On alternative solution methods for solving approximate subproblems in SAO

by

Marlize Cronje

Thesis presented in partial fulfilment of the requirements for the degree of Master of Science in Engineering (Mechanical) at Stellenbosch University

Supervisor: Prof. Albert A. Groenwold

December 2015

Declaration

By submitting this thesis electronically, I declare that the entirety of the work contained therein is my own, original work, that I am the owner of the copyright thereof (unless to the extent explicitly otherwise stated) and that I have not previously in its entirety or in part submitted it for obtaining any qualification.

Signature: MCronje ....
M. Cronje

Date: 11/09/2015 ....

Abstract

In this study, we focus our attention on sequential approximate optimization (SAO) techniques for large scale nonlinear constrained simulation based optimization problems, and aim to add to the efficiency and robustness of the SAO*i* algorithm, developed by Groenwold and Etman (2012). The SAO*i* algorithm construct diagonal quadratic subproblems using an intermediate variable (reciprocal and/or exponential) approach, which may be solved via a quadratic program (QP), or in the dual space.

In the first part of this study, we propose the use of a limited memory implementation of the Broyden, Fletcher, Goldfarb and Shanno updating formula for unconstrained optimization (l-BFGS) to approximate Hessian (second-order) information in an attempt to solve the quadratic approximate subproblems using a QP solver. Although the use of exact second-order information leads to more accurate quadratic approximation functions, the evaluation and storage of the generally fully populated Hessian matrix is not practical for large scale simulation based optimization. We then focus our attention on solving the approximate subproblems in the dual space, and consider the use of a projected adaptive cyclic Barzalai-Borwein (PACBB) method for bound constrained optimization to solve the approximate dual subproblems. Currently, the limited memory bound constrained l-BFGS-b solver is the only dual solver available in the SAO*i* algorithm. As a practical application, we attempt the topology optimization of the Messerschmitt-Bölkow-Blohm (MBB) beam.

Even though the proposed memoryless l-BFGS implementation does not retain the diagonal and separable nature of the approximate subproblems, the numerical results indicate that (although more expensive in terms of computational requirements) it is more efficient and robust than both the intermediate variable approach and the CPLEX solver. When compared to the implementation using exact Hessian information, only a slight increase in the number of iterations is noticed. A drawback of the current implementation is that (for very large scale optimization problems) the QP solver 'struggles' to return a solution to the quadratic subproblem when the Hessian approximation is dense and ill conditioned. It is worth exploring different choices of initial matrix and of the number of previous iterations used in the construction of the Hessian approximation. The use of other QP solvers should also be investigated.

The numerical results for the PACBB method (unfortunately) indicate that the l-BFGS-b dual solver is more efficient and robust than the proposed alternative. However, further investigation shows that the implemented PACBB method delivers a fast initial rate of convergence for the majority of test problems. The PACBB implementation quickly converges to within a neighbourhood of the local or global minimum, but then slows down and sometimes even 'struggles' to converge to the optimal solution. We propose that the PACBB implementation is used for the initial iterations and that an alternative method is then used to achieve a faster rate of convergence to the final solution. The use of other linesearch methods should also be explored.

A feasible solution is found in the application of the proposed PACBB method on the MBB beam problem. However, the l-BFGS implementation does not retain the conservative nature of the convex and separable approximate subproblem, and does not find an acceptable optimal solution to the MBB beam topology application.

Keywords: sequential approximate optimization (SAO); large scale optimization; simulation based optimization; quadratic program (QP); dual space; limited memory Broyden, Fletcher, Goldfarb and Shanno (l-BFGS); projected adaptive cyclic Barzalai-Borwein (PACBB); approximate subproblems.

Uittreksel

In hierdie studie fokus ons ons aandag op opeenvolgende benaderde optimering (*sequential approximate optimization (SAO)*) tegnieke vir grootskaalse lineêre beperkde simulasie gebaseerde optimeringsprobleme. Ons poog om die doeltreffendheid en robuustheid van die SAO*i* algoritme, ontwikkel deur Groenwold en Etman (2012), verder te verbeter. Die SAO*i* algoritme konstrueer diagonale kwadratiese subprobleme met behulp van 'n intermediêre veranderlike (resiproke en/of eksponensiële) benadering. Die subprobleme kan dan opgelos word deur middel van 'n kwadratiese program (*quadratic program (QP)*), of in die duale ruimte.

In die eerste deel van hierdie studie, maak ons gebruik van 'n beperkte geheue implementering van die Broyden, Fletcher, Goldfarb en Shanno opdaterings formule vir onbeperkte optimeringsprobleme (l-BFGS) om Hessiaan (tweede-orde) inligting te benader in 'n poging om die kwadratiese benaderde subprobleme op te los met behulp van 'n QP oplosser. Alhoewel die gebruik van presiese tweede-orde inligting tot meer akkurate kwadratiese benaderingsfunksies lei, is die evaluering en storing van die algemeen ten volle bevolkde Hessiaan matriks nie prakties vir grootskaalse simulasie gebaseerde optimeringsprobleme nie. Ons fokus hierna ons aandag op die oplossing van die benaderde subprobleme in die duale ruimte. Ons oorweeg die gebruik van 'n geprojekteerde aanpasbare sikliese Barzalai-Borwein (*projected adaptive cyclic Barzalai-Borwein (PACBB)*) metode vir begrensde optimering om die benaderde duale subprobleme op te los. Tans is die beperkte geheue l-BFGS-b oplosser die enigste duale oplosser beskikbaar in die SAO*i* algoritme. As 'n praktiese toepassing, poog ons dan die topologie optimering van die Messerschmitt-Bölkow-Blohm (MBB) balk.

Alhoewel die voorgestelde l-BFGS metode nie die diagonale en skeibare aard van die benaderde subprobleme behou nie (en dit duurder is in terme van die berekenings vereistes), dui die numeriese resultate daarop dat dit meer doeltreffend en robuust is as beide die intermediêre veranderlike benadering en die CPLEX oplosser. Wanneer ons dit vergelyk met die gebruik van presiese (ware) Hessiaan inligting, word net 'n effense toename in die aantal iterasies opgemerk. 'n Nadeel van die huidige implementering is dat (vir baie grootskaalse optimeringsprobleme) die QP oplosser 'sukkel' om 'n oplossing vir die kwadratiese subprobleme te vind wanneer die Hessiaan benadering dig en sleg gekondisioneerd is. Dit sal die moeite werd wees om verskillende keuses van aanvangsmatriks en van die aantal vorige iterasies wat gebruik word in die konstruksie van die Hessiaan benadering te verken. Die gebruik van ander QP oplossers moet ook ondersoek word.

Die numeriese resultate vir die PACBB implementering dui (ongelukkig) daarop dat die l-BFGS-b duale oplosser meer doeltreffend en robuust is as die voorgestelde alternatief. Verdere ondersoek wys egter dat die PACBB metode 'n vinniger aanvanklike tempo van konvergensie vir die meerderheid van die toets probleme lewer. Dit konvergeer vinnig tot binne 'n area van die plaaslike of globale minimum, maar word dan stadiger en 'sukkel' soms dan selfs om die optimale oplossing te vind. Ons stel voor dat die PACBB implementering gebruik word vir die aanvanklike iterasies en dat 'n alternatiewe metode dan ingespan word om 'n vinniger tempo van konvergensie tot die finale oplossing te verkry. Die gebruik van ander lynsoek metodes moet ook ondersoek word.

Die toepassing van die voorgestelde PACBB metode op die MBB balk probleem lewer 'n aanvaarbare oplossing. Die l-BFGS implementering behou egter nie die konserwatiewe aard van die konvekse en skeibare benaderde subprobleme nie en vind nie 'n aanvaarbare optimale oplossing vir die MBB balk topologie toepassing nie.

Sleutelwoorde: opeenvolgende benaderde optimering (*sequential approximate optimization (SAO)*); grootskaalse optimering; simulasie gebaseerde optimering; kwadratiese program (*quadratic program (QP)*); duale ruimte; beperkte geheue Broyden, Fletcher, Goldfarb en Shanno (l-BFGS); geprojekteerde aanpasbare sikliese Barzalai-Borwein (*projected adaptive cyclic Barzalai-Borwein (PACBB)*); benaderde subprobleme.

Acknowledgments

I would like to thank my supervisor, Prof. A.A. Groenwold, for his guidance and support throughout the entire duration of this study.

Contents

List of Figures

List of Tables

List of Symbols

$\mathbf{B}$	$n \times n$ Inverse Hessian matrix of the objective or Lagrangian function
f_0	Objective function
$\tilde{f}_0$	Approximation to the objective function
f_0^*	Objective function value at the reported optimum point
f_j	Equality or inequality contraint functions
$\tilde{f}_j$	Approximations to the constraint functions
$\lvert f_j^* \rvert$	Maximum constraint violation at the reported optimum point
$\boldsymbol{g}$	First derivative of the objective function
$\mathbf{H}$	$n \times n$ Hessian matrix of the objective or Lagrangian function
$\mathbf{I}$	$n \times n$ Identity matrix
k	Number of outer iterations
k^*	Number of outer iterations performed to find the reported optimum point
l	Number of inner iterations
l^*	Number of inner iterations performed to find the reported optimum point
m	Number of constraint functions
n	Number of optimization variables
$\boldsymbol{p}$	Vector of length n representing a search direction
r	Number of previous iterations used in the l-BFGS Hessian approximation
$\boldsymbol{x}$	Vector of length n containing optimization variables
x_i	Single optimization variable
$\check{x}_i$	Lower bound on a single optimization variable
$\hat{x}_i$	Upper bound on a single optimization variable
α	Step length
$\boldsymbol{\nabla}^2 f_0$	Second derivative of the objective function

Chapter 1

Introduction

Optimization is an important tool used in engineering design and the analysis of engineering systems. It is the minimization or maximization of an *objective*, which is expressed as a scalar *objective function* $f(\boldsymbol{x}) : \Re^n \rightarrow \Re$ that represents a quantitative measure of the performance of the system under study. The objective depends on certain characteristics of the system, referred to as *design variables*. The goal is to find those values of the variables that optimize the objective. The n real variables contained in $\boldsymbol{x} \in \Re^n$ are often restricted, or *constrained*, in some way. These constraints can be simple bounds placed on the variables (*box-constrained optimization*), or scalar functions of $\boldsymbol{x}$ that define certain equations and inequalities that the variables must satisfy (*constrained optimization*). *Unconstrained optimization* problems are those containing no restrictions at all on the values of the variables.

A powerful collection of optimization algorithms (especially those aimed at unconstrained optimization problems) has been developed over the years. Although unconstrained optimization problems arise directly in many practical applications, the majority of optimization problems contain constraints on the design variables. Due to the success of unconstrained optimization algorithms, constrained optimization problems are often reformulated by replacing the constraints by penalization terms added to the objective function, and the problem is solved using one of these unconstrained algorithms. (The *Lagrangian* method, which is an exact penalty approach that largely preserves smoothness and reduces the possibility of ill conditioning inherent in other penalization methods, is used in this study.)

We focus our attention on sequential approximate optimization (SAO) techniques. We aim to add to the efficiency and robustness of the existing SAO*i* algorithm, developed by Groenwold and Etman (2012). Similar to other SAO methods, the SAO*i* algorithm is aimed at solving large scale nonlinear constrained simulation based optimization problems, which require huge computational resources (Groenwold and Etman, 2009*b*). The optimization of systems or structures modelled using the finite element method (FEM) or computational fluid dynamics (CFD) simulations are typical examples of these large scale simulation based problems. Although primarily used to solve unimodal (structural) optimization problems, the SAO*i* algorithm may also be used to optimize smooth multimodal problems. This is made possible by using a multi-start strategy, combined with a Bayesian acceptance condition (Snyman and Fatti, 1987; Zielinsky, 1981). In this acceptance condition, the assumption is made that the region of attraction of the global optimum is comparable to, or larger than the regions of attraction of other local minima.

In SAO, the main idea is to sequentially approximate all complex objective and constraint functions present in the original optimization problem by simpler analytical functions (*approximation functions*), which (hopefully) represent the true functions well. The resulting approximate optimization problem is called an approximate sub-optimization problem or an approximate subproblem. A sequence of these approximate subproblems can, under certain assumptions and conditions, be shown to converge to the solution of the original problem (Groenwold and Etman, 2009*b*). Any suitable continuous programming method can be used to solve a given approximate subproblem. The obtained solution or optimal point then becomes the starting point for the next approximate subproblem. This results in an iterative process, which is stopped when either the sequence of iterates converges, or when some maximum number of iterations have been reached. As in other SAO algorithms, the SAO*i* algorithm constructs diagonal quadratic subproblems using an intermediate variable (reciprocal and/or exponential) approach. Due to the nature of the approximations used, the subproblems may then be solved via a quadratic program (QP), or in the dual space.

1.1 Objectives

In the first part of this study, we propose the use of a limited memory implementation of the Broyden, Fletcher, Goldfarb and Shanno (l-BFGS) updating formula for unconstrained optimization to approximate Hessian (second-order) information in an attempt to solve the approximate subproblems using a QP solver. Currently, the SAO*i* algorithm has interfaces to more than one (commercial) QP solver, all of which require Hessian or second-order information. Although the use of exact second-order information leads to more accurate quadratic approximation functions, the evaluation and storage of the generally fully populated $n \times n$ Hessian matrix is not practical for large scale simulation based optimization. The explicit computation of the exact Hessian can be cumbersome, error-prone, and expensive. l-BFGS methods have been particularly efficient when used in very large scale unconstrained optimization. Although it is still not clear if this holds true in the constrained case (Morales, 2002), we aim to answer this question and attempt to add to the robustness and efficiency of the existing SAO*i* algorithm.

In the second part of this study, we focus our attention on solving the approximate subproblems in the dual space. We consider the use of a projected adaptive cyclic Barzalai-Borwein (PACBB) method for box-constrained optimization (Zhang and Hager, 2004) to solve the approximate dual subproblem. Although the approximate dual subproblems can be solved using any bound constrained optimization algorithm, there is only one solver available in the SAO*i* algorithm, namely the limited memory bound constrained l-BFGS-b solver (Byrd *et al.*, 1994*a*, 1995).

As a practical implementation, we attempt the solution of a topology optimization problem. Topology optimization is the introduction of topological features into a structure in an attempt to optimize the material distribution in the given design space, subject to any number of linear and/or nonlinear inequality constraints. The aim is to find a purely discrete or black-and-white solution. In this study, we aim to find the optimal material distribution in the Messerschmitt-Bölkow-Blohm (MBB) beam subject to a single linear constraint.

1.2 Outline

In Chapter 2, an overview of *Quazi-Newton* methods aimed at unconstrained optimization is given. We discuss BFGS Hessian formulations and then focus our attention on l-BFGS implementations. This is followed by a basic introduction to SAO techniques (aimed at constrained optimization) and the SAO*i* algorithm. The focus here is on the solution of the approximate subproblems using a quadratic program and the need to develop more efficient and robust approaches to solve these subproblems.

In Chapter 3, the proposed l-BFGS method and the implementation thereof in the existing SAO*i* algorithm are discussed in more detail. A series of test problems are used to measure the performance of the l-BFGS approach against that of two other methods. The performance is measured in terms of the relative number of iterations performed to obtain an optimum solution and the relative accuracy of the obtained solution. The numerical results are summarized in a set of tables and some concluding remarks are provided regarding the efficiency and robustness of the implemented method.

In Chapter 4, an overview of different Barzalai-Borwein (BB) methods is given. We briefly discuss the cyclic BB (CBB) and the adpative CBB (ACBB) methods, from which the projected ACBB (PACBB) method is derived. We then turn our attention to the SAO*i* algorithm and give a basic description of the solution of the approximate subproblems in the dual space. We propose the use of the PACBB method aimed at bound constrained optimizaton as an alternative to the l-BFGS-b dual solver currently used to solve the approximate subproblems.

In Chapter 5, we discuss the PACBB method and the implementation thereof in the existing SAO*i* algorithm in more detail. A series of test problems are used to measure the performance of the PACBB against that of the l-BFGS-b dual solver. Again, the performance is measured in terms of the relative number of iterations performed to obtain an optimum solution and the relative accuracy of the obtained solution. The numerical results are summarized in a set of tables and concluding remarks are provided.

In Chapter 6, a brief description of topology optimization is given and we report on the results of the application of the proposed methods on the MBB beam topology optimization problem.

Finally, in Chapter 7, concluding remarks and recommendations for further study are provided.

All results were generated on a dual core 64-bit 1.86 GHz Intel-based Dell machine with 2 GB of memory and the operating system that was used is OpenSuse 11.4. The code was implemented in double-precision FORTRAN, using the open-source gfortran-4.5 compiler.

Chapter 2

SAO Using Limited Memory BFGS Hessian Formulations

2.1 Introduction

In *unconstrained optimization*, *quasi-Newton methods* are well established methods for updating approximations to a Hessian matrix **H**. Essentially, these methods use historic gradient information to construct an approximation to the Hessian matrix based on the iteration trajectory. Approximating the Hessian, rather than evaluating the true Hessian, is done for any number of reasons. The computational and storage requirements associated with the evaluation of the true Hessian matrix, but also the difficulties associated with obtaining the Hessian matrix during *black-box optimization*, are some of the main reasons. (The term *black-box optimization* derives from the optimization of complex physical or engineering phenomena using 'black-box' simulation packages for which the source code is not available, *e.g.* commercial FEM codes or CFD packages.)

Arguably, the best known approximate formulations are the BFGS updating formula (of rank 2), and the symmetric rank 1 (SR1) formula. For very large scale unconstrained optimization, l-BFGS have been particularly efficient. However, it is still not clear if this holds true in the constrained case (Morales, 2002). In this study, we aim to answer this question. We explore the use of a l-BFGS method in an existing SAO algorithm (the SAO*i* algorithm, developed by Groenwold *et al.*) for solving large scale nonlinear constrained optimization problems.

2.2 Quasi-Newton Methods for Unconstrained Optimization

We first focus our attention on unconstrained optimization and consider problems of the form

$$\min \; f(\boldsymbol{x}) \tag{2.2.1}$$

where $\boldsymbol{x} \in \Re^n$ is a real vector containing $n \geq 1$ design variables, and $f(\boldsymbol{x}) : \Re^n \rightarrow \Re$ is a smooth objective function for which the gradient $\boldsymbol{g}(\boldsymbol{x}) = \boldsymbol{\nabla} f(\boldsymbol{x})$ is available. There are no restrictions at all on the values of the n variables contained in $\boldsymbol{x}$.

Starting at some point $\boldsymbol{x}_0$, optimization algorithms generate a sequence of iterates $\boldsymbol{x}_k$, $k = 0, 1, 2, ...,$ that terminate when either no more progress can be made or when a solution point has been approximated with sufficient accuracy. Two fundamental strategies for moving from the current point $\boldsymbol{x}_k$ to a new iterate $\boldsymbol{x}_{k+1}$ are the *line search* and the *trust region* strategies. The optimization algorithms focused on here follow the line search strategy.

Line search methods generate new iterates by

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \alpha_k \boldsymbol{p}_k, \tag{2.2.2}$$

where $\boldsymbol{p}_k \in \Re^n$ is a search direction and α_k is a positive scalar denoted the step length. The success of a line search algorithm depends on the effective choice of the search direction $\boldsymbol{p}_k$, as well as the step length α_k.

To ensure that the function f can be reduced along the chosen search direction $\boldsymbol{p}_k$, the majority of line search methods require $\boldsymbol{p}_k$ to be a descent direction for which $\boldsymbol{p}_k^T \boldsymbol{\nabla} f_k < 0$. The search direction often has the form

$$\boldsymbol{p}_k = -\,\mathbf{H}_k^{-1} \boldsymbol{\nabla} f_k, \tag{2.2.3}$$

where $\mathbf{H}_k$ is a $n \times n$ symmetric, nonsingular matrix. When $\mathbf{H}_k$ is positive definite, we have

$$\boldsymbol{p}_k^T \boldsymbol{\nabla} f_k = -\boldsymbol{\nabla} f_k^T \mathbf{H}_k^{-1} \boldsymbol{\nabla} f_k < 0, \tag{2.2.4}$$

and $\boldsymbol{p}_k$ is a descent direction.

In general, the step length α_k is found by approximately solving the following one-dimensional minimization problem

$$\min_{\alpha > 0} f(\boldsymbol{x}_k + \alpha \boldsymbol{p}_k). \tag{2.2.5}$$

In computing the step length α_k, we face a tradeoff between choosing α_k to give a substantial reduction of f and not spending too much time making the choice. In general, it is too expensive to find even a local minimizer of $f(\boldsymbol{x}_k+\alpha\boldsymbol{p}_k)$ to moderate precision. Therefore, we use an inexact line search that identifies a step length that achieves adequate reductions in f at minimal cost. Typically, inexact line search algorithms try a sequence of possible values for the step length, and stop to accept one of these values when certain conditions are satisfied. The *Wolfe conditions*, for example, stipulate that α_k should give sufficient decrease in the objective function (the *Armijo condition*) and that unacceptably short steps should be ruled out to ensure that the algorithm makes reasonable progress (the *curvature condition*). These conditions can be enforced in most line search methods, and are particularly important in the implementation of quasi-Newton methods. However, even with these conditions enforced, line search algorithms are often still expensive, requiring a large number of function evaluations. Alternative step sizes determined within a few simple calculations have been suggested. In Chapter 4, we discuss the Barzalai-Borwein (BB) methods, which suggest such an alternative step size for the steepest descent method.

Steepest descent methods use a step length of $\alpha_k = 1$ and simply set $\mathbf{H}_k$ equal to the identity matrix $\mathbf{I}$, *i.e.*

$$\boldsymbol{p}_k = -\nabla f_k. \tag{2.2.6}$$

In Newton's method, $\mathbf{H}_k$ is the exact Hessian $\mathbf{H}_k = \nabla^2 f(\boldsymbol{x}_k)$. As in the steepest descent methods, a unit step length is used in most line search implementations of Newton's method. However, this value is adjusted using an inexact line search algorithm when it does not produce a satisfactory reduction in the value of f.

Although the Newton direction has a fast rate of local convergence (typically quadratic), the explicit computation of the exact Hessian $\mathbf{H}_k = \nabla^2 f(\boldsymbol{x}_k)$ can be cumbersome, error-prone, and expensive. If the exact Hessian is not positive definite, $\mathbf{H}_k^{-1} = (\nabla^2 f(\boldsymbol{x}_k))^{-1}$ may not exist and the Newton direction will not be defined. Even if it *is* defined, it may not satisfy the descent property $\nabla f(\boldsymbol{x}_k)^T\boldsymbol{p}_k < 0$ and will not be a suitable search direction.

Quasi-Newton methods provide attractive alternative search directions. These methods avoid calculation of the exact Hessian while still attaining a super-linear rate of convergence. They use an approximation to the Hessian which is updated after each iteration. These updates take account of the additional knowledge gained during the iteration and make use of the fact that changes in the gradient $\mathbf{g} = \nabla f$ provide useful information about the second derivative $\nabla^2 f(\boldsymbol{x})$ along the search direction. In quasi-Newton methods, the difference between successive Hessian approximations $\mathbf{H}_k$ and $\mathbf{H}_{k+1}$ should be low rank, and the new approximation $\mathbf{H}_{k+1}$ should satisfy the *secant equation*

$$\mathbf{H}_{k+1}\boldsymbol{s}_k = \boldsymbol{y}_k, \tag{2.2.7}$$

where

$$\boldsymbol{s}_k = \boldsymbol{x}_{k+1} - \boldsymbol{x}_k, \tag{2.2.8}$$

$$\boldsymbol{y}_k = \nabla f_{k+1} - \nabla f_k. \tag{2.2.9}$$

An additional condition imposed on $\mathbf{H}_{k+1}$ is that of symmetry, which is motivated by the symmetry of the exact Hessian.

Arguably, the best known approximate formulations are the SR1 and the BFGS updating formulae. A brief discussion of the BFGS method, with the focus on limited memory implementations thereof, follows.

2.2.1 BFGS Hessian Formulations

As hinted at, one of the most popular formulae used to update $\mathbf{H}_{k+1}$ is the BFGS formula, which is given by

$$\mathbf{H}_{k+1} = \mathbf{H}_k - \frac{\mathbf{H}_k\boldsymbol{s}_k\boldsymbol{s}_k^T\boldsymbol{H}_k}{\boldsymbol{s}_k^T\mathbf{H}_k\boldsymbol{s}_k} + \frac{\boldsymbol{y}_k\boldsymbol{y}_k^T}{\boldsymbol{y}_k^T\boldsymbol{s}_k}. \tag{2.2.10}$$

The BFGS update satisfies the *secant equation* (2.2.7), maintains symmetry, and the difference between successive approximations is a rank-two matrix. It also generates positive definite approximations whenever the initial approximation $\mathbf{H}_0$ is positive definite and $\boldsymbol{s}_k^T\boldsymbol{y}_k > 0$.

Since the inverse Hessian $\mathbf{H}_k^{-1}$ is required for the determination of the *quasi-Newton* search direction (2.2.3), some practical implementations approximate the inverse, instead of the Hessian itself, to avoid the need to factorize the approximation at each iteration. The inverse Hessian approximation $\mathbf{B}_k \overset{def}{=} \mathbf{H}_k^{-1}$ is given by

$$\mathbf{B}_{k+1} = (\mathbf{I} - \rho_k\boldsymbol{s}_k\boldsymbol{y}_k^T)\mathbf{B}_k(\mathbf{I} - \rho_k\boldsymbol{y}_k\boldsymbol{s}_k^T) + \rho_k\boldsymbol{s}_k\boldsymbol{s}_k^T, \qquad \rho_k = \frac{1}{\boldsymbol{y}_k^T\boldsymbol{s}_k}. \tag{2.2.11}$$

The formula for calculating $\boldsymbol{p}_k$ then becomes

$$\boldsymbol{p}_k = -\,\mathbf{B}_k\nabla f_k. \tag{2.2.12}$$

Note that the matrix-vector multiplication is much simpler than the factorization/ back-substitution procedure needed when using (2.2.10) together with (2.2.3) to determine the search direction.

Due to memory constraints, the BFGS approach is not affordable in the setting of large scale optimization. Limited memory BFGS (l-BFGS) methods, which are variations of the standard BFGS approach, overcome this difficulty and are discussed next.

2.2.2 l-BFGS Hessian Formulations

l-BFGS methods are aimed at large scale optimization problems whose Hessian matrices cannot be computed or stored at a reasonable cost. The amount of storage required by these methods can be controlled, and no knowledge of the sparsity structure of the true Hessian matrix or of the separability of the problem is needed. (This is not as attractive as one may think at first glance. Indeed, the l-BFGS method is oblivious about the original structure, and it is of some interest to develop effective methods that observe the original sparsity structure.)

l-BFGS methods construct a positive definite approximation to the Hessian matrix, using curvature information from only the most recent iterations. Instead of storing fully dense $n \times n$ approximations obtained using the complete iteration history, these methods saves only a few vectors of length n that represent the approximations implicitly. These vectors contain gradient information from only the most recent iterations. Gradient information from earlier iterations, which is (arguably) less likely to be relevant to the actual behavior of the Hessian at the current iteration, is discarded. (If gradient information from only the current and the previous iteration is used in the l-BFGS method, it is considered a *memoryless* approach.) This reduces storage requirements, making the method linear in n. For unconstrained optimization, recent investigations have reported that limited memory implementations can often outperform implementations that update using the complete iteration history. As hinted at, this can (presumably) be attributed to the fact that older information might not be relevant at the current point.

l-BFGS methods have proved to be efficient, robust, and relatively inexpensive in terms of computational effort for a very large number of unconstrained optimization test problems. In this study, we propose the use of l-BFGS Hessian approximations in a SAO algorithm for constrained optimization. The (standard) l-BFGS approach is discussed next.

The (standard) l-BFGS approach

The l-BFGS method (Liu and Nocedal, 1989) is almost identical to the (standard) BFGS method with the only difference being in the matrix update. To avoid storage of the full Hessian $\mathbf{H}_k$, the l-BFGS method stores a certain number (say r)

of vector pairs $\{\boldsymbol{s}_i, \boldsymbol{y}_i\}$ where

$$\boldsymbol{s}_i = \boldsymbol{x}_{i+1} - \boldsymbol{x}_i, \tag{2.2.13}$$

$$\boldsymbol{y}_i = \nabla f_{i+1} - \nabla f_i. \tag{2.2.14}$$

At iterate $\boldsymbol{x}_k$, the initial inverse Hessian matrix $\mathbf{B}_{k,0}$ (usually a positive multiple of the identity matrix to ensure positive definiteness) is updated r times using the (standard) BFGS updating formula and the r most recent vector pairs $\{\boldsymbol{s}_i, \boldsymbol{y}_i\}$, $i = k-r, k-r+1, \ldots, k-1$. (In constrast to the BFGS approach, the initial matrix $\mathbf{B}_{k,0}$ is allowed to vary from iteration to iteration.) The updating formula (2.2.10) then becomes

$$\begin{aligned}\mathbf{B}_k &= (\mathbf{V}_{k-1}^T \ldots \mathbf{V}_{k-r}^T)\mathbf{B}_{k,0}(\mathbf{V}_{k-r} \ldots \mathbf{V}_{k-1}) \\ &+ \rho_{k-r}(\mathbf{V}_{k-1}^T \ldots \mathbf{V}_{k-r+1}^T)\boldsymbol{s}_{k-r}\boldsymbol{s}_{k-r}^T(\mathbf{V}_{k-r+1} \ldots \mathbf{V}_{k-1}) \\ &+ \rho_{k-r+1}(\mathbf{V}_{k-1}^T \ldots \mathbf{V}_{k-r+2}^T)\boldsymbol{s}_{k-r+1}\boldsymbol{s}_{k-r+1}^T(\mathbf{V}_{k-r+2} \ldots \mathbf{V}_{k-1}) \\ &+ \ldots \\ &+ \rho_{k-1}\boldsymbol{s}_{k-1}\boldsymbol{s}_{k-1}^T,\end{aligned} \tag{2.2.15}$$

where

$$\rho_k = \frac{1}{\boldsymbol{y}_k^T \boldsymbol{s}_k} \qquad \text{and} \qquad \mathbf{V}_k = \mathbf{I} - \rho_k \boldsymbol{y}_k \boldsymbol{s}_k^T. \tag{2.2.16}$$

It is possible to derive a recursive procedure to compute the product $\mathbf{B}_k \nabla f_k$ efficiently (Nocedal and Wright, 2006), which allows for the computation of the search direction (2.2.12) to be performed very economically. For the implementation of the l-BFGS method into the existing SAO*i* algorithm, however, we require the direct Hessian approximation of the Lagrangian $\mathbf{H}_k$. Unfortunately, there appears to be no analogous recursion for $\mathbf{H}_k$. Compact representations of l-BFGS matrices are available for both the inverse and the direct Hessian, which overcome this difficulty (Byrd *et al.*, 1994*b*).

Compact l-BFGS representations

Several compact representations of l-BFGS approximations to the inverse and the direct Hessian have been derived for the efficient implementation of l-BFGS methods. One of these representations for the direct Hessian approximation are discussed here.

At iterate $\boldsymbol{x}_k$, the approximation to the direct Hessian $\mathbf{H}_k$ can be constructed by updating the initial matrix r times using the $2r$ vectors $\{\boldsymbol{s}_{k-r}, \ldots, \boldsymbol{s}_{k-1}\}$ and

$\{\boldsymbol{y}_{k-r}, \ldots, \boldsymbol{y}_{k-1}\}$. If the initial matrix $\mathbf{H}_{k,0} = \sigma_k \mathbf{I}$ (Liu and Nocedal, 1989), with σ_k some positive scalar, $\mathbf{H}_k$ is given by

$$\mathbf{H}_k = \sigma_k \mathbf{I} - \begin{bmatrix} \sigma_k \mathbf{S}_k & \mathbf{Y}_k \end{bmatrix} \begin{bmatrix} \sigma_k \mathbf{S}_k^T \mathbf{S}_k & \mathbf{L}_k \\ \mathbf{L}_k^T & -\mathbf{D}_k \end{bmatrix}^{-1} \begin{bmatrix} \sigma_k \mathbf{S}_k^T \\ \mathbf{Y}_k^T \end{bmatrix} \tag{2.2.17}$$

where $\mathbf{S}_k$ and $\mathbf{Y}_k$ are then $n \times r$ matrices defined by

$$\mathbf{S}_k = [\mathbf{s}_{k-r}, \ldots, \mathbf{s}_{k-1}], \qquad \mathbf{Y}_k = [\mathbf{y}_{k-r}, \ldots, \mathbf{y}_{k-1}], \tag{2.2.18}$$

while $\mathbf{L}_k$ and $\mathbf{D}_k$ are the $r \times r$ matrices

$$\mathbf{L}_{k,(i,j)} = \begin{cases} \mathbf{s}_{k-r-1+i}^T \mathbf{y}_{k-r-1+j} & \text{if } i > j, \\ 0 & \text{otherwise,} \end{cases} \tag{2.2.19}$$

$$\mathbf{D}_k = \operatorname{diag}[\mathbf{s}_{k-r}^T \mathbf{y}_{k-r}, \ldots, \mathbf{s}_{k-1}^T \mathbf{y}_{k-1}]. \tag{2.2.20}$$

For iterations where $k < r$, r is simply replaced by k in the above formulae.

After the new iterate $\boldsymbol{x}_{k+1}$ is generated, $\mathbf{S}_{k+1}$ and $\mathbf{Y}_{k+1}$ are obtained by deleting the oldest pair $\{\boldsymbol{s}_{k-r}, \boldsymbol{y}_{k-r}\}$ from, and adding the new vector pair $\{\boldsymbol{s}_k, \boldsymbol{y}_k\}$ to $\mathbf{S}_k$ and $\mathbf{Y}_k$. This method therefore always keeps the r most recent vector pairs to define the iteration matrix. Since it has been observed in practice that small values of r give satisfactory results, this approach is suitable for very large scale optimization problems.

A (straightforward) l-BFGS approach

Although different variants of the direct l-BFGS formula are available, we will focus on a version of the compact form of the l-BFGS updating formula (2.2.17), which simplifies the matrix multiplications needed to update the approximate Hessian $\mathbf{H}_k$ by reducing (2.2.17) to (simpler) vector multiplications (Byrd *et al.*, 1994*b*). Unfortunately, this approach increases the amount of multiplications needed to determine $\mathbf{H}_k$ from $2rn$ to $\frac{3}{2}r^2 n$.

In a typical iteration k, the l-BFGS approximation to the Hessian matrix $\mathbf{H}_k$ is obtained by updating a starting matrix r times using the r most recent vector pairs

$$(\boldsymbol{s}_{k-r}, \boldsymbol{y}_{k-r}), \ \ldots, (\boldsymbol{s}_{k-1}, \ \boldsymbol{y}_{k-1}),$$

where $\mathbf{s}_i$ and $\mathbf{y}_i$, $i = k - r, \ldots, k - 1$, are given by (2.2.13) and (2.2.14) respectively. The direct Hessian approximation is then given by

$$\mathbf{H}_k = \mathbf{H}_{k,0} + \sum_{i=k-r}^{k-1} [\boldsymbol{b}_i \boldsymbol{b}_i^T - \boldsymbol{a}_i \boldsymbol{a}_i^T], \tag{2.2.21}$$

where $\boldsymbol{a}_i$ and $\boldsymbol{b}_i$ are vectors of length n. These vectors can be computed for $i = k - r,\ k - r + 1,\ ...,\ k - 1$ by

$$\mathbf{b}_i = \frac{\mathbf{y}_i}{(\mathbf{y}_i^T \mathbf{s}_i)^{1/2}} \tag{2.2.22}$$

and

$$\mathbf{a}_i = \frac{\tilde{\mathbf{a}}_i}{(\mathbf{s}^T{}_i \tilde{\mathbf{a}}_i)^{1/2}}, \tag{2.2.23}$$

where

$$\tilde{\mathbf{a}}_i = \mathbf{H}_{k,0} \boldsymbol{s}_i + \sum_{j=0}^{i-1} [(\boldsymbol{b}_j^T \boldsymbol{s}_i)\boldsymbol{b}_j - (\boldsymbol{a}_j^T \boldsymbol{s}_i)\boldsymbol{a}_j]. \tag{2.2.24}$$

To implement a *memoryless* BFGS method, where gradient information from only the current and previous iterates is used and the starting matrix is only updated once, r should simply be set equal to 1.

As mentioned, l-BFGS methods construct a positive definite approximation to the Hessian matrix. To ensure that the approximate Hessian is indeed positive definite, the chosen starting matrix $\mathbf{H}_{k,0}$ must be positive definite and the following curvature condition must hold:

$$\boldsymbol{s}_k^T \boldsymbol{y}_k > 0. \tag{2.2.25}$$

(An update is only performed if (2.2.25) holds.) To satisfy (2.2.25), the starting matrix $\mathbf{H}_{k,0}$ is often simply set to the identity matrix $\mathbf{I}$, or some positive multiple of the identity matrix $\sigma_k \mathbf{I}$.

2.3 SAO Methods for Constrained Optimization

As mentioned, *constrained optimization* is the minimization or maximization of a scalar objective function $f(\boldsymbol{x}) : \Re^n \rightarrow \Re$, where the n real variables contained in $\boldsymbol{x} \in \Re^n$ are restricted, or *constrained*, in some way. These constraints can be simple bounds placed on the variables, or scalar functions of $\boldsymbol{x}$ that define certain equations and/or inequalities that the variables must satisfy. In this study, we consider (large scale) nonlinear inequality-constrained programming problems $\mathrm{P_{NLP}}$ of the form

$$\begin{aligned} &\min_{\boldsymbol{x}} f_0(\boldsymbol{x}) \\ &\text{subject to } f_j(\boldsymbol{x}) \le 0,\ j = 1, 2, \ldots, m, \\ &\qquad \check{x}_i \le x_i \le \hat{x}_i,\ i = 1, 2, \ldots, n, \end{aligned} \tag{2.3.1}$$

where $f_0(\boldsymbol{x})$ represents a real-valued scalar objective function, and $f_j(\boldsymbol{x})$, $j = 1, 2, \ldots, m$, represents m inequality real-valued scalar constraint functions. The objective and constraint functions depend on the n real design variables $\boldsymbol{x} = \{x_1, x_2, \ldots, x_n\}^T \in \Re^{\mathrm{n}}$. $\check{x}_i$ indicates the lower bound and $\hat{x}_i$ the upper bound of continuous real variable x_i. A number of methods can be used to solve $\mathrm{P}_{\mathrm{NLP}}$, but we focus on *sequential approximate optimization* (SAO) techniques.

SAO methods are aimed at solving large scale nonlinear constrained simulation based optimization problems, which require huge computational resources (Groenwold and Etman, 2009*b*). In SAO, the main idea is to sequentially approximate all complex objective and constraint functions present in the original optimization problem by simpler analytical functions (approximation functions), which (hopefully) represent the true functions well. The resulting approximate optimization problem is called an approximate sub-optimization problem or an approximate subproblem. A sequence of these approximate subproblems can, under certain assumptions and conditions, be shown to converge to the solution of the original problem (Groenwold and Etman, 2009*b*). Any suitable continuous programming method can be used to solve a given approximate subproblem. The obtained solution or optimal point then becomes the starting point for the next approximate subproblem. This results in an iterative process, which is stopped when either the sequence of iterates converges, or when some maximum number of iterations have been reached.

Sequential quadratic programming (SQP) methods construct quadratic approximations to the objective function

$$\tilde{f}_0(\boldsymbol{x}) = f_0(\boldsymbol{x}_k) + \boldsymbol{\nabla}^T f_0(\boldsymbol{x}_k)(\boldsymbol{x} - \boldsymbol{x}_k) + \frac{1}{2}(\boldsymbol{x} - \boldsymbol{x}_k)^T \mathbf{H}_k(\boldsymbol{x} - \boldsymbol{x}_k), \tag{2.3.2}$$

and linear approximations to the constraint functions

$$\tilde{f}_j(\boldsymbol{x}) = f_j(\boldsymbol{x}_k) + \boldsymbol{\nabla}^T f_j(\boldsymbol{x}_k)(\boldsymbol{x} - \boldsymbol{x}_k), \; j = 1, 2, \ldots, m. \tag{2.3.3}$$

(The Hessian in the quadratic approximation $\mathbf{H}_k$ is the (exact) Hessian of the Lagrangian, and not that of the objective function.) Although the use of exact second-order information leads to more accurate approximation functions, the evaluation and storage of the generally fully populated $n \times n$ Hessian matrix is not practical for large scale simulation based optimization. Even when n is small, the Hessian might be difficult to attain (especially in the case of *black-box optimization*).

To avoid the problems associated with using (true) Hessian information, SAO methods make use of convex separable approximation functions. The aim is to achieve an accuracy comparible to that of SQP methods, while keeping computational and storage requirements to a minimum. The approximations used are diagonal quadratic in nature, and are of the form

$$\tilde{f}(\boldsymbol{x}) = f(\boldsymbol{x}_k) + \boldsymbol{\nabla}^T f(\boldsymbol{x} - \boldsymbol{x}_k) + \frac{1}{2}\sum_{i=1}^{n} c_{k,i}\,(\boldsymbol{x} - \boldsymbol{x}_k)^2, \qquad (2.3.4)$$

where $c_{k,i}$, $i = 1, 2, \dots, n$, are the only unknowns. Full Hessian information is not used and interaction between the variables is ignored. To introduce desirable nonlinearities into the approximation functions (*i.e.* to determine appropriate values for the unknowns $c_{k,i}$, $i = 1, 2, \dots, n$), SAO methods make use of intermediate variables. Though the intermediate variable approach can in general of course not be expected to yield comparable accuracy to using full Hessian information, it is in part hoped that it will alleviate the shortcomings that arise from neglecting the interaction between the variables in the first place. Also, the approximations used in SAO are typically more accurate than the linear Taylor series expansion. Many different intermediate variables may be used, providing a range of possible approximation functions.

The efficiency and accuracy of SAO algorithms depend crucially on the approximation functions used, as well as (although sometimes to a lesser extent) on the solvers used for the approximate subproblems. As in SQP, a QP solver can be used for the approximate subproblems in SAO. However, SAO methods almost invariably use a dual statement due to Falk (Falk, 1967). This enables SAO methods to outperform SQP methods when the number of constraints is far less than the number of design variables. In SAO, different approximation functions (and solvers) may be optimal for different problems. We will now focus our attention on the SAO*i* algorithm, developed by Groenwold and Etman (2012), which provides for a range of possible approximation functions and allows subproblems to be solved via a QP or in the dual space.

2.4 The SAO*i* Algorithm

The SAO*i* algorithm is aimed at large scale 'simulation-based' optimization (Groenwold and Etman, 2012). It is an algorithm for bounded inequality constrained nonlinear optimization problems $\mathrm{P}_{\mathrm{NLP}}$ (2.3.1). The SAO*i* algorithm is primarily aimed at unimodal (structural) optimization problems, but smooth multimodal problems may also be optimized.

2.4.1 Multimodal Problems

For multimodal optimization problems, the SAO*i* algorithm uses a multi-start strategy, combined with a Bayesian acceptance condition (Snyman and Fatti, 1987).

Multi-start methods repeatedly invoke a local optimization method using different starting points and end the process when some prescribed termination rule is satisfied. In the SAO*i* algorithm, the multi-start strategy is terminated when the probability of convergence to the global optimum is larger than, or equal to some prescribed confidence level value. Typically, confidence level values of 0.99 to 0.999 are used. Upon termination, the overall minimum of the function values $f^*_{0,\ell}$, $\ell = 1, 2, \ldots$, is then used as the approximation to the global minimum f^*_0, *i.e.*

$$f^*_0 = \min\{f^*_{0,\ell},\ \ell = 1, 2, \ldots\}, \tag{2.4.1}$$

where ℓ represents the number of starting points used and $f^*_{0,\ell}$ are assumed to be feasible local minima determined at each starting point.

Multi-start methods are based on the assumption that the region of attraction of the global optimum is comparable to, or larger than, the region of attraction of any other local optimum. We define the *region of convergence* of a local minimum $\boldsymbol{x}^*_k$ as the set of all points $\boldsymbol{x}$ which will result in convergence to $\boldsymbol{x}^*_k$ when used as starting points for a given algorithm. R^*_k denotes the region of convergence of local minimum $\boldsymbol{x}^*_k$ and ξ^*_k the associated probability that a sample point will be selected in R^*_k. In the same way, the region of convergence and the associated probability for the global minimum $\boldsymbol{x}^*$ are denoted by R^* and ξ^* respectively. We can then make the assumption (which is probably true for many algorithms and functions of practical interest) that

$$\xi^* \ \geq \ \xi^*_k \text{ for all local minima } \boldsymbol{x}^*_k \tag{2.4.2}$$

The Bayesian stopping criterion is based on this assumption and is given by the following theorem (Snyman and Fatti, 1987).

Theorem 3.1 *Let r be the number of sample points falling within the region of convergence of the current overall minimum $\tilde{f}$ after $\tilde{n}$ points have been sampled. Then, under assumption (2.4.2) and a statistically non-informative prior distribution, the probability P_r that $\tilde{f}$ corresponds to f^* may be obtained from*

$$P_r\left[\tilde{f} = f^*\right] \geq q(\tilde{n}, r) \ = \ 1 - \frac{(\tilde{n}+1)!(2\tilde{n}-r)!}{(2\tilde{n}+1)!(\tilde{n}-r)!}. \tag{2.4.3}$$

On the basis of the above theorem, the adopted *stopping rule* becomes

$$\text{STOP when } P_r\left[\tilde{f} = f^*\right] \geq q^*, \tag{2.4.4}$$

where q^* represents some prescribed confidence level. As mentioned, confidence level values of 0.99 to 0.999 are typically used. An outline of the proof of (2.4.3), which closely follow the presentation by Snyman and Fatti (1987), is given next.

If $\tilde{n}^*$ denotes the number of points sampled to achieve the desired confidence level and ξ^* the probability that a sample point will be selected in the region of convergence for the global minimum, the probability that at least one point, $\tilde{n} \geq 1$, has converged to f^* is

$$P_r\left[\tilde{n}^* \geq 1|\tilde{n}, r\right] = 1 - (1 - \xi^*)^{\tilde{n}}. \tag{2.4.5}$$

In the Bayesian approach, we specify a prior probability distribution for ξ^* to characterize the uncertainty about its value. Using the sample information ($\tilde{n}$ and r), this distribution can be modified to form a posterior probability distribution. If we let $p_*(\xi^*|\tilde{n}, r)$ be the posterior probability distribution of ξ^*, (2.4.5) becomes

$$\begin{aligned} P_r\left[\tilde{n}^* \geq 1|\tilde{n}, r\right] &= \int_0^1 \left[1 - (1 - \xi^*)^{\tilde{n}}\right] p_*(\xi^*|\tilde{n}, r) d\xi^* \\ &= 1 - \int_0^1 (1 - \xi^*)^{\tilde{n}} p_*(\xi^*|\tilde{n}, r) d\xi^*. \end{aligned} \tag{2.4.6}$$

Although we know that the r sample points converge to the current overall minimum, we do not know whether this minimum corresponds to the global minimum f^*. If we utilize the assumption made earlier (2.4.2) and note that $(1 - \xi)^{\tilde{n}}$ is a decreasing function of ξ, we can replace ξ^* in the above integral by ξ. We then have

$$P_r\left[\tilde{n}^* \geq 1|\tilde{n}, r\right] \geq \int_0^1 \left[1 - (1 - \xi)^{\tilde{n}}\right] p(\xi|\tilde{n}, r) d\xi. \tag{2.4.7}$$

By using Bayes theorem, we then obtain

$$p(\xi|\tilde{n}, r) = \frac{p(r|\xi, \tilde{n})p(\xi)}{\int_0^1 p(r|\xi, \tilde{n})p(\xi)d\xi}. \tag{2.4.8}$$

If we take into consideration that the $\tilde{n}$ points are sampled at random and that each point has a probability ξ of converging to the current overall minimum, r has a binomial distribution with parameter ξ and $\tilde{n}$. This means that

$$p(r|\xi, \tilde{n}) = \binom{\tilde{n}}{r} \xi^r (1 - \xi)^{\tilde{n} - r}. \tag{2.4.9}$$

By substituting (2.4.9) and (2.4.8) into (2.4.7), we get

$$P_r\left[\tilde{n}^* \geq 1|\tilde{n}, r\right] \geq 1 - \frac{\int_0^1 \xi^r (1 - \xi)^{2\tilde{n} - r} p(\xi) d\xi}{\int_0^1 \xi^r (1 - \xi)^{\tilde{n} - r} p(\xi) d\xi}. \tag{2.4.10}$$

The beta distribution (with parameters a and b) is a suitable flexible prior distribution $p(\xi)$ for ξ, *i.e.*

$$p(\xi) = \left[1/\beta(a,b)\right]\xi^{a-1}(1-\xi)^{b-1},\ 0 \le \xi \le 1. \tag{2.4.11}$$

Using (2.4.11) gives us

$$\begin{aligned} P_r\left[\tilde{n}^* \ge 1|\tilde{n},r\right] &\ge 1 - \frac{\Gamma(\tilde{n}+a+b)\Gamma(2\tilde{n}-r+b)}{\Gamma(2\tilde{n}+a+b)\Gamma(\tilde{n}-r+b)} \\ &= 1 - \frac{(\tilde{n}+a+b-1)!(2\tilde{n}-r+b-1)!}{(2\tilde{n}+a+b-1)!(\tilde{n}-r+b-1)!}. \end{aligned} \tag{2.4.12}$$

If we assume a prior expectation of 1 (*i.e.* $a = b = 1$), (2.4.12) simplifies to

$$P_r\left[\tilde{f} = f^*\right] \ge q(\tilde{n},r) = 1 - \frac{(\tilde{n}+1)!(2\tilde{n}-r)!}{(2\tilde{n}+1)!(\tilde{n}-r)!}. \tag{2.4.13}$$

2.4.2 QP Subproblems

As in other SAO algorithms, the SAO*i* algorithm constructs diagonal quadratic subproblems (2.3.4) using an intermediate variable (reciprocal and/or exponential) approach. Due to the nature of the approximations used, the subproblems may then be solved via a QP, or in the dual space.

In dual SAO methods, the main difficulty is to select accurate approximation functions that satisfy the requiremetns needed to invoke duality. The current academic version of the SAO*i* algorithm, in part based on the dual of Falk (Falk, 1967), uses separable approximation functions, and the subproblems can very efficiently be solved in the dual space. (Therefore, it is effectively impossible for SQP methods to compete with SAO methods if the number of constraints is considerably less than the number of design variables.) Although the approximate dual subproblems can be solved using any bound constrained optimization algorithm, there is only one solver available, namely the limited memory bound constrained l-BFGS-b solver (Byrd *et al.*, 1994*a*, 1995), in the current academic version of the SAOi algorithm. In the second part of this study, we consider the use of a projected adaptive cyclic Barzalai-Borwein (PACBB) method for box-constrained optimization (Zhang and Hager, 2004) to solve the approximate dual subproblem (Chapter 4).

The (diagonal) quadratic nature of the aprroximate subproblems (2.3.4) also makes it easy to construct (diagonal) QP subproblems. The QP subproblems $\mathrm{P_{QP}}$ are given by

$$\begin{aligned} \min_{\boldsymbol{s}} \ \bar{f}_0(\boldsymbol{s}) &= f_0(\boldsymbol{x}_k) + \boldsymbol{\nabla}^T f_0(\boldsymbol{x}_k)\boldsymbol{s} + \frac{1}{2}\boldsymbol{s}^T\mathbf{H}_k\boldsymbol{s} \\ \text{subject to} \ \bar{f}_j(\boldsymbol{s}) &= f_j(\boldsymbol{x}_k) + \boldsymbol{\nabla}^T f_j(\boldsymbol{x}_k)\boldsymbol{s} \le 0, \qquad j = 1,2,\dots,m, \end{aligned} \tag{2.4.14}$$

where $\boldsymbol{s} = \boldsymbol{x} - \boldsymbol{x}_k$ and $\mathbf{H}_k$ is the Hessian of the Lagrangian at $\boldsymbol{x}_k$ (Groenwold and Etman, 2012). For more details, refer to Etman *et al.* (2009).

To avoid the (cumbersome, error-prone, and expensive) computation of the exact Hessian $\mathbf{H}_k$, the current version of the SAO*i* algorithm use only approximate diagonal terms, estimated from suitable intervening variable expressions for the objective and constraint functions, to approximate $\mathbf{H}_k$. As hinted at, the use of diagonal curvatures (*i.e.* neglecting the interaction between variables) can of course not be expected to yield comparable accuracy to using full Hessian information. Therefore, we are motivated to develop a more 'universal' SAO algorithm that provides a 'better' approximation to the full (exact) Hessian $\mathbf{H}_k$, while keeping computational and storage requirements at a minimum. Since the amount of storage required by a l-BFGS approach can be controlled, and no knowledge of the sparsity structure of the true Hessian matrix $\mathbf{H}_k$ or of the separability of the problem is needed, we propose the use of a l-BFGS method to approximate Hessian information.

The QP subproblems P_{QP} can be solved using any QP solver. Currently, the SAO*i* algorithm has interfaces to the (commercial) NAG QP solver E04NQF, and to the QP solvers LSQP, QPA, QPB and QPC available in GALAHAD (Gould *et al.*, 2004). (GALAHAD is not open source, but is freely available for academic use.) For the purpose of this study, the QPB solver has been found to be more efficient than the other available QP solvers. The QPB solver uses a primal dual feasible interior-point trust region method for quadratic programming. The quadratic programming problem involves the minimization of a diagonal quadratic objective function subject to linear inequality constraints and simple bound constraints. It solves the problem in two phases. The first phase involves finding an initial feasible point for the set of constraints, while the second phase aims to maintain the feasibility while iterating towards the optimum point. For the implementation of the QPB solver in the SAO*i* algorithm, Hessian information of the Lagrangian of the quadratic approximate subproblem should be determined (or approximated). Since the methods applied in the QPB solver already caters for the bound constraints and the dual variables used, any unconstrained optimization method can be used to approximate the Hessian information. It is 'unfortunate' that the l-BFGS method proposed in this study does not take advantage of the sparsity structure of the true Hessian, as the QPB solver takes full advantage of any zero elements in the Hessian.

In the next chapter, we discuss the implementation of a (straightforward) variant of the l-BFGS method into the existing SAO*i* algorithm to approximate Hessian information when solving the quadratic approximate subproblems using the QPB solver. This is followed by a summary of the results obtained using a series of test problems to evaluate the proposed method.

Chapter 3

l-BFGS: Implementation and Results

3.1 Introduction

We propose the use of a l-BFGS formulation in a SAO algorithm for constrained optimization. To maintain a positive definite approximation to the full Hessian of the Lagrangian of the diagonal quadratic approximate subproblem, we implement a (straightforward) variant of the l-BFGS method (Chapter 2). The approximate subproblem is then solved using the GALAHAD QPB solver. Although more expensive in terms of computational requirements, this approach proved to be more suitable for implementation in the current version of the SAO*i* algorithm. Two different choices of the initial matrix, as well as the impact of the number of previous iterations used in the updating formulae, are also explored.

A range of inequality and equality constrained optimization problems, including unimodal, multimodal and other specialized examples, are used to measure the performance of the implemented method (subsolver set to 27 in the SAO*i* algorithm) against that of the intermediate variable approach (subsolver set to 25 in the SAO*i* algorithm) and a CPLEX optimization solver. (Detail on the optimization problems used is available upon request.) The intermediate variable approach constructs a (separable) diagonal quadratic approximate subproblem, which is solved using the GALAHAD LSQP solver. The CPLEX optimization solver makes use of a Barrier method, a primal Simplex method and a dual Simplex method to solve Quadratic Programming optimization problems with linear constraints. The Barrier method is an interior point method that searches for an optimum by traversing within the feasible domain using an unconstrained optimization method such as the Newton method. The Simplex methods are active set methods. Unlike interior point methods, active set methods search along the boundary of the feasible domain to find an optimum. Although originally developed to solve Linear Programming (LP) problems, Simplex methods have been extended to solve convex Quadratic Programming optimization problems. In addition to using the three different methods used in CPLEX, a functionality is also available for the treatment of infeasible optimization problems. The intermediate variable approach constructs a (separable) diagonal quadratic approximate subproblem, which is solved using the GALAHAD LSQP solver.

Detail on the (straightforward) l-BFGS algorithm, as implemented in the existing SAO*i* algorithm, follows. (Refer to Appendix 7.1 for the FORTRAN source code.)

3.2 The l-BFGS Algorithm

Let $\boldsymbol{x}_k$ be the current iteration. If $k = 1$, simply let $\mathbf{H}_k = \mathbf{I}$. Otherwise, given $\mathbf{x}_i$ and $\mathbf{g}_i$ for the r most recent iterations, approximate the Hessian $\mathbf{H}_k$ of the Lagrangian as follows.

```
Step 1  For $i = k - r, k - r + 1, \dots, k - 1$
            $\boldsymbol{s}_i \leftarrow \boldsymbol{x}_{i+1} - \boldsymbol{x}_i$ ;
            $\boldsymbol{y}_i \leftarrow (\nabla f_{0,i+1} - \nabla f_{0,i}) + \sum_{j=1}^{m} \lambda_j (\nabla f_{j,i+1} - \nabla f_{j,i})$ ;
        end (for)

Step 2  If $\mathbf{s}_{k-1}^T \mathbf{y}_{k-1} = 0$ for all $i = k - r, k - r + 1, \dots, k - 1$
          $\mathbf{H}_k \leftarrow \mathbf{0}$ ;
          break (do not perform steps 3, 4, and 5)
        end (if)

Step 3  If $\mathbf{s}_{k-1}^T \mathbf{y}_{k-1} > 0$
            $\sigma_k \leftarrow \mathbf{s}_{k-1}^T \mathbf{y}_{k-1} / \mathbf{s}_{k-1}^T \mathbf{s}_{k-1}$ ;
            $\mathbf{H}_{k,0} \leftarrow \sigma_k \mathbf{I}$ ;
        else
            $\mathbf{H}_{k,0} \leftarrow \mathbf{I}$ ;
        end (if)

Step 4  For $i = k - r, k - r + 1, \dots, k - 1$
          If $\mathbf{s}_i^T \mathbf{y}_i > 0$
            $\mathbf{b}_i \leftarrow \mathbf{y}_i / (\mathbf{y}_i^T \mathbf{s}_i)^{1/2}$ ;
            $\tilde{\mathbf{a}}_i \leftarrow \mathbf{H}_{k,0} \boldsymbol{s}_i + \sum_{j=0}^{i-1} [(\boldsymbol{b}_j^T \boldsymbol{s}_i) \boldsymbol{b}_j - (\boldsymbol{a}_j^T \boldsymbol{s}_i) \boldsymbol{a}_j]$ ;
            $\mathbf{a}_i \leftarrow \tilde{\mathbf{a}}_i / (\mathbf{s}^T{}_i \tilde{\mathbf{a}}_i)^{1/2}$ ;
          else
            $\mathbf{b}_i \leftarrow \mathbf{0}$ ;
            $\tilde{\mathbf{a}}_i \leftarrow \mathbf{0}$ ;
          end (if)
        end (for)
```

Step 5 Compute
$$\mathbf{H}_k \leftarrow \mathbf{H}_{k,0} + \textstyle\sum_{i=k-r}^{k-1} [\boldsymbol{b}_i \boldsymbol{b}_i^T - \boldsymbol{a}_i \boldsymbol{a}_i^T] \; ;$$

3.3 Results

In the following sections, k^* represents the number of outer iterations and l^* the number of inner iterations performed to find the reported optimum point $\boldsymbol{x}^*$. f_0^* denotes the objective function value and $|f_j^*|$ the maximum constraint violation at $\boldsymbol{x}^*$. $||\boldsymbol{x}^*||$ is the Euclidian norm of $\boldsymbol{x}^*$.

3.3.1 A Unimodal Example: *2-Bar Default* Problem

Example Problem Formulation

This is a simple problem that involves the simultaneous shape and sizing design of a *2-bar truss* (Svanberg, 1995). Analytically, the problem may be expressed as

$$\begin{aligned} \min_{\boldsymbol{x}} f_0(\boldsymbol{x}) &= x_1 \sqrt{1 + x_2^2}\,, \\ \text{subject to } \; f_1(\boldsymbol{x}) &= 0.124 \sqrt{1 + x_2^2} \left(\frac{8}{x_1} + \frac{1}{x_1 x_2} \right) - 1 \le 0\,, \\ f_2(\boldsymbol{x}) &= 0.124 \sqrt{1 + x_1^2} \left(\frac{8}{x_1} - \frac{1}{x_1 x_2} \right) - 1 \le 0\,, \\ & 0.2 \le x_1 \le 4.0\,, \\ & 0.1 \le x_2 \le 1.6\,. \end{aligned} \tag{3.3.1}$$

Running the SAOi algorithm with exact Hessian information

Using exact second-order information of the original optimization problem (3.3.1) to construct the Hessian of the Lagrangian of the diagonal quadratic approximate subproblem, gives

$$\begin{aligned} \mathbf{H}_k = & \begin{bmatrix} 0 & \frac{x_2}{\sqrt{1+x_2^2}} \\ \frac{x_2}{\sqrt{1+x_2^2}} & \frac{x_1}{(1+x_2^2)^{\frac{3}{2}}} \end{bmatrix} \\ & + \lambda_{1,k}\, 0.124 \begin{bmatrix} \sqrt{1 + x_2^2}(\frac{16}{x_1^3} + \frac{2}{x_1^3 x_2}) & (1 + x_2^2)^{-\frac{1}{2}}(\frac{1}{x_1^2 x_2^2} - \frac{8x_2}{x_1^2}) \\ (1 + x_2^2)^{-\frac{1}{2}}(\frac{1}{x_1^2 x_2^2} - \frac{8x_2}{x_1^2}) & (1 + x_2^2)^{-\frac{3}{2}}(\frac{8}{x_1} + \frac{3}{x_1 x_2} + \frac{2}{x_1 x_2^3}) \end{bmatrix} \\ & + \lambda_{2,k}\, 0.124 \begin{bmatrix} \sqrt{1 + x_2^2}(\frac{16}{x_1^3} - \frac{2}{x_1^3 x_2}) & (1 + x_2^2)^{-\frac{1}{2}}(-\frac{1}{x_1^2 x_2^2} - \frac{8x_2}{x_1^2}) \\ (1 + x_2^2)^{-\frac{1}{2}}(-\frac{1}{x_1^2 x_2^2} - \frac{8x_2}{x_1^2}) & (1 + x_2^2)^{-\frac{3}{2}}(\frac{8}{x_1} - \frac{3}{x_1 x_2} - \frac{2}{x_1 x_2^3}) \end{bmatrix}, \end{aligned} \tag{3.3.2}$$

k	l	f_0	$\lvert f_j\rvert$	$\lvert\lvert \boldsymbol{x}_{k-1} - \boldsymbol{x}_k\rvert\rvert$
0	0	1.6770510E+00	0.00E+00	
1	0	1.4043183E+00	1.94E-01	3.24E-01
2	0	1.4830853E+00	4.92E-02	8.94E-02
3	0	1.4867258E+00	1.79E-02	7.41E-02
4	0	1.5049828E+00	2.47E-03	3.26E-02
5	0	1.5086085E+00	2.91E-05	4.03E-03
6	0	1.5086524E+00	2.82E-09	4.27E-05

Table 3.1: Iteration results: *2-bar truss* problem using exact Hessian information

k	l	f_0	$\lvert f_j\rvert$	$\lvert\lvert \boldsymbol{x}_{k-1} - \boldsymbol{x}_k\rvert\rvert$
0	0	1.6770510E+00	0.00E+00	
1	0	1.4043183E+00	1.94E-01	3.24E-01
2	0	1.4255354E+00	7.53E-02	9.90E-02
3	0	1.4790102E+00	2.00E-02	8.22E-02
4	0	1.5083294E+00	5.07E-04	2.48E-02
5	0	1.5082815E+00	2.46E-04	1.28E-02
6	0	1.5086523E+00	8.39E-08	4.00E-04
7	0	1.5086524E+00	6.18E-11	6.22E-06

Table 3.2: Iteration results: *2-bar truss* problem using the proposed l-BFGS approach

where $\lambda_{1,k}$ and $\lambda_{2,k}$ are the Lagrangian multipliers for $f_1(\mathbf{x})$ and $f_2(\mathbf{x})$ respectively.

The iteration results for the *2-bar truss* problem (3.3.1) when using exact second-order information to construct the Hessian (3.3.2) are given in Table 3.1. Convergence to an objective function value $f_0^* = 1.5086524$ (with a maximum constraint violation $|f_j^*| = 2.82E - 09$) is achieved within six outer iterations.

Running the SAO*i* algorithm using a memoryless BFGS approach

Using the l-BFGS approach (with $r = 2$) to approximate second-order information results in only one additional iteration (Table 3.2). Convergence to $f_0^* = 1.5086524$ (with $|f_j^*| = 6.18E - 11$) is achieved within seven outer iterations. The additional iteration is a small price to pay if it means that we are able to avoid the (cumbersome and error-prone) computation of exact second-order information, while achieving increased accuracy w.r.t. the maximum constraint violation.

Influence of choice of $\mathbf{H}_{k,0}$ and the value of r on iteration results

The influence of the choice of starting matrix $\mathbf{H}_{k,0}$ and the amount of previous iteration points used to construct the approximate Hessian is now explored. The iteration results for $\mathbf{H}_{k,0} = \mathbf{I}$ and $\mathbf{H}_{k,0} = \sigma_k\mathbf{I}$ for different values of r are given in Table 3.3. The results indicate that (for this problem) setting the starting matrix equal to the identity matrix, *i.e.* $\mathbf{H}_{k,0} = \mathbf{I}$, leads to an increase in the amount of iterations performed for small values of r. Also, increasing the value of r does not lead to increased accuracy irrespective of the starting matrix used. On the contrary, even though the same minimum function value is achieved within the same amount of iterations for all values of r, the memoryless approach has the smallest value for the maximum constraint violation. However, for higher values it has no effect on the number of iterations, but does lead to a slightly higher maximum constraint violation.

	$\mathbf{H}_{k,0} = \mathbf{I}$		$\mathbf{H}_{k,0} = \sigma_k\mathbf{I}$	
r	k^*	f_0^* $\lvert f_j^*\rvert$	k^*	f_0^* $\lvert f_j^*\rvert$
2	7	1.5086524E+00 6.17987883E-11	10	1.50865242E+00 4.97883512E-10
3	7	1.5086524E+00 5.08165887E-09	8	1.50865242E+00 1.95310434E-12
4	7	1.5086524E+00 2.27637287E-09	7	1.50865240E+00 1.48225567E-08
5	7	1.5086524E+00 2.17933804E-09	7	1.50865241E+00 2.41487719E-09
6	7	1.5086524E+00 2.19295782E-09	7	1.50865240E+00 1.39842267E-08
7	7	1.5086524E+00 2.19297491E-09	7	1.50865240E+00 1.39846998E-08

Table 3.3: Iteration results: *2-bar truss* problem: different values of r and different starting matrices

3.3.2 A Unimodal Example: *CCSA* Problems of Svanberg

Example Problem Formulation

The two *ccsa* problems (*ccsa1* and *ccsa2*) discussed next are *artificial* problems proposed by Svanberg (Svanberg, 2002). The general structure of these problems resembles the corresponding structure of topology optimization problems. The problems are non-convex with a large number of optimization variables, upper and lower bounds on all variables, a relatively small number of inequality constraints, and the Hessian matrices of the objective and constraint functions are dense. However, the problems are not geniune structural optimization problems that require a finite element approach, but are explicitly stated *academic* problems. Problem *ccsa1* consists of a strictly convex objective function subject to strictly concave constraints, and problem *ccsa2* consists of a strictly concave objective subject to strictly convex constraints. The problems are given below.

ccsa1:

$$\begin{aligned} \min_{x} f_0(\boldsymbol{x}) &= \boldsymbol{x}^T\boldsymbol{S}\boldsymbol{x}\,, \\ \text{subject to } f_1(\boldsymbol{x}) &= \frac{n}{2} - \boldsymbol{x}^T\boldsymbol{P}\boldsymbol{x} \leq 0\,, \\ f_2(\boldsymbol{x}) &= \frac{n}{2} - \boldsymbol{x}^T\boldsymbol{Q}\boldsymbol{x} \leq 0\,, \\ -1 \leq x_i &\leq 1\,, \qquad i = 1,2,\ldots,n \end{aligned} \tag{3.3.3}$$

ccsa2:

$$\begin{aligned} \min_{x} f_0(\boldsymbol{x}) &= -\boldsymbol{x}^T\boldsymbol{S}\boldsymbol{x}\,, \\ \text{subject to } f_1(\boldsymbol{x}) &= \boldsymbol{x}^T\boldsymbol{P}\boldsymbol{x} - \frac{n}{2} \leq 0\,, \\ f_2(\boldsymbol{x}) &= \boldsymbol{x}^T\boldsymbol{Q}\boldsymbol{x} - \frac{n}{2} \leq 0\,, \\ -1 \leq x_i &\leq 1\,, \qquad i = 1,2,\ldots,n \end{aligned} \tag{3.3.4}$$

S, **P** and **Q** are $n \times n$ symmetric, positive definite matrices. All elements are constants and are given by

$$\begin{aligned} s_{ij} &= \frac{2 + \sin 4\pi\alpha_{ij}}{(1 + |i-j|)\ln n}, \qquad i,j = 1,2,\ldots,n \\ p_{ij} &= \frac{1 + 2\alpha_{ij}}{(1 + |i-j|)\ln n}, \qquad i,j = 1,2,\ldots,n \\ q_{ij} &= \frac{3 - 2\alpha_{ij}}{(1 + |i-j|)\ln n}, \qquad i,j = 1,2,\ldots,n \\ \alpha_{ij} &= \frac{i + j - 2}{2n - 2}, \qquad i,j = 1,2,\ldots,n. \end{aligned} \tag{3.3.5}$$

For this study, we use $n = 200$. The starting points are set to $x_i = 0.5 \forall i$ and $x_i = 0.25 \forall i$ for problems *ccsa1* and *ccsa2* respectively.

Running the SAOi algorithm with exact Hessian information

Using exact second-order information of the original optimization problems (3.3.3 and 3.3.4) to construct the Hessian of the Lagrangian of the diagonal quadratic approximate subproblems, gives

$$\textit{ccsa1:} \qquad \mathbf{H}_k = 2\boldsymbol{S} - 2\lambda_{1,k}\boldsymbol{P} - 2\lambda_{2,k}\boldsymbol{Q}, \tag{3.3.6}$$

$$\textit{ccsa2:} \qquad \mathbf{H}_k = -2\boldsymbol{S} + 2\lambda_{1,k}\boldsymbol{P} + 2\lambda_{2,k}\boldsymbol{Q}, \tag{3.3.7}$$

where $\lambda_{1,k}$ and $\lambda_{2,k}$ are the Lagrangian multipliers for $f_1(\mathbf{x})$ and $f_2(\mathbf{x})$ respectively, and **S**, **P** and **Q** determined using (3.3.5).

The iteration results for the *ccsa1* and *ccsa2* problems (3.3.3 and 3.3.4) when using exact second-order information to construct the Hessians (3.3.6 and 3.3.7) are given in Tables 3.4 and 3.5. For problem *ccsa1*, convergence to an objective function value $f_0^* = 5.1045787E + 01$ (with a maximum constraint violation $|f_j^*| = 0.00E + 00$) is achieved within 15 outer iterations. For problem *ccsa2*, convergence to an objective function value $f_0^* = -1.4895421E + 02$ (with a maximum constraint violation $|f_j^*| = 1.05E - 08$) is achieved within 16 outer iterations.

Running the SAO*i* algorithm using a memoryless BFGS approach

Using the l-BFGS approach (with $r = 2$) to approximate second-order information results in additional iterations for both problems (Tables 3.6, 3.7, 3.8 and 3.9). For problem *ccsa1*, convergence to $f_0^* = 5.1045788E + 01$ (with $|f_j^*| = 0.00E + 00$) is achieved within 74 outer iterations. For problem *ccsa2*, convergence to $f_0^* = -1.4895421E + 02$ (with $|f_j^*| = 5.24E - 08$) is achieved within 80 outer iterations. Although the additional iterations seem like a high price to pay, it still means that we are able to avoid the (cumbersome and error-prone) computation of exact second-order information while achieving the same level of accuracy.

Influence of choice of $\mathbf{H}_{k,0}$ and the value of r on iteration results

The influence of the choice of starting matrix $\mathbf{H}_{k,0}$ and the amount of previous iteration points used to construct the approximate Hessian is now explored. The iteration results for problems *ccsa1* and *ccsa2* for $\mathbf{H}_{k,0} = \mathbf{I}$ and $\mathbf{H}_{k,0} = \sigma_k\mathbf{I}$ for different values of r are given in Tables 3.10 and 3.11. The results indicate that (for these problems) setting the starting matrix equal to the identity matrix, *i.e.* $\mathbf{H}_{k,0} = \mathbf{I}$, leads to an increase in the amount of iterations performed for small

k	l	f_0	$\lvert f_j \rvert$	$\lVert \boldsymbol{x}_{k-1} - \boldsymbol{x}_k \rVert$
0	0	1.6637060E+02	0.00E+00	
1	0	9.5368269E+01	0.00E+00	3.13E+00
2	0	8.3182121E+01	0.00E+00	4.58E+00
3	0	6.5454020E+01	0.00E+00	2.63E+00
4	0	6.0404977E+01	0.00E+00	2.12E+00
5	0	5.8747846E+01	0.00E+00	1.43E+00
6	0	5.7722298E+01	0.00E+00	1.15E+00
7	0	5.7776694E+01	0.00E+00	1.38E+00
8	0	5.9295671E+01	0.00E+00	1.93E+00
9	0	5.6339797E+01	0.00E+00	1.99E+00
10	0	5.4248563E+01	0.00E+00	1.96E+00
11	0	5.3070556E+01	0.00E+00	1.99E+00
12	0	5.2244137E+01	0.00E+00	1.85E+00
13	0	5.1106556E+01	0.00E+00	5.12E-01
14	0	5.1046029E+01	0.00E+00	3.71E-02
15	0	5.1045787E+01	0.00E+00	3.74E-04

Table 3.4: Iteration results: *ccsa1* problem using exact Hessian information

k	l	f_0	$\lvert f_j \rvert$	$\lVert \boldsymbol{x}_{k-1} - \boldsymbol{x}_k \rVert$
0	0	-4.1592649E+01	0.00E+00	
1	0	-1.8521316E+02	6.05E+01	5.64E+00
2	0	-1.8158276E+02	3.78E+01	4.81E+00
3	0	-1.6269993E+02	1.67E+01	3.90E+00
4	0	-1.4997545E+02	6.86E+00	2.38E+00
5	0	-1.4744936E+02	4.72E+00	1.95E+00
6	0	-1.4499770E+02	2.15E+00	1.25E+00
7	0	-1.4600697E+02	2.71E+00	1.32E+00
8	0	-1.4780300E+02	3.95E+00	1.48E+00
9	0	-1.5164905E+02	6.46E+00	1.70E+00
10	0	-1.5262858E+02	5.63E+00	1.91E+00
11	0	-1.5356897E+02	5.08E+00	1.97E+00
12	0	-1.5382707E+02	4.53E+00	1.95E+00
13	0	-1.5153647E+02	2.29E+00	1.54E+00
14	0	-1.4905990E+02	9.30E-02	4.18E-01
15	0	-1.4895460E+02	3.43E-04	3.06E-02
16	0	-1.4895421E+02	1.05E-08	1.84E-04

Table 3.5: Iteration results: *ccsa2* problem using exact Hessian information

values of r. Increasing the value of r leads to an initial decrease in the amount of iterations performed irrespective of the starting matrix used. As r is increased further, only slight changes are noticed in the amount of iterations performed to reach the optimum point. Changes to the starting matrix used and the value of r do not lead to any significant changes in the accuracy acheived. However, for problem *ccsa2*, when the starting matrix is set equal to the identity matrix, *i.e.* $\mathbf{H}_{k,0} = \mathbf{I}$, and a memoryless approach ($r = 2$) is used, the iteration process terminates on the allowed maximum number of outer iterations performed and the desired optimum point is not obtained.

3.3.3 A Multimodal Example: *6-Hump Camelback* Problem

Example Problem Formulation

Analytically, the *6-hump camelback* problem can be expressed as

$$\begin{aligned} \min_{x} f_0(\boldsymbol{x}) &= \left(4 - 2.1x_1^2 + \frac{1}{3}x_1^4\right)x_1^2 + x_1x_2 + \left(-4 + 4x_2^2\right)x_2^2, \\ \text{subject to } & -3.0 \le x_1 \le 3.0, \\ & -2.0 \le x_2 \le 2.0. \end{aligned} \tag{3.3.8}$$

Running the SAOi algorithm with exact Hessian information

Using exact second-order information of the original optimization problem (3.3.8) to construct the Hessian of the diagonal quadratic approximate subproblem, gives

$$\mathbf{H}_k = \begin{bmatrix} 8 - 25.2x_1^2 + 10x_1^4 & 1 \\ 1 & -8 + 48x_2^2 \end{bmatrix}. \tag{3.3.9}$$

The iteration results for the *6-hump camelback* problem (3.3.8) when using exact second-order information to construct the Hessian (3.3.9) and invoking a global search pattern (a multi-start approach) is given in Table 3.12. A global optimum of $f_0^* = -1.0316285E + 00$ is found with a probability of convergence of 0.9908.

Running the SAOi algorithm using a memoryless BFGS approach

Using the proposed l-BFGS approach (with $r = 2$) to approximate second-order information, while still invoking a global search pattern, results in a decrease in the amount of trials performed and an increased probability that a global optimum has been found (Table 3.13). A global optimum of $f_0^* = -1.0316285E+00$ is found with a probability of convergence of 0.9959.

3.3.4 Influence of choice of $\mathbf{H}_{k,0}$ and the value of r on l-BFGS performance

From the above results (obtained using only a limited amount of test problems), we can conclude that the use of exact Hessian information may (sometimes) lead to a significant decrease in the number of iterations performed and to increased accuracy. However, there seems to be problems for which the increase in the number of iterations is insignificant and no notable decrease in the accuracy is obtained. Based on the limited amount of testing done, it is still recommended that approximate Hessian information is used in the approximate subproblems to avoid the calculation of the exact Hessian.

For the unimodal test problems performed, the choice of initial matrix $\mathbf{H}_{k,0}$ has a great influence on the amount of iterations performed for low values of r. However, when r is increased, an initial decrease in the number of iterations is observed and the differences between the number of iterations performed for different initial matrices become insignificant. It is noted again that increasing r only leads to a significant decrease in the number of outer iterations performed for small values of r. However, any further increases in r does not seem to hold any advantage in terms of the number of iterations performed or the accuracy achieved.

3.3.5 Performance Measurement of a Memoryless Approach

In this section, the performance of the the memoryless l-BFGS approach using $r = 2$ (subsolver 27) is compared to that of the CPLEX solver and the intermediate variable approach (subsolver 25) using a series of problems available for testing in the SAO*i* algorithm. The numerical results are given in Tables 3.14 and 3.15. Based on the number of outer iterations, the numerical comparison of the memoryless l-BFGS method with the other two methods can be summarized as follows:

- The memoryless l-BFGS implementation converges in less iterations than the intermediate variable approach in 25 problems, while the intermediate variable approach converges in less iterations in 15 problems. It converges in the same amount of iterations as the intermediate variable approach in 10 problems.
- The memoryless l-BFGS implementation converges in less iterations than the CPLEX solver in 30 problems, while the CPLEX solver converges in less iterations in 13 problems. It converges in the same amount of iterations as the CPLEX solver in 5 problems.

From the numerical results, the memoryless l-BFGS implementation is more efficient and robust than both the intermediate variable approach and the CPLEX solver. Especially considering that (unlike the methods to which it is being compared) the l-BFGS implementation does not apply any preconditioning. (However, we are reminded of the *no free lunch* (NFL) theorems for optimization (Wolpert and Macready, 1997), which establish that all algorithms that search for the optimum of an objective function perform exactly the same when averaged over all possible objective functions. In simpler terms, if algorithm A outperforms algorithm B on some optimization problems, then there must exist as many other problems where B outperforms A.) The l-BFGS implementation is definitely a feasible method for approximating second-order information when solving the approximate subproblems using a QP solver, with no need to use the complete iteration history or store fully dense $n \times n$ Hessian approximations. It is relatively simple to implement. However, a drawback of the l-BFGS implementation is that it is more expensive in terms of computational requirements than the other two methods and that it does not (always) retain the conservative nature of the convex and separable approximate subproblems. For very large scale optimization problems, the QPB solver 'struggles' to return a solution to the diagonal quadratic subproblem when the approximation to the Hessian is dense and ill conditioned. In order to overcome this problem, in depth knowledge of the GALAHAD solver will be required or the use of other QP solvers should be explored.

k	l	f_0	$\lvert f_j \rvert$	$\lVert \boldsymbol{x}_{k-1} - \boldsymbol{x}_k \rVert$
0	0	1.6637060E+02	0.00E+00	
1	0	1.0292595E+02	0.00E+00	4.88E+00
2	0	8.5132023E+01	0.00E+00	3.96E+00
3	0	7.2993430E+01	0.00E+00	2.77E+00
4	0	6.1698793E+01	0.00E+00	1.35E+00
5	0	5.8959243E+01	0.00E+00	4.77E-01
6	0	5.8860789E+01	0.00E+00	9.54E-01
7	0	6.0002158E+01	0.00E+00	1.59E+00
8	0	5.8903674E+01	0.00E+00	9.69E-01
9	0	5.8311719E+01	0.00E+00	6.58E-01
10	0	5.7751890E+01	0.00E+00	3.03E-01
11	0	5.7616276E+01	0.00E+00	2.23E-01
12	0	5.7688739E+01	0.00E+00	6.92E-01
13	0	6.8308855E+01	0.00E+00	2.61E+00
14	0	6.1304263E+01	0.00E+00	2.06E+00
15	0	5.7960899E+01	0.00E+00	6.63E-01
16	0	5.7272260E+01	0.00E+00	3.10E-01
17	0	5.7120269E+01	0.00E+00	2.27E-01
18	0	5.7132639E+01	0.00E+00	5.76E-01
19	0	5.7525411E+01	0.00E+00	1.03E+00
20	0	6.1056489E+01	0.00E+00	1.54E+00
21	0	5.8217374E+01	0.00E+00	1.17E+00
22	0	5.6541384E+01	0.00E+00	3.01E-01
23	0	5.6356599E+01	0.00E+00	3.09E-01
24	0	5.8909191E+01	0.00E+00	1.49E+00
25	0	5.6536162E+01	0.00E+00	6.94E-01
26	0	5.5904576E+01	0.00E+00	3.71E-01
27	0	5.5662530E+01	0.00E+00	3.69E-01
28	0	8.8188257E+01	0.00E+00	3.67E+00
29	0	6.4612635E+01	0.00E+00	2.52E+00
30	0	5.7433517E+01	0.00E+00	1.27E+00
31	0	5.5058681E+01	0.00E+00	4.95E-01
32	0	5.4775396E+01	0.00E+00	9.88E-01
33	0	6.7832124E+01	0.00E+00	3.40E+00
34	0	5.8273041E+01	0.00E+00	2.54E+00
35	0	5.3853417E+01	0.00E+00	1.05E+00
36	0	5.3079539E+01	0.00E+00	9.91E-01
37	0	5.2887663E+01	0.00E+00	1.66E+00
38	0	5.8617141E+01	0.00E+00	2.47E+00
39	0	5.3545150E+01	0.00E+00	1.62E+00
40	0	5.1300982E+01	0.00E+00	2.95E-01

Table 3.6: Iteration results: *ccsa1* problem using the proposed l-BFGS approach

k	l	f_0	$\lvert f_j \rvert$	$\lVert \boldsymbol{x}_{k-1} - \boldsymbol{x}_k \rVert$
41	0	5.1230397E+01	0.00E+00	3.12E-01
42	0	5.1170924E+01	0.00E+00	4.22E-01
43	0	5.1925077E+01	0.00E+00	8.77E-01
44	0	5.1388706E+01	0.00E+00	5.91E-01
45	0	5.1074193E+01	0.00E+00	1.02E-01
46	0	5.1058949E+01	0.00E+00	4.71E-02
47	0	5.1054658E+01	0.00E+00	6.94E-02
48	0	5.1052234E+01	0.00E+00	1.32E-01
49	0	5.1064140E+01	0.00E+00	2.02E-01
50	0	5.1053000E+01	0.00E+00	8.36E-02
51	0	5.1047144E+01	0.00E+00	2.79E-02
52	0	5.1046232E+01	0.00E+00	9.13E-03
53	0	5.1046150E+01	0.00E+00	1.84E-02
54	0	5.1046857E+01	0.00E+00	4.41E-02
55	0	5.1046466E+01	0.00E+00	2.17E-02
56	0	5.1046003E+01	0.00E+00	1.01E-02
57	0	5.1045902E+01	0.00E+00	4.45E-03
58	0	5.1046231E+01	0.00E+00	2.64E-02
59	0	5.1046156E+01	0.00E+00	1.44E-02
60	0	5.1045916E+01	0.00E+00	8.05E-03
61	0	5.1045852E+01	0.00E+00	3.64E-03
62	0	5.1046015E+01	0.00E+00	1.46E-02
63	0	5.1045883E+01	0.00E+00	6.51E-03
64	0	5.1045826E+01	0.00E+00	8.78E-04
65	0	5.1045823E+01	0.00E+00	3.74E-03
66	0	5.1046101E+01	0.00E+00	3.02E-02
67	0	5.1045975E+01	0.00E+00	1.51E-02
68	0	5.1045846E+01	0.00E+00	6.99E-03
69	0	5.1045806E+01	0.00E+00	3.00E-03
70	0	5.1045796E+01	0.00E+00	2.76E-03
71	0	5.1045798E+01	0.00E+00	4.00E-03
72	0	5.1045792E+01	0.00E+00	1.61E-03
73	0	5.1045789E+01	0.00E+00	7.79E-04
74	0	5.1045788E+01	0.00E+00	3.14E-04

Table 3.7: Iteration results: *ccsa1* problem using the proposed l-BFGS approach (continued)

k	l	f_0	$\lvert f_j \rvert$	$\lvert\lvert \boldsymbol{x}_{k-1} - \boldsymbol{x}_k \rvert\rvert$
0	0	-4.1592649E+01	0.00E+00	
1	0	-1.6946254E+02	4.47E+01	4.34E+00
2	0	-1.8687914E+02	4.26E+01	4.56E+00
3	0	-1.6840234E+02	2.56E+01	3.85E+00
4	0	-1.5001685E+02	8.48E+00	1.88E+00
5	0	-1.4195693E+02	7.29E-01	6.42E-01
6	0	-1.4321370E+02	1.34E+00	1.07E+00
7	0	-1.4278391E+02	8.41E-01	7.96E-01
8	0	-1.4289605E+02	7.28E-01	7.59E-01
9	0	-1.4455869E+02	2.28E+00	1.27E+00
10	0	-1.4506873E+02	2.47E+00	1.24E+00
11	0	-1.4494744E+02	2.59E+00	1.19E+00
12	0	-1.4390998E+02	1.16E+00	6.76E-01
13	0	-1.4295356E+02	1.50E-01	2.53E-01
14	0	-1.4286054E+02	3.39E-02	1.63E-01
15	0	-1.4734597E+02	5.57E+00	1.91E+00
16	0	-1.4654784E+02	2.94E+00	1.30E+00
17	0	-1.4325725E+02	1.46E-01	2.83E-01
18	0	-1.4342852E+02	2.03E-01	4.16E-01
19	0	-1.4442976E+02	9.23E-01	9.02E-01
20	0	-1.4419629E+02	5.42E-01	5.90E-01
21	0	-1.4390177E+02	1.44E-01	3.18E-01
22	0	-1.4548855E+02	1.27E+00	9.83E-01
23	0	-1.4721234E+02	2.72E+00	1.35E+00
24	0	-1.4823747E+02	3.03E+00	1.39E+00
25	0	-1.5394690E+02	7.74E+00	2.03E+00
26	0	-1.4921472E+02	2.46E+00	1.18E+00
27	0	-1.4651824E+02	2.59E-01	3.81E-01
28	0	-1.5115006E+02	3.56E+00	1.48E+00
29	0	-1.4919560E+02	1.26E+00	9.12E-01
30	0	-1.5122329E+02	2.54E+00	1.37E+00
31	0	-1.5837590E+02	1.02E+01	2.74E+00
32	0	-1.5573936E+02	5.31E+00	1.94E+00
33	0	-1.4914026E+02	3.58E-01	4.30E-01
34	0	-1.4891334E+02	1.20E-01	2.88E-01
35	0	-1.4907125E+02	1.94E-01	4.25E-01
36	0	-1.5154536E+02	2.57E+00	1.43E+00
37	0	-1.5099107E+02	1.64E+00	9.79E-01
38	0	-1.4900194E+02	6.78E-02	1.84E-01
39	0	-1.4894450E+02	1.31E-02	8.39E-02
40	0	-1.4894231E+02	6.15E-03	7.81E-02

Table 3.8: Iteration results: *ccsa2* problem using the proposed l-BFGS approach

k	l	f_0	$\|f_j\|$	$\|\|\boldsymbol{x}_{k-1} - \boldsymbol{x}_k\|\|$
41	0	-1.4905650E+02	1.07E-01	3.14E-01
42	0	-1.4899384E+02	3.25E-02	1.21E-01
43	0	-1.4895282E+02	5.81E-04	2.01E-02
44	0	-1.4895480E+02	1.55E-03	3.85E-02
45	0	-1.4895449E+02	1.20E-03	3.33E-02
46	0	-1.4895389E+02	3.78E-04	1.87E-02
47	0	-1.4895421E+02	3.88E-04	2.55E-02
48	0	-1.4895514E+02	9.55E-04	4.22E-02
49	0	-1.4895813E+02	3.65E-03	8.02E-02
50	0	-1.4895681E+02	2.91E-03	4.09E-02
51	0	-1.4895602E+02	1.42E-03	3.08E-02
52	0	-1.4895413E+02	2.99E-05	5.79E-03
53	0	-1.4895426E+02	1.01E-04	1.09E-02
54	0	-1.4895486E+02	6.36E-04	2.65E-02
55	0	-1.4895521E+02	9.39E-04	3.14E-02
56	0	-1.4895537E+02	1.03E-03	3.52E-02
57	0	-1.4895599E+02	2.54E-03	3.98E-02
58	0	-1.4895679E+02	2.06E-03	3.54E-02
59	0	-1.4895420E+02	2.02E-06	1.12E-03
60	0	-1.4895420E+02	5.49E-07	9.29E-04
61	0	-1.4895423E+02	2.05E-05	7.46E-03
62	0	-1.4895512E+02	8.84E-04	4.86E-02
63	0	-1.4895465E+02	4.34E-04	1.48E-02
64	0	-1.4895433E+02	1.22E-04	8.10E-03
65	0	-1.4895419E+02	4.63E-06	2.15E-03
66	0	-1.4895421E+02	1.94E-05	5.25E-03
67	0	-1.4895442E+02	2.09E-04	1.80E-02
68	0	-1.4895441E+02	1.82E-04	1.42E-02
69	0	-1.4895430E+02	8.36E-05	9.29E-03
70	0	-1.4895426E+02	6.79E-05	4.88E-03
71	0	-1.4895423E+02	1.77E-05	2.80E-03
72	0	-1.4895421E+02	6.34E-07	5.81E-04
73	0	-1.4895421E+02	2.36E-07	5.68E-04
74	0	-1.4895429E+02	7.53E-05	1.13E-02
75	0	-1.4895426E+02	7.12E-05	6.76E-03
76	0	-1.4895427E+02	4.95E-05	5.84E-03
77	0	-1.4895421E+02	6.59E-07	9.60E-04
78	0	-1.4895428E+02	8.33E-05	8.61E-03
79	0	-1.4895428E+02	5.13E-05	5.86E-03
80	0	-1.4895421E+02	5.24E-08	2.62E-04

Table 3.9: Iteration results: *ccsa2* problem using the proposed l-BFGS approach (continued)

	$\mathbf{H}_{k,0} = \mathbf{I}$		$\mathbf{H}_{k,0} = \sigma_k\mathbf{I}$	
r	k^*	f_0^* $\lvert f_j^*\rvert$	k^*	f_0^* $\lvert f_j^*\rvert$
2	217	5.1045788E+01 0.00E+00	74	5.1045788E+01 0.00E+00
3	65	5.10457867E+01 0.00E+00	70	5.10457872E+01 0.00E+00
4	59	5.10457871E+01 0.00E+00	65	5.1045787E+01 0.00E+00
5	59	5.1045787E+01 0.00E+00	62	5.10457870E+01 0.00E+00
6	61	5.1045787E+01 0.00E+00	55	5.1045787E+01 0.00E+00
7	57	5.1045787E+01 0.00E+00	57	5.1045787E+01 0.00E+00
8	56	5.1045787E+01 0.00E+00	52	5.1045787E+01 0.00E+00
9	57	5.1045787E+01 0.00E+00	51	5.1045787E+01 0.00E+00
10	58	5.1045787E+01 0.00E+00	55	5.1045787E+01 0.00E+00

Table 3.10: Iteration results: *ccsa1* problem: different values of r and different starting matrices

	$\mathbf{H}_{k,0} = \mathbf{I}$		$\mathbf{H}_{k,0} = \sigma_k\mathbf{I}$	
r	k^*	f_0^* $\lvert f_j^*\rvert$	k^*	f_0^* $\lvert f_j^*\rvert$
2	1000	-1.7303174E+02 2.29E+01	80	-1.4895421E+02 5.24E-08
3	63	-1.4895421E+02 1.61E-07	68	-1.4895421E+02 1.70E-07
4	59	-1.4895421E+02 1.39E-07	66	-1.4895421E+02 8.29E-08
5	58	-1.4895421E+02 4.57E-08	58	-1.4895421E+02 1.33E-08
6	57	-1.4895421E+02 7.12E-08	64	-1.4895421E+02 1.33E-08
7	60	-1.4895421E+02 4.68E-08	59	-1.4895421E+02 1.30E-08
8	61	-1.4895421E+02 2.90E-08	61	-1.4895421E+02 6.32E-09
9	57	-1.4895421E+02 1.10E-07	55	-1.4895421E+02 9.42E-08
10	59	-1.4895421E+02 5.17E-08	52	-1.4895421E+02 8.62E-08

Table 3.11: Iteration results: *ccsa2* problem: different values of r and different starting matrices

Trials	Successes	Probability	f_0^*	Optimum f_0^*
1	1	0.6667	2.1043E+00	2.1042503E+00
2	1	0.6667	-1.0316E+00	-1.0316285E+00
3	1	0.6667	-2.1546E-01	-1.0316285E+00
4	2	0.8810	-1.0316E+00	-1.0316285E+00
5	2	0.8810	2.1043E+00	-1.0316285E+00
6	2	0.8810	-2.1546E-01	-1.0316285E+00
7	3	0.9487	-1.0316E+00	-1.0316285E+00
8	3	0.9487	-2.1546E-01	-1.0316285E+00
9	3	0.9487	-2.1546E-01	-1.0316285E+00
10	4	0.9773	-1.0316E+00	-1.0316285E+00
11	5	0.9908	-1.0316E+00	-1.0316285E+00

Table 3.12: Iteration results: *6-hump camelback* problem using exact Hessian information

Trials	Successes	Probability	f_0^*	Optimum f_0^*
1	1	0.6667	-1.0316E+00	-1.0316285E+00
2	2	0.9000	-1.0316E+00	-1.0316285E+00
3	3	0.9714	-1.0316E+00	-1.0316285E+00
4	3	0.9714	2.1043E+00	-1.0316285E+00
5	4	0.9870	-1.0316E+00	-1.0316285E+00
6	5	0.9959	-1.0316E+00	-1.0316285E+00

Table 3.13: Iteration results: *6-hump camelback* problem using the proposed l-BFGS approach

	SUBSOLVER 25		**CPLEX**		**SUBSOLVER 27**	
OPTIMIZATION PROBLEM	k^*	f_0^* $\lvert f_j^* \rvert$	k^*	f_0^* $\lvert f_j^* \rvert$	k^*	f_0^* $\lvert f_j^* \rvert$
CAM-DESIGN-1	8	-4.4933202E+00 1.4771203E-07	8	-4.3452565E+00 0.0000000E+00	8	-4.4933197E+00 9.8044940E-08
SVAN-CANT	6	1.3399564E+00 0.0000000E+00	6	1.3399857E+00 0.0000000E+00	13	1.3399564E+00 2.6282398E-10
TOROPOV- SVAN-CANT	8	1.3103301E+00 1.0436096E-13	8	1.3103301E+00 1.5598633E-12	9	1.3188486E+00 0.0000000E+00
SVAN-CCSA-1	123	5.1045815E+01 0.0000000E+00	165	5.1045926E+01 0.0000000E+00	74	5.1045788E+01 0.0000000E+00
SVAN-CCSA-2	199	-1.4895417E+02 6.0769614E-08	301	-1.4895400E+02 0.0000000E+00	80	-1.4895421E+02 5.2421732E-08
2-BAR- DEFAULT	9	1.5086524E+00 1.8449848E-09	9	1.5088153E+00 0.0000000E+00	7	1.5086524E+00 6.1798788E-11
2-BAR- DEFAULT-MOD	9	1.5086524E+00 1.8449848E-09	9	1.5088153E+00 0.0000000E+00	7	1.5086524E+00 6.1798788E-11
12-CORNER- POLY	178	-2.7990380E+02 7.3242745E-11	183	-2.7990380E+02 0.0000000E+00	777	-2.7990381E+02 1.0999202E-09
ROSEN- BANANA-2D	123	4.0823240E-07 0.0000000E+00	123	4.0866740E-07 0.0000000E+00	17	5.5636784E-05 0.0000000E+00
SEMI- INF-A	8	5.3346873E+00 3.8931969E-11	-	- -	7	5.3346873E+00 4.6438280E-09
SEMI- INF-B	4	4.3011837E+00 0.0000000E+00	-	- -	4	4.3011837E+00 0.0000000E+00
VDPLAATS- CANTILEVER	12	6.3678100E+04 3.3084646E-13	12	6.3678100E+04 0.0000000E+00	18	6.3678100E+04 3.4416914E-14
VDPLAATS- CANT-NO-DISPL	8	5.4176212E+04 3.4239278E-13	8	5.4176213E+04 0.0000000E+00	10	5.4176212E+04 1.4210855E-14
VDPLAATS- CANT-NOTHING	9	5.4155571E+04 7.1054274E-15	9	5.4155571E+04 0.0000000E+00	9	5.4155571E+04 1.4210855E-14
BRANIN	535	3.9788736E-01 0.0000000E+00	611	3.9788736E-01 0.0000000E+00	220	3.9788736E-01 0.0000000E+00
6-HUMP- CAMEL-BACK	327	-1.0316285E+00 0.0000000E+00	53631	-1.0316285E+00 0.0000000E+00	241	-1.0316285E+00 0.0000000E+00
GOLDSTEIN- PRICE	199	3.0000000E+00 0.0000000E+00	199	3.0000000E+00 0.0000000E+00	219	3.0000001E+00 0.0000000E+00
GRIEWANK- 2D	3019	0.0000000E+00 0.0000000E+00	3064	0.0000000E+00 0.0000000E+00	3077	1.4561869E-01 0.0000000E+00
GRIEWANK- 10D	1059	2.6576408E-11 0.0000000E+00	1568	2.8086200E-11 0.0000000E+00	2712	1.1175949E-11 0.0000000E+00
HARTMAN-3	1851	-3.8627534E+00 0.0000000E+00	1406	-3.8627512E+00 0.0000000E+00	704	-3.8627820E+00 0.0000000E+00
HARTMAN-6	822	-3.3223680E+00 0.0000000E+00	828	-3.3223680E+00 0.0000000E+00	377	-3.3223680E+00 0.0000000E+00
PARTSTAMP-1	822	2.1856406E+01 0.0000000E+00	73	2.1856520E+01 0.0000000E+00	57	2.1856406E+01 0.0000000E+00
RASTRIGIN	244	-2.0000000E+00 0.0000000E+00	244	-2.0000000E+00 0.0000000E+00	990	-2.0000000E+00 0.0000000E+00
SCHUBERT	1462	-1.8673091E+02 0.0000000E+00	1462	-1.8673091E+02 0.0000000E+00	1143	-1.8673091E+02 0.0000000E+00

Table 3.14: Numerical comparison: l-BFGS method (unimodal and multimodal test problems)

	SUBSOLVER 25		**CPLEX**		**SUBSOLVER 27**	
OPTIMIZATION PROBLEM	k^*	f_0^* $\lvert f_j^* \rvert$	k^*	f_0^* $\lvert f_j^* \rvert$	k^*	f_0^* $\lvert f_j^* \rvert$
BOBYQA-01	6	3.2203053E+01 0.0000000E+00	6	3.2203053E+01 0.0000000E+00	7	3.2203053E+01 0.0000000E+00
NEWUOA-01	46	3.5168746E-03 0.0000000E+00	48	3.5169078E-03 0.0000000E+00	33	3.5168775E-03 0.0000000E+00
TOY-QP1	9	1.8093053E-18 0.0000000E+00	9	1.8093053E-18 0.0000000E+00	3	9.3251277E-22 0.0000000E+00
TOY-SALA1	4	1.2000000E+01 0.0000000E+00	5	1.2000131E+01 0.0000000E+00	4	1.2000000E+01 0.0000000E+00
SVAN-CCSA-1-CONCAVE	86	5.1045795E+01 0.0000000E+00	165	5.1045926E+01 0.0000000E+00	74	5.1045788E+01 0.0000000E+00
SVAN-CCSA-2-CONCAVE	80	-1.4895421E+02 1.2384771E-07	301	-1.4895400E+02 0.0000000E+00	80	-1.4895421E+02 5.2421732E-08
3-BAR	6	2.2233333E+01 2.0719314E-08	6	2.2233417E+01 0.0000000E+00	6	2.2233333E+01 2.0719225E-08
3-BAR-SAND	4	2.2233333E+01 7.1718880E-13	5	2.2233354E+01 7.5309799E-08	4	2.2233333E+01 1.4328218E-12
3-BAR-SAND SINGULAR	7	1.9333333E+01 1.8189894E-12	7	1.9333406E+01 1.5902877E-06	4	1.9333333E+01 5.4569682E-12
SVAN-CANT-EXACT-HESS	8	1.3399564E+00 9.8632214E-13	14	1.3399694E+00 0.0000000E+00	13	1.3399564E+00 2.6282398E-10
HOCK-43-EXACT-HESS	7	-4.4000000E+01 1.9076296E-11	7	-4.3999908E+0 0.0000000E+00	8	-4.4000000E+01 1.4757253E-09
HOCK-6	24	1.0965241E-10 1.5209147E-08	24	8.1120111E-15 1.6015733E-09	13	2.2728948E-15 1.7309043E-10
HOCK-7	11	-1.7320508E+00 1.9672264E-11	9	-1.7320508E+00 3.8675466E-08	12	-1.7320508E+00 3.9726569E-09
HOCK-32	4	1.0000000E+00 1.1102230E-16	5	1.0000426E+00 1.1102230E-16	4	1.0000000E+00 1.1102230E-16
HOCK-39	46	-1.0094263E+00 1.8324044E-02	8	-2.4382990E+00 4.8442362E+00	13	-1.0000000E+00 5.9640764E-09
HOCK-41	8	1.9259259E+00 0.0000000E+00	17	1.9259622E+00 0.0000000E+00	10	1.9259259E+00 0.0000000E+00
HOCK-47	26	4.8391176E-07 1.1783072E-09	42	1.2987530E-08 2.1205434E-09	36	-2.6714183E-02 1.4889121E-09
HOCK-48	11	7.9741708E-10 8.8817842E-16	11	7.9630913E-10 0.0000000E+00	10	8.4826713E-11 0.0000000E+00
HOCK-71	7	1.7014017E+01 1.6060397E-10	7	1.7014058E+01 1.3540102E-10	6	1.7014017E+01 7.4383166E-10
HOCK-80	8	5.3949847E-02 8.9094883E-09	8	5.3949852E-02 8.3786098E-09	7	5.3949848E-02 1.1194405E-09
HOCK-107	7	5.0550114E+03 3.0742520E-10	7	5.0550115E+03 4.6029758E-10	6	5.0550114E+03 4.2684167E-10
HOCK-111	162	-4.7761086E+01 3.0886405E-11	169	-4.7761087E+01 9.2382102E-11	127	-4.7761091E+01 1.3779866E-10
HOCK-112	151	-4.7758461E+01 2.2204460E-16	65	-4.7760495E+01 4.4408921E-16	53	-4.7760997E+01 4.4408921E-16
NONCONVEX1	6	1.6905989E+00 1.3322676E-14	6	1.6906433E+00 1.0658141E-14	6	1.6905989E+00 1.2878587E-14
NONCONVEX2	4	-8.8467550E+01 1.3889156E-10	4	-8.8467434E+01 3.1803484E-09	4	-8.8467550E+01 1.3889156E-10
NONCONVEX3	12	-8.0625000E+00 0.0000000E+00	12	-8.0624325E+00 0.0000000E+00	5	-8.0625000E+00 6.6613381E-16

Table 3.15: Numerical comparison: l-BFGS method (specialized and equality constrained test problems)

Chapter 4

PACBB Method To Solve Dual Subproblems in SAO

4.1 Introduction

In 1988, Barzalai and Borwein presented an alternative step size for the gradient method for unconstrained optimization problems (Barzalai and Borwein, 1988). Due to its simplicity, low memory requirements and inexpensive computations, the Barzalai-Borwein (BB) method has been used in many applications and good performance is observed on a wide variety of problems. It is specifically efficient in a neighbourhood of a local minimizer where the Hessian is strictly convex at the solution. Several authors have proposed variants to the BB-method which are aimed at solving large-scale unconstrained, as well as box-constrained, optimization problems. We explore the use of the projected adaptive cyclic Barzalai-Borwein (PACBB) method in the SAO*i* algorithm for constrained optimization to solve the approximate subproblems in the dual space. A brief description of the cyclic Barzalai-Borwein (CBB) and the adaptive cyclic Barzalai-Borwein (ACBB) methods, from which the PACBB method is derived, follows.

4.1.1 CBB Methods for Unconstrained Optimization

Consider unconstrained optimization problems of the form

$$\min f(\boldsymbol{x}), \qquad \boldsymbol{x} \in \Re^n \tag{4.1.1}$$

where the real valued scalar objective function $f(\boldsymbol{x}) : \Re^n \to \Re$ is sufficiently smooth and its gradient $\boldsymbol{g}(\boldsymbol{x}) = \nabla f(\boldsymbol{x})$ is available.

Steepest descent methods start from an initial point $\boldsymbol{x}_0$ and generate new iterates by

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k - \alpha_k \boldsymbol{g}_k, \tag{4.1.2}$$

where $\boldsymbol{x}_k$ is the k^{th} approximation to the optimal solution to (4.1.1), $\boldsymbol{g}_k$ is the gradient vector of $f(\boldsymbol{x})$ at $\boldsymbol{x}_k$ and α_k is a stepsize computed by solving

$$\alpha_k^{SD} \in \arg\min_{\alpha \in R} f(\boldsymbol{x}_k - \alpha \boldsymbol{g}_k). \tag{4.1.3}$$

It has been shown that the convergence of the (often slow) steepest descent method can be accelerated by reusing the exact steepest descent step in a cyclic fashion. This is especially true for optimization problems where the Hessian of $f(\boldsymbol{x})$ is ill-conditioned at a local minimum. The cyclic steepest descent step is given by

$$\alpha_{mk+i} = \alpha_{mk+1}^{SD} \qquad \text{for} \qquad i = 1, \ldots, m, \tag{4.1.4}$$

where α_{mk+1}^{SD} is the steepest descent step determined by (4.1.3) and $m \geq 1$ is an integer, which we will call the cycle length.

Barzilai and Borwein proposed the following step size

$$\alpha_k^{BB} = \frac{\boldsymbol{s}_{k-1}^T \boldsymbol{s}_{k-1}}{\boldsymbol{s}_{k-1}^T \boldsymbol{y}_{k-1}}, \qquad k \geq 2, \tag{4.1.5}$$

where $\boldsymbol{s}_{k-1} = \boldsymbol{x}_k - \boldsymbol{x}_{k-1}$ and $\boldsymbol{y}_{k-1} = \boldsymbol{g}_k - \boldsymbol{g}_{k-1}$. This choice of α_k is obtained by approximating the Hessian $\boldsymbol{H}_k$ by $\boldsymbol{H}_k(\alpha_k) = \frac{1}{\alpha_k}\boldsymbol{I}$ and imposing the quasi-Newton property

$$\alpha_k^{BB} = \min_{\alpha \in R} \{\boldsymbol{H}(\alpha)\boldsymbol{s}_{k-1} - \boldsymbol{y}_{k-1 2}\} \tag{4.1.6}$$

on $\boldsymbol{H}_k(\alpha_k)$. The improved performance of the cyclic steepest descent method compared to that of the ordinary steepest descent method, lead to the derivation of the cyclic BB (CBB) method for which the stepsize is determined by

$$\alpha_{mk+i} = \alpha_{mk+1}^{BB} \qquad \text{for} \qquad i = 1, \ldots, m, \tag{4.1.7}$$

where α_{mk+1}^{BB} is the BB step determined by (4.1.5) and the integer $m \geq 1$ is again the cycle length. The CBB method is superior to the cyclic steepest descent method in the sense that the stepsize is given by a simple fomula (4.1.5), whereas the steepest descent step is determined by solving an (often nontrivial) optimization problem (4.1.3). In the neighbourhood of a local minimizer, where the Hessian of $f(\boldsymbol{x})$ is strictly convex (positive definite), no line search is needed and the CBB method is locally linearly convergent (Dai *et al.*, 2005).

Since the choice of m has a significant impact on the performance of the CBB method, an adaptive choice for m has been proposed. The implementation of the CBB method that combines this adaptive choice of m and a nonmonotone line search, is called the adaptive CBB (ACBB) method. Although the ACBB method is not as straightforward as the other BB methods discussed thus far, it outperforms these methods in many applications. In the next section, we focus our attention on the projected adaptive cyclic Barzalai-Borwein (PACBB) method, which is derived from the ACBB method and is used to solve bound constrained optimization problems.

4.1.2 The PACBB Method for Bound Constrained Optimization

The ACBB method for solving unconstrained optimization problems can be modified to solve optimization problems of the form

$$\begin{aligned} &\min_{\boldsymbol{x}} f_0\left(\boldsymbol{x}\right) \\ &\text{subject to } \check{x}_i \leq x_i \leq \hat{x}_i,\ i = 1, 2, \ldots, n, \end{aligned} \tag{4.1.8}$$

where $f_0(\boldsymbol{x})$ represents a real-valued scalar objective function. The objective function depends on the n real design variables $\boldsymbol{x} = \{x_1, x_2, \ldots, x_n\}^T \in \Re^{\text{n}}$. $\check{x}_i$ indicates the lower bound and $\hat{x}_i$ the upper bound of continuous real variable x_i.

The projected ACBB (PACBB) method projects the ACBB iterates onto the feasible set and performs a nonmonotone line search between the current iterate and the projection point as illustrated in Figure 4.1 (Zhang and Hager, 2004).

At every PACBB iteration, the following steps need to be followed.

1. Take a step along the negative gradient $\boldsymbol{g}_k(\boldsymbol{x})$ to a point $\bar{\boldsymbol{x}}_k = \boldsymbol{x}_k - \bar{\alpha}_k \boldsymbol{g}_k$. The initial stepsize $\bar{\alpha}_k$ is the previous stepsize used or, if the cycle length has been reached, a newly calculated trial stepsize.
2. If the point $\bar{\boldsymbol{x}}_k$ lies outside the feasible set B, compute the projection ($\mathbf{P}_B(\bar{\boldsymbol{x}}_k)$) of $\bar{\boldsymbol{x}}_k$ onto B and determine the search direction $\boldsymbol{d}_k = \mathbf{P}_B(\bar{\boldsymbol{x}}_k) - \boldsymbol{x}_k$. Since the feasible set B is convex, the search direction $\boldsymbol{d}_k$ will be a descent direction.
3. Perform a nonmonotone line search along $\boldsymbol{d}_k$ between $\boldsymbol{x}_k$ and $\mathbf{P}_B(\bar{\boldsymbol{x}}_k)$. Accept the point $\mathbf{P}_B(\bar{\boldsymbol{x}}_k)$ (if possible) or backtrack towards $\boldsymbol{x}_k$.
4. If $\bar{\boldsymbol{x}}_k$ lies outside of B, compute the next initial stepsize $\bar{\alpha}_{k+1}$ using the BB formula (4.1.5).

The low memory requirements and the simplicity of the PACBB algorithm compared to other optimization algorithms, lead us to consider the use of the PACBB method in the SAO*i* algorithm to solve the approximate dual subproblems.

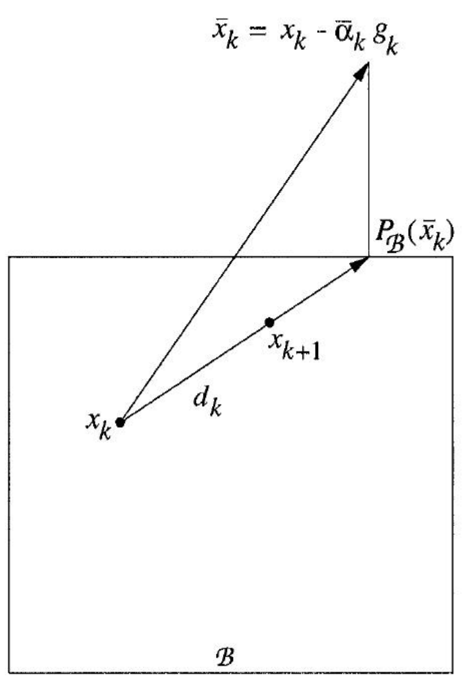
$\bar{x}_k = x_k - \bar{\alpha}_k g_k$
$P_{\mathcal{B}}(\bar{x}_k)$
x_{k+1}
d_k
x_k
$\mathcal{B}$

Figure 4.1: An illustration of the projected line search for a PACBB iteration

4.2 The SAO*i* Algorithm: Dual Subproblems

As mentioned, the SAO*i* algorithm not only provides for a range of possible approximation functions, but it allows subproblems to be solved via a quadratic program (QP) or in the dual space. In the current academic version of the SAO*i* algorithm, there is only one solver available for the bounded dual subproblems, namely the limited memory bound constrained l-BFGS-b solver. In the following sections, we discuss the approximate dual subproblems based on the dual of Falk and the implementation of the PACBB method in the SAO*i* algorithm as an alternative solver for the bounded dual subproblems.

Currently, the SAO*i* algorithm is restricted to separable primal diagonal quadratic approximation functions of the form

$$\tilde{f}(x) = f(x_k) + \nabla^T f(x - x_k) + \frac{1}{2}\sum_{i=1}^{n} c_{k,i}\,(x - x_k)^2, \tag{4.2.1}$$

where $c_{k,i}$, $i = 1, 2, \ldots, n$, are the only unknowns. (As mentioned in Chapter 2, SAO methods make use of intermediate variables to determine appropriate values for these unknowns.) Due to the separability of the primal approximate subproblems, it can be solved very efficiently in the dual space. This makes the SAO*i* algorithm very efficient if the number of constraints n_i is (far) less than the number of design variables n, *e.g.* 'classical' minimum compliance topology optimization problems where the number of design variables can easily run into the millions with only a single constraint present. Given the (convex) primal diagonal quadratic approximate subproblem (4.2.1), the approximate dual subproblem becomes

$$\max_{\boldsymbol{\lambda}} \{ \gamma(\boldsymbol{\lambda}) = \tilde{f}_{0,k}(\boldsymbol{x}(\boldsymbol{\lambda})) + \sum_{j=1}^{m} \lambda_j \tilde{f}_{j,k}(\boldsymbol{x}(\boldsymbol{\lambda})) \}, \tag{4.2.2}$$
$$\text{subject to } \lambda_j \geq 0, \qquad j = 1, 2, \ldots, m.$$

The notation $\boldsymbol{x}(\boldsymbol{\lambda})$ implies minimization with respect to $\boldsymbol{x}$ and the relationships between the primal variables $\boldsymbol{x}_i$, $i = 1, 2, \ldots, n$ and the dual variables λ_j, $j = 1, 2, \ldots, m$, are given by

$$x_i(\lambda) = \begin{cases} \beta_i(\boldsymbol{\lambda}) & \text{if } \check{x}_{k,i} < \beta_i(\boldsymbol{\lambda}) < \hat{x}_{k,i}, \\ \check{x}_{k,i} & \text{if } \beta_i(\boldsymbol{\lambda}) \leq \check{x}_{k,i}, \\ \hat{x}_{k,i} & \text{if } \beta_i(\boldsymbol{\lambda}) \geq \hat{x}_{k,i}, \end{cases} \tag{4.2.3}$$

and

$$\beta_i(\boldsymbol{\lambda}) = x_{k,i} - \left(c_{k,2i_0} + \sum_{j=1}^{m} \lambda_j c_{k,2i_j} \right)^{-1} \left(\frac{\partial f_{0,k}}{\partial x_i} + \sum_{j=1}^{m} \lambda_j \frac{\partial f_{j,k}}{\partial x_i} \right). \tag{4.2.4}$$

In the current academic version of the SAO*i* algorithm, the limited memory bound constrained l-BFGS-b solver (developed by Zhu and co-workers) is the only available solver for the approximate dual subproblem. Despite the discontinuities in the second order derivatives that result from the modified definition domain of the Falk dual when the fixed primal variables become free (and vice versa), the l-BFGS-b solver has been shown to be highly efficient for solving the Falk-like dual approximate subproblems. Although the l-BFGS-b solver is robust, it does occasionally fail and the need for the development of additional solvers exists (Groenwold and Etman, 2012). As an alternative to the l-BFGS-b solver, we propose the implementation of a PACBB solver. In the next chapter, we present the PACBB algorithm (Zhang and Hager, 2004) as implemented in the existing SAO*i* algorithm to solve the bound constrained subproblems. This is followed by a summary of the results obtained using a series of test problems to evaluate the proposed method.

Chapter 5

PACBB: Implementation and Results

5.1 Introduction

We propose the use of the PACBB method (Zhang and Hager, 2004) in the SAO*i* algorithm for constrained optimization to solve the bound constrained approximate dual subproblems. As mentioned, the SAO*i* algorithm only has one solver available for the bounded dual subproblem, namely the limited memory bound constrained l-BFGS-b solver. A series of inequality constrained optimization problems, including unimodal and other specialized examples, are used to measure the performance of the implemented method against that of the l-BFGS-b solver.

For the PACBB implementation, the golden section method is used to perform the nonmonotone line search between the current iterate and the projection point if the projection point is not in the feasible set. The golden section method is easy to implement and is found to be reliable even for poorly conditioned problems. Unfortunately, the method requires a relatively large number of function evaluations and the use of other line search methods for the PACBB implementation should be explored.

5.2 The PACBB Algorithm

Detail on the PACBB algorithm, as implemented into the existing SAO*i* algorithm, follows. We closely follow the algorithm as described by Zhang and Hager (2004). (Refer to Appendix 7.2 for the FORTRAN source code.)

Step 0 Initialize some parameters

1. Set $k := 1$.
2. Determine $x_1 \in \Re^n$. If $x_1 \notin B$, replace x_1 by its projection $\mathbf{P}_B(x_1)$.
3. Given $0 < \alpha_{min} < \alpha_{max}$, pick up $\alpha_1^{(1)} \in [\alpha_{min}, \alpha_{max}]$.
4. Set variables $l := 0$, $p := 0$, $m := 0$, $\bar{l} := 0$ and $f_{min} = f_r = f_c := f(x_1)$.
5. Set positive integers $P > M > L$ and $\bar{M}$, as well as constants $0 < \sigma_1 < \sigma_2 < 1$, $\gamma_1 \geq 1$, $\gamma_2 \geq 1$, $\beta > 0$ and $\epsilon \geq 0$.

Step 1 If the stopping condition $\|\mathbf{P}_B(x_k - g_k) - x_k\|_\infty \leq \epsilon$ holds, do not continue with the iteration process

Step 2 Compute the search direction and check if the first trial stepsize $\alpha_k^{(1)}$ is truncated or not

1. Let $d_k = \mathbf{P}_B(x_k - \alpha_k^{(1)} g_k(\mathbf{x}_k)) - \mathbf{x}_k$;
2. If $d_k^{(i)}$ and $g_k^{(i)}$ denote the *i-th* component of the vectors $\mathbf{d}_k$ and $\mathbf{g}_k$ respectively, and $0 < |d_k^{(i)}| < \alpha_k^{(1)} |g_k^{(i)}|$ for some $1 \leq i \leq n$, let $\bar{l} = \bar{l} + 1$.

Step 3 Compute a stepsize α_k using an adaptive nonmonotone line search

(i) If $l = L$, update f_r using

$$f_r = \begin{cases} f_c, & \text{if } \frac{f_{max} - f_{min}}{f_c - f_{min}} > \gamma_1; \\ f_{max}, & \text{otherwise} \end{cases} \tag{5.2.1}$$

and set $l := 0$. If $p > P$, update f_r using

$$f_r = \begin{cases} f_{max}, & \text{if } f_{max} > f(\mathbf{x}_k) \text{ and } \frac{f_r - f(\mathbf{x}_k)}{f_{max} - f(\mathbf{x}_k)} \geq \gamma_2; \\ f_r, & \text{otherwise} \end{cases} \tag{5.2.2}$$

(ii) If

$$m = 0 \text{ and } f(\mathbf{x}_k + \alpha_k^{(1)} \mathbf{d}_k) \leq f_r + \delta \alpha_k^{(1)} \mathbf{g}_k^T \mathbf{d}_k, \tag{5.2.3}$$

or

$$f(\mathbf{x}_k + \alpha_k^{(1)} \mathbf{d}_k) \leq \min\{f_{max}, fr\} + \delta \alpha_k^{(1)} \mathbf{g}_k^T \mathbf{d}_k, \tag{5.2.4}$$

let $m = m + 1$, $\alpha_k = \alpha_k^{(1)}$ and $p = p + 1$, and go to step v.

(iii) Let $p := 0$ and $\alpha_{old} = \alpha_k^{(1)}$.

(*iv*) Compute a stepsize $\alpha_{new} \in [\sigma_1\alpha_{old}, \sigma_2\alpha_{old}]$ using a backtrack (or other) line search algorithm. If

$$f(\mathbf{x}_k + \alpha_{new}\mathbf{d}_k) \leq \min\{f_{max}, fr\} + \delta\alpha_{new}\mathbf{g}_k^T\mathbf{d}_k, \tag{5.2.5}$$

let $\bar{l} = \bar{l} + 1$ and $\alpha_k = \alpha_{new}$, and go to Step *v*. Otherwise, let $\alpha_{old} = \alpha_{new}$ and repeat Step *iv*.

(*v*) Let $f_{k+1} = f(\mathbf{x}_k + \alpha_k\mathbf{d}_k)$.

(*vi*) If $f_{k+1} < f_{min}$, let $f_c = f_{min} = f_{k+1}$ and $l := 0$; otherwise, let $l = l + 1$.

(*vii*) If $f_{k+1} > f_c$, let $f_c = f_{k+1}$.

(*viii*) Compute f_{max} using

$$f_{max} = \max_{0 \leq i \leq \min\{k, M-1\}} f(\mathbf{x}_{k-i}), \tag{5.2.6}$$

but replacing k with $k + 1$.

Step 4 Update the estimation by letting $\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k\mathbf{d}_k$.

Step 5 Get the first trial step size for the next iteration α_k

(*i*) Let $\mathbf{s}_k = \mathbf{x}_{k+1} - \mathbf{x}_k$ and *index* $= 0$.

(*ii*) If $m \geq \bar{M}$ or $\bar{l} \geq 1$ or $k = 1$ or $||\mathbf{s}_k||_2 \geq \max\{\frac{1}{||\mathbf{P}_B(x_k - g_k) - \mathbf{x}_k||_\infty}, 1\}$, let

$$\mathbf{y}_k = \mathbf{g}_{k+1} - \mathbf{g}_k \qquad \text{and} \qquad index = 1;$$

else if $||\mathbf{s}_k||_2 \leq \min\{\frac{1}{||\mathbf{P}_B(x_k - g_k) - \mathbf{x}_k||_\infty}, 1\}$ and $\frac{\mathbf{s}_k^T\mathbf{y}_k}{||\mathbf{s}_k||_2||\mathbf{y}_k||_2} \geq \beta$, let

$$\mathbf{y}_k = \mathbf{g}_{k+1} - \mathbf{g}_k \qquad \text{and} \qquad index = 1;$$

else if $||\mathbf{s}_k||_2 \leq \min\{\frac{1}{||\mathbf{P}_B(x_k - g_k) - \mathbf{x}_k||_\infty}, 1\}$, let

$$\mathbf{y}_k = \mathbf{g}_{k+1} - \mathbf{g}_k$$

(*iii*) If *index* $= 1$ and $\mathbf{s}_k^T\mathbf{y}_k \leq 0$ and $m \geq 1.5\bar{M}$, let

$$\bar{l} = 0 \qquad \text{and} \qquad \alpha_{k+1}^{(1)} = \max\{\frac{1}{||\mathbf{P}_B(x_k - g_k) - \mathbf{x}_k||_\infty}, \alpha_k\} \qquad \text{and} \qquad m = 0;$$

else if *index* $= 1$ and $\mathbf{s}_k^T\mathbf{y}_k > 0$, let

$$\bar{l} = 0 \qquad \text{and} \qquad \alpha_{k+1}^{(1)} = \max\{\alpha_{min}, \min\{\frac{\mathbf{s}_k^T\mathbf{s}_k}{\mathbf{s}_k^T\mathbf{y}_k}, \alpha_{max}\}\} \qquad \text{and} \qquad m = 0;$$

Step 6 Let $k := k + 1$ and go to Step 1.

5.2.1 Performance Measurement of the PACBB Method

In this section, the performance of the proposed PACBB method (subsolver set to 5 in the SAO*i* algorithm) is compared to that of the l-BFGS-b dual solver (subsolver set to 1 in the SAO*i* algorithm) using a range of test problems. (Detail on the optimization problems is available upon request.) The numerical results are given in Table 5.1 and 5.2. The PACBB method is applied using different values for the non-negative tolerance (f_{tol}) used to determine whether a sufficient decrease in the objective function has been reached. Based on the iteration results for the PACBB method, it is noted that the method delivers a relatively fast initial rate of convergence for the majority of the test problems. It quickly converges to within a neighbourhood of the local or global minimum, but then slows down and sometimes even 'struggles' to converge to a solution. The possibility of using the PACBB method for the initial steps and then implementing a more robust method to find the final solution should be explored.

However, we aim to improve the iteration results by 'forcing' convergence using a trust region method. The trust region method used in the SAO*i* algorithm is based on the filtered acceptance of iterates, and is inspired by Fletcher, Leyffer and their co-workers (Fletcher and Leyffer, 1998, 2002; Fletcher *et al.*, 1998, 2002*a*,*b*). For convex problems, this method may be used to enforce global convergence. For non-convex problems, the use of this method guarentees convergence to at least some local minimum. The numerical results are given in Table 5.3 and 5.4. Although the use of the trust region method results in a decrease in the number of iterations for some of the test problems, the numerical results still indicate that the existing l-BFGS-b dual solver is more efficient and robust. We hope to investigate the possibility of improving the iteration results for the PACBB implementation through the use of a different line search method in a future study.

	SUBSOLVER 1 **with** $f_{\text{tol}} = 10^{-10}$	**SUBSOLVER 5** **with** $f_{\text{tol}} = 10^{-10}$	**SUBSOLVER 5** **with** $f_{\text{tol}} = 10^{-05}$	**SUBSOLVER 5** **with** $f_{\text{tol}} = 10^{-03}$
OPTIMIZATION PROBLEM	k^* f_0^* $\lvert f_j^*\rvert$	k^* f_0^* $\lvert f_j^*\rvert$	k^* f_0^* $\lvert f_j^*\rvert$	k^* f_0^* $\lvert f_j^*\rvert$
SVAN-CANT	5 1.3399564E+00 0.0000000E+00	33 1.3399449E+00 2.5624629E-05	5 1.3399414E+00 3.3385387E-05	4 1.3399587E+00 0.0000000E+00
TOROPOV-SVAN-CANT	8 1.3103301E+00 0.0000000E+00	1000 1.3103327E+00 0.0000000E+00	13 1.3103107E+00 4.4418299E-05	10 1.3103333E+00 0.0000000E+00
SVAN-CCSA-1	119 5.1045822E+01 0.0000000E+00	1000 5.0447248E+01 1.0458309E+00	1000 5.0447248E+01 1.0458309E+00	35 5.7888594E+01 0.0000000E+00
SVAN-CCSA-2	190 -1.4895415E+02 0.0000000E+00	1000 -1.4895014E+02 5.3160272E-04	1000 -1.4895014E+02 5.3160272E-04	20 -1.4867394E+02 0.0000000E+00
2-BAR-DEFAULT	6 1.5086524E+00 0.0000000E+00	6 1.5086526E+00 0.0000000E+00	5 1.5086459E+00 4.2890572E-06	4 1.5086441E+00 5.5403273E-06
2-BAR-DEFAULT-MOD	6 1.5086524E+00 0.0000000E+00	6 1.5086526E+00 0.0000000E+00	5 1.5086459E+00 4.2890572E-06	4 1.5086441E+00 5.5403273E-06
12-CORNER-POLY	155 -2.7990380E+02 0.0000000E+00	1000 -2.7923304E+02 4.2654633E-02	64 -2.7965767E+02 0.0000000E+00	11 -2.7738955E+02 0.0000000E+00
VDPLAATS-CANTILEVER	12 6.3678065E+04 1.1128471E-06	1000 6.6444172E+04 2.3700326E+00	369 6.5588421E+04 7.4776187E-01	26 5.5876919E+04 3.0918443E+00
VDPLAATS-ANT-NO-DISPL	10 5.4176211E+04 3.1007144E-06	1000 5.5891724E+04 4.3666144E-01	1000 5.5891724E+04 4.3666144E-01	32 5.5738289E+04 6.1941169E-01
VDPLAATS-CANT-NOTHING	10 5.4155571E+04 1.5908248E-06	1000 5.6006326E+04 2.4262228E+00	1000 5.6006326E+04 2.4262228E+00	18 5.5370514E+04 2.6532321E-01
FLEURY-WEIGHT-1	67 9.5000005E+02 0.0000000E+00	262 9.5000061E+02 0.0000000E+00	262 9.5000061E+02 0.0000000E+00	88 1.8233466E+02 4.0292556E+03

Table 5.1: Numerical comparison: PACBB method (unimodal test problems)

	SUBSOLVER 1 with $f_{\text{tol}} = 10^{-10}$	**SUBSOLVER 5 with $f_{\text{tol}} = 10^{-10}$**	**SUBSOLVER 5 with $f_{\text{tol}} = 10^{-05}$**	**SUBSOLVER 5 with $f_{\text{tol}} = 10^{-03}$**
OPTIMIZATION	k^*	k^*	k^*	k^*
PROBLEM	f_0^*	f_0^*	f_0^*	f_0^*
	$\lvert f_j^*\rvert$	$\lvert f_j^*\rvert$	$\lvert f_j^*\rvert$	$\lvert f_j^*\rvert$
	8	1000	13	10
TOROPOV-SVAN-	1.3103301E+00	1.3103327E+00	1.3103107E+00	1.3103333E+00
CANT-VLSO	0.0000000E+00	0.0000000E+00	4.4418299E-05	0.0000000E+00
	25	102	25	23
FLEURY-WEIGHT-	9.5000005E+04	9.5000136E+04	9.4999293E+04	9.5000577E+04
1-VLSO	4.5261004E-09	0.0000000E+00	8.1429259E-03	5.0495517E-01
	10	1000	16	6
DENSE-H	3.2864175E+04	3.28641969E+04	3.2906465E+04	3.51260147E+04
	2.2447677E-10	0.00000000E+00	3.7152169E-03	2.30997871E+00
	5	9	9	6
TOY-SALA1	1.2000000E+01	1.2000000E+01	1.2000000E+01	1.2000923E+01
	0.0000000E+00	0.0000000E+00	0.0000000E+00	0.0000000E+00
	5	7	7	4
3-BAR	2.2233333E+01	2.2233497E+01	2.2233497E+01	2.2393181E+01
	4.0268899E-11	0.0000000E+00	0.0000000E+00	0.0000000E+00
	3	2	2	2
3-BAR-	1.1102439E+01	1.1102439E+01	1.1102439E+01	1.1102439E+01
SAND	7.1171573E+03	7.0131776E+03	7.0131776E+03	7.0131776E+03
	3	10	10	10
3-BAR-SAND	0.0000000E+00	0.0000000E+00	0.0000000E+00	0.0000000E+00
SINGULAR	1.0000000E+04	1.0000000E+04	1.0000000E+04	1.0000000E+04
	8	1000	8	8
SVAN-CANT-	1.3399564E+00	1.3399523E+00	1.3399529E+00	1.3399529E+00
EXACT-HESS	0.0000000E+00	9.0131347E-06	7.7342459E-06	7.7342459E-06
	2	1000	1000	5
HOCK-43-	-4.4000000E+01	-4.4000719E+01	-4.4000719E+01	-4.3999276E+01
EXACT-HESS	8.1001872E-11	3.1141149E-04	3.1141149E-04	0.0000000E+00

Table 5.2: Numerical comparison: PACBB method (specialized test problems)

	SUBSOLVER 1 **with** $f_{\text{tol}} = 10^{-10}$	**SUBSOLVER 5** **with** $f_{\text{tol}} = 10^{-10}$	**SUBSOLVER 5** **with** $f_{\text{tol}} = 10^{-05}$
OPTIMIZATION PROBLEM	k^* f_0^* $\lvert f_j^*\rvert$	k^* f_0^* $\lvert f_j^*\rvert$	k^* f_0^* $\lvert f_j^*\rvert$
SVAN-CANT	5 1.3399564E+00 0.0000000E+00	33 1.3399449E+00 2.5624629E-05	33 1.3399449E+00 2.5624629E-05
TOROPOV-SVAN-CANT	8 1.3103301E+00 0.0000000E+00	1000 1.3103327E+00 0.0000000E+00	1000 1.3103311E+00 0.0000000E+00
SVAN-CCSA-1	119 5.1045822E+01 0.0000000E+00	1000 5.0447248E+01 1.0458309E+00	1000 5.4402784E+01 0.0000000E+00
SVAN-CCSA-2	190 -1.4895415E+02 0.0000000E+00	1000 -1.4895014E+02 5.3160272E-04	1000 -1.3761002E+02 0.0000000E+00
2-BAR-DEFAULT	6 1.5086524E+00 0.0000000E+00	6 1.5086526E+00 0.0000000E+00	6 1.5086526E 0.0000000E+00
2-BAR-DEFAULT-MOD	6 1.5086524E+00 0.0000000E+00	6 1.5086526E+00 0.0000000E+00	6 1.5086526E+00 0.0000000E+00
12-CORNER-POLY	155 -2.7990380E+02 0.0000000E+00	1000 -2.7923304E+02 4.2654633E-02	297 -3.0862372E+02 3.5941036E+00
VDPLAATS-CANTILEVER	12 6.3678065E+04 1.1128471E-06	1000 6.6444172E+04 2.3700326E+00	132 6.3579530E+04 3.1068743E-03
VDPLAATS-ANT-NO-DISPL	10 5.4176211E+04 3.1007144E-06	1000 5.5891724E+04 4.3666144E-01	81 5.4179377E+04 1.4126940E-04
VDPLAATS-CANT-NOTHING	10 5.4155571E+04 1.5908248E-06	1000 5.6006326E+04 2.4262228E+00	95 5.2782097E+04 1.1702158E+00
FLEURY-WEIGHT-1	67 9.5000005E+02 0.0000000E+00	262 9.5000061E+02 0.0000000E+00	374 9.4999917E+02 8.7814291E-04

Table 5.3: Numerical comparison: PACBB method with trust region filter (unimodal test problems)

	SUBSOLVER 1 **with** $f_{\text{tol}} = 10^{-10}$	**SUBSOLVER 5** **with** $f_{\text{tol}} = 10^{-10}$	**SUBSOLVER 5** **with** $f_{\text{tol}} = 10^{-05}$
OPTIMIZATION PROBLEM	k^* f_0^* $\|f_j^*\|$	k^* f_0^* $\|f_j^*\|$	k^* f_0^* $\|f_j^*\|$
TOROPOV-SVAN-CANT-VLSO	8 1.3103301E+00 0.0000000E+00	1000 1.3103327E+00 0.0000000E+00	1000 1.3103311E+00 0.0000000E+00
FLEURY-WEIGHT-1-VLSO	25 9.5000005E+04 4.5261004E-09	102 9.5000136E+04 0.0000000E+00	28 9.4999411E+04 9.2117148E-03
DENSE-H	10 3.2864175E+04 2.2447677E-10	1000 3.28641969E+04 0.00000000E+00	1000 3.2598179E+04 4.0965626E+02
TOY-SALA1	5 1.2000000E+01 0.0000000E+00	9 1.2000000E+01 0.0000000E+00	8 1.2000448E+01 0.0000000E+00
3-BAR	5 2.2233333E+01 4.0268899E-11	7 2.2233497E+01 0.0000000E+00	7 2.2233497E+01 0.0000000E+00
BAR-SAND	3 1.1102439E+01 7.1171573E+03	2 1.1102439E+01 7.0131776E+03	5 1.1102439E+01 7.3483496E+03
BAR-SAND-SINGULAR	3 0.0000000E+00 1.0000000E+04	10 0.0000000E+00 1.0000000E+04	2 0.0000000E+00 1.0000000E+04
SVAN-CANT-EXACT-HESS	8 1.3399564E+00 0.0000000E+00	1000 1.3399523E+00 9.0131347E-06	73 1.3399567E+00 0.0000000E+00
HOCK-43-EXACT-HESS	2 -4.4000000E+01 8.1001872E-11	1000 -4.4000719E+01 3.1141149E-04	56 -4.3999937E+01 0.0000000E+00

Table 5.4: Numerical comparison: PACBB method with trust region filter (specialized test problems)

Chapter 6

Topology Optimization Application

In the following sections, we closely follow the presentation by Groenwold *et al.* entitled *A quadratic approximation for structural topology optimization* (International Journal for Numerical Methods in Engineering (2010); 82(4):505-524). We also include detail from the presentation by Etman *et al.* entitled *On diagonal QP subproblems for sequential approximate optimization* (Proc. Eighth World Congress on Structural and Multidisciplinary Optimization; Lisboa, Portugal (2009); Paper 1065).

6.1 Introduction

Topology optimization is the introduction of topological features into a structure in an attempt to optimize the material distribution in a given design space, for a given set of loads and boundary conditions. These conditions are derived from a prescribed set of performing targets and are expressed as linear and/or nonlinear inequality constraints. Topology optimization is implemented through the use of finite element methods for the analysis and discretization of the design space. Although we seek a purely discrete or black-and-white solution, the discrete optimization problem is intractable and often only near-discrete solutions are found.

In the following section, we introduce a very well-known form of the topology optimization problem, namely the 'classical' minimum compliance topology optimization problem, potentially subject to m linear constraints.

6.2 Minimum Compliance Topology Optimization

Minimum compliance topology optimization problems are aimed at finding the stiffest possible black-and-white design, subject to any number of constraints. The popular solid isotropic material with penalization (SIMP) method (independently proposed by Bendsøe (1989), and Rozvany and Zhou (1991)) is arguably the simplest strategy for these problems. The problem is expressed as

$$\begin{aligned}
&\min_{x} \ f_0(x) = \mathbf{q}^T\mathbf{K}\mathbf{q} = \sum_{i=1}^{n} (x_i)^p \mathbf{q}_i^T \mathbf{K}_i \mathbf{q}_i, && \\
&\text{subject to } f_j(x) \leq 0, && j = 1, 2, \ldots, m, \\
&\qquad\qquad \mathbf{K}\mathbf{q} = \mathbf{r}, && \\
&\qquad\qquad 0 < \check{x}_i \leq x_i \leq 1, && i = 1, 2, \ldots, n, \qquad (6.2.1)
\end{aligned}$$

where $f_0(x)$ represents the compliance objective function, and $f_j(x)$, $j = 1, 2, \ldots, m$, represents the m linear or nonlinear constraint functions. $\mathbf{q} = \mathbf{q}(x)$ and $\mathbf{r}$ respectively represent the finite element displacement and force vectors, while $\mathbf{K} = \mathbf{K}(x)$ represents the global assembled finite element stiffness matrix. It is assumed that the loads $\mathbf{r}$ are design independent. There are n finite elements in the mesh, and subscript i is used to indicate the elemental quantities and operators. The n design variables contained in $x \in \Re^n$ represent the elemental densities. p is the SIMP penalty parameter. (See Bendsøe and Sigmund (2003) for details.)

If we set $m = 1$ and formulate the linear volume constraint

$$f_1(x) = \frac{v(x)}{v_0} - \bar{v} = \frac{1}{v_0} \sum_{i=1}^{n} v_i x_i - \bar{v} \leq 0, \qquad (6.2.2)$$

we recover the well-known 'standard' minimum compliance topology optimization problem. In (6.2.2), $v(x)$ represents the material or final structural volume, v_i the elemental volumes, v_0 the total volume of the design space, and $\bar{v}$ the prescribed limit on the final volume fraction.

If we assume that the loads $\mathbf{r}$ are independent of the design variables, the derivatives of compliance are

$$\frac{\partial f_0}{\partial x_i} = -p(x_i)^{p-1} q_i^T \mathbf{K}_i \mathbf{q}_i < 0, \qquad (6.2.3)$$

and for the 'standard' compliance problem constraint

$$\frac{\partial f_1}{\partial x_i} = \frac{\partial v}{\partial x_i} = \frac{v_i}{v_0}. \qquad (6.2.4)$$

6.3 SAO for Topology Optimization

Topology optimization problems are normally solved using optimality criterion (OC) methods or (dual) SAO algorithms. Equivalent to a dual SAO method, the OC method commonly used in topology optimization is based on a linear (exact) representation of the (volume) constraint and a single-point strictly convex and separable exponential approximation to the (compliance) objective function. When only a single linear constraint is present (typically the volume constraint), OC methods are a popular class of solution methods in structural topology optimization problems. However, if more than one constraint is present, OC methods are considered cumbersome and SAO methods are resorted to.

As a solution strategy for problem (6.2.1), SAO seeks to construct successive approximate analytical subproblems $P[k], k = 1, 2, 3, \ldots$, at successive approximations $\boldsymbol{x}_k$ to the solution $\boldsymbol{x}^*$. The solution of subproblem $P[k]$ is denoted $\boldsymbol{x}_k^* \in \Re^n$ and is obtained using any suitable continuous programming method. Thereafter, $\boldsymbol{x}_{k+1} = \boldsymbol{x}_k^*$. The iterative application of $P[k]$ is performed until convergence is achieved (or until a prescribed maximum number of iterations have occured). The approximate subproblems can be expressed in terms of the primal variables or in dual form.

6.3.1 Primal Approximate Subproblem

The continuous primal approximate subproblem $P_P[k]$ constructed at $\boldsymbol{x}_k$, for given objective function $f_0(\boldsymbol{x})$ and constraints $f_j(\boldsymbol{x}), j = 1, 2, \ldots, m$, is

$$\begin{aligned} \min\ & \tilde{f}_0(\boldsymbol{x}) \\ \text{subject to} & \tilde{f}_j(\boldsymbol{x}) \leq 0, \qquad j = 1, 2, \ldots, m, \\ & \check{x}_i \leq x_i \leq \hat{x}_i, \quad j = 1, 2, \ldots, m, \end{aligned} \tag{6.3.1}$$

where $\tilde{f}_j(\boldsymbol{x})$, $j = 0, 1, 2, \ldots, m$, represent suitable approximation functions to the objective function $f_0(\boldsymbol{x})$ and the m linear and/or nonlinear constraint functions $f_j(\boldsymbol{x}), j = 1, 2, \ldots, m$. The primal approximate subproblem (6.3.1) has n unknowns, m constraints and $2n$ bound constraints. A number of techniques for constrained nonlinear programming may be used to solve (6.3.1). In addition, for a prescribed move limit $\delta > 0$, we set the upper and lower bounds in (6.3.1) to

$$\begin{aligned} \check{x}_i \leftarrow \check{x}_{k,i} &= \max\left(x^*_{(k-1),i} - \delta, x_{\min}\right), \\ \hat{x}_i \leftarrow \hat{x}_{k,i} &= \max\left(x^*_{(k-1),i} + \delta, 1\right). \end{aligned} \tag{6.3.2}$$

For the sake of simplicity, we assume that δ is expressed as an infinity norm, and is constant in the n dimensions.

A number of approximating functions may be used in the primal approximate subproblem (6.3.1). SAO methods use approximations of the form

$$\tilde{f}(\boldsymbol{x}) = f(\boldsymbol{x}_k) + \nabla^T f(\boldsymbol{x} - \boldsymbol{x}_k) + \frac{1}{2}\sum_{i=1}^{n} c_{k,i}\,(\boldsymbol{x} - \boldsymbol{x}_k)^2, \tag{6.3.3}$$

where $c_{k,i}$, $i = 1, 2, \ldots, n$, are the only unknowns. This is simply a second order (quadratic) Taylor series expansion in which only the diagonal Hessian terms are retained (making the approximations separable). Due to the computational effort associated with calculating and\or storing the coupling terms, these are neglected. The approximation (6.3.3) is considered *convex* if $c_{k,i} \geq 0 \forall i$, and *strictly convex* if $c_{k,i} > 0 \forall i$. To introduce desirable nonlinearities into the approximation functions (*i.e.* to determine appropriate values for the unknowns $c_{k,i}$, $i = 1, 2, \ldots, n$), SAO methods make use of intermediate variables. Many different intermediate variables may be used, providing a range of possible approximation functions. The SAO*i* algorithm allows for the use of reciprocal and exponential intermediate variables. In our implementation of the proposed methods, reciprocal intermediate variables are used to construct the approximate subproblems.

6.3.2 Dual Approximate Subproblem

In topology optimization, the number of design variables n is usually significantly higher than the number of constraints m, and the approximate subproblems can be solved very efficiently in the dual space. The main difficulty is to select accurate approximation functions that satisfy the requirements needed to invoke duality. The dual approximate subproblem $P_D[k]$ due to Falk, is

$$\begin{aligned} &\max_{\boldsymbol{\lambda}} \{ \gamma(\boldsymbol{\lambda}) = \tilde{f}_0(\boldsymbol{x}(\boldsymbol{\lambda})) + \sum_{j=1}^{m} \lambda_j \tilde{f}_j(\boldsymbol{x}(\boldsymbol{\lambda})) \}, \\ &\text{subject to} \qquad \lambda_j \geq 0, \qquad j = 1, 2, \ldots, m. \end{aligned} \tag{6.3.4}$$

The dual approximate subproblems $P_D[k]$ only require the determination of the m unknowns λ_j subject to the m non-negativity (bound) constraints. These bound constrained subproblems may be solved using any optimization algorithm able to handle the non-negativity constraints on λ_j, $j = 1, 2, \ldots, m$, and the discontinuous second derivatives which arise when variables are freed from the bounds $\check{x}_i$ and $\hat{x}_i$ on the n primal variables, or *vice versa* (Fleury, 1979). The reader is referred to Falk (1967); Fleury (1993*a*,*b*); Duysinx (2008) and Groenwold and Etman (2008) for more detail. Here, we merely note that strict convexity of the primal approximate subproblem $P_P[k]$ is a sufficient (but not a necessary) condition for invoking duality.

SAO algorithms mainly differ in terms of the approximation functions used in the approximate subproblems. The current academic version of the SAO*i* algorithm, in part based on the dual of Falk (1967), uses convex separable approximation functions and the subproblems can easily be cast in an efficient dual form (Section 4.2).

SAO methods aimed at topology (and structural) optimization often use reciprocal-like approximations in a dual setting. The method of moving asymptotes (MMA) and the convex linearization method (CONLIN) are important and telling examples. Both these algorithms use reciprocal-like approximations and generate a series of convex nonlinear programming (NLP) approximate subproblems. The reciprocal-like approximations are obtained by substituting reciprocal intermediate variables into a first-order (linear) Taylor series expansion. The reciprocal intermediate variables introduce nonlinearity (without having to use second-order or Hessian information) and fairly accurately match the nonlinearity present in many objective and constraint functions encountered in structural and topology optimization. The separability and convexity of these approximations may also be used to develop instances of the dual formulation of Falk, which is very efficient for applications where the number of design variables is significantly larger than the number of constraints.

Other separable approximations based on intervening variables have been proposed. Approximations (even if separable) that result in non-convex approximation functions or in the inablility to arrive at an analytical primal-dual relation may hinder the utilization of the Falk dual, and the resulting approximate subproblems are typically solved in their primal form by means of an appropriate mathematical programming algorithm. In addition, non-convex approximations result in the loss of the conservative nature of the approximate subproblems. (An approximate subproblem is considered *conservative* if the objective and constraint function values become greater than or equal to the original function values at the optimal solution of the subproblem. This implies that the optimal solution of the subproblem is a feasible solution to the original problem, with the objective function value lower than the value at the previous iterate.)

Recently, it has been found that (diagonal) quadratic Taylor series approximations to the original nonlinear reciprocal-like (or exponential) approximations may perform equally well, or sometimes even better, than the original approximations (Groenwold and Etman, 2010; Groenwold *et al.*, 2009; Groenwold and Etman, 2009*a*). In Groenwold and Etman (2010), it is demonstrated that diagonal quadratic approximations to the reciprocal and exponential approximate objective functions can be used successfully in topology optimization.

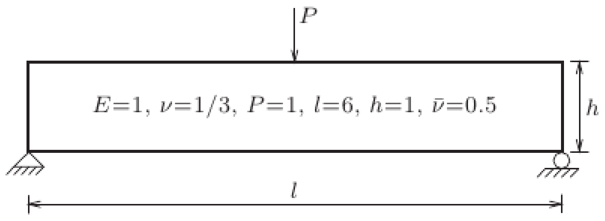

Figure 6.1: The MBB beam (unit thickness; plane stress)

6.4 The MBB Beam

We now consider the well-known minimum compliance topology optimization MBB beam (Figure 6.1). If we use the popular SIMP material model (Bendsøe, 1989), the problem can be expressed as in (6.2.1)

$$\begin{aligned}
\min_{x}\; & f_0(x) = \mathbf{q}^T\mathbf{K}\mathbf{q} = \sum_{i=1}^{n_{et}} (x_i)^p \mathbf{q}_i^{eT}\mathbf{K}_0^e\mathbf{q}_i^e, \\
\text{subject to}\; & f_1 \equiv v - \overline{v} = \sum_{i=1}^{n_{et}} x_i^e v_i^e - \overline{v} \leqslant 0, \\
& \mathbf{K}\mathbf{q} = \mathbf{r}, \\
& \mathbf{0} < \check{x} \leqslant x \leqslant \mathbf{1}.
\end{aligned} \tag{6.4.1}$$

Again, $f_0(x)$ represents the nonlinear objective function and $f_1(x)$ the linear volume constraint. $\mathbf{q} = \mathbf{q}(x)$ and $\mathbf{r}$ represent the global assembled finite element displacement and force vectors respectively, while $\mathbf{K} = \mathbf{K}(x)$ represents the global finite element stiffness matrix. Superscript e indicates the elemental quantities and n_{el} the number of finite elements. x contains the n_{el} elemental densities (the design variables), with $\check{x}$ the lower bounds on x. v represents the structural volume, while $\overline{v}$ is a prescribed limit on the volume. We use a volume fraction $\overline{v} = 0.5$. We set $x_{\min} = 10^{-3}\,\forall\, i$ in (6.3.2), and use a constant move limit $\delta = 0.2$. Due to symmetry, the discretization reported are for half the beam. In the linear mesh independence filter, the filter radius r_{mesh} is fixed at 8% of the beam height.

For the topology optimization of the MBB problem, we use the quadratic replacement to the reciprocal approximation for the nonlinear approximate objective function $\tilde{f}_0(\mathbf{x})$, and the spherical quadratic approximation based on function values for the linear volume constraint function $f_1(\mathbf{x})$. We then make use of the proposed l-BFGS approach to solve the QP primal approximate subproblems $P_P[k]$, and then go on to use the proposed PACBB method to solve the dual approximate subproblems $P_D[k]$.

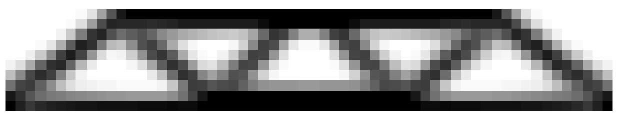

Figure 6.2: MBB beam design: intermediate variable approach with 30 x 10 grid ($k^* = 150; f_0^* = 2.57997005\text{E+02}; |f_j^*| = 4.44089210\text{E-16}$)

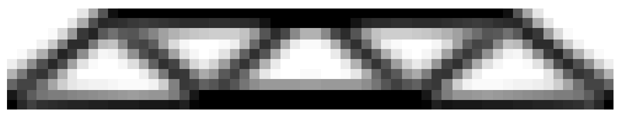

Figure 6.3: MBB beam design: proposed l-BFGS approach with 30 x 10 grid ($k^* = 150; f_0^* = 2.61325414\text{E+02}; |f_j^*| = 0$)

6.5 Topology Optimization Using the Proposed l-BFGS Approach

In this section, we attempt the optimal design of the MBB beam using the proposed (memoryless) l-BFGS approach to approximate Hessian information to solve the quadratic approximate subproblems in the SAO*i* algorithm. We compare the design obtained using the proposed l-BFGS approach (Figure 6.3) with that of the optimal design obtained using the intermediate variable approach (Figure 6.2). The latter retains the convex and separable reciprocal approximations, whereas the proposed l-BFGS implementation does not. The Hessian approximations used in the l-BFGS approach are fully populated and we do not use a diagonal approximation. This means that we loose the conservative nature of the approximate subproblem and the GALAHAD solver 'struggles' to converge to the optimal solution. Since using a less refined grid in the l-BFGS implementation leads to a significant decrease in the CPU time, we use a grid of 30x10 for the comparison. However, a more refined grid is possible when the intermediate variable approach is used, and the results are included here (Figure 6.4). This in itself makes the intermediate variable approach superior to the proposed l-BFGS implementation in the MBB beam application. A maximum of 150 outer iterations is used for both methods.

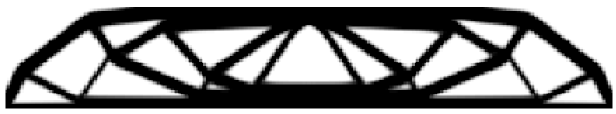

Figure 6.4: MBB beam design: intermediate variable approach with 150 x 50 grid
($k^* = 150; f_0^* = 1.94072436\text{E}+02; |f_j^*| = 4.66293670\text{E}-15$)

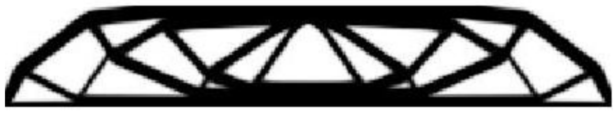

Figure 6.5: MBB beam design: l-BFGS-b dual solver with 150 x 50 grid
($k^* = 150; f_0^* = 1.94074216\text{E}+02; |f_j^*| = 0$)

6.6 Topology Optimization Using the Proposed PACBB Approach

We now attempt the optimal design of the MBB beam using the proposed PACBB approach to solve the quadratic approximate subproblems present in the SAO*i* algorithm in the dual space. We compare the design with that of the optimal design obtained when using the l-BFGS-b dual solver. A maximum of 150 outer iterations and a grid of 150x50 are used for both methods. The optimal design obtained using the proposed PACBB method (Figure 6.6) differs from that obtained using the l-BFGS-b dual solver (Figure 6.5). However, in terms of the objective function value at the optimum point after 150 iterations, the PACBB method compares well with the l-BFGS-b dual solver. Fortunately, the PACBB implementation retains the convex and separable nature of the reciprocal approximation and the expected optimal design is achieved.

6.7 Numerical Comparison

From the comparisons of the optimal solutions obtained for the MBB beam using the proposed methods, it can be concluded that the PACBB approach outperformed the proposed l-BFGS method in the topology application. Based

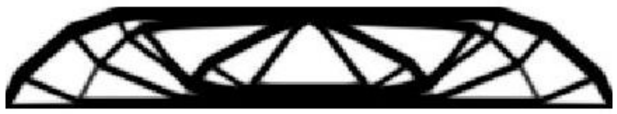

Figure 6.6: MBB beam design: proposed PACBB approach with 150 x 50 grid ($k^* = 150; f_0^* = 1.94853884\text{E+}02; |f_j^*| = 1.33924035\text{E-}05$)

on the numerical results reported on in Chapters 3 and 5, we would expect the opposite. However, as explained, the l-BFGS implementation does not retain the conservative nature of the approximate subproblem (whereas the PACBB dual solver does) and 'struggles' to return an optimal solution.

The minimum compliance topology optimization problem appears deceptively simple, since there is only one constraint, which has to be active for obvious reasons (else we would have $x_i = 1 \forall i$ and the design would be all solid, without any voids). However, the dual objective is discontinuous in the second derivatives, which is a result of the primary variables becoming fixed when free, or vice versa. Considering the large number of primary variables, the implications are obvious.

Chapter 7

Conclusion

In the previous chapters, two alternative approaches for the solution of the quadratic approximate subproblems in the SAO*i* algorithm are proposed and implemented. The application on a topology optimization problem is also discussed. We now summarize the conclusions and give recommendations with regards to future study.

7.1 Conclusions

The numerical results for the memoryless l-BFGS implementation indicate that the proposed method is more efficient and robust than both the intermediate variable approach and the CPLEX solver. Also, when compared to the implementation using exact Hessian information, only a slight increase in the number of iterations is noticed. This is a small price to pay if it means that we are able to avoid the (cumbersome and error-prone) computation of exact second-order information. The l-BFGS is definitely a viable solution to solve the QP subproblems in the SAOi algorithm, with no need to use the complete iteration history or to store fully dense $n \times n$ Hessian approximations. Although it is relatively simple to implement, the choice of starting matrix and the number of previous iteration points used to construct the approximate Hessian are both critical. The 'wrong' choice of these can lead to a great increase in the number of iterations required to reach a solution. Another drawback of the l-BFGS implementation is that it is more expensive in terms of computational requirements compared to the intermediate variable approach and the CPLEX solver. Also, for very large scale optimization problems, the QP solver 'struggles' to return a solution to the diagonal quadratic subproblem when the approximation to the Hessian is dense and ill conditioned. In order to overcome this problem, in depth knowledge of the GALAHAD solver will be required or a different QP solver should be used.

The numerical results for the PACBB method (unfortunately) indicate that the l-BFGS-b dual solver is more efficient and robust than the proposed alternative. However, further investigation shows that the implemented PACBB method delivers a fast initial rate of convergence for the majority of the test problems. This can be derived from the numerical results given for different values of the non-negative tolerance (f_{tol}) used to determine whether a sufficient decrease in the objective function has been reached. For a large number of the test problems, a large decrease in the number of iterations is achieved if an accuracy of only 10^{-03} is used. The PACBB implementation quickly converges to within a neighbourhood of the local or global minimum, but then slows down and sometimes even 'struggles' to converge to the optimal solution. Although the application of a trust region filter results in a slight improvement of the iteration results, the numerical results still indicate that the l-BFGS-b dual solver is superior to the proposed method.

From the comparisons of the optimal solutions obtained for the MBB beam using the proposed methods, it can be concluded that the PACBB approach outperforms the proposed l-BFGS method in the topology application. Based on the concluding remarks made above, we would expect the opposite. However, as mentioned, the l-BFGS implementation does not retain the convex and separable nature of the approximate subproblem (whereas the PACBB dual solver does) and 'struggles' to return an optimal solution.

7.2 Recommendations and Future Work

From the findings made in this study, we recommend that the following are considered for future study:

- For the l-BFGS implementation, investigate the 'problems' experienced by the QP solver in very large scale optimization when the approximation to the Hessian is dense and ill conditioned, and consider the use of alternative QP solvers to overcome this.
- Investigate the influence of the choice of the initial Hessian matrix and the number of previous iteration points used to construct the approximate Hessian on the iteration results.
- For the proposed PACBB approach, explore the use of a different (less expensive) line search method to perform the nonmonotone line search between the current iterate and the projection point if the projection point is not in the feasible set.

- Investigate the possibility of using the implemented PACBB method for the initial steps until the iteration point lies within an 'acceptable' neighbourhood of the optimal point, and then using a more efficient and robust method to achieve a faster rate of convergence to the final solution.

List of References

Barzalai, J. and Borwein, J. (1988). Two point step size gradient methods. *IMA J. Numer. Anal.*, vol. 8, pp. 141–148.

Bendsøe, M. (1989). Optimal shape design as a material distribution problem. *Struct. Optim.*, vol. 1, pp. 193–202.

Bendsøe, M. and Sigmund, O. (2003). *Topology Optimization: Theory, Methods and Applications*. Springer, Berlin.

Byrd, R., Lu, P. and Nocedal, J. (1994*a*). l-BFGS-b: FORTRAN subroutines for large scale bound constrained optimization. Tech. Rep. NAM-11, Northwestern University, EECS Department.

Byrd, R., Lu, P., Nocedal, J. and Zhu, C. (1995). A limited memory algorithm for bound constrained optimization. *SIAM J. Scient. Comput.*, vol. 16, pp. 1190–1208.

Byrd, R., Nocedal, J. and Schnabel, R. (1994*b*). Representations of quasi-Newton matrices and their use in limited memory methods. *Mathematical Programming*, vol. 63, pp. 129–156.

Dai, Y., Hager, W., Schittkowski, K. and Zhang, H. (2005). Cyclic Barzilai-Borwein stepsize method for unconstrained optimization.

Duysinx, P. (2008 May). Solution of topology optimization problems with sequential convex programming. Tech. Rep., LTAS - Automotive Engineering, Institute of Mechanics and Civil Engineering, University of Liège.

Etman, L.F.P., Groenwold, A.A. and Rooda, J.E. (2009 June). On diagonal QP subproblems for sequential approximate optimization. In: *Proc. Eighth World Congress on Structural and Multidisciplinary Optimization*. Lisboa, Portugal. Paper 1065.

Falk, J. (1967). Lagrange multipliers and nonlinear programming. *J. Math. Anal. Appls.*, vol. 19, pp. 141–159.

Fletcher, R., Gould, N., Leyffer, S., Toint, P. and Wächter, A. (2002*a*). Global convergence of a trust-region SQP-filter algorithm for general nonlinear programming. *SIAM J. Optim.*, vol. 13, pp. 635–659.

Fletcher, R. and Leyffer, S. (1998 April). User manual for filterSQP. Numerical Analysis Report NA\181, Department of Mathematics, University of Dundee, Dundee, Scotland.

Fletcher, R. and Leyffer, S. (2002). Nonlinear programming without a penalty function. *Math. Program.*, vol. 91, pp. 239–269.

Fletcher, R., Leyffer, S. and Toint, P. (1998). On the global convergence of an SLP-filter algorithm. Technical Report 00/15, Department of Mathematics, University of Namur, Namur, Belgium.

Fletcher, R., Leyffer, S. and Toint, P. (2002*b*). On the global convergence of a filter-SQP algorithm. *SIAM J. Optim.*, vol. 13, pp. 44–59.

Fleury, C. (1979). Structural weight optimization by dual methods of convex programming. *Int. J. Numer. Meth. Eng.*, vol. 14, pp. 1761–1783.

Fleury, C. (1993*a*). Mathematical programming methods for constrained optimization: Dual methods. In: *Structural Optimization: Status and Promise*, Progress in Astronautics and Aeronautics, pp. 123–150. AIAA. Chapter 7.

Fleury, C. (1993*b*). Recent developments in structural optimization methods. In: *Structural Optimization: Status and Promise*, Progress in Astronautics and Aeronautics, pp. 183–208. AIAA. Chapter 9.

Gould, N., Orban, D. and Toint, P.L. (2004). GALAHAD, a library of thread-safe FORTRAN 90 packages for large-scale nonlinear optimization. *ACM Trans. Math. Software*, vol. 29, pp. 353–372.

Groenwold, A. and Etman, L. (2008). Sequential approximate optimization using dual subproblems based on incomplete series expansions. *Struct. Multidisc. Opt.*, vol. 36, pp. 547–570.

Groenwold, A. and Etman, L. (2009*a*). On the supremacy of reciprocal-like approximations in SAO - a case for quadratic approximations. In: *Proc. Eighth World Congress on Structural and Multidisciplinary Optimization*. Lisboa, Portugal. Paper 1062.

Groenwold, A. and Etman, L. (2010). A quadratic approximation for structural topology optimization. *International Journal for Numerical Methods in Engineering*, vol. 82, pp. 505–524.

Groenwold, A. and Etman, L. (2012). *The 'not-so-short' manual for the SAOi algorithm.*

Groenwold, A., Etman, L. and Wood, D. (2009). Approximated approximations for SAO. *Struct. Mult. Optim.* In press. Available online, DOI 10.1007/s00158-009-0406-0.

Groenwold, A.A. and Etman, L.F.P. (2009*b*). Globally convergent SAO algorithms for large scale simulation-based optimization. In: *Proc. Eighth World Congress on Structural and Multidisciplinary Optimization*. Lisboa, Portugal. Paper 1055.

Liu, D. and Nocedal, J. (1989). On the limited memory BFGS method for large scale optimization. *Mathematical Programming*, vol. 45, pp. 503–528.

Morales, J. (2002). A numerical study of limited memory BFGS methods. *Applied Mathematics Letters*, vol. 15, pp. 481–487.

Nocedal, J. and Wright, S. (2006). *Numerical Optimization*. Springer Series in Operations Research and Financial Engineering, 2nd edn. Springer.

Rozvany, G. and Zhou, M. (1991). Applications of COC method in layout optimization. In: Eschenauer, H., Mattheck, C. and Olhoff, N. (eds.), *Proc. Engineering Optimization in Design Processes*, pp. 59–70. Springer-Verlag, Berlin.

Snyman, J. and Fatti, L. (1987). A multi-start global minimization algorithm with dynamic search trajectories. *Optim. Theory Appl.*, vol. 54, pp. 121–141.

Svanberg, K. (1995). A globally convergent version of MMA without linesearch. In: Rozvany, G. and Olhoff, N. (eds.), *Proc. First World Congress on Structural and Multidisciplinary Optimization*, pp. 9–16. Goslar, Germany.

Svanberg, K. (2002). A class of globally convergent optimization methods based on conservative convex separable approximations. *SIAM J. Optim.*, vol. 12, pp. 555–573.

Wolpert, D. and Macready, W. (1997). No free lunch theorems for optimization. *Evolutionary Computation, IEEE Transactions on*, vol. 1, pp. 67–82.

Zhang, H. and Hager, W. (2004). PACBB: A projected adaptive cyclic Barzilai-Borwein software for box constrained optimization. In: *Proceedings of SIAM Gators Student Conference, Advances in Applied Mathematics*, pp. 1–2. SIAM Gators.

Zielinsky, R. (1981). A statistical estimate of the structure of multiextremal problems. *Math. Program*, vol. 21, pp. 348–356.

Appendices

Source Code for l-BFGS Implementation

FORTRAN code for l-BFGS implementation in the SAO*i* algorithm (version 0.8.8)

The driver for the Galahad QP solvers used to solve the diagonal quadratic approximate subproblems (*drive_galQP*) is called from the secondary SAO driver (*sao_dense*). The FORTRAN code for *drive_galQP* is given below, and the code for the l-BFGS implementation (*QPform_H* = 0) is included. To invoke the proposed l-BFGS implementation, set *subsolver* to 27 in *enforce.f* and *QPform_H* to 0 in *dgalQP.f*. The value of *n_lbfgs* in *enforcce.f* determines the number of previous iteration points used to approximate Hessian information, with *n_lbfgs* = 2 representing the memoryless approach.

```
      subroutine drive_galQP (n,x,m,nq,f_a,c_a,h_a,xlam,x_h,
     &                        acurvn,bcurvn,ccurvn,
     &                        x_l,x_u,f,c,h,gf,gc,gh,ksubiter,
     &                        xkkt,flam,cstage0,ostring,xlam_h,
     &                        nsx,ictrl,lctrl,rctrl,message,yhi,
     &                        outerloop,n2,n3,nm,kmn,xm,gfm,gcm,
     &                        ierr1,eqn,lin)
      implicit none
      include          'ctrl.h'
      character*16     A_string,H_string
      character*90     ostring
      logical          kkt_local,cstage0
      logical          eqn(*),lin(*)
      integer          m,nq,n,n2,n3,ierr,a_ne,h_ne,ksubiter,outerloop
      integer          i,j,k,QPform_A,QPform_H,sdr,relaxed,nsx,message
      integer          A_row(n*m),A_col(n*m),A_ptr(m+1),ierr1
      integer          H_row(n2),H_col(n2),H_ptr(n+1),nm,kmn
      double precision w(n),xlamj,bji,gh(nq,n)
      double precision x(n),xi(n),x_li(n),x_ui(n)
      double precision x_h(n),x_l(n),x_u(n),xlam(m)
      double precision acurvn(n),bcurvn(m,n),gc_h(m,n),gf_h(n)
      double precision ccurvn(nq,n),f,c(m),h(nq),gf(n),gc(m,n)
      double precision f_a,c_a(m),h_a(nq),xkkt,gf_a(n),gc_a(m,n)
      double precision y(m),z(n),xlam_h(m),yhi
      double precision flam,c_l(m),c_u(m),A_val(n*m),g(n)
      double precision H_val(n2)
      double precision xm(n,nm),gfm(n,nm),gcm(m,n,nm)
!
```

```
      integer          nr,A_rowrel((n+1)*m),A_colrel((n+1)*m)
      integer          H_rowrel(n3),H_colrel(n3),H_ptrrel(n+2)
      double precision xr(n+1),xr_u(n+1),xr_l(n+1),gfr(n+1),wrel(n+1)
      double precision gcr(m,n+1),x_lirel(n+1),x_uirel(n+1)
      double precision A_valrel((n+1)*m),xirel(n+1),zrel(n+1)
      double precision H_valrel(n3)
      include          'ctrl_get.inc'
!
! Set QPform_H = 0 to use l-BFGS implementation (subsolver = 27)
!
      ierr = 0
      QPform_A = 2 ! 0 = dense /
   ! 1 = coordinate /
   ! 2 = sparse_by_rows
      QPform_H = 0 ! 0 = dense /
   ! 1 = coordinate /
   ! 2 = sparse_by_rows /
   ! 3 = diagonal
!
! Galahad lsqp solves diagonal QP problems only
      if (subsolver.eq.25) QPform_H = 3
!
      if (pen1.eq.0.d0) then
        sdr = 1
      else
        sdr = 3
      endif
!
      if (sdr.eq.1) then
!
        call QP_pre (n,m,a_ne,c,acurvn,
     &              bcurvn,xlam,gc,x,xi,x_l,x_u,x_li,x_ui,
     &              z,c_l,c_u,w,A_row,A_col,A_val,A_ptr,
     &              A_string,QPform_A,n,QPform_H,H_string,
     &              H_row,H_col,H_ptr,H_val,h_ne,ictrl,
     &              lctrl,rctrl,nm,kmn,xm,gfm,gcm,outerloop,eqn,lin)

! call the Galahad QP solvers (double)
        if     (subsolver.eq.25) then
          call galahad_lsqp (n,m,a_ne,f_a,c_l,c_u,x_li,
     &                      x_ui,A_row,A_col,A_ptr,xi,gf,
     &                      w,xlam,z,A_val,A_string,ierr,
     &                      ksubiter,f)
        elseif (subsolver.eq.26) then
          call galahad_qpa  (n,m,a_ne,f_a,c_l,c_u,x_li,
     &                      x_ui,A_row,A_col,A_ptr,xi,gf,
     &                      w,xlam,z,A_val,A_string,ierr,
     &                      ksubiter,H_string,H_row,H_col,
     &                      H_ptr,H_val,h_ne,f)
        elseif (subsolver.eq.27) then
          call galahad_qpb  (n,m,a_ne,f_a,c_l,c_u,x_li,
     &                      x_ui,A_row,A_col,A_ptr,xi,gf,
     &                      w,xlam,z,A_val,A_string,ierr,
     &                      ksubiter,H_string,H_row,H_col,
     &                      H_ptr,H_val,h_ne,f)
        elseif (subsolver.eq.28) then
          call galahad_qpc  (n,m,a_ne,f_a,c_l,c_u,x_li,
     &                      x_ui,A_row,A_col,A_ptr,xi,gf,
     &                      w,xlam,z,A_val,A_string,ierr,
     &                      ksubiter,H_string,H_row,H_col,
     &                      H_ptr,H_val,h_ne,f)
        endif
!
      else if (sdr.eq.2) then
!
        stop ' sdr.eq.2 is no longer possible '
!
      elseif (sdr.eq.3) then
!
        nr = n+1
        do i=1,n
          xr(i) = x(i)
          xr_u(i) = x_u(i)
          xr_l(i) = x_l(i)
          gfr(i) = gf(i)
        enddo
        xr(nr) = 0.d0
        xr_u(nr) = ymax
        xr_l(nr) = 0.d0
        gfr(nr) = pen1
!
        do j=1,m
          do i=1,n
            gcr(j,i) = gc(j,i)
          enddo
        enddo
        do j=1,m
          gcr(j,nr) = -1.d0
        enddo
!
        call QP_pre (nr,m,a_ne,c,acurvn,bcurvn,xlam,gcr,xr,xirel,xr_l,
     &              xr_u,x_lirel,x_uirel,zrel,c_l,c_u,wrel,A_rowrel,
     &              A_colrel,A_valrel,A_ptr,A_string,QPform_A,n,
     &              QPform_H,H_string,H_rowrel,H_colrel,H_ptrrel,
     &              H_valrel,h_ne,ictrl,lctrl,rctrl,nm,kmn,xm,gfm,gcm,
     &              outerloop,eqn,lin)

! call the Galahad QP solvers (double)
        xlam = -xlam
        if     (subsolver.eq.25) then
          call galahad_lsqp (nr,m,a_ne,f_a,c_l,c_u,x_lirel,
     &                      x_uirel,A_rowrel,A_colrel,A_ptr,xirel,
     &                      gfr,wrel,xlam,zrel,A_valrel,A_string,ierr,
     &                      ksubiter,f)
        elseif (subsolver.eq.26) then
          call galahad_qpa  (nr,m,a_ne,f_a,c_l,c_u,x_lirel,
     &                      x_uirel,A_rowrel,A_colrel,A_ptr,xirel,
     &                      gfr,wrel,xlam,zrel,A_valrel,A_string,ierr,
     &                      ksubiter,H_string,H_rowrel,H_colrel,
     &                      H_ptrrel,H_valrel,h_ne,f)
        elseif (subsolver.eq.27) then
          call galahad_qpb  (nr,m,a_ne,f_a,c_l,c_u,x_lirel,
     &                      x_uirel,A_rowrel,A_colrel,A_ptr,xirel,
     &                      gfr,wrel,xlam,zrel,A_valrel,A_string,ierr,
     &                      ksubiter,H_string,H_rowrel,H_colrel,
     &                      H_ptrrel,H_valrel,h_ne,f)
        elseif (subsolver.eq.28) then
          call galahad_qpc  (nr,m,a_ne,f_a,c_l,c_u,x_lirel,
     &                      x_uirel,A_rowrel,A_colrel,A_ptr,xirel,
     &                      gfr,wrel,xlam,zrel,A_valrel,A_string,ierr,
     &                       ksubiter,H_string,H_rowrel,H_colrel,
     &                       H_ptrrel,H_valrel,h_ne,f)
        endif
!
        do i=1,n
         xi(i) = xirel(i)
        enddo
        yhi = xirel(nr)
!
      else if (sdr.eq.4) then
!
        stop ' sdr.eq.4 is no longer possible '
!
      else
!
        stop ' sdr invalid in drive_gqp'
!
      endif

! update multipliers
      do j=1,m
        xlam(j)= max(-biglam,min(-xlam(j),biglam))
      enddo

! update x in current point
      do i=1,n
        x(i)=min(x_u(i),max(x_l(i),x(i)+xi(i)))
      enddo
!
      message = ierr
!
      return
      end subroutine drive_galQP
!-----------------------------------------------------------------------!
      subroutine QP_pre (n,m,a_ne,c,acurvn,
     &                   bcurvn,xlam,gc,x,xi,x_l,x_u,x_li,x_ui,
     &                   z,c_l,c_u,w,A_row,A_col,A_val,A_ptr,
     &                   A_string,QPform_A,n1,QPform_H,H_string,
     &                   H_row,H_col,H_ptr,H_val,h_ne,
     &                   ictrl,lctrl,rctrl,nm,kmn,xm,gfm,gcm,outerloop,
     &                   eqn,lin)
      implicit none
      include          'ctrl.h'
      integer          n,m,a_ne,h_ne,ierr,i,j,QPform_A,QPform_H,n1,k
      integer          A_row(*),A_col(*),A_ptr(*),nm,kmn,l,q,test
      integer          H_row(*),H_col(*),H_ptr(*),outerloop
      logical          eqn(*),lin(*)
      double precision w(*),xlamj,bji,x(*),xi(*),x_li(*),x_ui(*)
      double precision x_l(*),x_u(*),xlam(*),gc(m,*),z(*),c(*)
      double precision c_l(*),c_u(*),A_val(*),acurvn(*),bcurvn(m,*)
      double precision H_val(*),ddot,temp(n,nm-1)
      double precision xm(n1,nm),gfm(n1,nm),gcm(m,n1,nm)
      double precision s(n,nm-1),y(n,nm-1),sdoty(nm-1)
      double precision sdots,max_Hval
      double precision b(n,nm-1),a(n,nm-1),temp1,temp2,temp3
      character*16     A_string
      character*16     H_string
      include          'ctrl_get.inc'

! initialize n side constraints
```

```
      do i=1,n
        z(i) = 0.d0
      enddo

! m constraint lower bounds
      do j=1,m
        if (eqn(j)) then
          c_l(j) = -c(j)
        else
          c_l(j) = -1.d20
        endif
      enddo

! m constraint upper bounds
      do j=1,m
        c_u(j) = -c(j)
      enddo

! Construct the diagonal Hessian for SAOi
      do i=1,n1
        w(i) = acurvn(i)
        do j=1,m
          bji   = bcurvn(j,i)
          xlamj = xlam(j)
!         if (bji.gt.0.d0.and.xlamj.gt.0.d0) then
! now replaced by below !
          if (xlamj.gt.0.d0) then
            w(i) =  w(i) + bji*xlamj
          endif
        enddo
        w(i) = max(0.d0,w(i))
      enddo
      if (n-n1.eq.0) then
        ! do nothing
      else if (n-n1.eq.1) then
        w(n) = pen2
      else
        stop ' error in constructing the diagonal Hessian'
      endif

! Transform the Hessian to one of four QP forms.
! 0, 1, 1 are not sensible for diagonal; for development only !
      if (QPform_H.eq.1) then
        H_string = 'COORDINATE'
        h_ne = n
        do i=1,n
          H_val(i) = w(i)
          H_row(i) = i
          H_col(i) = i
        enddo
!
      else if (QPform_H.eq.2) then
        H_string = 'SPARSE_BY_ROWS'
        h_ne = n
        do i=1,n
          H_val(i) = w(i)
          H_col(i) = i
          H_ptr(i) = i
        enddo
        H_ptr(n+1) = n+1
```

```
!
      else if (QPform_H.eq.3) then
        H_string = 'DIAGONAL'
        ! do nothing,
! the QP solvers will construct H%val from array w above
!
! l-BFGS implementation - constructs a 'DENSE' Hessian approximation
!
      else ! 0 or anything else
!
        H_string = 'DENSE'
!
        h_ne = (n1*n1+n1)/2
!
! Approximate Hessian by identity matrix for first iterations
!
        if (outerloop.eq.1) then
!
          k = 0
          do i=1,n1
            do j=1,i
              k = k+1
              if (i.eq.j) then
                H_val(k) = 1.d0
              else
                H_val(k) = 0.d0
              endif
            enddo
          enddo
!
        else
!
! Approximate Hessian using l-BFGS for other iterations
!
          do j=1,kmn-1
            do i=1,n1
              s(i,j) = xm(i,kmn-j) - xm(i,kmn-j+1)
            enddo
          enddo
!
          do l=1,m+1
!
            if (l.gt.1) then
               j = l - 1
            endif
            do j=1,kmn-1
              do i=1,n1
                if (l.eq.1) then
                  y(i,j) = gfm(i,kmn-j) - gfm(i,kmn-j+1)
                else
                  temp(i,j) = (gcm(l-1,i,kmn-j) - gcm(l-1,i,kmn-j+1))
                  y(i,j) = y(i,j) + xlam(l-1)*temp(i,j)
                endif
              enddo
            enddo
!
         enddo
!
         do j=1,kmn-1
           sdoty(j) = 0.d0
           do i=1,n1
             sdoty(j) = sdoty(j) + s(i,j)*y(i,j)
           enddo
         enddo
         sdots = 0.d0
         do i=1,n1
            sdots = sdots + s(i,kmn-1)*s(i,kmn-1)
         enddo
!
         test = 0
         do j=1,kmn-1
           if (sdoty(j).ne.0.d0) then
             test = test + 1
           endif
         enddo
!
! If all sdoty = 0, approximate Hessian by zero matrix
!
         if (test.eq.0) then
!
             do j=1,h_ne
               H_val(j) = 0.d0
            enddo
!
         else
!
! If some sdoty > 0, use info from last kmn iterations for which
! sdoty > 0 to update approximate Hessian of Lagrangian
!
            k = 0
            do i=1,n1
               do j=1,i
                  k = k+1
                  if (i.eq.j) then
                     if (sdoty(kmn-1).gt.0.d0) then
                        H_val(k) = sdoty(kmn-1)/max(sdots,1.d-10)
                     else
                        H_val(k) = 1.d0
                     endif
                  else
                     H_val(k) = 0.d0
                  endif
               enddo
            enddo
            if (k.ne.h_ne) stop ' DENSE incorrectly assembled '
!
            do j=1,kmn-1
!
! If sdoty > 0, calculate a and b vectors
             if (sdoty(j).gt.0.d0) then
!
                  do i=1,n1
                     b(i,j) = y(i,j)/max(dsqrt(sdoty(j)),1.d-10)
                  enddo

                  k = 0
                  do i=1,n1
                     do l=1,i
                        k = k+1
                        if (i.eq.l) then
                           a(i,j) = H_val(k)*s(i,j)
                        else
                           a(i,j) = 0.d0
                        endif
                     enddo
                  enddo
                  if (j.gt.1) then
                     do q=1,j-1
                       temp1 = 0.d0
                       temp2 = 0.d0
                       do i=1,n1
                          temp1 = temp1 + b(i,q)*s(i,j)
                          temp2 = temp2 + a(i,q)*s(i,j)
                       enddo
                       do i=1,n1
                          a(i,j) = a(i,j) + temp1*b(i,q) - temp2*a(i,q)
                       enddo
                     enddo
                  endif
                  temp3 = 0.d0
                  do i=1,n1
                    temp3 = temp3 + s(i,j)*a(i,j)
                  enddo
                  do i=1,n1
                    a(i,j) = a(i,j)/max(dsqrt(temp3),1.d-10)
                  enddo
!
               else ! If sdoty <= 0, set a=0 and b=0

                  do i=1,n1
                    b(i,j) = 0.d0
                    a(i,j) = 0.d0
                  enddo
!
              endif
!
            enddo
!
              k = 0
              do i=1,n1
                do j=1,i
                  k = k+1
                  do q=1,kmn-1
                     H_val(k) = H_val(k) + b(i,q)*b(j,q)
     &                                         -a(i,q)*a(j,q)
                  enddo
                enddo
              enddo
              if (k.ne.h_ne) stop ' DENSE incorrectly assembled '
!
            endif
!
        endif
!
        if (n-n1.eq.0) then
          ! do nothing
        else if (n-n1.eq.1) then
          do i=1,n
            if (i.ne.n) then
              H_val(h_ne+i) = 0.d0
            else
              H_val(h_ne+i) = pen2
            endif
          enddo
          h_ne = h_ne + n
        else
          stop ' error in constructing the diagonal Hessian'
        endif
!
      endif

! l-BFGS implementation complete
!
! Transform the Jacobian to one of three QP forms
      if (QPform_A.eq.1) then
        A_string = 'COORDINATE'
        call co_ordinate_A(n,m,gc,A_row,A_col,A_val,a_ne)
!
      else if (QPform_A.eq.2) then
        A_string = 'SPARSE_BY_ROWS'
        call sparse_by_rows_A(n,m,gc,A_ptr,A_col,A_val,a_ne)
!
      else ! 0 or anything else
        A_string = 'DENSE'
        do j=1,m
          do i=1,n
            A_val(n*(j-1)+i) = gc(j,i)
          enddo
        enddo
        a_ne = n*m
      endif

! Initialise move limits
      do i=1,n
        x_li(i)=x_l(i)-x(i)
        x_ui(i)=x_u(i)-x(i)
      enddo

! create copy of x
      call dcopy(n,x,1,xi,1)

      return
      end subroutine QP_pre
!-----------------------------------------------------------------------!
      subroutine co_ordinate_A(n,m,gc,A_row,A_col,A_val,a_ne)
      implicit none
      integer          i,j,n,m,a_ne
      integer          A_row(n*m),A_col(n*m)
      double precision A_val(n*m),gc(m,n)
!
      a_ne = 0
      do j=1,m
        do i=1,n
          if (gc(j,i).ne.0.d0) then
             a_ne = a_ne+1
             A_val(a_ne) = gc(j,i)
             A_row(a_ne) = j
             A_col(a_ne) = i
          endif
        enddo
      enddo
!
      return
      end subroutine co_ordinate_A
!-----------------------------------------------------------------------!
      subroutine sparse_by_rows_A(n,m,gc,A_ptr,A_col,A_val,a_ne)
      implicit none
      integer          i,j,n,m,a_ne,ptr
      integer          A_ptr(m+1),A_col(n*m)
      double precision A_val(n*m),gc(m,n)
!
!  ne  : number of entries
!  ptr : first entry of each row in A_val
!
      a_ne = 0
      do j=1,m   ! loop through rows
        A_ptr(j) = a_ne + 1
        do i=1,n   ! loop through columns
          if (gc(j,i).ne.0.d0) then
             a_ne = a_ne+1
             A_val(a_ne) = gc(j,i)
             A_col(a_ne) = i
          endif
        enddo
      enddo

! A%ptr(m+1) holds the total number of entries plus one
      A_ptr(m+1) = a_ne+1
      return
      end subroutine sparse_by_rows_A
!-----------------------------------------------------------------------!
```

Source Code for PACBB Implementation

FORTRAN code for PACBB implementation in the SAO*i* algorithm (version 0.8.8)

For the implementation of the proposed PACBB dual solver, we developed the *drive_pacbb* driver. It is called from the secondary SAO driver (*sao_dense*). To invoke the proposed PACBB implementation, set *subsolver* to 5 in *enforce.f*.

```
! SAOi:
! SAOi: Mon, April 01, 2013, Marlize Cronje, Stellenbosch
! SAOi: Driver for PACBB to solve Falk-like DQ subproblems.
! SAOi: Sections of this code was taken from the l-bfgs-b code.
! SAOi:
c     --------------------------------------------------------------
c                SIMPLE DRIVER FOR PACBB (Version 1.0)
c     --------------------------------------------------------------
c
c PACBB is a code for solving large nonlinear optimization
c problems with simple bounds on the variables.
c
c References:
c
c [1] H. Zang and W.W. Hager ``PACBB: A Projected Adaptive
c Cyclic Barzilai-Borwein software for
c box constrained optimization'',
c SIAM Gators, Advances in Applied Mathematics,
c Proceedings of Siam Gators Student
c Conference, pp. 1-2, 1994.
c
c NOTE: The user should adapt the subroutine 'timer' if
c 'etime' is not available on the system.
c
c     **************

      subroutine drive_pacbb(nprimal,xprimal,ni,nq,f_a,c_a,
     &                      h_a,iact,nact,x,x_h,
     &                      acurv,bcurv,ccurv,acurvn,
     &                      bcurvn,ccurvn,x_l,x_u,
     &                      f,c,h,gf,gc,gh,ksubiter,
     &                      xkkt,flam,cstage0,ostring,xlam_h,
     &                      ns,ictrl,lctrl,rctrl,message,yhi,
     &                      outerloop,eqn,lin,maxit,maxeval,maxLS)
```

```
!  This driver calls the PACBB code for the Falk dual
      implicit         none
      include          'ctrl.h'
      integer          umax
      parameter        (umax = 8)
      character*60     csave
      character*12     task
      character*90     ostring
      logical          eqn(*),lin(*)
      logical          kkt_local,cstage0,warn
      logical          lgsave(3)
      integer          ni,nq,nprimal,ksubiter,n,iprints
      integer          i,j,nact,iact(nact),nbd(ni),iwa(3*ni)
      integer          outerloop,kN
      integer          ns,ncol(ns),nrow(ns),icnt,message,ifree
      integer          maxit,maxeval,maxLS,psave(14)
      double precision bsave(29),fvector(maxit+1)
      double precision gcvec(ns),bvec(ns),xold(ni),gold(ni)
      double precision flambda,factr,pgtol,yhi
      double precision l(ni),u(ni),g(ni),time1,time2
      double precision dsave(29),xprimal(nprimal),flam
      double precision x_h(nprimal),x_l(nprimal),x_u(nprimal),x(ni)
      double precision acurv,bcurv(ni),ccurv(nq),xlam_h(ni)
      double precision acurvn(nprimal),bcurvn(ni,nprimal)
      double precision ccurvn(nq,nprimal),xlamsml,xlambig
      double precision f,c(ni),h(nq),gf(nprimal),gc(ni,nprimal)
      double precision f_a,c_a(ni),h_a(nq),gh_a(nq,nprimal)
      double precision xkkt,gf_a(nprimal),gc_a(ni,nprimal)
      double precision gh(nq,nprimal),fold,xtemp(ni),ftemp
      double precision timef1,timef2,timef3
      double precision dtemp(ni),d(ni)
      double precision x_al(ni),x_au(ni),f_al,f_au
      double precision a1,a2,f1,f2,al,au
      double precision gcl,NN,x1(ni),x2(ni)
      data             kkt_local /.false./
      include          'ctrl_get.inc'

! construct the sparse matrices
      call sparset (nprimal,ni,gf,gc,ns,ncol,nrow,gcvec,bvec,bcurvn,
     &              ictrl,lctrl,rctrl)

! no output whatsoever
      iprints = -10

! specify the dimension n (For the Falk dual,
! the number of variables is equal to the
! number of inequality constraints.)
      n      = ni

! set bounds on the dual variables
      do 10 i = 1,ni
        nbd(i) = 2
        if (eqn(i)) then
          l(i)   =  max(-biglam,x(i)-DualTrustRadius)
        else
          l(i)   =  max(0.d0   ,x(i)-DualTrustRadius)
        endif
        u(i)     =  min(biglam ,x(i)+DualTrustRadius)
   10 continue
```

```
! start the iteration by initializing task and timer
      call timer(time1)
      task = 'START '
      ostring=' subproblem terminated with standard settings'
      message=0

  111 continue

! call the PACBB code
       call PACBB(n,maxit,maxeval,maxLS,x,l,u,task,lgsave,nbd,
     &      iprint,psave,bsave,fvector,flambda,g,dtemp,d,xold,fold,gold,
     &      xtemp,ftemp,x_al,x_au,x1,x2,al,au,a1,a2,f_al,
     &      f_au,f1,f2,NN,kN,gcl)
!
       if (task(1:2).eq.'FG'.or.task(1:2).eq.'GS') then
 ! check the time limit
         call timer(time2)
         if (time2 - time1 .gt. tlimit) then
           task='STOP: CPU time exceeds limit.'
           ostring=' subproblem terminated on time limit'
           message=1
         else ! construct the falk dual
            call falk (nprimal,ni,xprimal,x,x_l,x_u,x_h,f,c,
     &              gf,gc,f_a,c_a,acurv,bcurv,acurvn,bcurvn,g,
     &              ksubiter,iact,nact,flambda,ns,ncol,nrow,
     &              gcvec,bvec,ictrl,lctrl,rctrl,yhi,xlam_h)
            goto 111
         endif
 ! end of the loop
      endif

! calculate the approximate KKT conditions on the subproblem level
      flam=-flambda
      if (kkt_local) then
        call gradf_a (nprimal,xprimal,f_a,x_h,acurv,acurvn,f,gf,gf_a)
        call gradc_a (nprimal,ni,xprimal,c_a,iact,nact,x_h,
     &               bcurv,bcurvn,c,gc,gc_a)
        call form_kkt (xkkt,nprimal,ni,nq,xprimal,iact,nact,x_l,x_u,
     &                gf_a,gc_a,gh_a,x,xlamsml,xlambig,ifree,
     &                ictrl,lctrl,rctrl)
        write(*,*) ' Local KKT-condition : ',xkkt,xlamsml,xlambig,ifree
      endif
!
      return
      end subroutine drive_pacbb

c====================== The end of driver1 ===========================

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

subroutine PACBB(n,maxit,maxeval,maxLS,x,xl,xu,task,lgsave,nbd,
     &          iprint,psave,bsave,fvector,f,g,dtemp,d,xold,fold,gold,
     &          xtemp,ftemp,x_al,x_au,x1,x2,al,au,a1,a2,f_al,
     &          f_au,f1,f2,NN,kN,gcl)

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

c Algorithm based on PACBB algorithm described in 'PACBB:
c A Projected Adaptive Cyclic Barzilai-Borwein software for
```

```
c box constrained optimization' by H. Zhang and W.W. Hager.
c This subroutine solves box-constrained optimization problems,
c i.e. optimization problems with simple lower and upper bounds
c placed on each of the n design
c variables.
c     n is an integer variable.
c       On entry n is the dimension of the problem.
c       On exit n is unchanged.
c     x is a double precision array of dimension n.
c        On entry x is an approximation to the solution.
c        On exit x is the current approximation.
c     xl is a double precision array of dimension n.
c       On entry xl is the lower bound on x.
c       On exit xl is unchanged.
c     xu is a double precision array of dimension n.
c       On entry xu is the upper bound on x.
c       On exit xu is unchanged.
c     f is a double precision variable.
c       On first entry f is unspecified.
c       On final exit f is the value of the function at x.
c     g is a double precision array of dimension n.
c       On first entry g is unspecified.
c       On final exit g is the value of the gradient at x.
c     maxit is the maximum number of iterations allowed
c     maxeval is the maximum number of function evaluations allowed
c     maxLS is the maximum number of function evaluations allowed in
c           the linesearch per pacbb iteration
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

        implicit                none

        integer                 n,i,k,it,maxit,feval,maxeval,LSfeval
        integer                 maxLS,pp,mm,ll,mbar,l,p,m,lbar,ind
        integer                 tempi,psave(14),stp,counter,inc_l,kN

        logical                 prjctd, cnstnd, boxed, lgsave(3)
        integer                 iprint, nbd(n), iwhere(n)

        character*12            task

        double precision        c,tolx,tolf,atrial,gamma,a_min
        double precision        a_max,md,mbard,bsave(29)
        double precision        sig1,sig2,tol1,tol2,gam1,gam2,beta
        double precision        x(n),xl(*),xu(*),f,g(n),fmin
        double precision        fr,fc,fmax,tempxg(n)
        double precision        temp(n),normtemp,normx,tempxag(n)
        double precision        d(n),fratio
        double precision        delta,aold,anew,s(n),snorm,alfa
        double precision        y(n),ynorm,sy,ss,ratio2,xold(n)
        double precision        ftemp,xtemp(n),dd(n)
        double precision        fvector(maxit+1),xnorm,gnorm
        double precision        fold,cond1,cond2,slope,atemp,gold(n)
        double precision        dtemp(n),cslope,dnorm,sbgnrm,ddum
        double precision        x_al(n),x_au(n),f_al,f_au
        double precision        a1,a2,f1,f2,al,au
        double precision        gcl,NN,x1(n),x2(n)

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

! write headings for results
!        write(6,1001)

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

!returning from drive_pacbb after determining function values

        if (task.ne.'START ') goto 2001

! Algorithm 2.2, Step 0
! Give the starting point and initialize some parameters !

        c = 1.0d-1           ! the slope modification parameter in
                             !(0,1) for the backtracking linesearch
        gamma = 8.d-1        ! the backstepping parameter in (0,1)
                             !for the backtracking linesearch
        tolx = 1.0d-10       ! termination tolerance on progress in
                             ! terms of function/parameter changes
        tolf = 1.0d-5        ! termination tolerance on the
                             ! first-order optimality
        tol1 = 1.0d-6        ! termination tolerance on the
                             ! 'stopping condition'
        tol2 = 1.0d-16       ! tolerance used for division

! (i) !

        a_min = 1.0d-6
        a_max = 1.0d1   ! 0 < a_min < a_max
        sig1 = 2.0d-1
        sig2 = 8.0d-1   ! 0 < sig1 < sig2 < 1 (not as suggested)
        k = 0
        it = 0
        feval = 0

! (ii) !

        pp = 40
        mm = 8
        ll = 3          ! P, M and L positive integers
                        ! with L <= 5 and P >= 4M >= 8L (as suggested)
        gam1 = mm/ll    ! gam1 >= 1, taken as gam1 = M/L (as suggested)
        gam2 = pp/mm    ! gam2 >= 1, taken as gam2 = P/M (as suggested)


! (iii) !

        mbar = 4        ! Mbar a positive integer (as suggested)
        beta = 975.d-3  ! beta > 0 (as suggested)

! (iv) !
! x replaced by its projection if not within specified bounds / feasible region
        call active(n,xl,xu,nbd,x,iwhere,iprint,prjctd,
     &              cnstnd,boxed)

! (v) !

        l = 0        ! Set l = 0
        p = 0        ! Set p = 0
        m = 0        ! Set m = 0
        lbar = 0     ! Set lbar = 0
```

```
! go back to driver to determine function and gradient values at x
        task = 'FG_INI'

        goto 2000

 2001   continue

! Restore local variables (psave stores all integer variables,
! lgsave stores all logical fields and bsave stores all
! double precision variables)

        prjctd      = lgsave(1)
        cnstnd      = lgsave(2)
        boxed       = lgsave(3)
        it          = psave(1)
        feval       = psave(2)
        l           = psave(3)
        p           = psave(4)
        m           = psave(5)
        lbar        = psave(6)
        pp          = psave(7)
        mm          = psave(8)
        ll          = psave(9)
        mbar        = psave(10)
        k           = psave(11)
        maxLS       = psave(12)
        LSfeval     = psave(13)
        stp         = psave(14)
        atrial      = bsave(1)
        fmin        = bsave(2)
        fr          = bsave(3)
        fc          = bsave(4)
        fmax        = bsave(5)
        c           = bsave(6)
        gamma       = bsave(7)
        tolx        = bsave(8)
        tolf        = bsave(9)
        tol1        = bsave(10)
        tol2        = bsave(11)
        a_min       = bsave(12)
        a_max       = bsave(13)
        sig1        = bsave(14)
        sig2        = bsave(15)
        gam1        = bsave(16)
        gam2        = bsave(17)
        beta        = bsave(18)
        xnorm       = bsave(19)
        gnorm       = bsave(20)
        slope       = bsave(21)
        aold        = bsave(22)
        cslope      = bsave(23)
        dnorm       = bsave(24)
        atemp       = bsave(25)
        anew        = bsave(26)
        cond1       = bsave(27)
        cond2       = bsave(28)
        delta       = bsave(29)

        if (task.eq.'NXT_IT') goto 1000
        if (task.eq.'FG_TRL') goto 2002
        if (task(1:4).eq.'GS_I') goto 2003
        if (task(1:4).eq.'GS_F') goto 2004

        feval = 1
        call norm2(x,n,xnorm)
        call norm2(g,n,gnorm)

! Set fmin = fr = fc = f(x1)
        fmin = f
        fr = f
        fc = f
        fmax = f

        fvector(1) = f

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

! Algorithm 2.2 Step 1
! Check if the stopping condition holds

        do i = 1,n
! determine (x - g) and, if it is not within
! the defined bounds, replace it by its projection ...
           tempxg(i) = x(i) - g(i)
        enddo
          call active(n,xl,xu,nbd,tempxg,iwhere,iprint,prjctd,
     &               cnstnd,boxed)
        do i = 1,n
           temp(i) = tempxg(i) - x(i)
        enddo
        call norminf(temp,n,normtemp)
! check if stopping condition holds
        if (normtemp.lt.tol1) then
           write(*,*) 'Stopping condition holds.'
           write(*,*) 'Stopped before iteration process could start.'
           it = k
           task = 'NEW_X '
        endif

        if (task.eq.'NEW_X') goto 2000
! determine the first trial stepsize (alfa1(1))
           call norminf(x,n,normx)
           if (normx.ne.0.d0) then
              atrial = normx/max(normtemp,tol2)
           else
              atrial = 1.d0/max(normtemp,tol2)
           endif
! ensure that a_min <= atrial <= a_max
           if (atrial.lt.a_min) then
              atrial = a_min
           endif
           if (atrial.gt.a_max) then
              atrial = a_max
           endif

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

 1000  continue     !start of loop

           fold = f
           do i = 1,n
              xold(i) = x(i)
              gold(i) = g(i)
           enddo

           if (task.eq.'NXT_IT') then

              fvector(it+1) = f

              call norm2(x,n,xnorm)
              call norm2(g,n,gnorm)

! Determine (x - g) and, if it is not within the
! defined bounds, replace it by its projection
              do i = 1,n
                 tempxg(i) = x(i) - g(i)
              enddo
              call active(n,xl,xu,nbd,tempxg,iwhere,iprint,prjctd
     &                ,cnstnd,boxed)
              do i = 1,n
                 temp(i) = tempxg(i) - x(i)
              enddo
              call norminf(temp,n,normtemp)

           endif

           k = k + 1
           it = k

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

! Algorithm 2.2, Step 2
! Compute the searching
! direction and check if the first trial stepsize
! is truncated or not

! Determine (xk - atrial)*gk) and, if it is not within the
! defined bounds, replace it by its projection
           do i = 1,n
              tempxag(i) = x(i) - atrial*g(i)
           enddo
           call active(n,xl,xu,nbd,tempxag,iwhere,iprint,prjctd,
     &               cnstnd,boxed)
! Compute the searching direction dk = P(x - a*g) - x
          do i = 1,n
              d(i) = tempxag(i) - x(i)
          enddo
          inc_l = 0
          do i = 1,n
            ddum = atrial*dabs(g(i))
            dd(i) = dabs(d(i))
! Check if the first trial stepsize is truncated
! and adjust the value of lbar if necessary
            if (dd(i).gt.0.d0.and.ddum.gt.dd(i).and.inc_l.ne.1) then
               inc_l = 1
            endif
          enddo
          if (inc_l.eq.1) then
             lbar = lbar + 1
          endif
```

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

! Algorithm 2.2, Step 3
! Compute a stepsize alfak and update fr, fmin and more
! using Algorithm 2.1 ... An adaptive nonmonotone line search

! Algorithm 2.1, Step 1
! Possibly reset the reference value
          if (l.eq.ll) then   ! (i) !
             if (fc.ge.fmin) then
                fratio = (fmax-fmin)/max((fc-fmin),tol2)
             else
                fratio = (fmax-fmin)/min((fc-fmin),-tol2)
             endif
             if (fratio.gt.gam1) then
                 fr = fc
             else
                 fr = fmax
             endif
             l = 0
          endif

          if (p.gt.pp) then   ! (ii) !
             if (fmax.gt.f) then
                fratio = (fr-f)/max((fmax-f),tol2)
                if (fratio.ge.gam2) then
                   fr = fmax
                endif
             endif
          endif

! Algorithm 2.1, Step 2
! Test the first trial stepsize atrial with
! different conditions associated with the cycle number m

          delta = 1.d-4
          cond1 = 0.d0
          cond2 = 0.d0
          slope = 0.d0
          do i = 1,n
             slope = slope + g(i)*d(i)
          enddo
          cond1 = fr + delta*atrial*slope
          cond2 = min(fmax,fr) + delta*atrial*slope
          do i = 1,n
             xtemp(i) = x(i)
             x(i) = xtemp(i) + atrial*d(i)
          enddo
          ftemp = f

          task = 'FG_TRL'

          goto 2000

 2002     ontinue

          feval = feval + 1

          if ((m.eq.0.and.f.le.cond1).or.f.le.cond2) then
```

```
            m = m + 1
            alfa = atrial
             p = p + 1
         else
             p = 0  ! completing step 3
         endif

         if (p.ne.0) goto 0002 ! skipping step 3

! Algorithm 2.1, Step 3
! Test other trial stepsizes until some stepsize
! is satisfactory (skip this step if atrial
! satisfies conditions)

         aold = atrial ! (i) !
         atemp = min(a_max,(sig2*aold))  ! (ii) !
         do i = 1,n
            dtemp(i) = atemp*d(i)
            x(i) = xtemp(i) + dtemp(i)
         enddo

         ftemp = f

         LSfeval = 0

         task = 'GS_ILS'

 2003    continue

         if (task(5:6).eq.'LD') goto 2007

!           call backtrack_linesearch(x,n,d,f,g,slope,c,gamma,tolx,
!      &         sig1,maxLS,atemp,LSfeval,a_min,aold,xtemp,
!      &          ftemp,task,stp,dtemp,cslope,dnorm,
!      &         lgsave,bsave,psave)

            call goldsec_linesearch(x,n,d,f,g,sig1,sig2,
    &            maxLS,xtemp,atemp,LSfeval,a_min,aold,a_max,
    &            ftemp,gcl,kN,NN,x_al,x_au,f_al,f_au,f1,f2,
    &             x1,x2,task,al,au,a1,a2)

           goto 2000

!if linesearch done...

 2007      continue

           anew = atemp
           feval = feval + LSfeval
           cond2 = min(fmax,fr) + delta*anew*slope

           counter = 0

 2005      continue

           counter = counter + 1

           aold = anew
           atemp = min(a_max,(sig2*aold))
           task = 'GS_FLS'
 2004      continue

           if (task(5:6).eq.'LD') goto 2008

!             call backtrack_linesearch(x,n,d,f,g,slope,c,gamma,tol2,
!      &                   sig1,maxLS,atemp,LSfeval,a_min,aold,xtemp,
!      &                   ftemp,task,stp,dtemp,cslope,dnorm,
!      &                   lgsave,bsave,psave)

              call goldsec_linesearch(x,n,d,f,g,sig1,sig2,
     &                   maxLS,xtemp,atemp,LSfeval,a_min,aold,a_max,
     &                   ftemp,gcl,kN,NN,x_al,x_au,f_al,f_au,f1,f2,
     &                   x1,x2,task,al,au,a1,a2)

              goto 2000

 2008         continue

             if (aold.gt.1.d-1*atrial) then
                if (atemp.ge.1.d-1*atrial.and.atemp.le.9.d-1*aold) then
                   anew = atemp
                endif
             else
                anew = 5.d-1*aold
             endif
             feval = feval + LSfeval
             cond2 = min(fmax,fr) + delta*anew*slope

             if (f.gt.cond2.and.counter.le.10) goto 2005

 2006        continue

             lbar = lbar + 1
             alfa = anew

! Algorithm 2.1, Step 4
! Possibly update the best value found and the
! candidate value

 0002        continue

             if (f.lt.fmin) then ! (ii) !
                fc = f
                fmin = f
                l = 0
             else
                l = l + 1
             endif

             if (f.gt.fc) then ! (iii) !
                fc = f
             endif

             fvector(it+1) = f

             tempi = min(it+1,mm-1)
             tempi = it + 1 - tempi
             fmax = fvector(tempi)
             do i = (tempi+1),(it+1)
```

```
                fmax = max(fvector(i),fmax)
             enddo

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

! Algorithm 2.2, Step 5
! Compute and choose the first trial stepsize

             do i = 1,n
                s(i) = x(i) - xold(i)
             enddo
             call norm2(s,n,snorm)
             ind = 0
             ratio2 = 1.d0/max(normtemp,tol2)
             if (m.ge.mbar.or.lbar.ge.1.or.k.eq.1) then ! (ii) !
                do i = 1,n
                   y(i) = g(i) - gold(i)
                enddo
                call norm2(y,n,ynorm)
                ind = 1
             elseif (snorm.ge.max(ratio2,1.d0)) then
                do i = 1,n
                   y(i) = g(i) - gold(i)
                enddo
                call norm2(y,n,ynorm)
                ind = 1
             elseif (snorm.le.min(ratio2,1.d0)) then
                sy = 0.d0
                do i = 1,n
                   y(i) = g(i) - gold(i)
                   sy = sy + s(i)*y(i)
                enddo
                call norm2(y,n,ynorm)
                if (sy/max((snorm*ynorm),tol2).ge.beta) then
                   ind = 1
                endif
             endif

             if (ind.eq.1) then  ! (iii) !
                sy = 0.d0
                ss = 0.d0
                do i = 1,n
                   sy = sy + s(i)*y(i)
                   ss = ss + s(i)*s(i)
                enddo
                lbar = 0
                md = m
                mbard = 15.d-1*mbar
                if (sy.le.0.d0.and.md.ge.mbard) then
                   atrial = max(ratio2,alfa)
                   m = 0
                else
                   atrial = max(a_min,min((ss/max(sy,tol2)),a_max))
                   m = 0
                endif
             endif

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

! Algorithm 2.2, Step 6
! Set k = k + 1 and goto Step 1 ... iteration process

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

             call projgr(n,xl,xu,nbd,x,g,sbgnrm)
! terminate if  |proj g|/(1 + |f|) < 1.0d-10.
             ddum = (1.d-10*(1.0d0 + dabs(f)))
             if (sbgnrm.le.ddum.and.task.ne.'NEW_X ') then
!               write(*,*) ' '
!                write(*,*) 'Iteration process stopped: '
!                write(*,*) 'Projected gradient sufficiently small'
!                write(*,*) ' '
                 it = k
                 task = 'NEW_X '
             endif

             ddum = 1.0d-10*max(dabs(fold),dabs(f))
             if ((dabs(fold-f)).le.ddum.and.task.ne.'NEW_X ') then
!                write(*,*) ' '
!                write(*,*) 'Iteration process stopped: '
!                write(*,*) 'Relative reduction in f sufficiently small'
!                write(*,*) ' '
                 it = k
                 task = 'NEW_X '
             endif

             if (k.eq.maxit.and.task.ne.'NEW_X ') then
!               write(*,*) ' '
!               write(*,*) 'Iteration process stopped: '
!               write(*,*) 'Exceeded max number of iterations'
!               write(*,*) ' '
                it = k
                task = 'NEW_X '
             endif

             if (feval.ge.maxeval.and.task.ne.'NEW_X ') then
!               write(*,*) ' '
!               write(*,*) 'Iteration process stopped: '
!               write(*,*) 'Exceeded max number of func. evaluations'
!               write(*,*) ' '
                it = k
!               call finalresults(n,x,xl,xu,it,feval,xopt,x0,
!      &                 maxit,maxeval,maxLS)
                task = 'NEW_X '
             endif

             if (abs(normtemp).le.tol1.and.task.ne.'NEW_X ') then
!               write(*,*) ' '
!               write(*,*) 'Iteration process stopped: '
!               write(*,*) 'Projection norm below tolerance.'
!               write(*,*) ' '
                it = k
                task = 'NEW_X '
             endif

             if (gnorm.le.tolf.and.task.ne.'NEW_X ') then
!               write(*,*) ' '
!               write(*,*) 'Iteration process stopped: '
!               write(*,*) 'Point satisfies optimality condition'
!               write(*,*) ' '
```

```
                it = k
                task = 'NEW_X '
             endif

!            if (task.eq.'NEW_X') then
!               write(*,*) 'pacbb iterations ', it
!            endif
             if (task.eq.'NEW_X ') goto 2000

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

! Some extra stuff before setting k = k + 1 .... !

             task = 'NXT_IT'

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

             goto 1000  ! end of loop

 1001 format (5x,'PACBB ALGORITHM RESULTS',/,147('-'),/,1x,
     &     '  Iteration   Function eval.   Function value   ',
     &     '  xnorm           gnorm',/,147('-'))

 1002 format (5x,i5.2,i5.2,8x,i5.2,10x,
     &        es11.4,6x,es11.4,6x,es11.4)

 2000  continue

             if (task.eq.'NEW_X ') then

! Some extra stuff before setting k = k + 1 .... !

             fold = f
             do i = 1,n
                xold(i) = x(i)
                gold(i) = g(i)
             enddo

             fvector(it+1) = f

             call norm2(x,n,xnorm)
             call norm2(g,n,gnorm)

! Determine (x - g) and, if it is not within the
! defined bounds, replace it by its projection ...
             do i = 1,n
                tempxg(i) = x(i) - g(i)
             enddo
             call active(n,xl,xu,nbd,tempxg,iwhere,iprint,prjctd
     &                 ,cnstnd,boxed)
             do i = 1,n
                temp(i) = tempxg(i) - x(i)
             enddo
             call norminf(temp,n,normtemp)

          endif

! Save local variables
          lgsave(1) = prjctd
          lgsave(2) = cnstnd
          lgsave(3) = boxed

          psave(1)  = it
          psave(2)  = feval
          psave(3)  = l
          psave(4)  = p
          psave(5)  = m
          psave(6)  = lbar
          psave(7)  = pp
          psave(8)  = mm
          psave(9)  = ll
          psave(10) = mbar
          psave(11) = k
          psave(12) = maxLS
          psave(13) = LSfeval
          psave(14) = stp

          bsave(1)  = atrial
          bsave(2)  = fmin
          bsave(3)  = fr
          bsave(4)  = fc
          bsave(5)  = fmax
          bsave(6)  = c
          bsave(7)  = gamma
          bsave(8)  = tolx
          bsave(9)  = tolf
          bsave(10) = tol1
          bsave(11) = tol2
          bsave(12) = a_min
          bsave(13) = a_max
          bsave(14) = sig1
          bsave(15) = sig2
          bsave(16) = gam1
          bsave(17) = gam2
          bsave(18) = beta
          bsave(19) = xnorm
          bsave(20) = gnorm
          bsave(21) = slope
          bsave(22) = aold
          bsave(23) = cslope
          bsave(24) = dnorm
          bsave(25) = atemp
          bsave(26) = anew
          bsave(27) = cond1
          bsave(28) = cond2
          bsave(29) = delta

          end subroutine PACBB

!         BBtime = cputime - time0

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

          subroutine norminf(temp,n,normtemp)

            implicit          none
            integer           n,i
            double precision  temp(n),normtemp

            normtemp = 0.d0
            do i = 1,n
               if (dabs(temp(i)).gt.normtemp) then
                  normtemp = dabs(temp(i))
               endif
            enddo

          end subroutine norminf

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

          subroutine norm2(temp,n,normtemp)

            implicit          none
            integer           n,i
            double precision  temp(n),normtemp

            normtemp = 0.d0
            do i = 1,n
              normtemp = normtemp + temp(i)*temp(i)
            enddo
            normtemp = dsqrt(normtemp)

          end subroutine norm2

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

          subroutine goldsec_linesearch(x,n,d,f,g,sig1,sig2,
     &                 maxLS,xtemp,atemp,LSfeval,a_min,aold,a_max,
     &                 ftemp,gcl,kN,NN,x_al,x_au,f_al,f_au,f1,f2,
     &                 x1,x2,task,al,au,a1,a2)

            implicit          none
            integer           n,maxLS,LSfeval,i,kN
            character*12      task
            double precision  x(n),f,g(n),x_al(n),x_au(n),f_al,f_au,aold
            double precision  xtemp(n),atemp,ftemp,sig1,sig2
            double precision  dnorm,d(n),a_min,a_max,a1,a2,f1,f2,al,au
            double precision  gcl,NN,x1(n),x2(n),ddum2,ddum1

            if (task(5:6).eq.'AL') goto 4001

            if (task(5:6).eq.'AU') goto 4002

            if (task(5:6).eq.'A1') goto 4003

            if (task(5:6).eq.'A2') goto 4004

            if (task(5:6).eq.'A3'.or.task(5:6).eq.'A4') goto 4006


! CHECK DIMENSIONS

            if (size(x).ne.size(d)) stop
!'The vectors sent to the golden section linesearch
! are not of the same dimension.'

! EXECUTE THE LINE SEARCH

            gcl = (3.d0 - dsqrt(5.d0))/2.d0
            NN = log(1.d-6)/log(1.d0-gcl) + 3.d0
            NN = ceiling(NN)

            al = max((sig1*aold),a_min)
            au = min(a_max,(sig2*aold))
            do i = 1,n
               x_al(i) = xtemp(i) + al*d(i)
               x(i) = x_al(i)
            enddo
            if (task(1:4).eq.'GS_I') then
               task = 'GS_IAL'
            else
               task = 'GS_FAL'
            endif

            goto 4000

4001    continue

            f_al = f

            do i = 1,n
               x_au(i) = xtemp(i) + au*d(i)
               x(i) = x_au(i)
            enddo
            if (task(1:4).eq.'GS_I') then
               task = 'GS_IAU'
            else
               task = 'GS_FAU'
            endif
            goto 4000

4002    continue

            f_au = f

            a1 = (1.d0 - gcl)*al + gcl*au
            a2 = gcl*al + (1.d0-gcl)*au
            do i = 1,n
               x1(i) = xtemp(i) + a1*d(i)
               x(i) = x1(i)
            enddo
            if (task(1:4).eq.'GS_I') then
               task = 'GS_IA1'
            else
               task = 'GS_FA1'
            endif
            goto 4000

4003    continue

            f1 = f

            do i = 1,n
               x2(i) = xtemp(i) + a2*d(i)
               x(i) = x2(i)
            enddo
            if (task(1:4).eq.'GS_I') then
               task = 'GS_IA2'
            else
```

```
             task = 'GS_FA2'
          endif
          goto 4000

 4004   continue

          f2 = f

          LSfeval = 4

          kN = 3

! do while (kN.lt.NN.and.LSfeval.lt.maxLS)
 4005   continue

          if (f1.gt.f2) then

            al = a1
            f_al = f1
            a1 = a2
            f1 = f2
            a2 = gcl*al + (1.d0-gcl)*au
            do i = 1,n
               x2(i) = xtemp(i) + a2*d(i)
               x(i) = x2(i)
            enddo
            if (task(1:4).eq.'GS_I') then
               task = 'GS_IA3'
            else
               task = 'GS_FA3'
            endif

         else

            au = a2
            f_au = f2
            a2 = a1
            f2 = f1
            a1 = (1.d0-gcl)*al + gcl*au
            do i = 1,n
              x1(i) = xtemp(i) + a1*d(i)
              x(i) = x1(i)
            enddo
            if (task(1:4).eq.'GS_I') then
               task = 'GS_IA4'
            else
               task = 'GS_FA4'
            endif

        endif

        goto 4000

 4006   continue

        if (task(5:6).eq.'A3') then
           f2 = f
        elseif (task(5:6).eq.'A4') then
           f1 = f
        endif
```

```
       LSfeval = LSfeval + 1

       kN = kN + 1

       ddum1 = dabs(f_au - f_al)
       ddum2 = 1.0d-10*max(dabs(f_au),dabs(f_al))

       if (kN.lt.NN.and.LSfeval.lt.maxLS.and.ddum1.gt.ddum2) goto 4005

       atemp = (al + au)/2.d0
       do i = 1,n
          x(i) = xtemp(i) + atemp*d(i)
       enddo

       LSfeval = LSfeval + 1

       if (task(1:4).eq.'GS_I') then
          task = 'GS_ILD'
       else
          task = 'GS_FLD'
       endif

 4000  continue

       end subroutine goldsec_linesearch

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```