for the digital human model to fall within a so-called Foot Support Polygon (FSP) defined in the plane of the ground between ground contact points. The ZMP, as the name suggests, is the point in the plane of the ground where the resultant tangential moments of the active forces are 0. Therefore, the location of the ZMP can be calculated relatively straight forwardly by: taking sum of moments equal to 0 along the axes that define the ground plane and solving the resulting equations for the ZMP location along each axis. Constraining the ZMP to fall within the FSP is accomplished by requiring that the cross product of each vector extending from a corner of the FSP to the ZMP with a vector extending from the same corner of the FSP to the adjacent FSP corner in the counterclockwise direction, be negative, since the right hand rule for cross products requires this for any point lying in the polygon.

6.3.5 Arm-Leg Coupling

During walking, it is generally the case that arm swing on one side counteracts leg swing on the other side in order to reduce trunk moment in the vertical direction and thus help balance the upper body. Xiang et al. (2009) enforce this behavior for their one step formulation by considering the left arm and right leg as individual pendulums and placing a coupling constraint on them. Mathematically this is accomplished by requiring that the product of the left arm pendulum's direction of swing and the right leg pendulum's direction of swing is positive, since they move together whether it be forward or backward. It is important to note that, while the swing direction of the left arm and right leg is explicitly enforced by the arm-leg coupling constraint, the swing angle of the left arm and right leg is determined by the optimization process.

6.3.6 Self-Avoidance

In order to avoid contact of the arm with the body, a self-avoidance constraint between the wrist and hip is enforced. Mathematically this is accomplished by placing appropriately sized spheres at the wrist and hip joints and requiring that the distance between their surfaces always be greater than zero. It is interesting to note that this so-called sphere filling technique is general purpose in nature and may be applied wherever it is a requirement that two surfaces not intersect.

6.3.7 Symmetry Conditions

As the current formulation is for one step of human walking, it is desirable that the initial and final postures and velocities satisfy certain symmetry conditions so a smooth continuous walking motion is generated when the motion for the first step is mirrored for the second step. This is accomplished by requiring that the joint angles and joint angle velocities for symmetric joints on opposite sides of the body and for self-symmetric joints be equal to one another at the beginning and end of the simulation.

6.3.8 Ground Clearance

Rather than arbitrarily imposing a maximum height of the swing leg to avoid foot drag motion, Xiang et al. (2009) use information gleaned from biomechanics experts that knee flexion during normal gait at mid-swing is roughly 60 degrees, regardless of age and gender, to create a constraint that avoids foot drag motion in a more natural way. Specifically, they require the joint angle at the knee to be within plus or minus 5 degrees of the 60 degree norm at the time of mid-swing.

6.3.9 Initial and Final Foot Contacting Position

The optimization process predicts the motion between two points, however, certain initial and final conditions must be specified to tell the optimization process what motion to predict. For the problem of simulating one step of human walking, foot contact positions make up these initial and final conditions. The exact positions are calculated based on the step length input by the user.

6.4 Numerical Results

Table 6.1 compares the performance of LASO's Augmented Lagrangian (AL) implementation and SNOPT's Sequential Quadratic Programming (SQP) implementation on the digital human walking problem just described. LASO and SNOPT are very competitive for the first 20 iterations. However, after iteration 20, SNOPT begins to achieve superlinear convergence and reaches the desired 1e-3 feasibility and optimality quickly. LASO, after iteration 20, continues to make slow progress, never achieving superlinear convergence, and fails to reach the desired 1e-3 feasibility and optimality. LASO's failure to reach superlinear convergence, however, is most likely due to a difference in the problem formulation used by LASO and SNOPT. SNOPT benefits when equality constraints can be relaxed, i.e. replaced by inequality constraints with tight lower and upper bounds. LASO, however, does not. Since, in this situation, it is difficult to identify the optimal active set. Research to improve LASO's procedure for identifying

the optimal active set is currently underway and will be published as part of a future paper.

Table 6.1 LASO and SNOPT Walking Problem Performance.

	LASO AL	SNOPT SQP
No. Iterations	50	26
No. Function Evaluations	101	52
Feasibility (Start)	1.7e-002	3.1e-002
Feasibility (Finish)	3.0e-003	8.9e-005
Optimality (Start)	1.4e-001	3.8e-002
Optimality (Finish)	6.0e-002	3.1e-004
Objective (Start)	7.6524520e+001	7.6524520e+001
Objective (Finish)	6.5472358+001	6.29010539+001
Merit (Start)	7.6524520e+001	7.6524520e+001
Merit (Finish)	6.4762536+001	6.3006027+001

CHAPTER VII

CONCLUSIONS AND AREAS FOR FURTHER RESEARCH

7.1 Conclusions

Bound-constrained, step size, and constrained algorithms have been developed that push the state-of-the-art. The bound-constrained formulation presented in chapter II, which used L-BFGS search directions, was found to converge fastest with the pre-conditioned conjugate gradient algorithm coming in second. The innovative efficient backtracking line search procedure proposed in chapter III was found to be quite effective at reducing the number backtracking steps and improving convergence, without incurring additional function evaluations. The novel hybrid line search / trust region approach proposed in chapter III reduces the number of function evaluations on poorly scaled problems, which consistently have poorly scaled L-BFGS Hessian matrices, and has little impact on other problems. The novel augmented Lagrangian algorithm proposed in chapter IV bridges the performance gap between augmented Lagrangian methods and SQP and IP methods. Key innovations include: (1) Decoupling of the search direction sub-problem into two simpler sub-problems capable of being solved directly; (2) Transformation of the equality constrained penalty parameter sub-problem into a simpler, yet equivalent, bound-constrained sub-problem; (3) The extension of bound-constrained optimization concepts to generally constrained algorithms. CPU parallelization was found to be most attractive for real time optimization when problems are not massively scaled, all high-dimensional data/operations cannot be stored/computed on the GPU, and/or custom GPU kernels cannot be written that combine multiple sequential operations into one. Conversely, GPU parallelization was found to be most attractive when problems are massively scaled, all high-dimensional data/operations can be stored/computed on the GPU, and/or custom GPU kernels can be written.

7.2 Areas for Further Research

Over the course of this research, two areas for further research have been identified: (1) Development of a holistic strategy for parallelization, which focuses on improving computational and convergence efficiency. One such strategy is suggested in section 7.2.1; (2) Development of a nonmonotone line search rule, which uses concepts from simulated annealing algorithms to allow more greedy line search steps in early iterations and less greedy line search steps in later iterations. One such strategy is suggested in section 7.2.2.

7.2.1 A Holistic Strategy for Parallelization

The parallelization scheme proposed here is a holistic approach to parallelization that combines many strategies for improving both computational and convergence efficiency in a way that is ideally suited for multi-core desktop workstations. Furthermore, the proposed scheme combines concepts from concurrent iterator and hybrid algorithm strategies to create a new multi-directional search strategy.

In line with the findings of Eldred and Hart (1998), preference is given to coarse-grained parallelism over fine-grained parallelism wherever a conflict between the two occurs. However, to allow for a high degree of coarse- and fine-grained parallelism on a multi-core workstation where only single-level parallelization is possible, coarse- and fine-grained calculations have been segmented to the maximum extent possible to create alternating periods of coarse-grained parallelization and fine-grained parallelization. This makes the resulting parallelization scheme well suited for a multitude of problem scenarios: expensive function evaluations but relatively few design variables, cheap function evaluations but very large number of design variables, expensive function evaluations and very large number of design variables.

The proposed parallelization scheme is shown in Figure 7.1. It starts by evaluating the objective function (fObj), objective gradient (gObj), constraint function (fCon), and constraint Jacobian (gCon) in parallel. Next, if the problem has relatively few design variables then L-BFGS, Preconditioned Conjugate Gradient, and potentially other search directions are calculated in parallel. If the problem has a very large number of design variables then the linear algebra required to update the current approximation of the Hessian and its inverse is calculated in parallel and the individual search directions are calculated sequentially. This so called multi-directional search strategy, which considers multiple search directions at every iteration, will increase the likelihood that the initial line search trial step is accepted and will help ensure that the maximum reduction in the cost function is achieved at every iteration. Last, fObj, gObj, fCon, and gCon are evaluated in parallel at every line search iteration. Only this time, if fObj finishes evaluating and the trial step is rejected, then the remaining gObj, fCon, and gCon processes are killed. The components of gObj, fCon, and gCon that were evaluated are used to perform partial updates to the Hessian and a partial gradient-based line search.

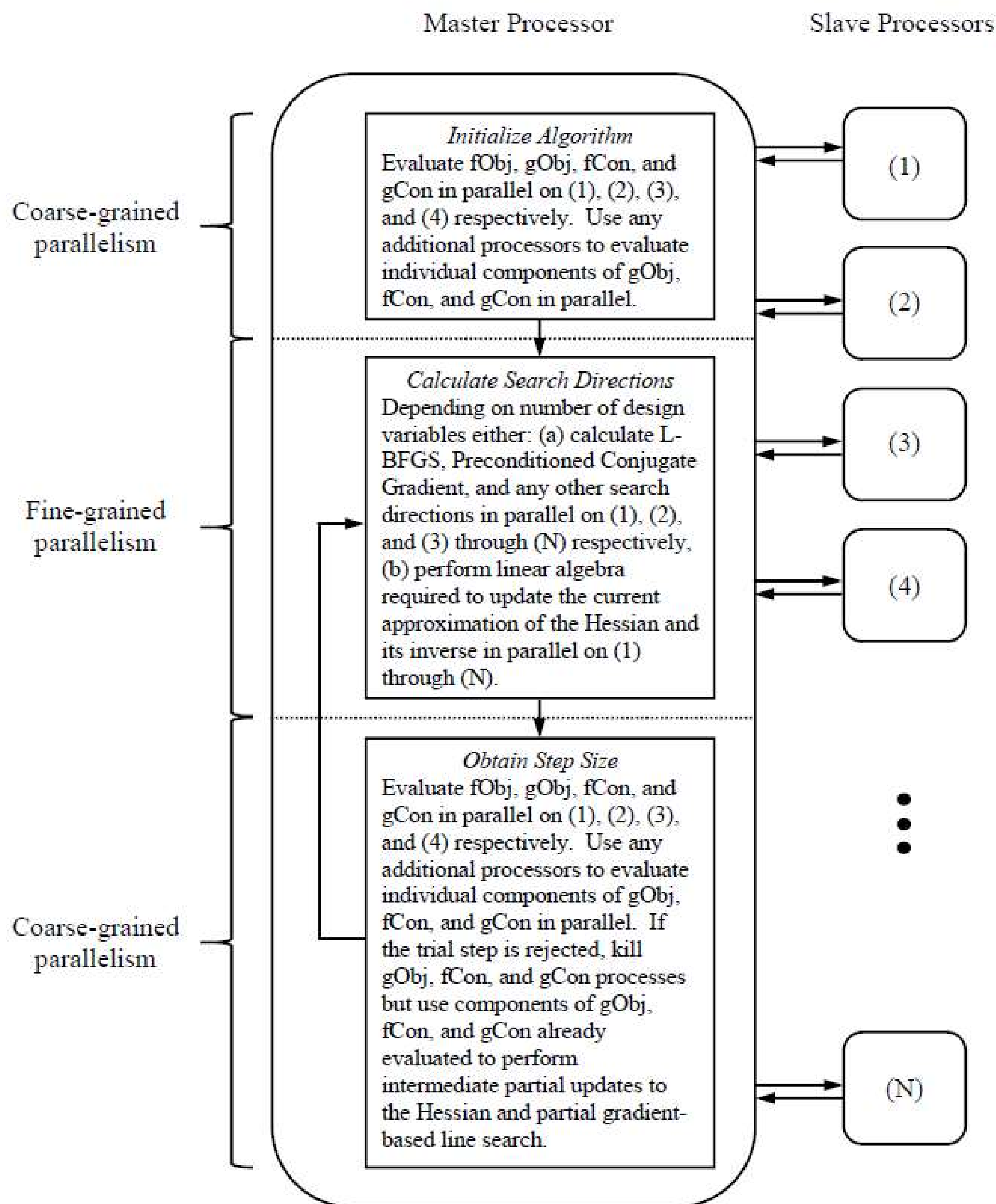

Figure 7.1 A Holistic Parallelization Scheme for Multi-Core Workstations.

7.2.2 Nonmonotone Line Search Rule

Nonmonotone step size schemes have proven to be more efficient, on average, than monotone techniques. Their property of not requiring a decrease at every step also

makes them less likely to converge to a local minimum. Both line search and trust region nonmonotone strategies have been developed and have been proven to be globally convergent. In general, they work by requiring a decrease against some other value than the cost function value at the previous iteration (e.g. the maximum cost function value over the previous 5-iterations). In this section, the literature surrounding nonmonotone techniques is reviewed and a new simulated annealing like nonmonotone rule is proposed.

7.2.2.1 Literature Review

Grippo, Lampariello, and Lucidi's (1986) nonmonotone line search technique is regarded as the traditional nonmonotone technique. They recognized that existing line search techniques, which required a decrease in the cost function at every iteration, could considerably slow the rate of convergence of Newton's method during intermediate stages of the minimization process by selecting step sizes not equal to unity. This behavior was particularly pronounced in problems with narrow curved valleys. Their solution was to require only that Armijo's line search condition be satisfied against the maximum cost function value for the previous ten iterations. Numerical results for their technique were quite promising with some problems exhibiting up to 50% reduction in the number of function evaluations and iterations. They also were able to make the following observations regarding their technique: it was most beneficial during intermediate stages of the minimization process, requiring monotonicity during the first two or three iterations yields the best results, considering the maximum cost function value for the previous five to ten iterations yields the best results.

Zhang and Hager (2004) proposed modifying Grippo, Lampariello, and Lucidi's (1986) approach by requiring that the average of all previous function values decreased instead of the maximum of the most recent function values. Their approach required fewer function and gradient evaluations, on average, than the traditional approach. They

computed their average using a convex combination method that allowed them to adjust the degree of nonmonotonicity. They found that their strategy performed best when the degree of nonmonotonicity was large far away from the optimum and small near the optimum. Their approach implicitly enforces some degree of monotonicity during the first few iterations since it is based on a cumulative average. Cui and Yang (2012) offered a generalization and development of Zhang and Hager's (2004) work that allowed them to prove global convergence under weaker conditions than previous works.

7.2.2.2 A Novel Simulated Annealing Like Rule

Here, a simulated annealing like nonmonotone rule is presented. Similar to Grippo, Lampariello, and Lucidi's (1986) work monotonicity is explicitly enforced during early iterations. Unlike previous works, a rigorous statistical approach is used to allow a high degree of nonmonotonicity far from the optimum and to decrease nonmonotonicity exponentially as the optimum is approached. The overall procedure is shown in Figure 7.2 and involves: performing a backtracking line search for the first two or three bound constrained iterations, setting a time variable for the current bound constrained iteration to monitor progress towards the optimum, calculating the current temperature, checking the strong Wolfe conditions and, if not satisfied on the first line search iteration, checking a simulated annealing like condition.

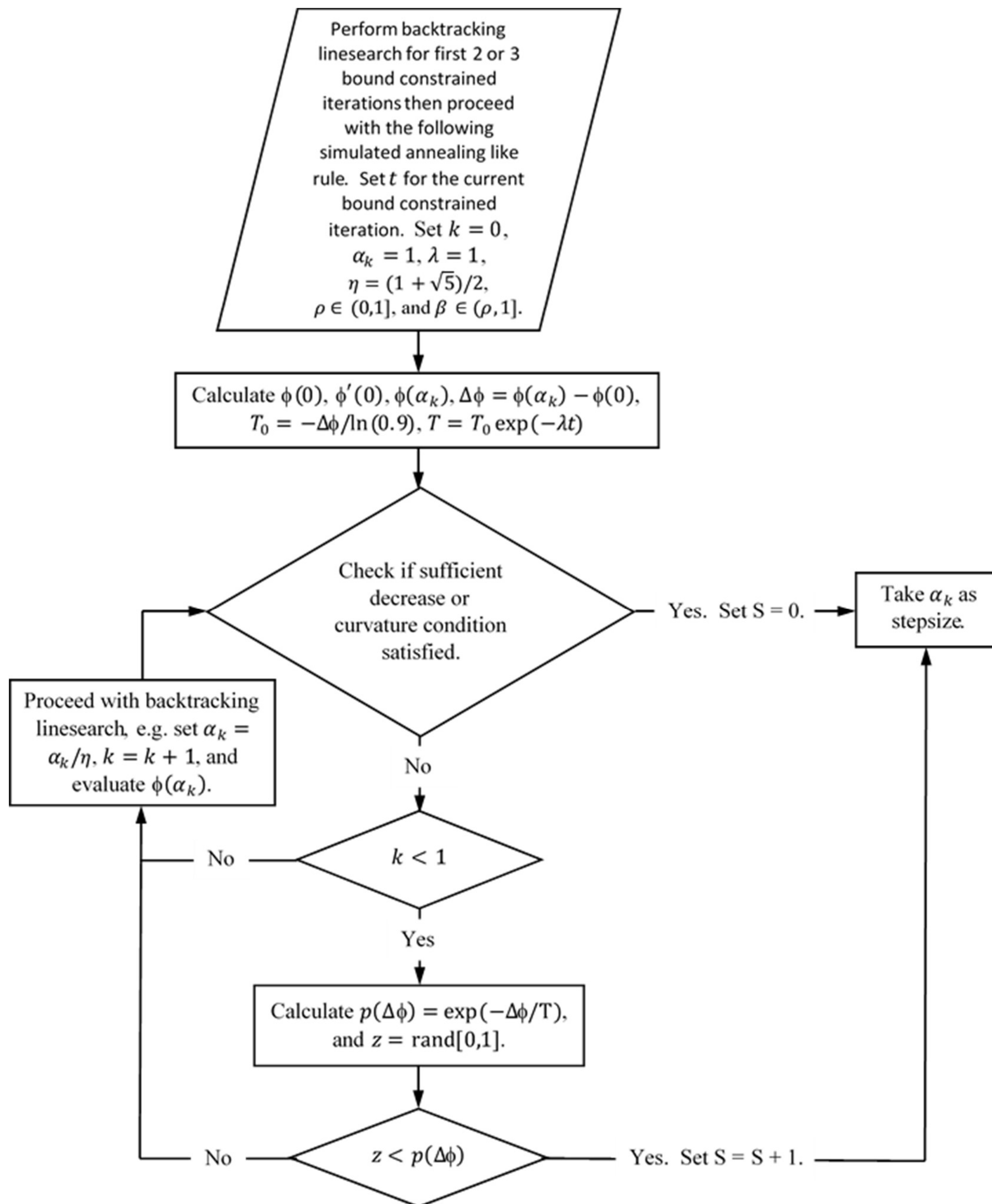

Figure 7.2 Simulated Annealing Like Nonmonotone Rule Flow Diagram.

REFERENCES

Armijo, L. (1966). Minimization of Functions Having Lipschitz Continuous First Partial Derivatives. *Pacific Journal of Mathematics*, 16(1), 1-3.

Arora, J, S. (2017). *Introduction to Optimum Design (4th ed.)*. Cambridge, MA: Academic Press.

Arora, J, S. Chahande, A, l. Paeng, J, K. (1991). Multiplier methods for engineering optimization. *International Journal for Numerical Methods in Engineering,* 32(7), 1485-1525.

Benson, H, Y. Shanno, D, F. Vanderbei, R, J. (2002). A Comparative Study of Large-Scale Nonlinear Optimization Algorithms. *Operations Research and Financial Engineering*, 1-35.

Bongartz, I. Conn, A, R. Gould, N, I, M. Saunders, M. Toint, Ph, L. (1997). A Numerical Comparison Between The Lancelot and Minos Packages For Large-Scale Nonlinear Optimization: The Complete Results, 1-19.

Boyd, S. (2016). Constrained Least Squares, 1-20. https://stanford.edu/class/ee103/lectures/constrained-least-squares/constrained-least-squares_slides.pdf

Buckley, A, G. (1978). A combined conjugate gradient quasi-Newton minimization algorithm. *Mathematical Programming,* 15, 200-210.

Byrd, R, H. Hribar, M, E. Nocedal, J. (1999). An interior point algorithm for large scale nonlinear programming. *SIAM Journal on Optimization*, 9(4), 877-900.

Byrd, R, H. Schnabel, R, B. Schultz, G. A. (1988a). Using parallel function evaluations to improve Hessian approximation for unconstrained optimization. *Annals of Operations Research,* 14, 167-193.

Byrd, R, H. Schnabel, R, B. Schultz, G. A. (1988b). Parallel quasi-Newton methods for unconstrained optimization. *Mathematical Programming: Series A and B,* 42(2), 273-306.

Conn, A, R. Gould, G, I, M. Toint, Ph, L. (1992). LANCELOT: A Fortran Package for Large-Scale Nonlinear Optimization (Release A). *Springer Series in Computational Mathematics*, 17, New York: Springer Verlag.

Cui, Z. Wu, B. (2011). A new self-adaptive trust region method for unconstrained optimization. *Journal of Vibration and Control,* 18(9), 1303-1309.

Cui, Z. Yang, Z. (2012). A nonmonotone line search method and its convergence for unconstrained optimization. *Journal of Vibration and Control,* 19(4), 517-520.

Dai, Y, H. Yuan, Y. (1999). A Nonlinear Conjugate Gradient Method with a Strong Global Convergence Property. *SIAM Journal on Optimization,* 10(1), 177-182.

Dai, Y, H. Yuan, Y. (2001). An Efficient Hybrid Conjugate Gradient Method for Unconstrained Optimization. *Annals of Operations Research,* 103(1-4), 33-47.

Dennis, J, E. Schnabel, R. B. (1996). Numerical Methods for Unconstrained Optimization and Nonlinear Equations. *Classics in Applied Mathematics*, 390.

Dolan, E, D. More, J, J. (2002). Benchmarking Optimization Software with Performance Profiles. *Mathematical Programming*, 91(2), 201-213.

Eldred, M, S. Hart, W, E. (1998). Design and Implementation Of Multi-level Parallel Optimization On The Intel Teraflops. *Proceedings of the 7th AIAA/USAF/NASA/ISSMO Symposium on Multidisciplinary Analysis and Optimization*, 44-54.

Eldred, M, S. Hart, W, E. Bohnhoff, W, J. Romero, V, J. Hutchinson, S, A. Salinger, A, G. (1996). Utilizing Object-Oriented Design to Build Advanced Optimization Strategies with Generic Implementation. *Proceedings of the 6th AIAA/USAF/NASA/ISSMO Symposium on Multidisciplinary Analysis and Optimization*, 1568-82.

Eldred, M, S. Schimel, B, D. (1999). Extended Parallelism Models for Optimization On Massively Parallel Computers. *New Mexico Sandia National Laboratories.*

Fletcher, R. (1987). *Practical Methods of Optimization Volume 1: Unconstrained Optimization*. New York: John Wiley & Sons.

Fletcher, R. (2000). *Practical Methods of Optimization (2nd Ed)*. England: John Wiley & Sons.

Fletcher, R. Reeves, C, M. (1964). Function minimization by conjugate gradients. *The Computer Journal*, 7(2), 149-154.

Fu, F. Sun, W. (2005). Nonmonotone adaptive trust-region method for unconstrained optimization problems. *Applied Mathematics and Computation, 163(*1), 489-504.

Ghattas, O. Orozco, C, E. (1997). A parallel reduced Hessian SQP method for shape optimization. *Multidisciplinary Design Optimization: State of the Art*, 133-152.

Gill, P, E. Murray, W. Saunders, M, A. (2002). SNOPT: An SQP Algorithm for Large-Scale Constrained Optimization. *SIAM Journal on Optimization*, 12(4), 979-1006.

Gill, P, E. Murray, W. Saunders, M, A. Wright, M. H. (1986). Some Theoretical Properties of an Augmented Lagrangian Merit Function, 1-22.

Griewank, A. Toint, Ph, L. (1982). Partitioned variable metric updates for large structured optimization problems. *Numerical Mathematics*, 39(1), 119-137.

Grippo, L. Lampariello, F. Lucidi, S. (1986). A nonmonotone line search technique for Newton's method. *SIAM Journal on Numerical Analysis,* 23(4), 707-716.

Hager, W, W. Zhang, H. (2005). A new conjugate gradient method with guaranteed descent and an efficient line search. *SIAM Journal on Optimization, 16(1),* 170-192.

Han, S, P. (1977). Dual variable metric algorithms for constrained optimization. *SIAM Journal on Control and Optimization,* 15(4), 546-565.

Hei, L. (2003). A Self-Adaptive Trust Region Algorithm. *Journal of Computational Mathematics,* 21(2), 229-236.

Hestenes, M, R. (1969). Multiplier and Gradient Methods. *Journal of Optimization Theory and Applications,* 4(5), 303-320.

Hestenes, M, R. Stiefel, E. (1952). Methods of Conjugate Gradients for Solving Linear Systems. *Journal of Research of the National Bureau of Standards,* 49(6), 409-436.

Liu, J. (2013). An improved trust region method for unconstrained optimization. *Journal of Vibration and Control,* 19(5), 643-648.

Liu, J. Ma, C. (2012). A nonmonotone trust region method with new inexact line search unconstrained optimization. *School of Mathematics and Computer Science, 64(*1), 1-20.

Liu, D. C. Nocedal, J. (1989). On The Limited Memory BFGS Method For Large Scale Optimization. *Mathematical Programming*, 45, 503-528.

Liu, Y, L. Storey, C, S. (1991). Efficient Generalized Conjugate Gradient Algorithms, Part 1: Theory. *Journal of Optimization Theory and Applications,* 69(1), 129-137.

Mayne, D, Q. Polak, E. (1982). A superlinearly convergent algorithm for constrained optimization problems. Algorithms for Constrained Minimization of Smooth Nonlinear Functions. *Mathematical Programming Studies,* 16, 45-61.

Murtagh, B, A. Saunders, M, A. (2003). MINOS 5.51 User's Guide. Stanford University.

Morales, J, L. (2002). A Numerical Study of Limited Memory BFGS Methods. *Applied Mathematics Letters*, 15(4), 481-487.

Nash, S, G. Sofer, A. (1989). Block truncated-Newton methods for parallel optimization. *Mathematical Programming: Series A and B,* 45(3), 529-546.

Nash, S, G. Sofer, A. (1991). A general purpose parallel algorithm for unconstrained optimization. *SIAM Journal on Optimization, 1(4),* 530-547.

Nazareth, L. (1979). A Relationship between the BFGS and Conjugate Gradient Algorithms and its implications for New Algorithms. *SIAM Journal on Numerical Analysis,* 16(5), 794-800.

Nocedal, J. (1980). Updating Quasi-Newton Matrices with Limited Storage. *Mathematics of Computation*, 35(151), 773-782.

Nocedal, J. Wright, S, J. (2006). *Numerical Optimization: Second Edition*. New York: Springer Science.

Nocedal, J. Ya-xiang, Y. (1998). Combining Trust Region and Line Search Techniques. *Advances in Nonlinear Programming*, 14, 153-175.

NVIDIA. (2017). cuBLAS CUDA Toolkit Documentation. http://docs.nvidia.com/cuda/cublas/index.html

Ou, Y. (2011). A hybrid trust region algorithm for unconstrained optimization. *Applied Numerical Mathematics*, 61(7), 900-909.

Perry, A. (1977). A Class of Conjugate Gradient Algorithms with a Two-Step Variable Metric Memory. *Center for Mathematical Studies in Economics and Management Science*, 269, Northwestern University.

Polyak, B, T. The Conjugate Gradient Method in Extreme Problems. (1969). *USSR Computational Mathematics and Mathematical Physics,* 9(4), 94-112.

Powell, M, J, D. (1969). A method for nonlinear constraints in minimization problems. *Optimization*, 283-298.

Rockafellar, R, T. (1973). A dual approach to solving nonlinear programming problems by unconstrained optimization. *Mathematical Programming,* 6(1), 354-373.

Sanderson, C. Curtin, R. (2016). Armadillo: a template-based C++ library for linear algebra. *Journal of Open Source Software, 1*, 26.

Sang, Z. Sun, Q. (2011). A new non-monotone self-adaptive trust region method for unconstrained optimization. *Journal of Applied Mathematics and Computing, 35(*1-2), 53-62.

Schwartz, A. Polak, E. (1997). Family of projected descent methods for optimization problems with simple bounds. *Journal of Optimization Theory and Applications*, 92(1), 1-31.

Shanno, D, F. (1978). On the Convergence of a New Conjugate Gradient Algorithm. *SIAM Journal on Numerical Analysis*, 15(6), 1247-1257.

Shi, Z, J. Guo, J, H. (2008). A new trust region method for unconstrained optimization. *Journal of Computational and Applied Mathematics,* 213, 509-520.

Tapia, R, A. (1977). Diagonalized multiplier methods and quasi-Newton methods for constrained optimization. *Journal of Optimization Theory and Applications,* 22(2), 135-194.

Touati-Ahmed, D. Storey, C. Efficient Hybrid Conjugate Gradient Techniques. *Journal of Optimization Theory and Applications,* 64(2), 379-397.

Vanderbei, R, J. (1999). LOQO User's Manual – Version 3.10. *Optimization Methods and Software,* 12, 485-514.

Wächter A. Biegler, L, T. (2006). On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming. *Mathematical Programming,* 106(1), 25-57.

Wenyu, S. Ya-Xiang, Y. (2006). *Optimization Theory and Methods: Nonlinear Programming*. Heidelberg, Germany: Springer.

Wolfe, P. (1969). Convergence Conditions for Ascent Methods. *SIAM Review*, 11(2), 226-235.

Wolfe, P. (1971). Convergence Conditions for Ascent Methods. II: Some Corections. *SIAM Review*, 13(2), 185-188.

Xiang, Y. Arora, J, S. Rahmatalla, S. Abdel-Malek, K. (2009). Optimization-based dynamic human walking prediction: One step formulation. *International Journal for Numerical Methods in Engineering*, 79(6), 667-695.

Xiang, Y. Rahmatalla, S. Chung, H. Kim, J. Bhatt, R. Mathai, A. Beck, S. Marler, T. R. Yang, J. Arora, J, S. Abdel-Malek, K. Obusek, J. (2008). Optimization-based Dynamic Human Lifting Prediction. *SAE Technical Paper*, 2008-01-1930.

Yuan, G. Meng, S. Wei, Z. (2009). A Trust-Region-Based BFGS Method with Line Search Technique for Symmetric Nonlinear Equations. *Advances in Operations Research*, 2009(909753), 1-22.

Yu, Z. Wang, C. Yu, J. (2004). Combining trust region and linesearch algorithm for equality constrained optimization. *Journal of Applied Mathematics and Computing*, 14(1-2), 123-136.

Zhang, X. (2017). OpenBLAS: An optimized BLAS library. http://www.openblas.net/

Zhang, X. Zhang, J. Liao, L. (2002). An adaptive trust region method and its convergence. *Science in China Series A: Mathematics,* 45(5), 620-631.

Zhou, A. Zhu, Z. Fan, H. Qing, Q. (2011). Three New Hybrid Conjugate Gradient Methods for Optimization. *Applied Mathematics*, 2, 303-308.

Zhang, H. Hager, W, W. (2004). A Nonmonotone Line Search Technique and Its Application to Unconstrained Optimization. *SIAM Journal on Optimization,* 14(4), 1043-1056.

APPENDIX A

LASO USER GUIDE

LASO

LArge Scale Optimizer

Version 1: User’s Guide

Contents

1. The LASO Interface
    1.1. Quick-Start – Unconstrained Example
    1.2. Quick-Start – Constrained Example
    1.3. Calling LASO
    1.4. User-Supplied Subroutine `funobj`
    1.5. User-Supplied Subroutine `funcon`
    1.6. Optional Parameters and Configuration

1. The LASO Interface

LASO is a general purpose C++ optimization software package for the solution of small- and large-scale linear and nonlinear unconstrained and constrained optimization problems. LASO needs only objective and constraint functions to run, however, for optimal performance it is recommended that objective- and constraint-gradients be provided as well. LASO handles optimization problems of the form:

minimize $f(\mathbf{x})$

subject to $l \leq \begin{pmatrix} \mathbf{x} \\ c(\mathbf{x}) \end{pmatrix} \leq u$

This general formulation allows LASO to efficiently handle unconstrained, bound-constrained, or generally-constrained problems from a single LASO call without additional configuration. Equality constraints may be specified by setting the lower- and upper-bounds equal to one another. Strictly less-than or greater-than type constraints can be specified by setting the lower-bound equal to -10^{20} or the upper-bound equal to 10^{20}, respectively. Free constraints, which have neither lower nor upper bounds, can be specified by setting the lower-bound equal to -10^{20} and the upper-bound equal to 10^{20}.

1.1. Quick-Start – Unconstrained Example

LASO is typically called by: creating an optimization problem object, providing an initial estimate of the design variables, setting the lower- and upper-bounds for the problem, changing any optional configuration parameters desired from their default values, and running LASO on the optimization problem object. Creating an optimization problem object requires 4-input parameters: number of design variables `n`, number of constraints `ncnln`, subroutine `funobj`, and subroutine `funcon`. Subroutines `funobj` and `funcon` are discussed in detail later in this guide. For now, it is only important to know that `funobj` is a subroutine for calculating the objective function and its gradient and `funcon` is a subroutine for calculating the constraint functions and their gradients or Jacobian.

Here, we wish to solve the following unconstrained optimization problem:

minimize $f(\mathbf{x}) = (1 - x_1)^2 + 100(x_2 - x_1^2)^2$

which has gradients:

$$\frac{df(\mathbf{x})}{dx_1} = -2(1 - x_1) - 400x_1(x_2 - x_1^2)$$

$$\frac{df(\mathbf{x})}{dx_2} = 200(x_2 - x_1^2)$$

A simple C++ program that uses LASO to solve this problem may look like:

```c
#include "LASO.h" // Must include LASO library

/**
 * User provided subroutine funobj calculates objective function, fObj, and objective
 * function gradient, gObj.
 */
int funobj(   long int *mode , long int *n , double *x , double *fObj , double *gObj ,
              long int *nState )
{
       switch ( *mode )
       {
              case 0:
                     {
                            *fObj = pow(((double)1 - x[0]),(double)2) + (double)100*pow((-
                            pow(x[0],(double)2) + x[1]),(double)2);

                            return 0;
                     }
                     break;
              case 1:
                     {
                            gObj[0] = -(double)2*((double)1 - x[0]) - (double)400*x[0]*(-
                            pow(x[0],(double)2) + x[1]);

                            gObj[1] = (double)200*(-pow(x[0],(double)2) + x[1]);

                            return 0;
                     }
                     break;
       }
}

/**
 * User provided subroutine funcon is left empty for unconstrained problems.
 */
int funcon(   long int *mode , long int *ncnln , long int *n , long int *ldg ,
              long int *needc , double *x , double *fCon , double *gCon ,
              long int *nState )
{
       switch ( *mode )
       {
              case 0:
                     {
                            return 0;
                     }
                     break;
              case 1:
                     {
                            return 0;
                     }
                     break;
       }
}
```

```
/**
 * Main routine sets up and solves the problem.
 */
int main()
{
	long int n = 2; // Number of design variables, n, is set to 2
	long int ncnln = 0; // Number of constraints, ncnln, is set to 0

	// An optimization problem object, OPO, is created and initialized.
	LASO::OptimizationProblemObject OPO( &n , &ncnln , funobj , funcon );

	// Initial estimates of the design variables are set.
	OPO.xDesignVariables[0] = -(double)1.2;
	OPO.xDesignVariables[1] = (double)1;

	// Lower and Upper limits on the design variables are set.
	OPO.blLowerBound[0] = - pow( (double)10 , (double)20 );
	OPO.blLowerBound[1] = - pow( (double)10 , (double)20 );
	OPO.buUpperBound[0] = pow( (double)10 , (double)20 );
	OPO.buUpperBound[1] = pow( (double)10 , (double)20 );

	// An optimization solver object, OSO, is created and the solver is run on the
	// desired optimization problem object.
	LASO::OptimizationSolverObject OSO( OPO );
	OSO.run_Solver( OPO );
}
```

1.2. Quick-Start – Constrained Example

Here, we wish to solve the following constrained optimization problem:

minimize $f(\mathbf{x}) = (x_1 - 2)^2 + (x_2 - 1)^2$

which has gradients:

$$\frac{df(\mathbf{x})}{dx_1} = 2(x_1 - 2)$$

$$\frac{df(\mathbf{x})}{dx_2} = 2(x_2 - 1)$$

subject to $c_1(\mathbf{x}) \equiv 0 \leq 1 - 0.25x_1^2 - x_2^2$

and $c_2(\mathbf{x}) \equiv 1 + x_1 - 2x_2 = 0$

which have gradients:

$$\frac{dc_1(\mathbf{x})}{dx_1} = -0.5x_1$$

$$\frac{dc_1(\mathbf{x})}{dx_2} = -2x_2$$

and

$$\frac{dc_2(\mathbf{x})}{dx_1} = 1$$

$$\frac{dc_2(\mathbf{x})}{dx_2} = -2$$

A simple C++ program that uses LASO to solve this problem may look like:

```
#include "LASO.h" // Must include LASO library

/**
 * User provided subroutine funobj calculates objective function, fObj, and objective
 * function gradient, gObj.
 */
int funobj(   long int *mode , long int *n , double *x , double *fObj , double *gObj ,
              long int *nState )
{
       switch ( *mode )
       {
              case 0:
                     {
                            *fObj = pow((x[0]-(double)2),(double)2) +
                            pow((x[1]-(double)1),(double)2);

                            return 0;
                     }
                     break;
              case 1:
                     {
                            gObj[0] = (double)2 * (x[0]-(double)2);

                            gObj[1] = (double)2 * (x[1]-(double)1);

                            return 0;
                     }
                     break;
       }
}

/**
 * User provided subroutine funcon calculates constraint functions, fCon, and constraint
 * Jacobian, gCon.
 */
int funcon(   long int *mode , long int *ncnln , long int *n , long int *ldg ,
              long int *needc , double *x , double *fCon , double *gCon ,
              long int *nState )
{
       switch ( *mode )
       {
              case 0:
                     {
                            fCon[0] = -(double)1*(-(double)1 + (double)0.25 *
                            pow(x[0],(double)2) + pow(x[1],(double)2));

                            fCon[1] = (double)1 + x[0] - (double)2*x[1];

                            return 0;
                     }
```

```
                break;
            case 1:
                {
                    /// Note: Jacobian is stored in single dimensional array.
                    /// Numbering system is as follows: [ design variable number *
                    /// total number of constraints + constraint number ]
                    gCon[0*2+0] = -(double)1*((double)0.5 * x[0]);
                    gCon[1*2+0] = -(double)1*((double)2 * x[1]);

                    gCon[0*2+1] = (double)1;
                    gCon[1*2+1] = -(double)2;

                    return 0;
                }
                break;
    }
}

/**
 * Main routine sets up and solves the problem.
 */
int main()
{
    long int n = 2; // Number of design variables, n, is set to 2
    long int ncnln = 2; // Number of constraints, ncnln, is set to 2

    // An optimization problem object, OPO, is created and initialized.
    LASO::OptimizationProblemObject OPO( &n , &ncnln , funobj , funcon );

    // Initial estimates of the design variables are set.
    OPO.xDesignVariables[0] = (double)2;
    OPO.xDesignVariables[1] = (double)2;

    // Lower and Upper limits on the design variables and constraints are set.
    OPO.blLowerBound[0] = -pow( (double)10 , (double)20 );
    OPO.blLowerBound[1] = -pow( (double)10 , (double)20 );
    OPO.blLowerBound[2] = (double)0;
    OPO.blLowerBound[3] = (double)0;
    OPO.buUpperBound[0] = pow( (double)10 , (double)20 );
    OPO.buUpperBound[1] = pow( (double)10 , (double)20 );
    OPO.buUpperBound[2] = pow( (double)10 , (double)20 );
    OPO.buUpperBound[3] = (double)0;

    // An optimization solver object, OSO, is created and the solver is run on the
    // desired optimization problem object.
    LASO::OptimizationSolverObject OSO( OPO );
    OSO.run_Solver( OPO );
}
```

1.3. Calling LASO

The LASO interface has been designed to be intuitive and simple, but still offer a high-degree of flexibility to accommodate advanced use cases. As seen in previous sections, LASO is called by creating and configuring an optimization problem object, creating an optimization solver object, and running the solver on the optimization problem object. An optimization problem object is created with the following line of code:

```
LASO::OptimizationProblemObject OPO( &n , &ncnln , funobj , funcon );
```

Here, an optimization problem object, OPO, is created and initialized. Initializing OPO requires 4 user provided input parameters to be passed in by address: a long `int` containing the number of design variables, n; a long `int` containing the number of constraints, `ncnln`; user-supplied subroutine `funobj`; user-supplied subroutine `funcon`.

Next, OPO is configured for the problem at hand and any optional configuration parameters the user wishes to change are set. In configuring OPO for the problem at hand, initial estimates of the design variables as well as lower and upper limits on the design variables and constraints are set:

```
// Set initial estimates of the design variables
OPO.xDesignVariables[0] = (double)0;
.
.
.

// Set lower limits on the design variables and constraints
OPO.blLowerBound[0] = -pow( (double)10 , (double)20 );
.
.
.

// Set upper limits on the design variables and constraints
OPO.buUpperBound[0] = pow( (double)10 , (double)20 );
.
.
.
```

Last, an optimization solver object, OSO, is created and run on OPO:

```
LASO::OptimizationSolverObject OSO( OPO );
OSO.run_Solver( OPO );
```

1.4. User-Supplied Subroutine `funobj`

The user must supply subroutine `funobj`, which defines the objective function and (optionally, but strongly recommended) the objective function gradient. On every call, this subroutine must return appropriate values of the objective, `fObj`, and objective gradient, `gObj`. Similar to the example problems in the quick-start sections, the general form of `funobj` is as follows:

```
/**
 * User provided subroutine funobj calculates objective function, fObj, and objective
 * function gradient, gObj.
 */
int funobj(   long int *mode , long int *n , double *x , double *fObj , double *gObj ,
              long int *nState )
{
       switch ( *mode )
```

```
        {
                case 0:
                        {
                                *fObj = ...

                                return 0;
                        }
                        break;
                case 1:
                        {
                                gObj[0] = ...
                                .
                                .
                                .

                                return 0;
                        }
                        break;
        }
}
```

On entry:

mode	is set by LASO to 0 or 1 depending on whether the objective, fObj, or objective gradient, gObj, is needed, respectively.
n	is the number of design variables, which also describes the dimension of x and gObj, for a given problem. This number is provided by the user to LASO when a new optimization problem object is created and initialized. This number does not change during a LASO run and must not be changed by funobj.
x(n)	is an array of dimension n containing the values of the design variables for which fObj or gObj must be evaluated. It is LASO's job to determine these values. Therefore, it is important that funobj not alter any of these values.
nstate	is not used in this version of LASO.

On exit:

fObj	must contain the value of the objective function evaluated at x.
gObj(n)	must contain the components of the objective function gradient evaluated at x.

1.5. User-Supplied Subroutine funcon

The user must also supply subroutine funcon, which defines the constraint functions and (optionally, but strongly recommended) the constraint function gradients or Jacobian. On every call, this subroutine must return appropriate values of the constraint functions, fCon, and constraint function gradients, gCon. Unless a problem is unconstrained, in which case,

funcon does nothing. Similar to the example problems in the quick-start sections, the general form of funcon is as follows:

```
/**
 * User provided subroutine funcon calculates constraint functions, fCon, and constraint
 * Jacobian, gCon.
 */
int funcon(   long int *mode , long int *ncnln , long int *n , long int *ldg ,
              long int *needc , double *x , double *fCon , double *gCon ,
              long int *nState )
{
       switch ( *mode )
       {
              case 0:
                     {
                            fCon[0] = ...
                            .
                            .
                            .

                            return 0;
                     }
                     break;
              case 1:
                     {
                            /// Note: Jacobian is stored in single dimensional array.
                            /// Numbering system is as follows: [ design variable number *
                            /// total number of constraints + constraint number ]
                            gCon[design variable number * total number of constraints +
                            constraint number] = ...
                            .
                            .
                            .

                            return 0;
                     }
                     break;
       }
}
```

<u>On entry:</u>

mode	is set by LASO to 0 or 1 depending on whether the constraints, fCon, or Jacobian, gCon, is needed, respectively.
ncnln	is the number of constraints, which also describes the dimension of fCon for a given problem. The dimension of gCon is ncnln*n. The value of ncnln is provided by the user to LASO when a new optimization problem object is created and initialized. This number does not change during a LASO run and must not be changed by funcon.
n	is the number of design variables, which also describes the dimension of x for a given problem. This number is provided by the user to LASO when a new optimization problem object is created and initialized. This

	number does not change during a LASO run and must not be changed by `funcon`.
`ldg`	is not used in this version of LASO.
`needc`	is not used in this version of LASO.
`x(n)`	is an array of dimension `n` containing the values of the design variables for which `fCon` or `gCon` must be evaluated. It is LASO's job to determine these values. Therefore, it is important that `funcon` not alter any of these values.
`nstate`	is not used in this version of LASO.

On exit:

`fCon(ncnln)`	must contain the values of the constraint functions evaluated at `x`.
`gCon(ncnln*n)`	must contain the components of the constraint Jacobian evaluated at `x`.

1.6 Optional Parameters and Configuration

LASO's performance depends on a number of internal algorithmic parameters and configurations. Default values of these parameters and configurations should be suitable for most problems. However, the user may adjust these parameters and configurations, as desired, to fine tune LASO's performance on a particular problem. Each parameter and configuration is a named member variable of the optimization problem object class and may be accessed from an optimization problem object, `OPO`, as follows: `OPO.variableName` or `OPO.vectorName[]`.

`vEqualityLMs(n+ncnln)`	is a vector of dimension `n + ncnln` and of type `double` containing the equality constraint Lagrange multiplier values. `vEqualityLMs` defaults to a 0-vector, but initial estimates of `vEqualityLMs` may be set by the user if information is known a priori about the equality constraint activity near the optimum. Providing good estimates of `vEqualityLMs` ahead of time can reduce the number of constrained iterations and markedly improve LASO's performance on constrained problems.
`ulInequalityLMs(n+ncnln)`	is a vector of dimension `n + ncnln` and of type `double` containing the lower bound or greater than type inequality constraint Lagrange multiplier values. `ulInequalityLMs` defaults to a 0-vector, but initial estimates of `ulInequalityLMs` may be set by the user if information is known a priori about the inequality constraint activity near the optimum. Providing good estimates of `ulInequalityLMs` ahead of time can reduce the number

of constrained iterations and markedly improve LASO's performance on constrained problems.

`uuInequalityLMs(n+ncnln)` is a vector of dimension `n + ncnln` and of type `double` containing the upper bound or less than type inequality constraint Lagrange multiplier values. `uuInequalityLMs` defaults to a 0-vector, but initial estimates of `uuInequalityLMs` may be set by the user if information is known a priori about the inequality constraint activity near the optimum. Providing good estimates of `uuInequalityLMs` ahead of time can reduce the number of constrained iterations and markedly improve LASO's performance on constrained problems.

`epsilonConvergence` is the convergence or optimality criteria for the problem. It is of type `double` and has a default value of `10e-6`.

`etaArmijo` is the Armijo stepsize parameter used to calculate the standard Armijo's rule stepsize. It is of type `double` and defaults to the Golden Ratio, which is approximately `1.618`.

`configuration` is the search direction configuration parameter. It is of type `int` and defaults to 3, which is the L-BFGS search direction. Steepest descent and conjugate gradient search directions may be used by setting `configuration` to `1` and `2`, respectively.

`outputLevel` is the output level parameter. It is of type `int` and defaults to `1`, which outputs summary information about each major iteration. Level `0` outputs no information, which may be desirable in production settings. Level `2` outputs detailed information about each major iteration and level 3 outputs detailed information about each major and minor iteration.

`epsilonTolerance` is the tolerance or feasibility criteria for the problem. It is of type `double` and has a default value of `10e-6`.

`mCorrections` is the number of L-BFGS corrections to store. It is of type `int` and has a default value of `20`, which has been shown to be efficient on a wide range of problems. It may, however, be desirable to decrease the number of

corrections to store when dealing with very large problems where memory storage is an issue.

`equalityTolerance`	is the tolerance used to determine whether or not a constraint is an equality constraint. It is of type `double` and has a default value of `0.005`. A constraint will be treated as an equality constraint if the difference between the lower bound and upper bound is less than or equal to this value.
`rPenalty`	is the initial estimate of the penalty parameter used to penalize violated constraints in the Augmented Lagrangian. It is of type `double` and has a default value of `1`. This parameter is updated dynamically during L-BFGS iterations and is increased monotonically by a factor, `betaFactor`, during conjugate gradient and steepest descent iterations when feasibility has not improved.
`betaFactor`	is the factor that the penalty parameter is increased by when constraint violation has not improved. It is of type `double` and has a default value of `10`.
`rhoArmijo`	is the rho parameter used in Armijo's sufficient decrease condition. It is of type `double` and defaults to `10e-4` for L-BFGS and conjugate gradient iterations and `0.2` for steepest descent iterations.
`betaArmijo`	is the beta parameter used in Armijo's curvature condition. It is of type `double` and defaults to `0.9`.
`checkGradient`	is the parameter used to turn finite difference gradient checking on and off. It is of type `bool` and defaults to `FALSE`. If set to TRUE, the objective function gradient will be checked prior to problem start.
`checkJacobian`	is the parameter used to turn finite difference Jacobian checking on and off. It is of type `bool` and defaults to `FALSE`. If set to TRUE, the constraint function gradients / Jacobian will be checked prior to problem start.
`differenceMethod`	is the parameter used to specify the finite difference method to be used during gradient checking and numerical gradient calculations. It is of type `int` and defaults to `1`, which corresponds to the forward difference

method. The central and backward difference methods may be selected by setting `differenceMethod` to `2` and `3`, respectively.

`gradientProvided` is the parameter used to specify whether or not an analytical gradient of the objective function has been provided by the user. It is of type `bool` and defaults to `TRUE`. If set to `FALSE`, gradients will be calculated numerically by the finite difference method specified by `differenceMethod`.

`jacobianProvided` is the parameter used to specify whether or not analytical gradients of the constraints have been provided by the user. It is of type `bool` and defaults to `TRUE`. If set to `FALSE`, gradients of the constraints will be calculated numerically by the finite difference method specified by `differenceMethod`.