DEPARTMENT OF CIVIL ENGINEERING
UNIVERSITY OF WALES SWANSEA

# A GENERAL PURPOSE COMPUTATIONAL SHELL FOR SOLVING INVERSE AND OPTIMISATION PROBLEMS

## APPLICATIONS TO METAL FORMING PROCESSES

Igor Grešovnik
Dipl.Ing. Physics
Faculty of Mathematics and Physics
University of Ljubljana

THESIS SUBMITTED TO THE UNIVERSITY OF WALES
IN CANDIDATURE FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

C/Ph/239/00 APRIL 2000

DECLARATION

This work has not previously been accepted in substance for any degree and is not being concurrently submitted in candidature for any degree.

Signed (candidate)

Date

STATEMENT 1

This thesis is the result of my own investigations, except where otherwise stated.

Other sources are acknowledged by giving explicit references. A bibliography is appended.

Signed (candidate)

Date

STATEMENT 2

I hereby give consent of my thesis, if accepted, to be available for photocopying and for inter-library loan, and for the title and summary to be made available to outside organisations.

Signed (candidate)

Date

***To my parents***

# *Acknowledgments*

I would like to thank:

- my supervisor, Prof. D.R.J. Owen, for his guidance and advice;
- my mentor, Dr. Tomaž Rodič, for his guidance and support;
- the Slovenian Ministry of Science and Technology for financial support;
- my parents for encouragement and support;
- my colleagues in C3M for productive collaboration;
- Suzana for her patience and understanding.

## *Summary*

A general-purpose optimisation shell has been developed which utilises an arbitrary simulation system for solution of inverse and optimisation problems. Focus is directed at an open and flexible structure of the shell which allows application to a wide variety of problems and independent development of different solution tools, which can be easily integrated into the optimisation system. In this respect, an approach based on isolated treatment of different subproblems, which can be identified in the overall problem, is followed.

The developed concepts were applied to the solution of problems related to metal forming. The approach includes finite element analyses and related design sensitivity evaluations, which are applied in gradient based optimisation algorithms. The combined algorithms are examined by considering implementation of the overall optimisation system.

Examples include inverse estimation of constitutive parameters as well as optimisation of shape and thermo-mechanical processing parameters for thermo-mechanical systems which show highly nonlinear, transient, coupled and path dependent behaviour.

# *Contents*

# 1 INTRODUCTION

## *1.1 Motivation and Scope*

Optimisation is a logical step forward from direct analyses of products and processes. For linear problems this is already a well established field while active research in areas related to nonlinear, transient, coupled and path dependent problems has become feasible relatively recently[4]-[14] due to advances in computer science, numerical analysis[1]-[4] and sensitivity analysis[15],[16].

This work represents an attempt to develop a general purpose optimisation system which can be effectively adapted to problems emerging in engineering and science. A computational shell *Inverse* has been developed, which can be used in conjunction with existing simulation systems in order to perform optimisation tasks. A tempting challenge of designing an optimisation system that is flexible enough for general application has been taken up. A contemporary finite element based simulation environment *Elfen*[21] was utilised in the system, which enabled exploitation of state of the art achievements[22]-[25] in the field of numerical simulations in continuum mechanics.

A prime aim of the shell development was to provide a solution environment capable of solving a large variety of problems. Such an environment has been built around a traditionally structured simulation code, which was not primarily designed to solve optimisation problems and can not be easily restructured. This imposed a particular emphasis on an open and flexible approach to shell development and a tendency of division of problems to sets of separately treated sub-problems.

Under these circumstances the shell has evolved into a general optimisation programme whose scope is not limited in advance to solution of any specific kind of optimisation problems or to usilisation of a specific simulation software. It provides a framework for independent development of optimisation algorithms and other solution tools in the form of separate modules, which are readily integrated in a common optimisation environment. Its flexible user interface implemented through a file interpreter enables interaction of built-in optimisation algorithms and utilities with additional functionality provided in separate modules. The available utilities can be arbitrarily combined in order to define solution strategies for complex problems. These utilities include interfaces with simulation programmes, which provide access to simulation capabilities.

The shell is developed in the scope of a broader project, the aim of which is to build a complete general optimisation system based on advanced development concepts (Figure 1.1)[17]. This system will include a finite element solution environment developed from the very beginning with the intention of being applicable to solution of optimisation problems. This imposes an open and modular structure convenient for building interfaces with other programmes and acceptable to introducing changes in programme structure.

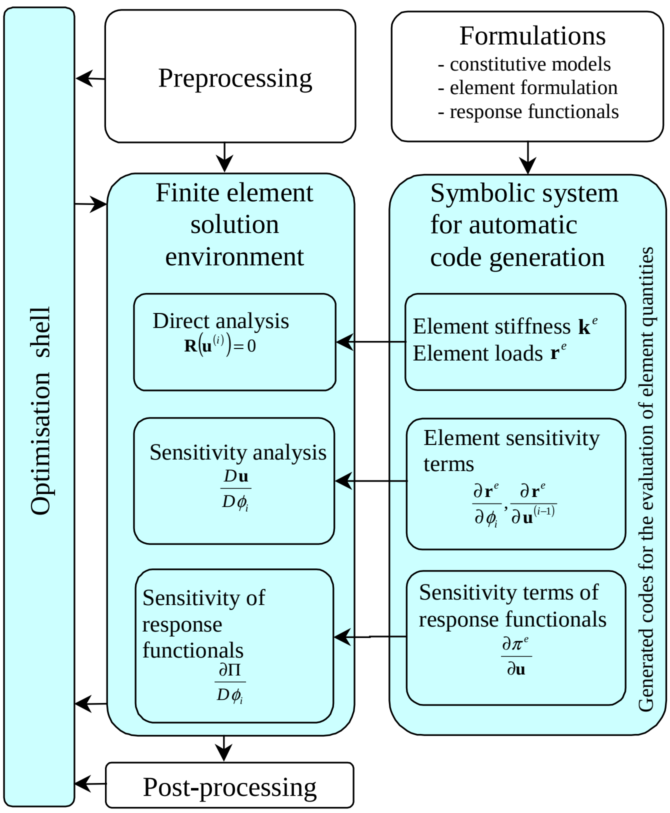



**Figure 1.1:** Outline of the idea of a complete optimisation system[17].

A particularly appealing feature is the use of a symbolic system for automatic generation of element quantities such as element stiffness, loads and sensitivity terms[18]. Element formulations, constitutive models and response functionals are defined on an abstract mathematical level. The system then derives the necessary relations and generates the appropriate functions which are directly incorporated in the simulation code through a standard interface. Such an approach enables quick incorporation of new models and is convenient for handling complexity of sensitivity analysis.

The adequacy of the developed concepts was tested on examples related to metal forming processes. Some typical optimisation problems which arise in this field were tackled, such as optimisation of pre-forms in two stage forming processes and optimal prestressing of cold forging tools.

It is acknowledged that numerical tools must be applied to such complex processes with certain care. Numerical simulations often fail to adequately replicate experimental results obtained in real metal forming processes or even in simple tests. The reason for this often lies in inaccuracy in parameters of physical models or processing conditions which are used as input data for simulations.

Obtaining accurate input data is not always a simple task. Direct evaluation of the required constitutive and processing parameters requires measurements which can not always be performed. This can be overcome by inverse analyses [19],[20], [11]-[14] where the parameters are estimated through minimisation of discrepancies between indirect measurements and simulation results by applying an optimisation algorithm. This approach has been illustrated on some inverse identification problems, for which the developed optimisation shell is also applicable.

## *1.2 Layout of the Thesis*

Chapter 2 introduces the basic aspects of numerical simulations with a main emphasis on sensitivity analysis for non-linear problems, which is performed by using the direct differentiation and adjoint method. A general formal basis is indicated and further references are quoted.

In chapter 3 the mathematical background of numerical methods for the solution of optimisation problems is given. Implementation of some commonly used algorithms is outlined. Discussion is limited to nonlinear programming algorithms which were used within the scope of this work. The aim of this chapter is to give a clear overview with some insight to the numerical complexity behind optimisation algorithms. In the final part of the chapter some practical algorithmic issues are highlighted.

Chapter 4 describes the optimisation shell, which was developed in the scope of the present work. The first part of this chapter is devoted to description of the shell basic concepts and function. This is supported by a brief description of the shell file interpreter and a short example in order to make the presented material less abstract.

The second part touches on the internal structure of the shell through a few representative implementation issues. The selection of the considered issues was made on the basis of significance for the shell concepts and connection with the material in the first part.

Chapter 5 provides some examples which were solved by applying the optimisation shell in conjunction with the finite element simulation programme *Elfen*. In the first example experimental results from a standard tension test are used for the inverse estimation of material hardening parameters. The second example is a simple test for studying the applicability of optimisation techniques in optimal pre-form design. In the third example optimal heating conditions of a two-stage forming process are evaluated. The fourth example is an industrial application where prestressing conditions for cold forging dies are optimised in order to prevent initiation of cracks in the tool and therefore improve the tool life. The chapter is concluded with some other problems, which have been solved using the shell by different authors.

The final part summarises the significance and the scope of the performed work. Some current deficiencies of the optimisation shell are pointed out and guidelines for further development are indicated.

## *References:*

[1] J. Huétink, F. P. T. Baijens (editors), *Simulation of Material Processing: Theory, Methods and Applications*, Proceedings of the 6th International Conference on Numerical Methods in Industrial Forming Processes – NUMIFORM '98 (held in Enschede, Netherlands), A. A. Balkema, Rotterdam, 1998.

[2] D. R. J. Owen, E. Oňate, E. Hinton (editors), *Computational Plasticity – Fundamentals and Applications*, Proceedings of the 5th International Conference on Computational Plasticity (held in Barcelona, Spain), CIMNE, Barcelona, 1997.

[3] M. Geiger (editor), *Advanced Technology of Plasticity*, Proceedings of the 6th International Conference on Technology of Plasticity (held in Nuremberg), Springer – Verlag, Berlin, 1999.

[4] G. Zhao, E. Wright, R. Grandhi, *Preform Die Shape Design in Metal Forming Using an Optimization Method*, *International Journal for Numerical Methods in Engineering*, vol. 40, p.p. 1213-1230, 1997.

[5] E. Wright, R. V. Grandhi, *Integrated Process and Shape Design in Metal Forming With Finite Element Sensitivity Analysis*, Design Optimisation: International Journal of Product & Process Improvement, Vol.1, No. 1, p.p. 55-78, MBC University Press, 1999.

[6] L. Fourment, J. L. Chenot, *Optimal Design for Non-steady-state Metal Forming Processes I. – Shape Optimization Method*, International Journal for Numerical Methods in Engineering, vol. 39, p.p. 33-50, 1996.

[7] L. Fourment, J. L. Chenot, *Optimal Design for Non-steady-state Metal Forming Processes II. – Application of Shape Optimization in Forging*, International Journal for Numerical Methods in Engineering, vol. 39, p.p. 33-50, 1996.

[8] L. Fourment, T. Balan, J.L. Chenot, *Shape optimal design in metal forming*, Proceedings of the Fifth International Conference on Numerical Methods in Industrial Forming Processes - Numiform'95, Balkema, pp. 557-562, 1995

[9] S. Chen, D. A. Tortorelli, *Three-Dimensional Shape Optimization With Variational Geometry*, Structural Optimization 13, p.p. 81-94, Springer-Verlag, 1997.

[10] C. E. K. Silva, E. Hinton, L. E. Vaz, J. Sienz, *2D Shape Optimization with Rate-Independent Elastoplasticity*, in D. R. J. Owen, E. Oňate, E. Hinton (editors), Computational Plasticity – Fundamentals and Applications, Proceedings of the 5th International Conference on Computational Plasticity (held in Barcelona, Spain), CIMNE, Barcelona, pp. 836-854, 1997.

[11] D. S. Schnur, N. Zabaras, *An Inverse Method for Determining Elastic Material Properties and Material Interface*, International Journal for Numerical Methods in Engineering, vol. 33, p.p. 2039-2057, 1992.

[12] J. C. Gelin, O. Ghouati, *Inverse Identification Methods for Material Parameters Estimation in Large Plastic Deformations*, in D. R. J. Owen, E. Oňate, E. Hinton (editors), Computational Plasticity – Fundamentals and Applications, Proceedings of the 4th International Conference on Computational Plasticity (held in Barcelona, Spain), CIMNE, Barcelona, pp. 767-778, 1995.

[13] A. M. Maniatty, M. F. Chen, *Shape Optimization for Steady Forming Processes*, in D. R. J. Owen, E. Oňate, E. Hinton (editors), Computational Plasticity – Fundamentals and Applications, Proceedings of the 4th International Conference on Computational Plasticity (held in Barcelona, Spain), CIMNE, Barcelona, pp. 719-730, 1995.

[14] A. Gavrus, E. Massoni, J. L. Chenot, *Computer Aided Rheology for Nonlinear Large strain Thermo-viscoplastic Behaviour Formulated as an Inverse Problem*, in Bui et al. (eds.): Inverse Problems in Engineering Mechanics, Paris, 1994.

[15] P. Michaleris, D. A. Tortorelli, C. A Vidal, *Tangent Operators and Design Sensitivity Formulations for Transient Non-Linear Coupled Problems with Applications to Elastoplasticity,* Int. Jour. For Numerical Methods in Engineering, vol. 37, pp. 2471-2499, John Wiley & Sons, 1994.

[16] M. Kleiber, H. Antunez, T. D. Hien, P. Kowalczyk, *Parameter Sensitivity in Nonlinear Mechanics – Theory and Finite Element Computations*, John Wiley & Sons, Chichester, 1997.

[17] T. Rodič, J. Korelc, M. Dutko, D.R.J. Owen, *Automatic optimisation of perform and tool design in forging*, ESAFORM bulletin, vol. 1, 1999.

[18] J. Korelc, *Automatic generation of finite-element code by simultaneous optimalization of expressions*, Theoretical Computer Science, 187, p.p. 231-248, 1997.

[19] H. D. Bui, M. Tanaka (editors), *Inverse Problems in Engineering Mechanics*, Proceedings of the 2nd International Symposium on

Inverse Problems – ISIP '94 (held in Paris, France), A. A. Balkema, Rotterdam, 1994.

[20] D. Delaunay, Y. Jarny, K. A. Woodbury (editors), *Inverse Problems in Engineering: Theory and Practice*, Proceedings of the 2nd International Conference (held in Le Croisic, France, 1996), Engineering Foundation, New York, 1998.

[21] *Elfen Implicit User Manual – Version 2.7*, Rockfield Software Ltd., Swansea, 1997.

[22] D. Perić, D. R. J. Owen, M. E. Honnor, *A model for finite strain elasto-plasticity based on logarithmic strains: Computational issues*, Comp. Meth. in Appl. Mech., vol. 94, pp. 35-61, 1992.

[23] D. Perić, D. R. J. Owen, *Computational Model for 3-D Contact Problems with Ffriction Based on the Penalty Method*, Int. J. Num. Meth. Engng. Vol. 35, pp 1289-1309, 1992.

[24] E. A. de Sousa Neto, D. Perić, G. C. Huang, D. R. J. Owen, *Remarks on the Stability on Enhanced Strain Elements in Finite Elasticity and Elastoplasticity*, in D. R. J. Owen, E. Oňate, E. Hinton (editors), Computational Plasticity – Fundamentals and Applications, Proceedings of the 4th International Conference on Computational Plasticity (held in Barcelona, Spain), CIMNE, Barcelona, pp. 361-372, 1995.

[25] N. Petrinić, *Aspects of Discrete Element Modelling Involving Facet-to-Facet Contact Detection and Interaction*, Ph. D. thesis, University of Wales, Swansea, 1996.

[26] S. Stupkiewicz, M. Dutko, J. Korelc, *Sensitivity Analysis for 2-D Frictional Contact Problems*, in ECCM '99 : European Conference on Computational Mechanics (held in Munich, Germany), Technische Universität München, 1999.

# 2 Sensitivity Analysis in Finite Element Simulations

## *2.1 Finite Element Simulations*

The aim of numerical simulations is to predict the behaviour of a system under consideration. In the finite element approach this is performed by solving a set of algebraic equations, which can be expressed in the residual form

$$\mathbf{R}(\mathbf{u}) = \mathbf{0}\,. \tag{2.1}$$

The above equations represent the discretised form of the governing equations including balance laws, constitutive equations, and initial and boundary conditions, which arise in mechanical, thermal, or electromagnetic problems. Unknowns $\mathbf{u}$ define approximate solution and are considered as the primary system response. System (2.1) represents a wide variety of problems and description of finite element techniques to solve particular problems are beyond the scope of this work. A large amount of literature covers this topic, e.g.. references [1] - [7]. This section is focused on basic aspects of sensitivity analysis[10]-[14] for nonlinear problems, which is crucial for efficient optimisation procedures.

The system (2.1) can be solved by the Newton-Raphson method, in which the following iteration is performed (chapter 3, [1]-[7]):

$$\frac{d\mathbf{R}}{d\mathbf{u}}\left(\mathbf{u}^{(i)}\right)\delta\,\mathbf{u} = -\mathbf{R}\left(\mathbf{u}^{(i)}\right), \tag{2.2}$$

$$\mathbf{u}^{(i+1)} = \mathbf{u}^{(i)} + \delta\,\mathbf{u}\,. \tag{2.3}$$

The term $\mathbf{R}\left(\mathbf{u}^{(i)}\right)$ is referred to as the residual (or load) vector and the term $\frac{d\mathbf{R}}{d\mathbf{u}}\left(\mathbf{u}^{(i)}\right)$ is referred to as the tangent operator (or tangential stiffness matrix).

For time dependent problems the iteration scheme given by (2.2) and (2.3) is not sufficient since the state of the system at different times must be determined. Time is usually treated differently to the spatial independent variables. The time domain is discretised according to the finite difference scheme in which approximate states are evaluated for discrete times ${}^{(1)}t, {}^{(2)}t, \ldots, {}^{(M)}t$. Solution for intermediate times is usually linearly interpolated within the intervals $\left[{}^{(n)}t, {}^{(n+1)}t\right]$ and time derivatives of the time dependent quantities are approximated by finite difference expressions.

The approximate solution for the *n*-th time step is obtained by solution of the residual equations

$$ {}^{(n)}\mathbf{R}\left({}^{(n)}\mathbf{u}, {}^{(n-1)}\mathbf{u}\right) = \mathbf{0}, \tag{2.4} $$

which are solved for each time step (or increment) for ${}^{(n)}\mathbf{u}$ while ${}^{(n-1)}\mathbf{u}$ is known from the previous time step. Dependence on earlier increments (${}^{(n-2)}\mathbf{u}$, etc.) is possible when higher order time derivatives are present in the continuum equations (e.g. [8]). The system (24) can again be solved by the Newton-Raphson method in which the following iteration is performed[1]:

$$ \frac{d\,{}^{(n)}\mathbf{R}}{d\,{}^{(n)}\mathbf{u}}\left({}^{(n)}\mathbf{u}^{(i)}\right)\delta\mathbf{u} = -{}^{(n)}\mathbf{R}\left({}^{(n)}\mathbf{u}^{(i)}\right), \tag{2.5} $$

$$ {}^{(n)}\mathbf{u}^{(i+1)} = {}^{(n)}\mathbf{u}^{(i)} + \delta\mathbf{u}. \tag{2.6} $$

The incremental scheme is not used only for transient but also for path dependent problems such as plasticity[22] where constitutive laws depend on evolution of state variables, which inherently calls for an incremental approach[9],[10]. Material response is not necessarily time dependent and the time can be replaced by some other parameter, referred to as pseudo time. Treatment of path dependent material behaviour requires introduction of additional internal state variables, which serve for description of the history effect.

The state of a continuum system is often defined by two distinct fields, e.g. the temperature and displacement fields. Two sets of governing equations define the solution for both types of variables. When neither of these variables can be

[1] The Euler backward integration scheme is considered here, but other schemes such as variable midpoint algorithms can also be incorporated.

eliminated by using one set of equations, both sets must be solved simultaneously and the system is said to be coupled. The approximate solution is obtained by solving two sets of residual equations in each time step:

$$^{(n)}\mathbf{R}\left(^{(n)}\mathbf{u},^{(n-1)}\mathbf{u},^{(n)}\mathbf{v},^{(n-1)}\mathbf{v}\right)=\mathbf{0} \tag{2.7}$$

and

$$^{(n)}\mathbf{H}\left(^{(n)}\mathbf{u},^{(n-1)}\mathbf{u},^{(n)}\mathbf{v},^{(n-1)}\mathbf{v}\right)=\mathbf{0}\,. \tag{2.8}$$

Different solution schemes[9],[16] include either solution of both systems simultaneously in an iteration system, or solution of the systems separately for one set of variables while keeping the other set fixed; the converged sets of variables are in this case exchanged between the two systems.

In the present work the developed optimisation methodology was applied to metal forming processes. Simulations of these processes must take into account complex path dependent and coupled material behaviour. A survey of modelling approaches fot this behaviour can be found in [9].

## 2.2 Sensitivity Analysis

For the purpose of optimisation the notion of parametrisation is introduced. We want to change the setup of the considered system either in terms of geometry, constitutive parameters, initial or boundary conditions, or a combination of these. A set of design parameters $\Phi=[\phi_1,\phi_2,...,,\phi_n]$ is used to describe the properties of the system which can be varied. The equations which govern the system and therefore the numerical solution depend on the design parmeters.

To define optimisation problems certain quantities of interest such as the objective and constraint functions must be defined. For many optimisation algorithms the derivatives of these quantities with respect to the design parameters (i.e. sensitivities) are important. Evaluation of these derivatives is the subject of sensitivity analysis[10]-[13], which is introduced in this section in terms of basic formalism. For this purpose, let us consider a general function, which is dependent on the design parameters which define the system of interest:

$$F(\Phi)=G(\mathbf{u}(\Phi),\Phi) \tag{2.9}$$

$F$ is referred to as the response functional and appears as a term in the objective or constraint functions. $F$ will be typically defined through a system response $\mathbf{u}$, but may in addition include explicit dependence on the design parameters, as is indicated by the right hand side of (2.9). One way of evaluating derivatives $dF/d\phi_i$ is numerical evaluation by the finite difference formula

$$\frac{dF}{d\phi_k}(\phi_1,\phi_2,...,\phi_n) \approx \frac{F(\phi_1,...,\phi_{k-1},\phi_k+\Delta\phi_k,\phi_{k+1}...,\phi_n)-F(\phi_1,...,\phi_{k-1},\phi_k,\phi_{k+1}...,\phi_n)}{\Delta\phi_k}. \quad (2.10)$$

Evaluation of each derivative requires additional evaluation of $F$ at a perturbed set of design parameters, which includes numerical evaluation of the system response $\mathbf{u}$ at the perturbed parameters. More effective schemes, which are incorporated in a solution procedure for evaluation of the system response, are described below.

Derivation of (2.9) with respect to a specific design parameter $\phi = \phi_k$ [1] gives

$$\frac{dF}{d\phi} = \frac{\partial G}{\partial \mathbf{u}}\frac{d\mathbf{u}}{d\phi} + \frac{\partial G}{\partial \phi}. \quad (2.11)$$

Derivatives $\partial G/\partial \mathbf{u}$ and $\partial G/\partial \phi$ are determined explicitly by definition of the functional $F$. The main task of the sensitivity analysis is therefore evaluation of the term $d\mathbf{u}/d\phi$, which is an implicit quantity because the system response $\mathbf{u}$ depends on the design parameters implicitly through numerical solution of the governing equations.

Let us first consider *steady state problems* where the approximate system response can be obtained by solution of a single set of non-linear equations (2.1). Since the system is parametrised, these equations depend on the design parameters and can be restated as

$$\mathbf{R}(\mathbf{u}(\Phi),\Phi) = \mathbf{0}. \quad (2.12)$$

This equation defines implicit dependence of the system response on the design parameters and will be used for derivation of formulae for implicit sensitivity terms.

In the *direct differentiation method* the term $d\mathbf{u}/d\phi$ is obtained directly by derivation of (2.12) with respect to a specific parameter $\phi$, which yields

[1] Index $k$ is suppressed in order to simplify the derived expressions.

$$\frac{\partial \mathbf{R}}{\partial \mathbf{u}}\frac{d\mathbf{u}}{d\phi}=-\frac{\partial \mathbf{R}}{\partial \phi}. \tag{2.13}$$

This set of linear equations must be solved for each design parameter in order to obtain the appropriate implicit term $d\mathbf{u}/d\phi$. This term is then substituted into (2.11) in order to obtain the derivative of the functional $F$ with respect to that parameter. The equation resembles (2.2), which is solved iteratively to obtain the approximate system response. According to this analogy (2.13) is often referred to as a pseudoproblem for evaluation of the implicit sensitivity terms, and the right-hand side $-\partial\mathbf{R}/\partial\phi$ is referred to as the pseudoload. As opposed to (2.2), (2.13) is solved only once at the end of the iterative scheme, because the tangent operator $\partial\mathbf{R}/\partial\mathbf{u}$ evaluated for the converged solution **u** (where equations (2.12) are satisfied) must be taken into account for evaluation of sensitivities. If the system of equations (2.2) is solved by decomposition of the stiffness matrix, then the decomposed tangent stiffness matrix from the last iteration can be used for solution of (2.13), which means that the additional the computational cost includes only back substitution. Evaluation of derivatives with respect to each design parameter therefore contributes only a small portion of computational cost required for solution of (2.12) as opposed to the finite difference scheme, where evaluation of the derivative with respect to each parameter requires a complete solution of (2.12) for the corresponding perturbed design. An additional complication is evaluation of the load vector $-\partial\mathbf{R}/\partial\phi$. It requires explicit derivation of the finite element formulation (more precisely the formulae for evaluation of element contributions to the stiffness matrix) with respect to design parameters, which must be incorporated in the numerical simulation.

An alternative method for evaluation of sensitivities is the *adjoint method*. In this method the implicit term $d\mathbf{u}/d\phi$ is eliminated from (2.11). An augmented functional

$$\tilde{F}(\Phi)=G(\mathbf{u}(\Phi),\Phi)-\lambda^T\mathbf{R}(\mathbf{u}(\Phi),\Phi) \tag{2.14}$$

is defined, where $\lambda$ is the vector[1] of Lagrange multipliers, which will be used for elimination of implicit sensitivity terms. $\tilde{F}=F$ because $\mathbf{R}=0$. Differentiation of (2.14) with respect to a specific design parameter $\phi$ yields

$$\frac{d\tilde{F}}{d\phi}=\frac{\partial G}{\partial \mathbf{u}}\frac{d\mathbf{u}}{d\phi}+\frac{\partial G}{\partial \phi}-\left(\frac{d\lambda}{d\phi}\right)^T\mathbf{R}-\lambda^T\left(\frac{\partial \mathbf{R}}{\partial \mathbf{u}}\frac{d\mathbf{u}}{d\phi}+\frac{\partial \mathbf{R}}{d\phi}\right). \tag{2.15}$$

[1] Vectors denoted by Greek letters are not typed in bold, but it should be clear from the context when some quantity is a vector and when scalar.

Since $\mathbf{R}=0$ by (2.12) and $\frac{\partial \mathbf{R}}{\partial \mathbf{u}}\frac{d\mathbf{u}}{d\phi}+\frac{\partial \mathbf{R}}{d\phi}=0$ by (2.13),

$$\frac{dF}{d\phi}=\frac{d\tilde{F}}{d\phi}. \tag{2.16}$$

The terms in (2.14) which include implicit derivatives are

$$\frac{\partial G}{\partial \mathbf{u}}\frac{d\mathbf{u}}{d\phi}-\lambda^T\frac{\partial \mathbf{R}}{\partial \mathbf{u}}\frac{d\mathbf{u}}{d\phi}=\left(\frac{\partial G}{\partial \mathbf{u}}-\lambda^T\frac{\partial \mathbf{R}}{\partial \mathbf{u}}\right)\frac{d\mathbf{u}}{d\phi} \tag{2.17}$$

These terms are eliminated from (2.15) by defining $\lambda$ so that the term in round brackets in (2.17) is zero. This is achieved if $\lambda$ solves the system

$$\left(\frac{\partial \mathbf{R}}{\partial \mathbf{u}}\right)^T\lambda=\left(\frac{\partial G}{\partial \mathbf{u}}\right)^T. \tag{2.18}$$

System (2.18) is referred to as the adjoint problem for the adjoint response $\lambda$ with the adjoint load $(\partial G/\partial \mathbf{u})^T$. Once multipliers $\lambda$ are evaluated, the derivative of $F$ with respect to a specific parameter $\phi$ is obtained as

$$\frac{dF}{d\phi}=\frac{d\tilde{F}}{d\phi}=\frac{\partial G}{\partial \phi}-\lambda^T\frac{\partial \mathbf{R}}{\partial \phi}. \tag{2.19}$$

The adjoint method requires the solution of the adjoint problem (2.18) for each response functional *F*. It is efficient when the number of response functionals is small compared to the number of design parameters.

A similar approach can be adopted for *transient problems* where sensitivities are evaluated within the incremental solution scheme. As for steady state problems the dependence on the design parameter is taken into account in the discretised governing equations (2.4):

$${}^{(n)}\mathbf{R}\left({}^{(n)}\mathbf{u}(\phi),{}^{(n-1)}\mathbf{u}(\phi),\phi\right)=\mathbf{0}. \tag{2.20}$$

It will be assumed that the response functional is defined through the response for the final time ${}^{(M)}t$, although it can be easily defined as a function of response for intermediate times[10],[12]:

$$F(\Phi)=G\left({}^{(M)}\mathbf{u}(\Phi),\Phi\right). \tag{2.21}$$

Derivation with respect to the parameter $\phi$ yields

$$\frac{dF}{d\phi} = \frac{dG}{d^{(M)}\mathbf{u}} \frac{D^{(M)}\mathbf{u}}{d\phi} + \frac{\partial G}{\partial \phi}. \qquad (2.22)$$

In the *direct differentiation method* the implicit derivative is obtained directly by derivation of (2.20), which yields (after setting the increment index to *M*)

$$\frac{\partial^{(M)}\mathbf{R}}{\partial^{(M)}\mathbf{u}} \frac{d^{(M)}\mathbf{u}}{d\phi} = -\left( \frac{\partial^{(M)}\mathbf{R}}{\partial^{(M-1)}\mathbf{u}} \frac{d^{(M-1)}\mathbf{u}}{d\phi} + \frac{\partial^{(M)}\mathbf{R}}{\partial \phi_i} \right) \qquad (2.23)$$

The pseudoload on the above equation contains the sensitivity of the response evaluated in the previous step. By applying the direct differentiation procedure back in time we see that the system

$$\frac{\partial^{(n)}\mathbf{R}}{\partial^{(n)}\mathbf{u}} \frac{d^{(n)}\mathbf{u}}{d\phi} = -\left( \frac{\partial^{(n)}\mathbf{R}}{\partial^{(n-1)}\mathbf{u}} \frac{d^{(n-1)}\mathbf{u}}{d\phi} + \frac{\partial^{(n)}\mathbf{R}}{\partial \phi_i} \right) \qquad (2.24)$$

must be solved for $d^{(i)}\mathbf{u} / d\phi$ after each time step (i.e. for $i$=1, 2, …, $M$) after convergence of the iteration (2.5) and (2.6), while the derivative of the initial condition $d^{(0)}\mathbf{u} / d\phi$ needed after the first increment is assumed to be known.

In the *adjoint method* the implicit terms are again eliminated by the appropriate definition of the Lagrange multipliers. The augmented functional is defined by combination of (2.21) and (2.20) for all increments:

$$F(\Phi) = G\left({}^{(M)}\mathbf{u}(\Phi), \Phi\right) - \sum_{n=1}^{M} {}^{(n)}\lambda(\Phi)^{T\,(n)}\mathbf{R}\left({}^{(n)}\mathbf{u}(\Phi), {}^{(n-1)}\mathbf{u}(\Phi), \Phi\right) \qquad (2.25)$$

Again $F = \tilde{F}$ follows from (2.20) and $\frac{dF}{d\phi} = \frac{d\tilde{F}}{d\phi}$ follows from (2.20) and (2.24). Derivation of (2.25) yields after rearrangement and some manipulation

$$
\begin{aligned}
\frac{dF}{d\phi} = \frac{d\tilde{F}}{d\phi} = \frac{\partial G}{\partial \phi} - \sum_{n=1}^{M} {}^{(n)}\lambda^{T} \frac{\partial^{(n)}\mathbf{R}}{\partial \phi} - {}^{(1)}\lambda^{T} \frac{d^{(1)}\mathbf{R}}{d^{(0)}\mathbf{u}} \frac{d^{(0)}\mathbf{u}}{d\phi} - \\
- \sum_{n=1}^{M-1} {}^{(n)}\lambda^{T} \frac{\partial^{(n)}\mathbf{R}}{\partial^{(n)}\mathbf{u}} \frac{d^{(n)}\mathbf{u}}{d\phi} + {}^{(n+1)}\lambda^{T} \frac{\partial^{(n+1)}\mathbf{R}}{\partial^{(n)}\mathbf{u}} \frac{d^{(n)}\mathbf{u}}{d\phi} - \\
- {}^{(M)}\lambda^{T} \frac{\partial^{(M)}\mathbf{R}}{\partial^{(M)}\mathbf{u}} \frac{d^{(M)}\mathbf{u}}{d\phi} + \left( \frac{\partial G}{\partial^{(M)}\mathbf{u}} \right)^{T} \frac{d^{(M)}\mathbf{u}}{d\phi}
\end{aligned}
\quad , \qquad (2.26)
$$

where the first line contains explicit terms and the other two lines contain implicit terms which must be eliminated.

Elimination of implicit terms from (2.26) is achieved by solution of the following set of adjoint problems for the Lagrange multiplier vectors:

$$
\begin{aligned}
&\left( \frac{\partial^{(M)}\mathbf{R}}{\partial^{(M)}\mathbf{u}} \right)^{T} {}^{(M)}\lambda = \frac{\partial G}{\partial^{(M)}\mathbf{u}}, \\
&\left( \frac{\partial^{(n)}\mathbf{R}}{\partial^{(n)}\mathbf{u}} \right)^{T} {}^{(n)}\lambda = -\left( \frac{\partial^{(n+1)}\mathbf{R}}{\partial^{(n)}\mathbf{u}} \right)^{T} {}^{(n+1)}\lambda,\ n = M-1, M-2, \ldots, M-1
\end{aligned}
\quad . \qquad (2.27)
$$

Once this is done, the functional derivative is obtained from

$$
\frac{dF}{d\phi} = \frac{d\tilde{F}}{d\phi} = \frac{\partial G}{\partial \phi} - \sum_{n=1}^{M} {}^{(n)}\lambda^{T} \frac{\partial^{(n)}\mathbf{R}}{\partial \phi} - {}^{(1)}\lambda^{T} \frac{d^{(1)}\mathbf{R}}{d^{(0)}\mathbf{u}} \frac{d^{(0)}\mathbf{u}}{d\phi} \qquad (2.28)
$$

Since the equations (2.27) are evaluated in the reverse order to the tangent operators, the complete problem must be solved before the sensitivity analysis can begin. This requires storage of converged (and possibly decomposed) tangent operators from all increments. The adjoint analysis may still be preferred when the number of the design parameters is significantly larger than the number of response functionals.

A similar derivation can be performed for coupled systems (i.e. equations (2.5) and (2. 6)). The procedure is outlined e.g. in [12], and [16]. In the direct method sensitivity of one field is expressed in terms of the sensitivity of another, which gives the dependent and the independent pseudoproblem. In the adjoint methods, two sets of Lagrange multipliers must be introduced, one for each corresponding equation. Two adjoint problems are solved for each set of multipliers for each increment, otherwise the procedure is the same as for non-coupled problems. Sensitivity analysis for various finite element formulations in metal forming is reviewed in [15] and [16] and discussed in detail in [10].

Sensitivity analysis significantly increases the complexity of the simulation code. One complication comes at the global level where the assembled problem is solved in the incremental/iterative scheme. Solution of the adjoint or pseudoproblems must be included in the scheme, which includes assembling of pseudoloads from element terms. This is followed by appropriate substitutions in order to evaluate the complete sensitivities. An additional complication in the adjoint method is that the converged tangent operators must be stored for increments, since solution of the adjoint problems is reversed in time. In this level the additional complexity can be relatively easily kept under control if the programme structure is sufficiently flexible. The number of necessary updates in the code which is primarily aimed for solution of the direct problem is small and the additional complexity in the programme flow chart is comparable to the complexity of the original flow chart.

A more serious problem is the complexity which arises on the element level, where element terms of the pseudoloads are evaluated, i.e. derivatives of the residual with respect to design parameters. The code should be able to evaluate the pseudoload for any parametrisation that might be used, which can include shape, material, load parameters, etc. Implementation of a general solution code which could provide response sensitivities for any possible set of parameters turns out to be a difficult task. It must be taken into account that such a code must include different complex material models and finite element formulations and that derivation of the process of evaluation of element residual terms with respect to any of the possible parameters can be itself a tedious task. Another complication which should not be overlooked is the evaluation of the terms $\partial G/\partial \mathbf{u}$. Although these are regarded as explicit terms, for complex functionals their evaluation is closely related to the numerical model and can include spatial and time integration and derivation of quantities dependent upon history parameters, with respect to the response $\mathbf{u}$.

The reasons outlined above make use of symbolic systems for automatic generation of element level code[17]-[20] (Figure 1.1) highly desirable. In the case of sensitivity analysis use of such systems enables implementation of new finite element formulations and physical models in times drastically shorter then would be needed for manual development. Additionally, use of these systems enables definition of functionals which are used in optimisation and the necessary sensitivity terms on abstract mathematical level where the basic formulation of the numerical model is defined. These definitions can be readily adjusted to new types of problems, because the necessary derivations are performed by the symbolic systems and the appropriate computer code is generated automatically. The system for automatic code generation is connected with a flexible solution environment framework (referred to as the finite element driver[21]) into which the generated code can be readily incorporated. The complexity of inherently combinatorial nature, which would arise in a static simulation code applicable for sensitivity analysis in general problems, can be avoided in this way.

## *References:*


[1] O. C. Zienkiewicz, R. Taylor, *The Finite Element Method*, Vol. 1, 2 (fourth edition), McGraw-Hill, London, 1991.

[2] K.J. Bathe, *Finite Element Procedures*, p.p. 697-745, Prentice Hall, New Jersey, 1996.

[3] J. Bonet, R.D Wood, *Nonlinear Continuum Mechanics for Finite Element Analysis*, Cambridge University Press, Cambridge, 1997.

[4] M. A. Crisfield, *Non-Linear Finite Element Analysis of Solids and Structures*, Vol. 1, John Wiley & Sons, Chichester, 1991.

[5] M. A. Crisfield, *Non-Linear Finite Element Analysis of Solids and Structures*, Vol. 2, John Wiley & Sons, Chichester, 1997.

[6] D. R. J. Owen, E. Hinton, *An Introduction to Finite Element Computations*, Pineridge Press, Swansea, 1979.

[7] D. R. J. Owen, E. Hinton, *Finite Elements in Plasticity*, Pineridge Press, Swansea, 1980.

[8] N. Petrinić, *Aspects of Discrete Element Modelling Involving Facet-to-Facet Contact Detection and Interaction*, Ph. D. thesis, University of Wales, Swansea, 1996.

[9] T. Rodič, *Numerical Analysis of Thermomechanical Processes During Deformation of Metals at High Temperatures*, Ph. D. thesis, University of Wales, Swansea, 1989.

[10] M. Kleiber, H. Antunez, T. D. Hien, P. Kowalczyk, *Parameter Sensitivity in Nonlinear Mechanics – Theory and Finite Element Computations*, John Wiley & Sons, Chichester, 1997.

[11] K. Dems, Z. Mroz, *Variational Approach to First and Second Order Sensitivity Analysis of Elastic Structures*, International Journal for Numerical Methods in Engineering, Vol. 21, p.p. 637-661, John Wiley & Sons, 1984.

[12] P. Michaleris, D. A. Tortorelli, C. A Vidal, *Tangent Operators and Design Sensitivity Formulations for Transient Non-Linear Coupled Problems with Applications to Elastoplasticity*, Int. Jour. For Numerical Methods in Engineering, vol. 37, pp. 2471-2499, John Wiley & Sons, 1994.

[13] D. A. Tortorelli, *Non-linear and Time Dependent Structural Systems: Sensitivity Analysis and Optimisation*, Material for the DCAMM

course (held in Lingby, Denmark), Technical University of Denmark, 1997.

[14] D. A. Tortorelli, P. Michaleris, *Design Sensitivity Analysis: Overview and Review*, Inverse Problems in Engineering, vol. 1, pp. 71-105, Overseas Publishers Association, 1994.

[15] S. Badrinarayanan, N. Zabaras, *A Sensitivity Analysis for the Optimal Design of Metal Forming Processes*, Comput. Methods Appl. Mech. Engrg. 129, pp. 319-348, Elsevier, 1996.

[16] I. Doltsinis, T. Rodič, *Process Design and Sensitivity Analysis in Metal Forming*, Int. J. Numer. Meth. Engng., vol. 45, p.p. 661-692, John Wiley & Sons, 1999.

[17] J. Korelc, *Automatic generation of finite-element code by simultaneous optimalization of expressions*, Theoretical Computer Science, 187, p.p. 231-248, 1997.

[18] J.Korelc, P. Wriggers, *Symbolic approach in computational mechanics*, in Computational Plasticity Fundamentals and Applications (ed. D.R.J. Owen, E. Onate and E. Hinton), pp. 286-304, CIMNE, Barcelona, 1997.

[19] J. Korelc, *Symbolic Approach in Computational Mechanics and its Application to the Enhanced Strain Method*, Ph. D. thesis, Technische Hochschule Darmstadt, 1996.

[20] T. Rodič, I. Grešovnik, D. Jelovšek, J. Korelc, *Optimisation of Prestressing of a Cold Forging Die by Using Symbolic Templates*, in ECCM '99 : European Conference on Computational Mechanics (held in Munich, Germany), pp. 362-372, Technische Universität München, 1999.

[21] T. Šuštar, *FEM Driver*, electronic document at http://www.c3m.si/driver/ , Ljubljana, 2000.

[22] J. Lubliner, *Plasticity Theory*, Macmillan Publishing Company, New York, 1990.

# 3 NUMERICAL OPTIMISATION

## 3.1 Introduction

### 3.1.1 Preliminaries

In general, optimisation problems can be stated as problems of minimisation of some function of the design parameters **x**, subjected to certain constraints, i.e.:

$$\begin{array}{lll} \textit{minimise} & f(\mathbf{x}), & \mathbf{x} \in \mathbb{R}^n \\ \textit{subject to} & c_i(\mathbf{x}) = 0, & i \in E \\ \textit{and} & c_j(\mathbf{x}) \geq 0, & j \in I\ , \end{array} \tag{3.1}$$

where $f(\mathbf{x})$ is the objective function and $c_i(\mathbf{x})$ and $c_j(\mathbf{x})$ are constraint functions[1]. Design parameters are also referred to as optimisation variables. The second line of (3.1) represents the equality constraints of the problem and the third line represents the inequality constraints. We have introduced two index sets, set $E$ of the equality constraint indices and set $I$ of the inequality constraint indices. The above problem is also referred to as the general nonlinear problem. Most of optimisation problems can be expressed in this form, eventually having multiple objective functions in the case of several conflicting design objectives.

Points $\mathbf{x}'$, which satisfy all constraints, are called feasible points and the set of all such points is called the feasible region. A point $\mathbf{x}^*$ is called a constrained local minimiser (or local solution of the above problem) if there exists some neighbourhood $\Omega$ of $\mathbf{x}^*$ such that $f(\mathbf{x}^*) \leq f(\mathbf{x}')$ for all feasible points $\mathbf{x}' \in \Omega, \mathbf{x}' \neq \mathbf{x}^*$. Such a point is called a strict local minimiser if the < sign is applied in place of $\leq$; a

[1] Number of optimisation variables will be denoted by $n$ throughout chapter 3.

slightly stronger definition of isolated local minimiser, which requires the minimiser to be the only local minimiser in some neighbourhood. Furthermore, **x*** is called the global solution or global constrained minimiser if $f\left(\mathbf{x}^*\right) \leq f\left(\mathbf{x}'\right)$ for all feasible points **x**'. This means that a global minimiser is the local solution with the least value of *f*.

Since the objective and constraint functions are in general nonlinear, the optimisation problem can have several constrained local minimisers $\mathbf{x}^*$. The goal of optimisation is of course to comply with the objective as much as possible, therefore the identification of the global solution is the most desirable. However, this problem is in general extremely difficult to handle. Actually there is no general way to prove that some point is a global minimiser. At best some algorithms are able to locate several local solutions and one can then take the best one of these. These methods are mostly based on some stochastic search strategy. Location of problem solutions is of a statistical nature, which inevitably leads to an enormous number of function evaluations needed to locate individual solutions with satisfactory accuracy and certainty. These methods are therefore usually not feasible for use with costly numerical simulations and are not included in the scope of this work. Currently the most popular types of algorithms for identifying multiple local solutions are the simulated annealing algorithms and genetic algorithms, briefly described in [9].

The optimisation problem can appear in several special forms dependent on whether the inequality or equality constraints are present or not, and whether the objective and constraint functions have some simple form (e.g. are linear or quadratic in the optimisation parameters). These special cases are interesting for mathematical treatment because it is usually possible to construct efficient solution algorithms that take advantage of the special structure.

In the cases related to this work, the objective and constraint functions are typically evaluated implicitly through a system response evaluated with complex numerical simulation. Here it can not be assumed that these functions will have any advantageous structure. At most there are cases with linear constraints or constraints that can be reduced to the form of simple bounds on variables, and in some cases it is possible to manage the problem without setting any constraints. Treatment of optimisation algorithms in this chapter will correspond to this fact. Some problems with special structure will however be considered since they appear as sub-problems in general algorithms. Example of this is the problem (3.1) with a quadratic objective function and linear constraint functions (the so called quadratic programming or QP problem), which often appears in algorithms for general constrained and unconstrained minimsation.

It proves that solution of the constrained problem is essentially more complex than solution of the unconstrained problem. Also theoretical treatment of the latter is in many aspects a natural extension of unconstrained minimisation, therefore the first part of this section is dedicated to the general unconstrained minimisation in multivariable space. Some attention is drawn to show parallels with solution of systems of nonlinear equations, which is the core problem in numerical simulations related to this work. The source of additional complexity that arises in practical unconstrained minimisation, as compared to the solution of nonlinear equations that appear in simulations, will be addressed. The aim of this section is to represent the theoretical background used in treatment of this complexity in order to assure satisfactory local and global convergence properties. Basic treatment of the one dimensional line search, a typical property of most practical algorithms, is also given in this context.

In the second part a more general constrained problem will be addressed. The additional mathematical background such as necessary and sufficient conditions will be given first. The two most commonly used approaches to constrained optimisation will then be described: sequential unconstrained minimisation and sequential quadratic programming.

The section is concluded with some practical considerations with regard to the present work. Some practical problems that can give rise to inadequacy of the described theory will be indicated. A problem strongly related to this work is optimisation in the presence of substantial amounts of numerical noise, which can cause serious difficulties to algorithms based on certain continuity assumptions regarding the objective and constraint functions.

### 3.1.2 Heuristic Minimisation Methods and Related Practical Problems

In the subsequent text the unconstrained problem is considered, namely

$$\textit{minimise} \qquad f(\mathbf{x}), \qquad \mathbf{x} \in \mathbb{R}^n \tag{3.2}$$

Throughout this chapter it is assumed that $f$ is at least a $\mathbb{C}^2$ function, i.e. twice continuously differentiable with respect to $\mathbf{x}$. Every local minimum is a stationary point of $f$, i.e. a point with zero gradient[1]:

$$\nabla f(\mathbf{x}^*) = \mathbf{g}(\mathbf{x}^*) = \mathbf{g}^* = 0\,. \tag{3.3}$$

Minimisation can therefore be considered as a solution of the above equation, which is essentially a system of nonlinear equations for gradient components

$$g_i(\mathbf{x}) = \frac{\partial f(\mathbf{x})}{\partial x_i} = 0\,,\ \ i = 1,\ldots n\,. \tag{3.4}$$

This is essentially the same system that arises in finite element simulation[34] and can be solved by the standard Newton method, for which the iteration is

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - \left(\nabla \mathbf{g}^{(k)}\right)^{-1} \mathbf{x}^{(k)}\,. \tag{3.5}$$

The notation $\mathbf{g}^{(k)} = \mathbf{g}\left(\mathbf{x}^{(k)}\right)$ is adopted throughout this work.

The method is derived from the Taylor series[30],[32] for $\mathbf{g}$ about the current estimate $\mathbf{x}^{(k)}$:

$$\mathbf{g}\left(\mathbf{x}^{(k)} + \delta\right) = \mathbf{g}^{(k)} + \nabla \mathbf{g}^{(k)} \delta + O\left(\left\|\delta\right\|^2\right) \tag{3.6}$$

Considering this as the first order approximation for $\mathbf{g}$ and equating it to zero we obtain the expression for step $\delta$ which should bring the next estimate close to the solution of (3.4)[1]:

$$\nabla \mathbf{g}^{(k)} \delta = -\mathbf{g}^{(k)}\,.$$

By setting $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \delta$ we obtain the above Newton Iteration.

The Newton method is known to be rapidly convergent[2], but suffers for a lack of global convergence properties, i.e. the iteration converges to the solution only in some limited neighbourhood, but not from any starting point. This is the fundamental reason that it is usually not applicable to optimisation without modifications. The problem can usually be elegantly avoided in simulations, either because of some nice physical properties of the analysed system that guarantee global convergence, or by the ability of making the starting guess arbitrarily close to the equilibrium point where the equations are satisfied. This is, for example, exploited in the solution of path dependent problems where the starting guess of the current iterate is the equilibrium of the previous, and this can be set arbitrarily close to the solution because of the continuous nature of the governing equations. Global convergence can be ensured simply by cutting down the step size, if necessary.

In practice, this is usually not at all case in optimisation. The choice of a good starting point typically depends only on a subjective judgment where the solution should be, and the knowledge used for this is usually not sufficient to choose the

[1] Notation $\mathbf{g}(\mathbf{x}) = \nabla f(\mathbf{x})$, $f^{(k)} = f\left(\mathbf{x}^{(k)}\right)$, $\mathbf{g}^{(k)} = \mathbf{g}\left(\mathbf{x}^{(k)}\right)$, etc. will be generally adopted throughout this text.

starting point within the convergence radius of Newton's method, especially due to the complex non-linear behaviour of $f$ and consequently **g**. Modifications to the method must therefore be made in order to induce global convergence[1], i.e. convergence from any starting guess.

One such modification arises from considering what properties the method must have in order to induce convergence to the solution. The solution $\mathbf{x}^*$ must be a limiting point of the sequence of iterations. This means that the distance between the iterates and the solution tends towards zero, i.e.

$$\lim_{k\to\infty} \left\| \mathbf{x}_k - \mathbf{x}^* \right\| = 0 . \tag{3.7}$$

This is satisfied if the above norm is monotonically decreasing and if the sequence has no accumulation point other than $\mathbf{x}^*$. When considering the minimisation problem and assuming that the problem has a unique solution, the requirements for a decreasing norm can be replaced (because of continuity of $f$) by the requirement that $f^{(k)}$ are monotonically decreasing. By such consideration, a basic property any minimisation algorithm should have, is the generation of descent iterates so that

$$f^{(k+1)} < f^{(k)} \quad \forall k . \tag{3.8}$$

This is closely related to the idea of line search, which is one of the elementary ideas in construction of minimisation algorithms. The idea is to minimise $f$ along some straight line starting from the current iterate. Many algorithms are centered on this idea, trying to generate a sequence of directions along which line searches are performed, such that a substantial reduction of $f$ is achieved in each line search and such that, in the limit, the rapid convergence properties of Newton's method are inherited.

An additional complication which limits the applicability of Newton's method is that the second derivatives of the objective function (i.e. first derivatives of its gradient) are required. These are not always directly available since double differentiation of numerical models is usually a much harder problem than single differentiation. Alternatively the derivatives can be obtained by straight numerical differentiation using small perturbation of parameters, but in many cases this is not applicable because numerical differentiation is very sensitive to errors in function evaluation[31],[33], and these can often not be avoided sufficiently when numerical models with many degrees of freedom are used. Furthermore, even if the Newton method converges, the limiting point is only guaranteed to be a stationary point of $f$,

[1] Herein the expression global convergence is used to denote convergence to a local solution from any given starting point. In some of the literature this expression is used to denote convergence to a global solution.

but this is not a sufficient condition for a local minimum, since it includes saddle points, which are stationary points but are not local minimisers.

The most simple algorithm that incorporates the idea of line search is sequential minimisation of the objective function in some fixed set of *n* independent directions in each iterate, most elementarily parallel to the coordinate axes. The requirement for *n* independent directions is obvious since otherwise the algorithm could not reach any point in $\mathbb{R}^n$. The method is called the alternating variables method and it seems to be adequate at a first glance, but turns out to be very inefficient and unreliable in practice. A simple illustration of the reasons for this is that the algorithm ignores the possibility of correlation between the variables. This causes the search parallel to the current search direction to destroy completely the property that the current point is the minimiser in previously used directions. This leads to oscillatory behaviour of the algorithm as illustrated in Figure 3.1.

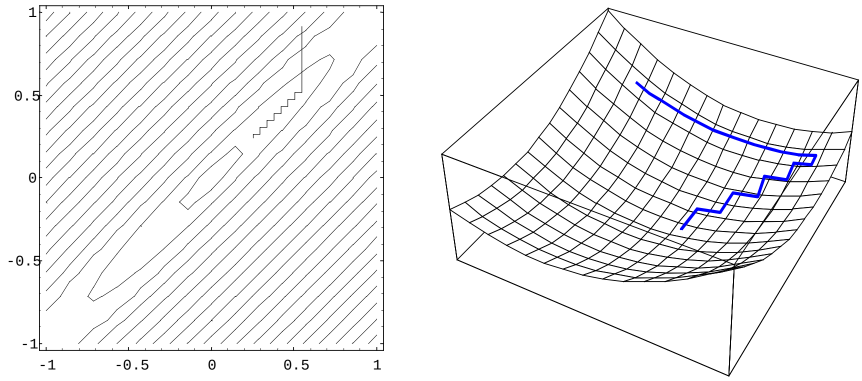



**Figure 3.1:** Oscillatory behaviour, which is likely to occur when using sequential minimisation in a fixed set of directions.

Another readily available algorithm is sequential minimisation along the current direction of the gradient of *f*. Again this seems to be a good choice, since the gradient is the direction of the steepest descent, i.e. the direction in which *f* decreases most rapidly in the vicinity of the starting point. With respect to this, the method is called the steepest descent method. In practice, however, the method suffers for similar problems to the alternating variables method, and the oscillating behaviour of this method is illustrated in Figure 3.2. The theoretical proof of convergence exists, but it can also be shown that locally the method can achieve an arbitrarily slow rate of linear convergence[1].

The above discussion clearly indicates the necessity for a more rigorous mathematical treatment of algorithms. Indeed the majority of the up-to-date algorithms have a solid mathematical background[1]-[7], [26] and partially the aim of this section is to point which are the most important features in the design of fast and reliable algorithms.

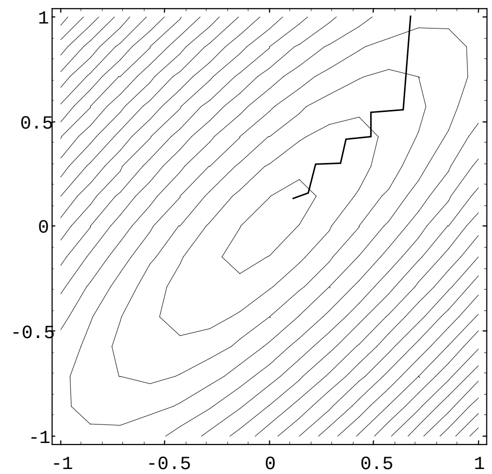



**Figure 3.2:** Oscillatory behaviour, which can occur when performing sequential line searches along the steepest descent directions.

## *3.2 Simplex Method*

One minimisation method that does not belong within the context of the subsequent text is the simplex method[12], [26],[1]. It has been known since the early sixties and could be classed as another heuristic method since it is not based on a substantial theoretical background.

The simplex method neither uses line searches nor is based on minimisation of some simplified model of the objective function, and therefore belongs to the class of direct search methods. Because of this the method does not compare well with other described methods with respect to local convergence properties. On the other hand, for the same reason it has some other strong features. The method is relatively insensitive to numerical noise and does not depend on some other properties of the objective function (e.g. convexity) since no specific continuity or other assumptions are incorporated in its design. It merely requires the evaluation of function values. Its performance in practice can be as satisfactory as any other non-derivative method, especially when high accuracy of the solution is not required and the local convergence properties of more sophisticated methods do not play so important role. In many cases it does not make sense to require highly accurate solutions of optimisation problems, because the obtained results are inevitably inaccurate with respect to real system behaviour due to numerical modeling of the system (e.g. discretisation and round-off errors or inaccurate physical models). These are definitely good arguments for considering practical use of the method in spite of the lack of good local convergence results with respect to some other methods.

The simplex method is based on construction of an evolving pattern of *n+1* points in $\mathbb{R}^n$ (vertices of a simplex). The points are systematically moved according to some strategy such that they tend towards the function minimum. Different strategies give rise to different variants of the algorithm. The most commonly used is the Nelder-Mead algorithm described below. The algorithm begins by choice of *n+1* vertices of the initial simplex ($\mathbf{x}_1^{(1)}, \ldots, x_{n+1}^{(1)}$) so that it has non-zero volume. This means that all vectors connecting a chosen vertex to the reminding vertices must be linearly independent, e.g.

$$\exists \lambda_i \neq 0 \Rightarrow \sum_{i=1}^{n} \lambda_i \left(\mathbf{x}_{i+1}^{(1)} - \mathbf{x}_1^{(1)}\right) \neq 0 .$$

If we have chosen $\mathbf{x}_1^{(1)}$, we can for example obtain other vertices by moving, for some distance, along all coordinate directions. If it is possible to predict several points that should be good according to experience, it might be better to set vertices to these points, but the condition regarding independence must then be checked.

Once the initial simplex is constructed, the function is evaluated at its vertices. Then one or more points of the simplex are moved in each iteration, so that each subsequent simplex consists of a better set of points:

**Algorithm 3.1:** The Nelder-Mead simplex method.

After the initial simplex is chosen, function values in its vertices are evaluated: $f_i^{(1)} = f\left(\mathbf{x}_i^{(1)}\right), i = 1, \ldots, n+1$.

Iteration $k$ is then as follows:

1. **Ordering step:** Simplex vertices are first reordered so that $f_1^{(k)} \leq f_2^{(k)} \leq \ldots \leq f_{n+1}^{(k)}$, where $f_i^{(k)} = f\left(\mathbf{x}_i^{(k)}\right)$.
2. **Reflection step:** The worst vertex is reflected over the centre point of the best $n$ vertices ($\overline{\mathbf{x}}^{(k)} = \frac{1}{n}\sum_{i=1}^{n}\mathbf{x}_i^{(k)}$), so that the reflected point $\mathbf{x}_r^{(k)}$ is

$$\mathbf{x}_r^{(k)} = \overline{\mathbf{x}}^{(k)} + \left(\overline{\mathbf{x}}^{(k)} - \mathbf{x}_{n+1}^{(k)}\right)$$

Evaluate $f_r^{(k)} = f\left(\mathbf{x}_r^{(k)}\right)$. If $f_1^{(k)} \le f_r^{(k)} < f_n^{(r)}$, accept the reflected point and go to 6.

3. **Expansion step:** If $f_r^{(k)} < f_1^{(k)}$, calculate the expansion

$$\mathbf{x}_e^{(k)} = \overline{\mathbf{x}}^{(k)} + 2\left(\mathbf{x}_r^{(k)} - \overline{\mathbf{x}}^{(k)}\right)$$

and evaluate $f_e^{(k)} = f\left(\mathbf{x}_e^{(k)}\right)$. If $f_e^{(k)} < f_r^{(k)}$, accept $\mathbf{x}_e^{(k)}$ and go to 6. Otherwise accept $\mathbf{x}_r^{(k)}$ and go to 6.

4. **Contraction step:** If $f_r^{(k)} \ge f_n^{(k)}$, perform contraction between $\overline{\mathbf{x}}^{(k)}$ and the better of $\mathbf{x}_{n+1}^{(k)}$ and $\mathbf{x}_r^{(k)}$. If $f_r^{(k)} < f_{n+1}^{(k)}$, set

$$\mathbf{x}_c^{(k)} = \overline{\mathbf{x}}^{(k)} + \frac{1}{2}\left(\mathbf{x}_r^{(k)} - \overline{\mathbf{x}}^{(k)}\right)$$

(this is called the outside contraction) and evaluate $f_c^{(k)} = f\left(\mathbf{x}_c^{(k)}\right)$. If $f_c^{(k)} \le f_r^{(k)}$, accept $\mathbf{x}_c^{(k)}$ and go to 6.
If in contrary $f_r^{(k)} \ge f_{n+1}^{(k)}$, set

$$\mathbf{x}_c^{(k)} = \overline{\mathbf{x}}^{(k)} - \frac{1}{2}\left(\overline{\mathbf{x}}^{(k)} - \mathbf{x}_{n+1}^{(k)}\right)$$

(inside contraction) and evaluate $f_c^{(k)}$. If $f_c^{(k)} < f_{n+1}^{(k)}$, accept $\mathbf{x}_c^{(k)}$ and go to 6.

5. **Shrink step:** Move all vertices except the best towards the best vertex, i.e.

$$\mathbf{v}_i^{(k)} = \mathbf{x}_1^{(k)} + \frac{1}{2}\left(\mathbf{x}_i^{(k)} - \mathbf{x}_1^{(k)}\right), i = 2, ..., n+1,$$

and evaluate $f_i^{'(k)} = f\left(\mathbf{v}_i^{(k)}\right), i = 2, ..., n+1$. Accept $\mathbf{v}_i^{(k)}$ as new vertices.

6. **Convergence check:** Check if the convergence criterion is satisfied. If so, terminate the algorithm, otherwise start the next iteration.

Figure 3.3 illustrates possible steps of the algorithm. A possible situation of two iterations when the algorithm is applied is shown in Figure 3.4. The steps allow the shape of the simplex to be changed in every iteration, so the simplex can adapt to the surface of *f*. Far from the minimum the expansion step allows the simplex to move rapidly in the descent direction. When the minimum is inside the simplex, contraction and shrink steps allow vertices to be moved closer to it.

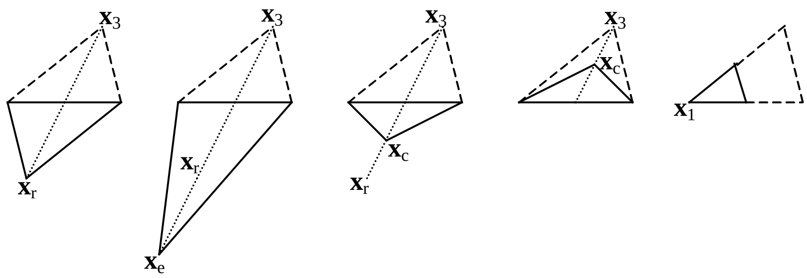



**Figure 3.3:** Possible steps of the simplex algorithm in two dimensions (from left to right): reflection, expansion, outside and inside contraction, and shrink.

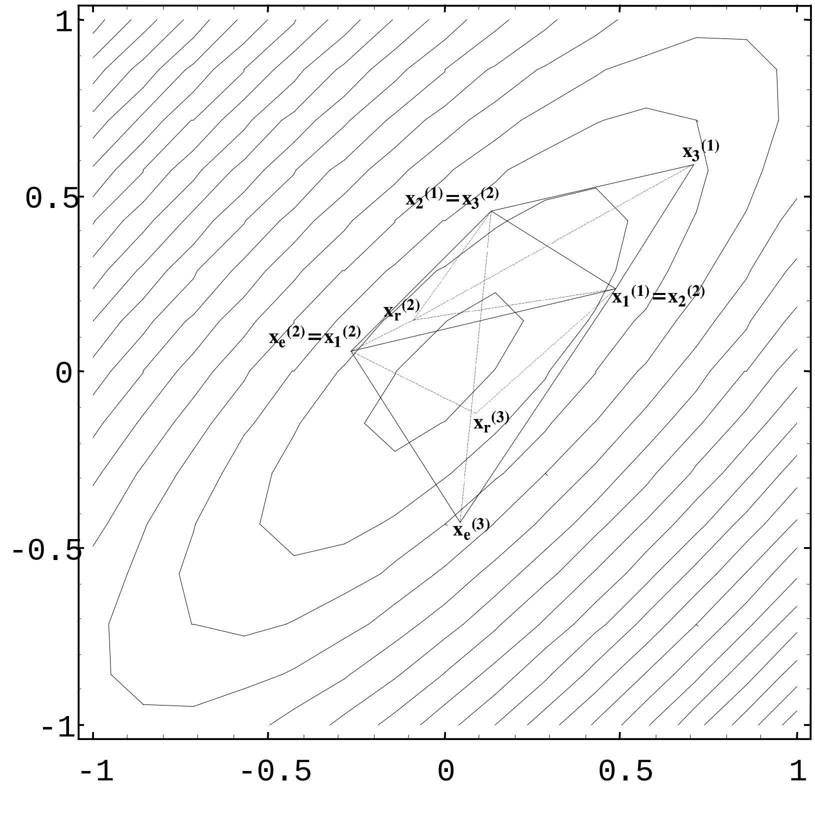



**Figure 3.4:** Example of evolution of the simplex.

There are basically two possibilities for the convergence criterion. Either that function values at vertices must become close enough or the simplex must becomes small enough. It is usually best to impose both criteria, because either of them alone can be misleading.

It must be mentioned that convergence to a local minimum has not been proved for the Nelder-Mead algorithm. Examples have been constructed for which the method does not converge[12]. However, the situations for which this was shown are quite special and unlikely to occur in practice. Another theoretical argument against the algorithm is that it can fail because the simplex collapses into a subspace, so that vectors connecting its vertices become nearly linearly dependent. Investigation of this phenomenon indicates that such behaviour is related to cases when the function to be minimised has highly elongated contours (i.e. ill conditioned Hessian). This is also a problematic situation for other algorithms.

The Nelder-Mead algorithm can be easily adapted for constrained optimisation. One possibility is to add a special penalty term to the objective function, e.g.

$$f^{'}(\mathbf{x}) = f(\mathbf{x}) + f_{n+1}^{(1)} - \sum_{i \in I} c_i(\mathbf{x}) + \sum_{i \in I} \left| c_j(\mathbf{x}) \right|, \tag{3.9}$$

where $f_{n+1}^{(1)}$ is the highest value of $f$ in the vertices of the initial simplex. Since subsequent iterates generate simplices with lower values of the function at vertices, the presence of this term guarantees that whenever a trial point in some iteration violates any constraints, its value is greater than the currently best vertex. The last two sums give a bias towards the feasible region when all vertices are infeasible. The derivative discontinuity of the terms with absolute value should not be problematic since the method is not based on any model, but merely on comparison of function values. A practical implementation is similar to the original algorithm. $f$ is first evaluated at the vertices of the initial simplex and the highest value is stored. Then the additional terms in (3.9) are added to these values, and in subsequent iterates $f$ is replaced by $f'$.

Another variant of the simplex method is the multidirectional search algorithm. Its iteration consists of similar steps to the Nelder-Mead algorithm, except that all vertices but the best one are involved in all operations. There is no shrink step and the contraction step is identical to the shrink step of the Nelder-Mead algorithm. Possible steps are shown in Figure 3.5. The convergence proof exists for this method[12], but in practice it performs much worse than the Nelder-Mead algorithm. This is due to the fact that more function evaluations are performed at each iteration and that the simplex can not be adapted to the local function properties as well as the former algorithm. The shape of the simplex can not change, i.e. angles between it edges remain constant (see Figure 3.5). The multidirectional search algorithm is better suited to parallel processing because $n$ function evaluations can always be performed simultaneously.

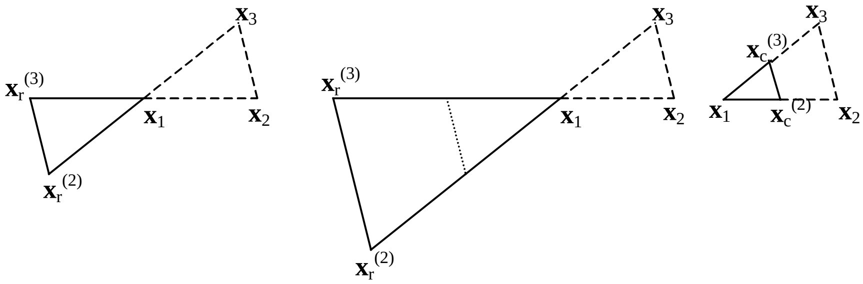



**Figure 3.5:** possible steps in the multidirectional search algorithm: reflection, expansion, and contraction.

## 3.3 Basic Mathematical Background

Construction of optimisation methods described further in this section is based on some model of the objective function and constraints. Such treatment of the problem arises to a large extent from the fact that locally every function can be developed into a Taylor series[30] about any point $x^{'}$:

$$f\left(x^{'}+h\right)=\sum_{n=0}^{\infty}\frac{h^{n}}{n!}f^{(n)}\left(x^{'}\right), \qquad (3.10)$$

where $f^{(n)}(x)=\frac{\partial^{n}}{\partial x^{n}}f(x)$ and $n!=1\cdot 2\cdot 3\cdot ...\cdot n$. This expression itself does not have a significant practical value. A more important fact is that

$$\lim_{n\to\infty}R_{n}(h)=0 \qquad (3.11)$$

and

$$\lim_{h \to 0} R_n(h) = 0\,, \tag{3.12}$$

where

$$R_n(h) = f\left(x^{'} + h\right) - S_n(h) \tag{3.13}$$

and

$$S_n(h) = \sum_{i=0}^{n} \frac{h^n}{n!} f^{(n)}\left(x^{'}\right). \tag{3.14}$$

This means that if we use only a few terms in the Taylor series, the error that we make tends to zero both when we increase the number of terms without limit for some fixed *h*, and when we take a fixed number of terms and decrease the step *h* towards zero. This follows from the result[30]

$$R_n(h) = \frac{h^{n+1}}{(n+1)!} f^{(n+1)}\left(x^{'} + \theta h\right), 0 < \theta < 1\,. \tag{3.15}$$

The above equation also holds if function *f* is only $\mathbb{C}^{n+1}$. This means that every sufficiently smooth function can be locally approximated by a simple polynomial function, which is sometimes more convenient for theoretical treatment than the original function.

A similar development is possible for a function of *n* variables[30]:

$$\begin{gathered} f\left(x_1^{'} + h_i, x_2^{'} + h_2, \ldots, x_n^{'} + h_n\right) = f\left(x_1^{'}, x_2^{'}, \ldots, x_n^{'}\right) + \\ \sum_{i=1}^{m} \frac{1}{i!}\left(h_1 \frac{\partial}{\partial x_1} + h_2 \frac{\partial}{\partial x_2} + \ldots + h_n \frac{\partial}{\partial x_n}\right)^i f\left(x_1, x_2, \ldots, x_n\right) + \,, \\ R_m\left(h_i, h_2, \ldots, h_n\right) \end{gathered} \tag{3.16}$$

where

$$\begin{gathered} R_m\left(h_1, \ldots, h_n\right) = \frac{1}{(n+1)!}\left(h_1 \frac{\partial}{\partial x_1} + \ldots + h_n \frac{\partial}{\partial x_n}\right)^{m+1} \quad . \\ f\left(x_1 + \theta_1 h_1, \ldots, x_n + \theta_n h_n\right), \quad 0 < \theta_i < 1,\ i = 1, \ldots, n \end{gathered} \tag{3.17}$$

In view of the beginning of this discussion, we can consider numerical optimisation as the estimation of a good approximation of the optimisation problem solution on the basis of limited information about the function, usually objective and constraint function values and their derivatives in some discrete set of points. The goal is to achieve satisfactory estimation with as little function and derivative evaluations as possible. Now we can use the fact that general functions can be locally approximated by simpler functions. Besides, functions of simple and known form (e.g. linear or quadratic) are completely described by a finite number of parameters. If we know these parameters, we know (in principle) all about the function, including minimising points.

There exists a clear correspondence between the above considerations and the design of optimisation algorithms. One thing to look at when constructing algorithms is how they perform on simple model functions, and proofs of local convergence properties based to a large extent on properties of the algorithms when applied to such functions[1]-[7].

Heuristically this can be explained by considering a construction of a minimisation algorithm in the following way. Use function values and derivatives in a set of points to build a simple approximation model (e.g. quadratic), which will be updated when new information is obtained. Consider applying an effective minimisation technique adequate for the model function. Since the model approximates the function locally, some information obtained in this way should be applicable to making decision where to set the next iterate when minimising the original function. In the limit, when the iterates approach the minimum, the model function should be increasingly better approximation and minima of the successively built models should be good guesses for the subsequent iterates.

In fact many algorithms perform in a similar manner. The difference is usually that models are not built directly, but the iterates are rather constructed in such a way that the algorithm has certain properties when applied to simple functions, e.g. termination in a finite number of steps. This ensures good local convergence properties. In addition some strategy must be incorporated which ensures global convergence properties of the algorithm. The remainder of this section will consider some mathematical concepts related to this. First, some basic notions will be introduced, and then some important algorithmic properties will be discussed.

### 3.3.1 Basic Notions

Quadratic model functions are the most important in the study of unconstrained minimisation. This is because the Taylor series up to quadratic terms is the simplest Taylor approximation that can have an unconstrained local minimum.

Keeping the terms up to the second order in (3.16) gives the following expression for a second order Taylor approximation:

$$f\left(\mathbf{x}^{'}+\mathbf{h}\right)\approx f\left(\mathbf{x}^{'}\right)+\mathbf{h}^{T}\nabla f\left(\mathbf{x}^{'}\right)+\frac{1}{2}\mathbf{h}^{T}\left[\nabla^{2} f\left(\mathbf{x}^{'}\right)\right]\mathbf{h}, \tag{3.18}$$

where

$$\nabla f\left(\mathbf{x}\right)=\mathbf{g}\left(\mathbf{x}\right)=\left[\frac{\partial f}{\partial x_1},\frac{\partial f}{\partial x_2},...,\frac{\partial f}{\partial x_n}\right]_{\mathbf{x}}^{T}$$

is the function gradient and

$$\nabla^{2} f\left(\mathbf{x}\right)=\mathbf{G}\left(\mathbf{x}\right)=\left(\nabla\nabla^{T}\right)f\left(\mathbf{x}\right)$$

is the Hessian matrix[1] of the function, i.e. matrix of function second derivatives,

$$\left[\nabla^{2} f\left(\mathbf{x}\right)\right]_{ij}=\mathbf{G}_{ij}\left(\mathbf{x}\right)=\frac{\partial^{2} f}{\partial x_i \partial x_j}\left(\mathbf{x}\right). \tag{3.19}$$

Notation $\mathbf{g}(\mathbf{x})=\nabla f\left(\mathbf{x}\right)$ and $\mathbf{G}(\mathbf{x})=\nabla^{2} f\left(\mathbf{x}\right)$ will be used throughout this text.

The idea of a line in $\mathbb{R}^n$ is important. This is a set of points

$$\mathbf{x}=\mathbf{x}(\alpha)=\mathbf{x}^{'}+\alpha\mathbf{s}, \tag{3.20}$$

where $\alpha\in\mathbb{R}$ is a scalar parameter, $\mathbf{x}^{'}$ is any point on the line and $\mathbf{s}$ is the direction of the line. $\mathbf{s}$ can be normalised, e.g. with respect to the Euclidian norm, i.e. $\sum_{i=1}^{n} s_i^2=1$.

It is often useful to study how a function defined in $\mathbb{R}^n$ behaves on a line. For this purpose, we can write

[1] In standard notation Operator $\nabla^2=\Delta=\nabla^T\nabla=\sum_{i=1}^{n}\frac{\partial^2}{\partial x_i^{\,2}}$ is the Laplace operator. However, in most optimisation literature this notation is used for the Hessian operator, and so is also used in this text.

$$f(\alpha) = f(\mathbf{x}(\alpha)) = f(\mathbf{x}' + \alpha \mathbf{s}). \tag{3.21}$$

From this expression we can derive direction derivative of *f*, i.e. derivative of the function along the line:

$$\frac{df(\alpha)}{d\alpha} = \sum_i \frac{dx_i}{d\alpha}\frac{\partial f}{\partial x_i} = \sum_i s_i \frac{\partial f}{\partial x_i} = (\nabla f(\mathbf{x}(\alpha)))^T \mathbf{s}\,.$$

This can be written as

$$\frac{df}{d\alpha} = \frac{df}{d\mathbf{s}} = \nabla f^T \mathbf{s}\,. \tag{3.22}$$

In a similar way the curvature along the line is obtained:

$$\frac{d^2 f(\alpha)}{d\alpha^2} = \frac{d}{d\alpha}\frac{df}{d\alpha} = \frac{d}{d\alpha}\sum_{i=1}^{n} s_i \frac{\partial f}{\partial x_i} =$$
$$\sum_{i=1}^{n} s_i \sum_{j=1}^{n} \frac{dx_j}{d\alpha}\frac{\partial^2 f}{\partial x_i \partial x_j} = \sum_{i=1}^{n}\sum_{j=1}^{n} s_i s_j \frac{\partial^2 f}{\partial x_i \partial x_j}$$

and so

$$\frac{d^2 f}{d\alpha^2} = \frac{d^2 f}{d\mathbf{s}^2} = \mathbf{s}^T \left(\nabla^2 f\right)\mathbf{s}\,. \tag{3.23}$$

A general quadratic function can be written in the form

$$q(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T \mathbf{G}\mathbf{x} + \mathbf{b}^T \mathbf{x} + c\,, \tag{3.24}$$

where **G** is a symmetric constant matrix, $\mathbf{b}^T$ a constant vector and *c* a constant scalar. The gradient of this function is

$$\nabla q(\mathbf{x}) = \mathbf{G}\mathbf{x} + \mathbf{b} \tag{3.25}$$

and the Hessian matrix is

$$\nabla^2 q(\mathbf{x}) = \mathbf{G}\,, \tag{3.26}$$

where the rule for gradient of a vector product

$$\nabla\left(\mathbf{u}^T\mathbf{v}\right) = \left(\nabla\mathbf{u}^T\right)\mathbf{v} + \left(\nabla\mathbf{v}^T\right)\mathbf{u};\ \ \mathbf{u} = \mathbf{u}(\mathbf{x}),\ \mathbf{v} = \mathbf{v}(\mathbf{x})$$

was applied.

We see that a quadratic function has a constant Hessian and its gradient is an affine function of **x**. As a consequence, for any two points the following equation relating the gradient in these points is valid:

$$\nabla q\left(\mathbf{x}''\right) - \nabla q\left(\mathbf{x}'\right) = \mathbf{G}\left(\mathbf{x}'' - \mathbf{x}'\right). \tag{3.27}$$

If **G** is nonsingular, a quadratic function has a unique stationary point ($\nabla q\left(\mathbf{x}'\right) = 0$):

$$\mathbf{x}' = -\mathbf{G}^{-1}\mathbf{b}\ , \tag{3.28}$$

which is also a minimiser if **G** is positive definite (see section 3.3.2). Taylor development about the stationary point gives another form for a quadratic function

$$q(\mathbf{x}) = \frac{1}{2}\left(\mathbf{x} - \mathbf{x}'\right)^T \mathbf{G}\left(\mathbf{x} - \mathbf{x}'\right) + c'\ , \tag{3.29}$$

where $c' = c - \frac{1}{2}\mathbf{x}'^T\mathbf{G}\mathbf{x}'$.

In this text a term linear function[1] will be used for functions of the form

$$l(\mathbf{x}) = \mathbf{a}^T\mathbf{x} + b\ , \tag{3.30}$$

where $\mathbf{a}^T$ is a constant vector and *b* a constant scalar. Such functions have a constant gradient

[1] Mathematically this is an affine function. Linear functions are those[30] for which $f(a\mathbf{x} + b\mathbf{y}) = af(\mathbf{x}) + bf(\mathbf{y})$ for arbitrary **x** and **y** in the definition domain and for arbitrary constants *a* and *b*. Affine functions are those for which $f(\mathbf{x}) - c$ is a linear function, where *c* is some constant. However, in the optimisation literature affine functions are often referred to simply as linear and this is also adopted in this text.

$$\nabla l(\mathbf{x}) = \mathbf{a} \tag{3.31}$$

and zero Hessian

$$\nabla^2 l(\mathbf{x}) = 0\,. \tag{3.32}$$

### 3.3.2 Conditions for Unconstrained Local Minima

Consider first a line through some point $\mathbf{x}^*$, i.e. $\mathbf{x}(\alpha) = \mathbf{x}^* + \alpha\mathbf{s}$. Let us define a scalar function of parameter $\alpha$ using values of function $f$ on this line as $f(\alpha) = f(\mathbf{x}(\alpha))$. If $\mathbf{x}^*$ is a local minimiser of $f(\mathbf{x})$, then 0 is clearly a local minimiser of $f(\alpha)$. From the Taylor expansion for a function of one variable about 0 then it follows[1] that *f* has zero slope and non-negative curvature at $\alpha = 0$. This must be true for any line through $\mathbf{x}^*$, and therefore for any $\mathbf{s}$. From (3.22) and (3.23) it then follows

$$\mathbf{g}^* = 0 \tag{3.33}$$

and

$$\mathbf{s}^T \mathbf{G}^* \mathbf{s} \geq 0 \quad \forall \mathbf{s}\,, \tag{3.34}$$

where the following notation is used: $f^* = f(\mathbf{x}^*)$, $\mathbf{g}(\mathbf{x}) = \nabla f(\mathbf{x})$, $\mathbf{g}^* = \mathbf{g}(\mathbf{x}^*)$, $\mathbf{G}(\mathbf{x}) = \nabla^2 f(\mathbf{x})$, and $\mathbf{G}^* = \mathbf{G}(\mathbf{x}^*)$. This notation will be used through this text, and similarly $f(\mathbf{x}^{(k)}) = f^{(k)}$, etc.

Since (3.33) and (3.34) are implied by assumption that $\mathbf{x}^*$ is a local minimiser of *f*, these are necessary conditions for $\mathbf{x}^*$ being a local minimiser. (3.33) is referred to a first order necessary condition and (3.34) as a second order necessary condition. This condition states that the Hessian matrix is positive semi-definite in a local minimum.

The above necessary conditions are not at the same time sufficient, i.e. these conditions do not imply $\mathbf{x}^*$ to be a local minimiser. Sufficient conditions can be stated in the following way[1]:

**Theorem 3.1:**

Sufficient conditions for a strict and isolated local minimiser $\boldsymbol{x}^*$ of f are that f has a zero gradient and a positive definite Hessian matrix in $\boldsymbol{x}^*$:

$$\mathbf{g}^* = 0 \tag{3.35}$$

*and*

$$\mathbf{s}^T \mathbf{G}^* \mathbf{s} > 0 \quad \forall \mathbf{s} \neq 0 \tag{3.36}$$

There are various ways how to check the condition (3.36). The most important for practical purposes are that[27],[29] **G** is positive definite, the Choleski factors of the $LL^T$ decomposition exist and all diagonal elements $l_{ii}$ are greater than zero, and the same applies for diagonal elements $d_{ii}$ of the $LDL^T$ decomposition. This can be readily verified on those algorithms which solve a system of equation with the system matrix **G** in each iteration, since one of these decompositions is usually applied to solve the system.

Some algorithms do not evaluate the Hessian matrix. These can not verify the sufficient conditions directly. Sometimes these algorithms check only the first order condition or some condition based on the progress during the last few iterations. It can usually be proved that under certain assumptions iterates still converges to a local minimum. Algorithms should definitely have the possibility of termination in a stationary point, which is not a minimum (usually in a saddle point with indefinite Hessian matrix). Some algorithms generate subsequent approximations of the Hessian matrix, which converge to the Hessian in the limit when iterates approach a stationary point. The condition can then be checked indirectly on the approximate Hessian. More details concerning this will be outlined in the description of individual algorithms.

### 3.3.3 Desirable Properties of Algorithms

A desired behaviour of an optimisation algorithm is that iterates move steadily towards the neighbourhood of a local minimser, then converge rapidly to this point and finally that it identifies when the minimiser is determined with a satisfactory accuracy and terminates.

Optimisation algorithms are usually based on some model and on some prototype algorithm. A model is some approximation (not necessarily explicit) of the objective function, which enables a prediction of a local minimiser to be made.

A prototype algorithm refers to the broad strategy of the algorithm. Two basic types are the restricted step approach and the line search approach, described in detail in the subsequent sections. There it will be also pointed out that the ideas of prototype algorithms are usually closely associated with global convergence.

Local convergence properties of an algorithm describe its performance in the neighbourhood of a minimum. If we define the error of the *k*-th iterate

$$\mathbf{h}^{(k)} = \mathbf{x}^{(k)} - \mathbf{x}^{*}, \tag{3.37}$$

it may be possible to state some limit results for $\mathbf{h}^{(k)}$. An algorithm is of course convergent if $\mathbf{h}^{(k)} \to 0$. If a limit

$$\lim_{k\to\infty} \frac{\left\|\mathbf{h}^{(k+1)}\right\|}{\left\|\mathbf{h}^{(k)}\right\|^{p}} = a \tag{3.38}$$

exists where $a > 0$ is some constant, then we say that the order of convergence is *p*. This definition can also be stated in terms of bounds if the limit does not exist: the order of convergence is *p* if

$$\frac{\left\|\mathbf{h}^{(k+1)}\right\|}{\left\|\mathbf{h}^{(k)}\right\|^{p}} \le a \tag{3.39}$$

for some constant $a > 0$ and for each *k* greater than some $k_{lim}$. An important cases are linear or first order convergence

$$\frac{\left\|\mathbf{h}^{(k+1)}\right\|}{\left\|\mathbf{h}^{(k)}\right\|} \le a \tag{3.40}$$

and quadratic or second order convergence

$$\frac{\left\|\mathbf{h}^{(k+1)}\right\|}{\left\|\mathbf{h}^{(k)}\right\|^{2}} \le a\,. \tag{3.41}$$

The constant *a* is called the rate of convergence and must be less than 1 for linear convergence. Linear convergence is only acceptable if the rate of convergence is small. If the order and rate are 1, the convergence is sublinear (slower than all linear convergence). This would be the case if $\left\|\mathbf{h}^{k}\right\| = 1/k$ .

When the order is 1, but the rate constant is 0, the convergence is superlinear (faster than all linear convergence), i.e.

$$\lim_{k\to\infty} \frac{\left\| \mathbf{h}^{(k+1)} \right\|}{\left\| \mathbf{h}^{(k)} \right\|} = 0\,. \tag{3.42}$$

Successful methods for unconstrained minimisation converge superlinearly.

Many methods for unconstrained minimisation are derived from quadratic models. They are designed so that they work well or exactly on a quadratic function. This is partially associated with the discussion of section 3.3.1: since a general function is well approximated by a quadratic function, the quadratic model should imply good local convergence properties. Because the Taylor series about an arbitrary point taken to quadratic terms will agree to a given accuracy with the original function on a greater neighbourhood than the series taken to linear terms, it is preferable to use quadratic information even remote from the minimum.

The quadratic model is most directly used in the Newton method (3.5), which requires the second derivatives. A similar quadratic model is used in restricted step methods. When second derivatives are not available, they can be estimated in various ways. Such quadratic models are used in the quasi-Newton methods.

Newton-like methods (Newton or quasi-Newton) use the Hessian matrix or its approximation in Newton's iteration (3.5). A motivation for this lies in the Dennis-Moré theorem, which states that superlinear convergence can be obtained if and only if the step is asymptotically equal to that of the Newton-Raphson method[1].

The quadratic model is also used by the conjugate direction methods, but in a less direct way. Nonzero vectors $\mathbf{s}^{(1)}, \mathbf{s}^{(2)}, ..., \mathbf{s}^{(n)}$ are conjugate with respect to a positive definite matrix **G**, when

$$\mathbf{s}^{(i)T} \mathbf{G} s^{(j)} = 0 \ \ \forall\, i \neq j\ . \tag{3.43}$$

Optimisation methods, which generate such directions when applied to a quadratic function with Hessian **G**, are called conjugate direction methods. Such methods have the following important property[1]:

**Theorem 3.2:**

A conjugate direction method terminates for a quadratic function in at most n exact line searches, and each $\mathbf{x}^{(k)}$ is a minimiser of that function in the set

$$\left\{ \mathbf{x};\, \mathbf{x} = \mathbf{x}^{(1)} + \sum_{j=1}^{k} \alpha_j \mathbf{s}^{(j)}\ ,\, \alpha_j \in \mathbb{R} \right\} \tag{3.44}$$

The above theorem states that conjugate direction methods have the property of quadratic termination, i.e. they can locate the minimising point of a quadratic function in a known finite number of steps. Many good minimisation algorithms can generate the set of conjugate directions, although it is not possible to state that superlinear convergence implies quadratic termination or vice versa. For example, some successful superlinearly convergent Newton-like methods do not possess this property.

It is useful to further develop the idea of conjugacy in order to gain a better insight in what it implies. We can easily see that $\mathbf{s}^{(i)}$ are linearly independent. If for example $\mathbf{s}^{(j)}$ was a linear combination of some other vectors $\mathbf{s}^{(k)}$, e.g.

$$\mathbf{s}^{(j)} = \sum_{k \neq j} \beta_k \mathbf{s}^{(k)} ,$$

multiplying this with $\mathbf{s}^{(j)T}\mathbf{G}$ would give

$$\mathbf{s}^{(j)T}\mathbf{G}\mathbf{s}^{(j)} = 0 ,$$

which contradicts the positive definiteness of **G**.

We can use vectors $\mathbf{s}^{(j)}$ as basis vectors and write any point as

$$\mathbf{x} = \mathbf{x}^{(1)} + \sum_{i=1}^{n} \alpha_i \mathbf{s}^{(i)} . \tag{3.45}$$

Taking into account this equation,(3.29) and conjugacy, the quadratic function from the theorem can be written as[1]

$$q(\alpha) = \frac{1}{2}\left(\mathbf{x}-\mathbf{x}^*\right)^T \mathbf{G}\left(\mathbf{x}-\mathbf{x}^*\right) = \frac{1}{2}\left(\alpha-\alpha^*\right)\mathbf{S}^T\mathbf{G}\mathbf{S}\left(\alpha-\alpha^*\right). \tag{3.46}$$

We have ignored a constant term in (3.29), which has no influence on further discussion, and written the minimiser $\mathbf{x}^*$ of $q$ as

$$\mathbf{x}^{(*)} = \mathbf{x}^{(1)} + \sum \alpha_i^* \mathbf{s}^{(i)} ,$$

[1] Notation $\alpha = [\alpha_1, \alpha_2, ..., \alpha_n]^T$ is used. Vectors denoted by Greek letters are not typed in bold, but it should be clear from the context when some quantity is vector and when scalar.

and **S** is a matrix whose columns are vectors $\mathbf{s}^{(i)}$. Since $\mathbf{s}^{(i)}$ are conjugate with respect to **G**, the product $\mathbf{S}^{\mathrm{T}}\mathbf{G}\mathbf{S}$ is a diagonal matrix with diagonal elements $d_i$, say, and therefore

$$q(\alpha)=\frac{1}{2}\sum_{i=1}^{n}\left(\alpha_i-\alpha_i^*\right)^2 d_i\ . \tag{3.47}$$

We see that conjugacy implies a coordinate transformation from **x**-space to $\alpha$-space in which **G** is diagonal. Variables in the new system are decoupled from the point of view that $q(\alpha)$ can be minimised by applying successive minimisations in coordinate directions, which results in a minimiser $\alpha^*$ corresponding to $\mathbf{x}^*$ in the **x** space. A conjugate direction method therefore corresponds to the alternating variable method applied in the new coordinate system. Enforcing conjugacy overcomes the basic problem associated with the alternating variable method, i.e. the fact that minimisation along one coordinate direction usually spoils earlier minimisations in other directions, which is the reason for oscillating behaviour of the method shown in Figure 3.1. Since a similar problem is associated with the steepest descent method, conjugacy can be successfully combined with derivative methods.

A side observation is that eigenvectors of **G** are orthogonal vectors conjugate to **G**. A quadratic function is therefore minimised by exact minimisation along all eigenvectors of its Hessian. Construction of the conjugate direction methods will show that there is no need to know eigenvectors of **G** in order to take advantage of conjugacy, but it is possible to construct conjugate directions starting with an arbitrary direction.

Another important issue in optimisation algorithms is when to terminate the algorithm. Since we can not check directly how close to the minimiser the current iterate is, the test can be based on conditions for a local minimum, for example

$$\left\|\mathbf{g}^{(k)}\right\|\le\varepsilon\ , \tag{3.48}$$

where $\varepsilon$ is some tolerance. Sometimes it is not easy to decide what magnitude to choose for $\varepsilon$, since a good decision would require some clue about the curvature in the minimum. The above test is also dependent on the scaling of variables. Another difficulty is that it can terminate in a stationary point that is not a minimum. When second derivative information is available, it should be used to exclude this possibility.

When the algorithm converges rapidly, tests based on differences between iterates can be used, e.g.

$$\left| x_i^{(k)} - x_i^{(k+1)} \right| \le \varepsilon_i \ \forall i \tag{3.49}$$

or

$$f^{(k)} - f^{(k+1)} \le \varepsilon \, . \tag{3.50}$$

These tests rely on a prediction how much at most *f* can be further reduced or **x** approached to the minimum.

The test

$$\frac{1}{2} \mathbf{g}^{(k)T} \mathbf{H}^{k} \mathbf{g}^{(k)}, \tag{3.51}$$

where **H** is the inverse Hessian or its approximation, is also based on predicted change of *f*.

Finally, the possibility of termination when the number of iterations exceeds some user supplied limit is a useful property of every algorithm. Even when good local convergence results exist for a specific algorithm, this is not necessarily a guarantee for good performance in practice. Function evaluation is always subjected to numerical errors and this can especially affect algorithmic performance near the solution where local convergence properties should take effect.

## *3.4 Line Search Subproblem*

### 3.4.1 Features Relevant for Minimisation Algorithms

The line search prototype algorithm sequentially minimises the objective function along straight lines. The structure of the *k*-th iteration is the following:

**Algorithm 3.2:** Iteration of a line search prototype algorithm.

1. Determine a search direction $\mathbf{s}^{(k)}$ according to some model.
2. Find $\alpha^{(k)}$, which minimises $f\left(\mathbf{x}^{(k)} + \alpha \mathbf{s}^{(k)}\right)$ and set $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha^{(k)} \mathbf{s}^{(k)}$.

Finding a minimum of *f* on a line is referred to as the line search subproblem.

In the minimum, slope $df/d\alpha$ must be zero, which from (3.22) gives

$$\nabla f^{(k+1)T}\mathbf{s}^{(k)} = 0. \tag{3.52}$$

If $\mathbf{s}^{(k)}$ satisfies the descent property

$$\mathbf{s}^{(k)T}\mathbf{g}^{(k)} < 0, \tag{3.53}$$

then the function can be reduced in the line search for some $\alpha^{(k)} > 0$ unless $\mathbf{x}^{(k)}$ is a stationary point. A line search method in which search directions satisfy the descent property is called the descent method.

The descent property is closely associated with global convergence and by suitable choice of a line search condition it is possible to incorporate it within a global convergence proof. Merely requiring that *f* is decreased in each iteration certainly does not ensure global convergence. On the other hand, expensive high accuracy line searches do not make sense, especially when the algorithm is far from the solution. Therefore conditions for line search termination must be defined so that they allow low accuracy line searches, but still enforce global convergence.

Let us write $f(\alpha) = f\left(\mathbf{x}^{(k)} + \alpha\,\mathbf{s}^{(k)}\right)$ and let $\overline{\alpha}^{(k)}$ denote the least positive $\alpha$ for which $f(\alpha) = f(0)$ (Figure 3.6). Negligible reductions can occur if we allow the line search to be terminated in points close to 0 or $\overline{\alpha}^{(k)}$. Line search conditions must exclude such points, impose significant reductions of *f*, guarantee that acceptable points always exist and can be determined in a finite number of steps, and should not exclude the minimising point $a^*$ when $f(\alpha)$ is quadratic with positive curvature.

These requirements are satisfied by the Goldstein conditions

$$f(\alpha) \le f(0) + \alpha\rho\, f'(0) \tag{3.54}$$

and

$$f(\alpha) \ge f(0) + \alpha(1-\rho) f'(0), \tag{3.55}$$

where $\rho \in \left(0, \frac{1}{2}\right)$ is some fixed parameter. (3.54) implies

$$f^{(k)} - f^{(k+1)} \geq -\rho\, \mathbf{g}^{(k)T} \delta^{(k)}, \tag{3.56}$$

where $\delta^{(k)} = \mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}$. The condition $\rho < 0.5$ ensures that when $f(\alpha)$ is quadratic, the minimiser is an acceptable point, but this is not true for a general function (Figure 3.6 also shows the case where the minimiser is not an acceptable point). This deficiency is dismissed with the Wolfe-Powell conditions

$$f(\alpha) \leq f(0) + \alpha\rho\, f'(0) \tag{3.57}$$

and

$$f'(\alpha) \geq \sigma\, f'(0), \tag{3.58}$$

where $\rho \in \left(0, \frac{1}{2}\right)$ and $\sigma \in (\rho, 1)$. This implies

$$\mathbf{g}^{(k+1)T} \delta^{(k)} \geq \sigma\, \mathbf{g}^{(k)T} \delta^{(k)}. \tag{3.59}$$

Let $\hat{\alpha} > 0$ be the least positive value for which the graph $f(\alpha)$ intersects the line $f(0) + \alpha\,\rho\, f'(0)$ (point *b* in Figure 3.6). If such a point exists, then an interval of acceptable points for the Wolfe-Powell conditions exists in $(0, \hat{\alpha})$.

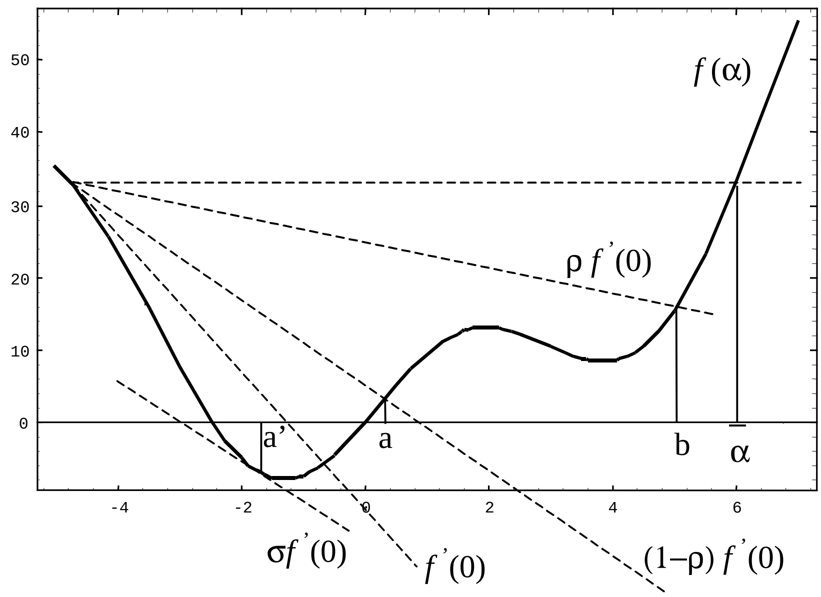



**Figure 3.6:** Line search conditions. [a,b] is the interval of acceptable points for Goldstein conditions, while [a’,b] is an interval of acceptable points for the Wolfe-Powell conditions. Slopes of auxiliary lines are denoted in the figure.

A two-sided test on the slope of *f* can also be used, i.e.

$$|f'(\alpha)| \leq -\sigma\, f'(0) \tag{3.60}$$

together with (3.57). An interval of acceptable points exists in $(0, \hat{\alpha})$ for this test, too, if $\hat{\alpha}$ exists.

To ensure global convergence, line searches must generate sufficient reduction of *f* in each iteration since otherwise non-minimising accumulation points of the sequence of iterates can exist. Fulfillment of this requirement depends on the applied line search criterion, but also on the line search directions. If these become orthogonal to the gradient, than no reduction of *f* can be made. It is advantageous to introduce some criterion to bound directions away from orthogonality to the gradient direction.

The angle criterion is defined as

$$\theta^{(k)} \leq \frac{\pi}{2} - \mu \quad \forall k\,, \tag{3.61}$$

where $\frac{\pi}{2} > \mu > 0$ is a fixed constant and $\theta^{(k)}$ is the angle between the gradient of *f* and the search direction, i.e.

$$\theta^{(k)} = \cos^{-1}\left( \frac{\left|\mathbf{g}^{(k)T}\mathbf{s}^{(k)}\right|}{\left\|\mathbf{g}^{(k)}\right\|_2 \left\|\mathbf{s}^{(k)}\right\|_2} \right). \tag{3.62}$$

The following global convergence theorem then holds[1]:

**Theorem 3.3**:

For a descent method with Wolfe-Powell line search conditions, if $\nabla f$ is uniformly continuous on the level set $\{\mathbf{x};\, f(\mathbf{x}) < f^{(1)}\}$ and if the angle criterion (3.61) holds, then either $f^{(k)} \to -\infty$ or $\mathbf{g}^{(k)} \to 0$.

Considering practical algorithms, the steepest descent method satisfies the angle criterion. Newton-like algorithms (section 3.5) define the search direction as

$$\mathbf{s}^{(k)} = -\mathbf{H}^{(k)}\mathbf{g}^{(k)}. \tag{3.63}$$

If $\mathbf{H}^{(k)}$ is positive definite, then $\mathbf{s}^{(k)}$ is a descent direction. In this case a sufficient condition is that the spectral condition number $\kappa^{(k)}$ of $\mathbf{H}^{(k)}$ is bounded above for every *k*. The spectral condition number of a matrix is the ratio between its largest and smallest eigenvalues ( $\lambda_1/\lambda_n$ ). The relations[29],[33] $\|\mathbf{Hg}\|_2 \leq \lambda_1\|\mathbf{g}\|_2$ and $\mathbf{g}^T\mathbf{HG} \geq \lambda_n\mathbf{g}^T\mathbf{g}$ hold for any matrix **H** and vector **g**, and this implies an estimation

$$\theta^{(k)} \leq \frac{\pi}{2} - \frac{1}{\kappa^{(k)}},$$

which implies the above statement.

A much weaker criterion can be used in place of the angle criterion in Theorem 3.3, namely

$$\sum_k \cos^2\theta^{(k)} = \infty . \qquad (3.64)$$

in this case $\liminf \left\|\mathbf{g}^{(k)}\right\| = 0$, which means that $\mathbf{g}^{(k)} \to 0$ on a subsequence[1].

### 3.4.2 Derivative Based Line Search Algorithms

Line search algorithms[1],[4] ,[13] consist of two parts. The first one is the bracketing stage, which finds a bracket, that is an interval known to contain acceptable points. The second part is the sectioning stage, in which a sequence of brackets whose length tends to zero is generated. It is advantageous to use some interpolation of $f(\alpha)$ in this stage in order to find an acceptable point which is close to the minimiser.

If *f* is not bounded below, it can happen that an interval of acceptable points does not exist. It is therefore advisable to supply a lower bound ( $\bar{f}$ , say) so that all points for which $f(a) \leq \bar{f}$ are considered acceptable. The line search can then be restricted to the interval $(0,\mu]$, where $\mu$ is the point at which the $\rho$-line reaches the level $\bar{f}$ , i.e.

$$\mu = \frac{\bar{f} - f(0)}{\rho\, f'(0)}. \qquad (3.65)$$

In the bracketing stage $\alpha_i$ is set in increasingly large jumps until a bracket $[a_i, b_i]$ on an interval of acceptable points is located. An algorithm suitable when objective function derivatives are available is given below.

**Algorithm 3.3:** Bracketing stage of line search.

Initially $\alpha_0 = 0$ and $\alpha_1$ is given so that $0 < \alpha_1 \le \mu$. For each i the following iteration is repeated:

1. Evaluate $f(\alpha_i)$ and $f'(\alpha_i)$.
2. If $f(\alpha_i) \le \bar{f}$ then terminate line search.
3. If $f(\alpha_i) > f(0) + \alpha_i \rho f'(0)$ or $f(\alpha_i) \ge f(\alpha_{i-1})$ then
   set $\alpha_i = \alpha_{i-1}$ and $b_i = \alpha_i$, terminate bracketing.
4. If $|f'(\alpha_i)| \le -\sigma f'(0)$ then terminate line search.
5. If $f'(\alpha_i) \ge 0$ then
   set $a_i = \alpha_i$ and $b_i = \alpha_{i-1}$, terminate bracketing.
6. If $\mu \le \alpha_{i-1} + 2(\alpha_i - \alpha_{i-1})$ then set $\alpha_{i+1} = \mu$, else
   choose $\alpha_{i+1} \in [\alpha_{i-1} + 2(\alpha_i - \alpha_{i-1}), \min(\mu, \alpha_i + \tau_1(\alpha_i - \alpha_{i-1}))]$

$\tau_1$ is a pre-set factor for which size of the jumps is increased, e.g. 10. Lines 2 to 5 terminate the bracketing stage, if a suitable bracket is located, or the whole line search, if an acceptable point or point for which $f(\alpha_i) \le \bar{f}$ is found. If neither of these situations take place in the current iteration, the search interval is extended (line 6). In this case it is convenient to choose $\alpha_{i+1}$ as a minimiser of some interpolation of $f(\alpha)$, e.g. a cubic polynomial constructed using $f(\alpha_{i-1})$, $f'(\alpha_{i-1})$, $f(\alpha_i)$ and $f'(\alpha_i)$.

If an acceptable point is not found in the bracketing stage, then a bracket $[a_i, b_i]$ is located, which contains an interval of acceptable points with respect to conditions (3.54) and (3.60). The bracket satisfies the following properties:

a. $a_i$ is the current best trial point that satisfies (3.54) (it is possible that $b_i < a_i$, i.e. the bracket is not necessarily written with ordered extreme points).
b. $(b_i - a_i) f'(a_i) < 0$, but $f'(a_i)$ does not satisfy (3.60).
c. either $f(b_i) > f(0) + b_i \rho f'(0)$ or $f(b_i) \ge f(a_i)$ or both.

The sectioning stage is then performed in which the bracket is sectioned so that the length of subsequently generated brackets $\lfloor a_j, b_j \rfloor$ tend to zero. In each iteration a new trial point $\alpha_j$ is chosen and the next bracket is either $\lfloor a_j, \alpha_j \rfloor$, $\lfloor \alpha_j, a_j \rfloor$, or $\lfloor \alpha_j, b_j \rfloor$ or, so that the above described properties remain valid. The algorithm terminates when the current trial point $\alpha_j$ is acceptable with respect to (3.54) and (3.60).

**Algorithm 3.4:** Sectioning stage of the line search

A bracket $[a_0, b_0]$ is first available from the bracketing stage. The *j*-th iteration is then:

1. Choose $\alpha_j \in \lfloor a_j + \tau_2 (b_j - a_j), b_j - \tau_3 (b_j - a_j) \rfloor$,
   evaluate $f(\alpha_j)$ and $f'(\alpha_j)$.
2. If $f(\alpha_j) > f(0) + \rho \alpha_j f'(0)$ or $f(\alpha_j) \geq f(a_j)$, then
   set $a_{j+1} = a_j$ and $b_{j+1} = \alpha_j$, begin the next iteration.
3. If $\left| f'(\alpha_j) \leq -\sigma f'(0) \right|$, then terminate the line search.
4. Set $a_{j+1} = \alpha_j$
   If $(b_j - a_j) f'(\alpha_j) \geq 0$ then set $b_{j+1} = a_j$, else set $b_{j+1} = b_j$.

$\tau_1$ and $\tau_2$ are prescribed constants ($0 \leq \tau_1 \leq \tau_3 \leq \frac{1}{2}$, $\tau_2 \leq \sigma$), which prevent $\alpha_j$ being arbitrarily close to $a_j$ or $b_j$. Then

$$\left| b_{j+1} - a_{j+1} \right| \leq (1 - \tau_2) \left| b_j - a_j \right| \tag{3.66}$$

holds and the interval length therefore tends to zero. Their values can be for example $\tau_2 = \frac{1}{10}$ and $\tau_3 = \frac{1}{2}$. The choice of $\alpha_j$ in line 1 can again be made by minimisation of some interpolation of $f(\alpha)$.

If $\sigma > \rho$ then the algorithm terminates in a finite number of steps with $\alpha_j$ which is an acceptable point with respect to (3.54) and (3.60) [1].

In practice it can happen that the algorithm does not terminate because of numerical errors in the function and its derivatives. It is therefore advisable to terminate if $(a_j - \alpha_j) f'(a_j) \le \varepsilon$, where $\varepsilon$ is some tolerance on *f*, with indication that no further progress can be made in the line search.

It is advantageous if a good initial choice $\alpha_1$ (i.e. close to the line minimiser) can be made before the beginning of the bracketing stage. Some algorithms can give an estimation of likely reduction in the objective function in the line search $\Delta f$. This can be used in the quadratic interpolation of *f*, giving

$$\alpha_1 = -2 \frac{\Delta f}{f'(0)}. \tag{3.67}$$

A suitably safeguarded reduction in the previous iterate can be used as estimation of $\Delta f$, e.g. $\Delta f = \max\left(f^{(k-1)} - f^{(k)}, 10\varepsilon\right)$, where $\varepsilon$ is the same tolerance as above. In the Newton-like methods (section 3.5) the choice $\alpha_1 = 1$ is significant in giving rapid convergence. Therefore the choice

$$\alpha_1 = \min\left(1, -2 \frac{\Delta f}{f'(0)}\right) \tag{3.68}$$

is usually made. The choice $\alpha_1 = 1$ is always made when iterates come close to the minimiser, if the method is superlinearly convergent.

### 3.4.3 Non-derivative Line Search Algorithms

If the line search is performed in an algorithm where derivatives are evaluated numerically by finite difference approximation, then $f'(\alpha)$ can also be approximated numerically and the line search strategy described in the previous section can be used. There also exist methods, which perform line searches, but do not use derivative information (e.g. direction set methods). In these methods non-derivative line search algorithms are used.

In the absence of derivative information, the criteria for acceptable points described in the previous section can not be applied. None-derivative line search methods rely on the fact that if we have three points *a*, *b* and *c* such that

$$a < b < c \wedge f(b) < f(a) \wedge f(b) < f(c), \tag{3.69}$$

then $f$ has at least one local minimum in the interval $[a,c]$. It is then possible to section this interval, keeping three points, which satisfy the above relation through iterates.

The non-derivative line search also consists of a bracketing and sectioning stage. In the bracketing stage a triple of points $\{a_1, b_1, c_1\}$ that satisfy (3.69) is found in the following way:

**Algorithm 3.5:** Sectioning stage of a non-derivative line search.

Given $\alpha_0 = 0$, $f_0 = f(\alpha_0)$, $\alpha_1$ and $f_1 = f(\alpha_1)$ such that $f_1 < f_0$, the *i*-th iteration is as follows:

1. Set $\alpha_i = \alpha_{i-1} + \zeta_1(\alpha_i - \alpha_{i-1})$, evaluate $f_i = f(\alpha_i)$.
2. If $f_i < \bar{f}$, accept $\alpha_i$ and terminate the line search.
3. If $f_i > f_{i-1}$, set $a_1 = \alpha_{i-2}$, $b_1 = \alpha_{i-1}$ and $c_1 = \alpha_i$, terminate bracketing.

Again $\bar{f}$ is some user defined value, so that the point with value of $f$ lesser than $\bar{f}$ is automatically accepted as the line minimum. $\zeta_1 > 1$ is some factor which ensures that trial intervals are progressively enlarged, e.g. $\zeta_2 = 2$.

The algorithm assumes that initially $f_1 < f_0$. If $f_1 > f_0$, the algorithm can simply change $\alpha_0$ and $\alpha_1$ before the first iteration begins. The other possibility is to try with $\alpha' = -\alpha_1$. If $f' = f(\alpha') > f_0$, we can immediately terminate bracketing with $a_1 = \alpha'$, $b_1 = \alpha_0 = 0$ and $c_0 = \alpha_1$, otherwise we change $\alpha_0$ and $\alpha_1$ and set $\alpha_3 = \alpha'$.

The sectioning stage (Figure 3.7) is described below.

**Algorithm 3.6:** Sectioning stage of a non-derivative line search.

Given a triple of points $\{a_1, b_1, c_1\}$, which satisfy (3.69), the *j*-th iteration is as follows:

1. Choose $\alpha_j \in \left\{ \begin{matrix} [a_j + \zeta_2(b_j - a_j), b_j - \zeta_2(b_j - a_j)] \ , \\ [b_j + \zeta_2(c_j - b_j), c_j - \zeta_2(c_j - b_j)] \end{matrix} \right\}$, evaluate $f^{(j)} = (\alpha_j)$.
2. If $f_j < \bar{f}$, then accept $\alpha_j$ and terminate the line search.
3. If $\alpha_j \in [b_{j-1}, c_{j-1}]$, interchange $a_j$ and $c_j$.

4. If $\alpha_j < b_j$, set $a_{j+1} = a_j$, $b_{j+1} = \alpha_j$ and $c_{j+1} = b_j$, go to line 6.
5. If $\alpha_j \geq b_j$, set $a_{j+1} = \alpha_j$, $b_{j+1} = b_j$ and $c_{j+1} = c_j$, go to line 6.
6. Check convergence criterion. If the criterion is satisfied, then terminate with $b_j$ as the line minimum.

A triple of points $\{a_j, b_j, c_j\}$ satisfies the condition (3.69) through all iterations. Parameter $\zeta_2$ ($0 < \zeta_2 < \frac{1}{2}$) ensures that the lengths of successive brackets $[a_j, c_j]$ tend to zero. $\zeta_2 = 0.1$ is a reasonable choice.

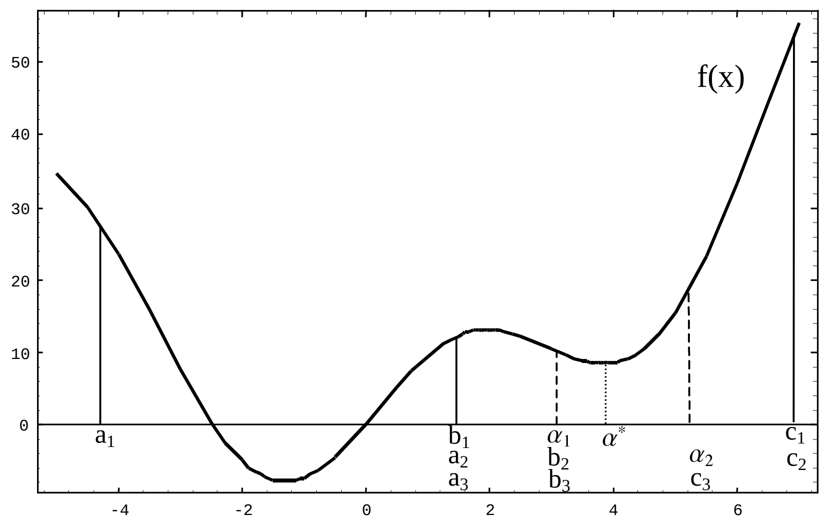



**Figure 3.7:** Sectioning stage of the non-derivative line search in the case when interpolation is not applied.

$\alpha_j$ can be chosen as the minimiser of a quadratic interpolation of *f*, i.e. a parabola through the points $(a_j, f(a_j))$, $(b_j, f(b_j))$ and $(c_j, f(c_j))$. The formula of such parabola is

$$p(\alpha) = f(a)\frac{(x-b)(x-c)}{(a-b)(a-c)} + f(b)\frac{(x-a)(x-c)}{(b-a)(b-c)} + f(c)\frac{(x-a)(x-b)}{(c-a)(c-b)}$$

and its minimiser is

$$\alpha_{\min} = b + \frac{1}{2}\frac{(b-a)^2[f(b)-f(c)]-(b-c)^2[f(b)-f(a)]}{(b-a)[f(b)-f(c)]-(b-c)[f(b)-f(a)]}, \tag{3.70}$$

where indices *j* have been omitted.

If $a_{\min}$ is an element of the acceptable interval in line 1 of the above algorithm, then $\alpha_j = \alpha_{\min}$ is set. Otherwise the longer $\alpha_j$ is obtained by sectioning the longer of both intervals. If the longer interval is $[a_j, b_j]$ , then

$$\alpha_j = a_j + \frac{\tau}{1+\tau}\left(b_j - a_j\right), \tag{3.71}$$

where $\tau$ is some fixed parameter such that $0.5 < \tau < 1 - \zeta_2$ .

A common choice is the golden section ratio $\tau = \left(1+\sqrt{5}\right)/2 \approx 1.618$ . It follows from the request that when a new point $\alpha_j$ is inserted in $[a_j, b_j]$ , both potentially new brackets have the same interval length ratio $\frac{c_{j+1} - b_{j+1}}{b_{j+1} - a_{j+1}}$ (i.e. $1/\tau$ ), which then gives $\tau = \left(1+\sqrt{5}\right)/2$ . This request can be applied when pure bracketing takes place and also the initial triple has the same interval length ratio.

The convergence can be checked either on function values, e.g.

$$\max\{(f(a_j) - f(b_j)), (f(c_j) - f(b_j))\} < \varepsilon \tag{3.72}$$

or on interval length, i.e.

$$\left|c_j - a_j\right| < \varepsilon \, . \tag{3.73}$$

## 3.5 Newton-like Methods

Newton-like methods are based on a quadratic model, more exactly on the second-order Taylor approximation (equation ) of $f(\mathbf{x})$ about $\mathbf{x}^{(k)}$. The basic ideas around this were explained in sections 3.1.2 and 3.3 and will be further developed in this section.

In section 3.1.2 Newton's method was derived from the solution of the system of equations

$$\nabla \mathbf{g}(\mathbf{x}) = 0 ,$$

where the iteration formula was derived from the first order Taylor's approximation of $\mathbf{g}(\mathbf{x})$, giving iteration formula (3.5). Two problems related with direct application of the method were mentioned there, i.e. lack of global convergence properties and explicit use of the second order derivative information regarding the objective functions. Some general ideas on how to overcome these problems were outlined in section 3.3 and will be further developed in this section for algorithms, which in principle stick with the basic idea of Newton's method.

In order to take over and develop the ideas given in section 3.3, let us start from the second order Taylor approximation of *f* itself, developed around the current iterate:

$$f\left(\mathbf{x}^{(k)} + \delta\right) \approx q^{(k)}(\delta) = f^{(k)} + \mathbf{g}^{(k)T}\delta + \frac{1}{2}\delta^T \mathbf{G}^{(k)}\delta . \tag{3.74}$$

Using the results of section 3.3, the stationary point of this approximation is a solution of a linear system of equations

$$\mathbf{G}^{(k)}\delta = -\mathbf{g}^{(k)} . \tag{3.75}$$

It is unique if $\mathbf{G}^{(k)}$ is non-singular and corresponds to a minimiser if $\mathbf{G}^{(k)}$ is positive definite. Newton's method is obtained by considering $\delta^{(k)}$ as solution of the above equation and setting the next guess to $\mathbf{x}^{(k)} + \delta^{(k)}$. The *k*-th iteration of Newton's method is then

1. Solve (3.75) for $\delta^{(k)}$,
2. Set $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \delta^{(k)}$.

This is well defined as a minimisation method only if $\mathbf{G}^{(k)}$ is positive definite in each iteration, and this can be readily checked if for example LDLT decomposition is used for solution of (3.75). However, even if $\mathbf{G}^{(k)}$ is positive definite, the method may not converge from any initial guess, and it can happen that $\left\{f^{(k)}\right\}$ do not even decrease.

Line search can be used to eliminate this problem. The solution of (3.75) then defines merely the search direction, rather than correction $\delta^{(k)}$. The correction is then obtained by line minimisation using algorithms described in section 3.4.2, and such a method is called Newton's method with line search. The direction of search is

$$\mathbf{s}^{(k)} = -\mathbf{G}^{(k)-1}\mathbf{g}^{(k)}. \tag{3.76}$$

If $\mathbf{G}^{(k)}$ and hence its inverse are positive definite, this defines a descent direction. If $\mathbf{G}^{(k)}$ is not positive definite, it may be possible to make a line search in $\pm\mathbf{s}^{(k)}$, but the relevance of searching in $-\mathbf{s}^{(k)}$ is questionable because this is not a direction towards a stationary point of $q(\delta)$. Furthermore, the method fails if any $\mathbf{x}^{(k)}$ is a saddle point of *f*. This gives $\mathbf{s}^{(k)} = 0$, although $\mathbf{x}^{(k)}$ is not a minimiser of *f*.

One possibility of how to overcome this problem is to switch to the steepest descent direction whenever $\mathbf{G}^{(k)}$ is not positive definite. This can be done in conjunction with the angle criterion (3.61) to achieve global convergence.

Minimising in the steepest descent directions can lead to undesired oscillatory behaviour where small reductions of *f* are achieved in each iteration. This happens because second order model information is ignored, as shown in section 3.3.3. The alternative approach is to switch between the Newton and steepest descent direction in a continuous way, controlling the influence of both through some adaptive weighting parameter. This can be achieved by adding a multiple of the unit matrix to $\mathbf{G}^{(k)}$ so that the search direction is defined as

$$\left(\mathbf{G}^{(k)} + \nu\,\mathrm{I}\right)\mathbf{s}^{(k)} = -\mathbf{g}^{(k)}. \tag{3.77}$$

Parameter $\nu$ is chosen so that $\mathbf{G}^{(k)} + \nu\,\mathrm{I}$ is positive definite. If $\mathbf{G}^{(k)}$ is close to positive definite, a small $\nu$ is sufficient and the method therefore uses the curvature information to a large extent. When large values of $\nu$ are necessary, the search directions becomes similar to the steepest descent direction $-\mathbf{g}^{(k)}$.

This method still fails when some $\mathbf{x}^{(k)}$ is a saddle point, and the second order information is not used in the best possible way. Further modification of the method incorporates the restricted step approach in which minimisation of the model quadratic function subjected to length restriction is minimised. This is a subject of section 3.7.

### 3.5.1 Quasi-Newton Methods

In the Newton-like methods discussed so far the second derivatives of $f$ are necessary and substantial problems arise when the Hessian matrix of the function is not positive definite. The second derivatives of $\mathbf{G}^{(k)}$ can be evaluated by numerical differentiation of the gradient vector. In most cases it is advisable that after this operation $\mathbf{G}$ is made symmetric by $\mathbf{G} = \frac{1}{2}\left(\overline{\mathbf{G}} + \overline{\mathbf{G}}^T\right)$, where $\overline{\mathbf{G}}$ is the finite difference approximation of the Hessian matrix. However, evaluation of $\mathbf{G}$ can be unstable in the presence of numerical noise, and it is also expensive, because quadratic model information built in the previous iterates is disregarded.

The above mentioned problems are avoided in so called quasi-Newton methods. In these methods $\mathbf{G}^{(k)-1}$ are approximated by symmetric matrices $\mathbf{H}^{(k)}$, which are updated from iteration to iteration using the most recently obtained information. Analogous to Newton's method with line search, line minimisations are performed in each iteration in the direction

$$\mathbf{s}^{(k)} = -\mathbf{H}^{(k)}\mathbf{g}^{(k)}. \tag{3.78}$$

By updating approximate $\mathbf{G}^{-1}$ rather than $\mathbf{G}$, a system of equations is avoided and the search direction is obtained simply by multiplication of the gradient vector by a matrix. An outline of the algorithm is given below:

**Algorithm 3.7:** General quasi-Newton algorithm.

Given a positive definite matrix $\mathbf{H}^{(1)}$, the $k$-th iteration is:

1. Calculate $\mathbf{s}^{(k)}$ according to (3.78).
2. Minimise $f$ along $\mathbf{s}^{(k)}$, set $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha^{(k)}\mathbf{s}^{(k)}$, where $\alpha^{(k)}$ is a line minimum.
3. Update $\mathbf{H}^{(k)}$ to obtain $\mathbf{H}^{(k+1)}$.

If no second derivative information is available at the beginning, $\mathbf{H}^{(1)}$ can be any positive definite matrix, e.g. $\mathbf{H}^{(1)} = \mathbf{I}$. The line search strategy described in section 3.4.2 can be used in line 2. If $\mathbf{H}^{(k)}$ is positive definite, the search directions are descent. This is desirable and the most important are those quasi-Newton methods, which maintain positive definiteness of $\mathbf{H}^{(k)}$.

The updating formula should explicitly use only first derivative information. Repeated updating should change arbitrary $\mathbf{H}^{(1)}$ to a close approximation of $\mathbf{G}^{(k)-1}$ The updating formula is therefore an attempt to augment the current $\mathbf{H}^{(k)}$ with second derivative information gained in the current iteration, i.e. by evaluation of $f$ and $\nabla f$ at two distinct points. In this context equation (3.27), which relates the Hessian matrix of a quadratic function with its gradient in two distinct points, requires attention.

Let us write

$$\delta^{(k)} = \mathbf{x}^{(k+1)} - \mathbf{x}^{(k)} \tag{3.79}$$

and

$$\gamma^{(k)} = \mathbf{g}^{(k+1)} - \mathbf{g}^{(k)}. \tag{3.80}$$

Using the Taylor series of **g** about $\mathbf{x}^{(k)}$ gives a relationship similar to (3.27), i.e.

$$\gamma^{(k)} = \mathbf{G}^{(k)}\delta^{(k)} + o\left(\left\|\delta^{(k)}\right\|\right). \tag{3.81}$$

The updating formula should therefore correct $\mathbf{H}^{(k+1)}$ so that the above relation would hold approximately with $\mathbf{H}^{(k+1)-1}$ in place of $\mathbf{G}^{(k)}$. This gives the so called quasi-Newton condition, in which the updating formula must satisfy

$$\mathbf{H}^{(k+1)}\gamma^{(k)} = \delta^{(k)}. \tag{3.82}$$

Since this condition gives only one equation, it does not uniquely define the updating formula and permits various ways of updating **H**. One possibility is to add a symmetric rank one matrix to $\mathbf{H}^{(k)}$, i.e.

$$\mathbf{H}^{(k+1)} = \mathbf{H}^{(k)} + \mathbf{u}\mathbf{u}^T. \tag{3.83}$$

Substituting this into (3.82) gives

$$\mathbf{H}^{(k)}\gamma^{(k)} + \mathbf{u}\mathbf{u}^T\gamma^{(k)} = \delta^{(k)}. \tag{3.84}$$

Since $\mathbf{u}^{(T)}\gamma^{(k)}$ is a scalar, matrix multiplication is associative and multiplication with a scalar is commutative, **u** must be proportional to $\delta^{(k)} - \mathbf{H}^{(k)}\gamma^{(k)}$. Writing

$$\mathbf{u} = a\left(\delta^{(k)} - \mathbf{H}^{(k)}\gamma^{(k)}\right)$$

and inserting this into (3.84) gives $a = 1 \Big/ \sqrt{\left(\delta^{(k)} - \mathbf{H}^{(k)}\gamma^{(k)}\right)^T \gamma^{(k)}}$ and therefore

$$\mathbf{H}^{(k+1)} = \mathbf{H}^{(k)} + \frac{\left(\delta^{(k)} - \mathbf{H}^{(k)}\gamma^{(k)}\right)\left(\delta^{(k)} - \mathbf{H}^{(k)}\gamma^{(k)}\right)^{(T)}}{\left(\delta^{(k)} - \mathbf{H}^{(k)}\gamma^{(k)}\right)^{(T)}\gamma^{(k)}}. \qquad (3.85)$$

This formula is called the rank one updating formula according to the above derivation.

For a quadratic function with positive definite Hessian the rank one method terminates in at most $n$+1 steps with $\mathbf{H}^{(n+1)} = \mathbf{G}^{-1}$, provided that $\delta^{(1)}, ..., \delta^{(n)}$ are independent and that the method is well defined[1]. The proof does not require exact line searches. Also the so called hereditary property can be established, i.e.

$$\mathbf{H}^{(i)}\gamma^{(j)} = \delta^{(j)}, \quad j = 1, 2, ..., i-1. \qquad (3.86)$$

A disadvantage is that in general the formula does not maintain positive definiteness of $\mathbf{H}^{(k)}$ and the dominator in (3.85) can become zero.

Better formulas can be obtained by allowing the correction to be of rank two. This can always be written[29],[31] as

$$\mathbf{H}^{(k+1)} = \mathbf{H}^{(k)} + \mathbf{u}\mathbf{u}^T + \mathbf{v}\mathbf{v}^T. \qquad (3.87)$$

Using this in the quasi-Newton condition gives

$$\delta^{(k)} = \mathbf{H}^{(k)}\gamma^{(k)} + \mathbf{u}\mathbf{u}^T\gamma^{(k)} + \mathbf{v}\mathbf{v}^T\gamma^{(k)}. \qquad (3.88)$$

**u** and **v** can not be determined uniquely. A straightforward way of satisfying the above equation is to set **u** proportional to $\delta^{(k)}$ and **v** proportional to $\mathbf{H}^{(k)}\gamma^{(k)}$. By solution of the equation separately for both groups of proportional vectors the Davidon – Fletcher - Powell or DFP updating formula is obtained:

$$\mathbf{H}_{DFP}^{(k+1)} = \mathbf{H} + \frac{\delta\delta^T}{\delta^T\gamma} - \frac{\mathbf{H}\gamma\gamma^T\mathbf{H}}{\gamma^T\mathbf{H}\gamma}. \qquad (3.89)$$

Indices $k$ have been omitted for the sake of simplicity (this approach will be adopted through this section) and the symmetry of **H** is used.

Another rank two updating formula can be obtained by considering updating and approximating $\mathbf{G}$ instead of $\mathbf{G}^{-1}$. Let us write $\mathbf{B}^{(k)} = \mathbf{H}^{(k)-1}$ and consider updating $\mathbf{B}^{(k)}$ in a similar way as $\mathbf{H}^{(k)}$ was updated according to the DFP formula. We require that the quasi-Newton condition (3.82) is preserved. This was true for the DFP formula, but now we are updating inverse of $\mathbf{H}^{(k)}$, therefore, according to (3.82), $\gamma^{(k)}$ and $\delta^{(k)}$ must be interchanged. This gives the formula

$$\mathbf{B}_{BFGS}^{(k+1)} = \mathbf{B} + \frac{\gamma\gamma^T}{\gamma^T\delta} - \frac{\mathbf{B}\delta\delta^T\mathbf{B}}{\delta^T\mathbf{B}\delta}. \tag{3.90}$$

We however still want to update $\mathbf{H}^{(k)}$ rather than $\mathbf{B}^{(k)}$, because a solution of system of equations is in this way avoided in the quasi-Newton iteration. The following updating formula satisfies $\mathbf{B}_{BFGS}^{(k+1)}\mathbf{H}_{BFGS}^{(k+1)} = \mathbf{I}$:

$$\mathbf{H}_{BFGS}^{(k+1)} = \mathbf{H} + \left(1 + \frac{\gamma^T\mathbf{H}\gamma}{\delta^T\gamma}\right)\frac{\delta\delta^T}{\delta^T\gamma} - \left(\frac{\delta\gamma^T\mathbf{H} + \mathbf{H}\gamma\delta^T}{\delta^T\gamma}\right). \tag{3.91}$$

This is called the Broyden – Fletcher – Goldfarb – Shanno or BFGS updating formula.

The BFGS and the DFP formula are said to be dual or complementary because the expressions for $\mathbf{B}^{(k+1)}$ and $\mathbf{H}^{(k+1)}$ in one are obtained by interchanging $\mathbf{B} \leftrightarrow \mathbf{H}$ and $\gamma \leftrightarrow \delta$ in the other. Such duality transformation preserves the quasi-Newton condition. The rank one formula is self-dual.

The DFP and BFGS updating formula can be combined to obtain the so called Broyden one-parameter family of rank two formulae:

$$\mathbf{H}_{\phi}^{k+1} = (1-\phi)\mathbf{H}_{DFP}^{(k+1)} + \phi\,\mathbf{H}_{BFGS}^{(k+1)}. \tag{3.92}$$

This family includes the DFP and BFGS and also rank 1 formula. The quasi-Newton method with a Broyden family updating formula has the following properties[1]:

1. For a quadratic function with exact line searches:
- The method terminates in at most *n* iterations with $\mathbf{H}^{(n+1)} = \mathbf{G}^{-1}$.
- Previous quasi-Newton conditions are preserved (hereditary property (3. 86)).
- Conjugate directions are generated, and conjugate gradients when $\mathbf{H}^{(0)} = \mathbf{I}$.
2. For general functions:
- The method has superlinear order of convergence.
- The method is globally convergent for strictly convex functions if exact line searches are performed.

The Broyden family updates maintain positive definiteness of $\mathbf{H}_{\phi}^{(k+1)}$ for $\phi \geq 0$.

Global convergence has also been proved for the BFGS method with inexact line searches subject to conditions (3.56) and (3.59), applied to a convex objective function[1]. The BFGS method with inexact line searches converges superlinearly if $\mathbf{G}^{(*)}$ is positive definite.

The BFGS method also shows good performance in numerical experiments. The method is not sensitive to exactness of line searches, in fact it is a generally accepted opinion that inexact line searches are more efficient with the BFGS method than near exact line searches. The contemporary optimisation literature[1],[4] suggests the BFGS method as preferable choice for general unconstrained optimisation based on a line search prototype algorithm.

### 3.5.2 Invariance Properties

It is important to study how optimisation algorithms perform when affine transformation of variables is made, i.e.

$$\mathbf{y} = \mathbf{A}\mathbf{x} + \mathbf{a}, \tag{3.93}$$

where **A** is nonsingular. This is a one-to-one mapping with inverse transformation

$$\mathbf{x} = \mathbf{A}^{-1}(\mathbf{y} - \mathbf{a}).$$

*f* can be evaluated either in **x** space (denoted by $f_x(\mathbf{x})$) or in **y** space (denoted by $f_y(\mathbf{y}) = f_x\left(\mathbf{A}^{-1}(\mathbf{y} - \mathbf{a})\right)$).

Applying the chain rule for derivation in **x** space gives

$$\frac{\partial}{\partial x_i} = \sum_{k=1}^{n} \frac{\partial y_k}{\partial x_i} \frac{\partial}{\partial y_k} = \sum_{k=1}^{n} \left(\mathbf{A}^T\right)_{ik} \frac{\partial}{\partial y_k}, \tag{3.94}$$

therefore $\nabla_x = \mathbf{A}^T \nabla_y$ and so

$$\mathbf{g}_x = \mathbf{A}^T \mathbf{g}_y. \tag{3.95}$$

Applying the gradient operator to the above equation then gives $\nabla_x \mathbf{g}_x^{\ T} = \mathbf{A}^T \nabla_y \mathbf{g}_y^{\ T} \mathbf{A}$, i.e.

$$\mathbf{G}_x = \mathbf{A}^T \mathbf{G}_y \mathbf{A} . \tag{3.96}$$

The notation $\mathbf{g}_y = \nabla_y f_y$, etc. was used, so that for example

$$\left[\mathbf{G}_y\right]_{ij} = \frac{\partial^2 f_y}{\partial y_i \partial y_j} .$$

The following theorem[1] applies to Newton-like methods:

**Theorem 3.4:**

If $\mathbf{H}^{(k)}$ transforms under transformation (3.93) as

$$\mathbf{H}_x^{(k)} = \mathbf{A}^{-1} \mathbf{H}_y^{(k)} \mathbf{A}^{-T} \quad \forall k , \tag{3.97}$$

then a Newton-like method with fixed step $\alpha^{(k)}$ is invariant under the transformation (3.93). A method is also invariant if $\alpha^{(k)}$ is determined by tests on $f^{(k)}$, $\mathbf{g}^{(k)T} \mathbf{s}^{(k)}$ or other invariant scalars.

Transformation (3.97) in the above theorem is obtained by inverting (3.96), since $\mathbf{H}^{(k)}$ approximate $\mathbf{G}^{(k)}$ in the quasi-Newton methods.

We see that the steepest descent method (treated as quasi-Newton method with $\mathbf{H}^{(k)} = \mathbf{I}$) is not invariant under transformation (3.93) because $\mathbf{I}$ does not transform correctly. Modified Newton methods are also not invariant because $\mathbf{G} + \nu \mathbf{I}$ does not transform correctly when $\nu > 0$.

For a quasi-Newton method to be invariant, $\mathbf{H}^{(1)}$ must be chosen so as to transform correctly (as (3.97)) and the updating formula must preserve the transformation property (3.97). Therefore, if $\mathbf{H}^{(1)} = \mathbf{I}$ is chosen, then invariance does not hold. $\mathbf{H}^{(1)} = \mathbf{G}\left(\mathbf{x}^{(1)}\right)^{-1}$ transforms correctly and therefore this choice does not affect invariance.

In order to show that a specific updating formula preserves the transformation property (3.97), we must show that $\mathbf{AH}_x^{(k)}\mathbf{A}^T = \mathbf{H}_y^{(k)}$ (which is (3.97) pre-multiplied by **A** and post-multiplied by $\mathbf{A}^T$) which implies $\mathbf{AH}_x^{(k+1)}\mathbf{A}^T = \mathbf{H}_y^{(k+1)}$. Let us do this for the DFP formula

$$\mathbf{H}_x^{(k+1)} = \mathbf{H}_x + \frac{\delta_x {\delta_x}^T}{{\delta_x}^T \gamma_x} - \frac{\mathbf{H}_x \gamma_x {\gamma_x}^T \mathbf{H}_x}{{\gamma_x}^T \mathbf{H}_x \gamma_x}. \tag{3.98}$$

We will pre-multiply the above equation by **A** and post-multiply it by $\mathbf{A}^T$ and use relations $\mathbf{A}\delta_x = \delta_y$ following from (3.93) and $\gamma_x = \mathbf{A}^T\gamma_y$ following from (3.95). We will consider individual terms in equation (3.98).

The first term on the right-hand side of (3.98) gives, after multiplication, $\mathbf{H}_y^{(k)}$ by assumption. Consider then the denominator of the second term:

$${\delta_x}^T\gamma_x = {\delta_x}^T\mathbf{A}^T\mathbf{A}^{-T}\gamma_x = \left(\mathbf{A}\delta_x\right)^T\gamma_y = {\delta_y}^T\gamma_y,$$

the denominator is invariant. The numerator after multiplication gives

$$\mathbf{A}\delta_x{\delta_x}^T\mathbf{A}^T = \delta_y{\delta_y}^T,$$

so the second term transforms correctly. Consider the denominator of the third term:

$${\gamma_x}^T\mathbf{H}_x\gamma_x = {\gamma_x}^T\mathbf{A}^{-1}\mathbf{AH}_x\mathbf{A}^T\mathbf{A}^{-T}\gamma_x = \left(\mathbf{A}^{-T}\gamma_x\right)^T\mathbf{H}_y\mathbf{A}^{-T}\gamma_x = {\gamma_y}^T\mathbf{H}_y\gamma_y,$$

the denominator is invariant under transformation. The numerator after multiplication is

$$\mathbf{AH}_x\gamma_x{\gamma_x}^T\mathbf{H}_x\mathbf{A}^T = \mathbf{AH}_x\mathbf{A}^T\mathbf{A}^{-T}\gamma_x{\gamma_x}^T\mathbf{A}^{-1}\mathbf{AH}_x\mathbf{A}^T = \mathbf{H}_y\gamma_y\left(\mathbf{A}^{-T}\gamma_x\right)^T\mathbf{H}_y = \mathbf{H}_y\gamma_y{\gamma_y}^T\mathbf{H}_y,$$

so the third term is also transformed correctly. $\mathbf{AH}_x^{(k+1)}\mathbf{A}^T = \mathbf{H}_y^{(k+1)}$ is valid since

$$\mathbf{AH}_x^{(k+1)}\mathbf{A}^T = \mathbf{H}_y + \frac{\delta_y{\delta_y}^T}{{\delta_y}^T\gamma_y} - \frac{\mathbf{H}_y\gamma_y{\gamma_y}^T\mathbf{H}_y}{{\gamma_y}^T\mathbf{H}_y\gamma_y}$$

and this is the DFP formula in the **y** space.

Similarly the preservation of (3.97) can be proved for all updating formulas in which the correction is a sum of rank one terms constructed from vectors $\delta$ and $\mathbf{H}\gamma$, multiplied by invariant scalars. Such versions are the BFGS formula and hence all Broyden family formulas.

The Broyden family (including BFGS and DFP) algorithms are therefore invariant under the affine transformation of variables (3.93), provided that $\mathbf{H}^{(1)}$ is chosen so as to transform correctly, i.e. as (3.97). However, even if $\mathbf{H}^{(1)}$ is not chosen correctly, after $n$ iterations we have $\mathbf{H}^{(n+1)} \approx \mathbf{G}^{(n+1)^{-1}}$, which is transforms correctly. The method therefore becomes close to the one in which invariance is preserved.

Invariance to an affine transformation of variables is a very important algorithmic property. Algorithms which have this property, are less sensitive to situations in which **G** is ill-conditioned, since an implicit transformation which transforms **G** to the unity matrix **I** can be introduced, which does not change the method. Algorithms that are not invariant, i.e. the steepest descent or the alternating variables method, can perform very badly when the Hessian is ill-conditioned.

When using methods which are not invariant, it can be advantageous to find a linear transformation which improves the conditioning of the problem[14].

If columns of $\mathbf{A}$ are eigenvectors of **G**, then **G** is diagonalised when transformation (3.96) is applied. Conditioning can be achieved by additional scaling of variables, i.e. by multiplication with a diagonal matrix. This approach is however not applicable in practice because it is usually difficult to calculate eigenvectors of **G**. For positive definite **G** the same effect is achieved by using Choleski factors of **G** as the transformation matrix. $\mathbf{G}_x = \mathbf{A}^T\mathbf{A}$ gives

$$\mathbf{G}_y = \mathbf{A}^{-T}\mathbf{G}_x\mathbf{A}^{-1} = \mathbf{A}^{-T}\mathbf{A}^T\mathbf{A}\mathbf{A}^{-1} = \mathbf{I}\,.$$

It is often possible to improve conditioning just by scaling the variables. In this case **A** is chosen to be a diagonal matrix so that $\mathbf{A}^2$ estimates $\mathbf{G}_x$ in some sense. $\mathbf{G}_y = \mathbf{A}^{-T}\mathbf{G}_x\mathbf{A}^{-1}$ (from (3.96)) is required to be close to the unity matrix in some sense. It can be required, for example, that $\left[\mathbf{G}_y\right]_{ii} = 1\,\forall i$. It is usually not necessary to explicitly perform the scaling, but **I** can be replaced in the methods by a suitable diagonal matrix. For example, the modified Newton method can be improved by using $\mathbf{G} + \nu\mathbf{A}^{-2}$ in place of $\mathbf{G} + \nu\,\mathrm{I}$.

## *3.6 Conjugate Direction Methods*

Optimisation algorithms described in this section are based on the result given in Theorem 3.2, which associates conjugacy and exact line searches with quadratic termination. These algorithms rely on an idealized assumption that exact line searches are performed as in Algorithm 3.2. This is possible for a quadratic function, but not in general. By using interpolation in the line search algorithm, it is still possible to locate a local minimum up to a certain accuracy, and this approach is used in practice with the conjugate direction methods. An argument which justifies this is that in the close neighbourhood of a minimum, quadratic interpolations of the objective functions will enable the line minimum to be located almost exactly, so that the inexact nature of the line search algorithm will not spoil local convergence properties, which are theoretically based on the assumption of exact line search.

In section 3.6.1 derivative based conjugate direction methods are described. In section 3.6.2 algorithms based on the idea of conjugacy, but in the absence of derivative information are treated. All the described methods generate conjugate directions when they are applied to a quadratic function.

### 3.6.1 Conjugate Gradient Methods

Conjugate gradient methods begin with line search along

$$\mathbf{s}^{(1)} = -\mathbf{g}^{(1)} \tag{3.99}$$

and then generate search directions $\mathbf{s}^{(k+1)}, k \geq 1$ from $-\mathbf{g}^{(k+1)}$, so that they are conjugate to $\mathbf{s}^{(1)}, ..., \mathbf{s}^{(k)}$ with respect to the Hessian matrix $\mathbf{G}$ when a method is applied to a quadratic function.

For a quadratic function it follows from (3.27) that

$$\gamma^{(k)} = \mathbf{G}\delta^{(k)}\,\forall k\,, \tag{3.100}$$

where $\gamma^{(k)} = \mathbf{g}^{(k+1)} - \mathbf{g}^{(k)}$ and $\delta^{(k)} = \mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}$, as usual. Conjugacy conditions (3.43) can therefore be written as

$$\mathbf{s}^{(i)T}\gamma^{(j)}=0 \quad j\neq i \tag{3.101}$$

since $\gamma^{j}=\mathbf{G}\delta^{(j)}=\mathbf{G}\alpha^{(j)}\mathbf{s}^{(j)}$. The last expression is a consequence of the fact that $\mathbf{x}^{(j+1)}$ is obtained by a line search performed from $\mathbf{x}^{(j)}$ along $\mathbf{s}^{(j)}$.

The above equation can be used to prove an important property. First we can see that

$$\mathbf{s}^{i^T}\mathbf{g}^{(i+1)}=0 \quad \forall i\,, \tag{3.102}$$

because exact line searches are used. By using the above equation and (3.101) we obtain

$$\begin{aligned}&\mathbf{s}^{(i)T}\mathbf{g}^{(k+1)}=\\&\mathbf{s}^{(i)(T)}\left(\mathbf{g}^{(k+1)}-\mathbf{g}^{(k)}+\mathbf{g}^{(k)}-\mathbf{g}^{(k-1)}+\ldots-\mathbf{g}^{(i+1)}+\mathbf{g}^{(i+1)}\right)=,\\&\mathbf{s}^{(i)(T)}\left(\gamma^{(k)}+\gamma^{(k)}+\ldots\gamma^{(i+1)}+\mathbf{g}^{(i+1)}\right)=0 \quad \forall i,k>i\end{aligned} \tag{3.103}$$

This means that $\mathbf{g}^{(k+1)}$ is orthogonal to all search directions of previous steps:

$$\mathbf{s}^{(i)T}\mathbf{g}^{(k+1)}=0 \quad \forall k, i\leq k\,. \tag{3.104}$$

This is actually the result of Theorem 3.2.

In the Fletcher-Reeves method $\mathbf{s}^{(k+1)}$ is obtained from $-\mathbf{g}^{(k+1)}$ by the extended Gramm-Schmidt orthogonalisation[27],[29] with respect to $\gamma^{(i)},\, j\leq k$, in order to satisfy conjugacy conditions (3.101). We can write[1]

$$\mathbf{s}^{(k+1)}=-\mathbf{g}^{(k+1)}+\sum_{j=1}^{k}\beta^{(j)}\mathbf{s}^{(j)}\,. \tag{3.105}$$

Multiplying the transpose of the above equation by $\gamma^{(i)}$ gives

$$\mathbf{s}^{(k+1)^T}\gamma^{(i)}=0=-\mathbf{g}^{(k+1)^T}\gamma^{(i)}+\beta^{(i)}\mathbf{s}^{(i)^T}\gamma^{(i)}, \tag{3.106}$$

[1] The derivation of the Fletcher-Reeves method was found to be not completely clear in some optimisation literature and is therefore included herein.

where (3.101) was taken into account. It follows that

$$\beta^{(i)} = \frac{\mathbf{g}^{(k+1)T}\gamma^{(i)}}{\mathbf{s}^{(i)T}\gamma^{(i)}} = \frac{\mathbf{g}^{(k+1)T}\left(\mathbf{g}^{(i+1)} - \mathbf{g}^{(i)}\right)}{\mathbf{s}^{(i)T}\left(\mathbf{g}^{(i+1)} - \mathbf{g}^{(i)}\right)}. \tag{3.107}$$

It follows from construction of $\mathbf{s}^{(k)}$ ((3.99) and (3.105)) that vectors $\mathbf{g}^{(1)}, ..., \mathbf{g}^{(k)}$ and $\mathbf{s}^{(1)}, ..., \mathbf{s}^{(k)}$ span the same subspace. Therefore, since $\mathbf{g}^{(k+1)}$ is orthogonal to the subspace spanned by $\mathbf{s}^{(1)}, ..., \mathbf{s}^{(k)}$ due to (3.104), it is also orthogonal to vectors $\mathbf{g}^{(1)}, ..., \mathbf{g}^{(k)}$, i.e.

$$\mathbf{g}^{(i)T}\mathbf{g}^{(k+1)} = 0 \quad \forall k, i \le k \tag{3.108}$$

We see that only $\beta^{(k)} \neq 0$ and that

$$\beta^{(k)} = \frac{\mathbf{g}^{(k+1)T}\left(\mathbf{g}^{(k+1)} - \mathbf{g}^{(k)}\right)}{\mathbf{s}^{(k)T}\left(\mathbf{g}^{(k+1)} - \mathbf{g}^{(k)}\right)} = -\frac{\mathbf{g}^{(k+1)T}\mathbf{g}^{(k+1)}}{\mathbf{s}^{(k)T}\mathbf{g}^{(k)}} \tag{3.109}$$

The denominator of the above equation can be obtained by substituting $\mathbf{s}^{(k)}$ by (3.105) with decreased indices and taking into account that only $\beta^{(k-1)}$ is non-zero, together with the established orthogonality properties:

$$\mathbf{s}^{(k)T}\mathbf{g}^{(k)} = \left(-\mathbf{g}^{(k)} + \beta^{(k-1)}\mathbf{s}^{(k-1)}\right)^T\mathbf{g}^{(k)} = -\mathbf{g}^{(k)T}\mathbf{g}^{(k)}.$$

Now we have

$$\beta^{(k)} = \frac{\mathbf{g}^{(k+1)T}\mathbf{g}^{(k+1)}}{\mathbf{g}^{(k)T}\mathbf{g}^{(k)}}. \tag{3.110}$$

The obtained results can be summarized in the following way:

**Theorem 3.5:**

The Fletcher-Reeves method with exact line searches terminates for a quadratic function at a stationary point $\mathbf{x}^{m+1}$ after $m \le n$ iterations. In addition, the following results hold for $1 \le i \le m$:

$$\mathbf{s}^{(i)T}\mathbf{G}\mathbf{s}^{(j)} = 0; \quad j = 1, 2, ..., i-1 \text{ (conjugate directions)}, \tag{3.111}$$

$$\mathbf{g}^{(i)^T}\mathbf{g}^{(j)} = 0;\;\; j = 1, 2, ..., i-1 \text{ (orthogonal gradients)} \tag{3.112}$$

and

$$\mathbf{s}^{(i)^T}\mathbf{g}^{(i)} = -\mathbf{g}^{(i)^T}\mathbf{g}^{(i)} \text{ (descent conditions).} \tag{3.113}$$

The termination must occur in at least $n$ iterations because in the opposite case $\mathbf{g}^{(n+1)} \neq 0$ would contradict the result that gradients are orthogonal.

When applied to a quadratic function with positive definite **G**, the Fletcher-Reeves method turns to be equivalent to the Broyden family of methods if $\mathbf{H}^{(1)} = \mathbf{I}$, the starting point is the same and exact line searches are performed in both methods[1],[4]. For non-quadratic functions line search Algorithm 3.4 is recommended with $\sigma = 0.1$. Resetting the search direction to the steepest descent direction periodically after every $n$ iterations is generally an accepted strategy in practice. When compared with quasi-Newton methods, conjugate gradient methods are less efficient and less robust and they are more sensitive to the accuracy of the line search algorithm. Methods with resetting are globally convergent and exhibit n-step superlinear convergence, i.e.

$$\lim_{k\to\infty} \frac{\left\| \mathbf{x}^{(k+n)} - \mathbf{x}^* \right\|}{\left\| \mathbf{x}^{(k)} - \mathbf{x}^* \right\|} = 0 \tag{3.114}$$

Some other formulas may be used instead of (3.110). Examples are the conjugate descent formula

$$\beta^{(k)} = \frac{\mathbf{g}^{(k+1)^T}\mathbf{g}^{(k+1)}}{\mathbf{g}^{(k)^T}\mathbf{s}^{(k)}} \tag{3.115}$$

and the Polak-Ribiere formula

$$\beta^{(k)} = \frac{\left(\mathbf{g}^{(k+1)} - \mathbf{g}^{(k)}\right)^T \mathbf{g}^{(k+1)}}{\mathbf{g}^{(k)^T}\mathbf{g}^{(k)}}. \tag{3.116}$$

Considering the derivation of the Fletcher-Reeves method, it can be seen that these formulas are equivalent to the Fletcher-Reeves formula when applied to quadratic functions with exact line searches. The conjugate descent formula has a strong descent property that $\mathbf{s}^{(k)T}\mathbf{g}^{(k)} < 0$ if $\mathbf{g}^{(k)} \neq 0$. The Polak-Ribiere formula is recommended when solving large problems[1].

Another possibility for conjugate gradient methods is to use symmetric projection matrices in the calculation of $\mathbf{s}^{(k+1)}$, which annihilate vectors $\gamma^{(k)}, ..., \gamma^{(k)}$:

$$\mathbf{s}^{(k)} = -\mathbf{P}^{(k)}\mathbf{g}^{(k)}, k = 1, 2, ..., n. \tag{3.117}$$

Initially

$$\mathbf{P}^{(1)} = \mathbf{I} \tag{3.118}$$

and subsequent $\mathbf{P}^{(k)}$ are updated as

$$\mathbf{P}^{(k+1)} = \mathbf{P}^{(k)} - \frac{\mathbf{P}^{(k)}\gamma^{(k)}\gamma^{(k)T}\mathbf{P}^{(k)}}{\gamma^{(k)T}\mathbf{P}^{(k)}\gamma^{(k)}}. \tag{3.119}$$

Again this method is equivalent to other described methods for quadratic functions. When applied to general functions, $\mathbf{P}^{(i)}$ must be reset to $\mathbf{I}$ every $n$ iterations since $\mathbf{P}^{(n+1)} = 0$. The method has the descent property $\mathbf{s}^{(k)T}\mathbf{g}^{(k)} \leq 0$, but has a disadvantage that matrix calculations are required in each iteration.

### 3.6.2 Direction Set Methods

Direction set methods[1],[26] generate conjugate directions with respect to the Hessian matrix G, when they are applied to a quadratic function, without use of derivative information. Construction of the conjugate direction is based on the following theorem[1]:

**Theorem 3.6: Parallel subspace property**

Let us have a quadratic function $q(\mathbf{x})$ with a positive definite Hessian matrix $\mathbf{G}$. Consider two parallel subspaces $S_1$ and $S_2$, generated by independent directions $\mathbf{s}^{(1)}, ..., \mathbf{s}^{(k)}$ ($k < n$) from the points $\mathbf{v}^{(1)}$ and $\mathbf{v}^{(2)}$, such that $S_1 \neq S_2$, i.e.

$$S_1 = \left\{ \mathbf{x}; \mathbf{x} = \mathbf{v}^{(1)} + \sum_{i=1}^{k} \alpha_i \mathbf{s}^{(i)} \ \forall \alpha_i \right\}$$

and similarly $S_2$. Let $\mathbf{z}^{(1)}$ be the point which minimises $q$ on $S_1$ and $\mathbf{z}^{(2)}$ the point which minimises $q$ on $S_2$ Then $\mathbf{z}^{(2)} - \mathbf{z}^{(1)}$ is conjugate to $\mathbf{s}^{(1)}, ..., \mathbf{s}^{(k)}$ with respect to $\mathbf{G}$, i.e. $\left(\mathbf{z}^{(2)} - \mathbf{z}^{(1)}\right)^T \mathbf{G}\, \mathbf{s}^{(i)} = 0,\ i = 1, ..., k$.

The above theorem is outlined for two dimensions in Figure 3.8. Although instructive, a two-dimensional representation is not completely satisfactory because the complement of the vector space spanned by $\mathbf{s}^{(1)}$ is one dimensional and therefore the line $\mathbf{z}^{(1)} + \alpha\left(\mathbf{z}^{(2)} - \mathbf{z}^{(1)}\right)$ contains a minimum of $q(x_i, x_2)$. Also the parallel subspaces are only one dimensional.

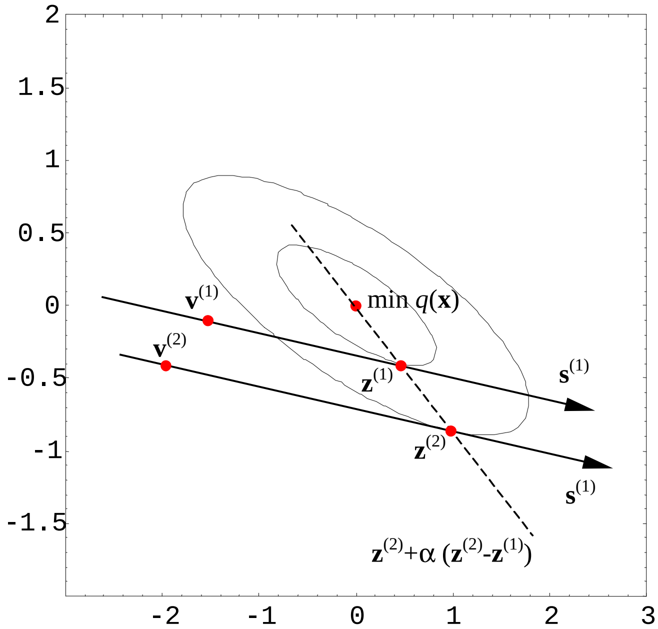



**Figure 3.8:** The parallel subspace property in two dimensions.

In the direction set methods, conjugate directions $\mathbf{s}^{(i)}$ are generated one by one by minimising the function in a subspace spanned by previously constructed $\mathbf{s}^{(1)}, ..., \mathbf{s}^{(i-1)}$ (giving $\mathbf{x}^{(i)}$, say), a parallel subspace is created by displacing the obtained minimum by a vector that is independent on $\mathbf{s}^{(1)}, ..., \mathbf{s}^{(i-1)}$ ($\mathbf{d}^{(i)}$, say), followed by minimising in that subspace (giving $\mathbf{z}^{(i)}$, say), and setting a new conjugate direction to $\mathbf{s}^{(i)} = \mathbf{z}^{(i)} - \mathbf{x}^{(i)}$. Since the directions $\mathbf{s}^{(1)}, ..., \mathbf{s}^{(i)}$ are conjugate, previously performed minimisation in directions $\mathbf{s}^{(1)}, ..., \mathbf{s}^{(i-1)}$ is not affected by moving along the line $\mathbf{z}^{(i)} + \alpha\left(\mathbf{z}^{(i)} - \mathbf{x}^{(i)}\right)$. Minimisation in the subspace

$$S_1^{(i)} = \left\{ \mathbf{x};\, \mathbf{x} = \mathbf{z}^{(i)} + \sum_{j=1}^{i} \alpha_j \mathbf{s}^{(j)} \;\; \forall \alpha_j \right\}$$

is therefore achieved by line minimisation along that line (this is well illustrated in Figure 3.8). This is of course valid only for quadratic functions, but the construction of an algorithm for general functions is based on the same arguments and is similar.

From the above construction it is clear that the direction $\mathbf{s}^{(i)}$ is a linear combination of vectors $\mathbf{d}^{(1)}, \mathbf{d}^{(2)}, ..., \mathbf{d}^{(i)}$, but is not contained in the subspace spanned by $\mathbf{d}^{(i+1)}, \mathbf{d}^{(i+2)}, ..., \mathbf{d}^{(n)}$

The above described procedure is followed in Smith's method. The cycle is repeated when the function is not quadratic. The method requires independent vectors $\mathbf{d}^{(1)}, \mathbf{d}^{(2)}, ..., \mathbf{d}^{(n)}$ to be supplied, which are used for the successive generation of parallel subspaces after minimisation in the current subspace. This is however not the most effective approach because directions are not treated equally. Line minimisations are performed more frequently in the directions that are constructed earlier.

This deficiency is abolished in Powell's method. Its cycle is as described above, except that a point of the parallel subspace is obtained by line minimisations along $\mathbf{d}^{(i)}, \mathbf{d}^{(i+1)}, ..., \mathbf{d}^{(n)}$ from $\mathbf{x}^{(i)}$ rather by just adding $\mathbf{d}^{(i)}$ to $\mathbf{x}^{(i)}$. Powell's method is also such that cycles can be continued when the function is not quadratic, while in Smith's method directions are restarted after each cycle. The algorithm is sketched below.

**Algorithm 3.8:** Powell's direction set algorithm.

Given independent directions $\mathbf{s}^{(1)}, \mathbf{s}^{(2)}, ..., \mathbf{s}^{(n)}$, the minimum point $\mathbf{x}^{(1)}$ along $\mathbf{s}^{(1)}$ is found by a line search. Then the following iteration is repeated for $i = 1, 2, ...$:

1. Find

$$\mathbf{z}^{(i)} = \mathbf{x}^{(i)} + \sum_{j=1}^{n} \alpha_j \mathbf{s}^{(j)}$$

by sequentially minimising $f$ along $\mathbf{s}^{(1)},\ \mathbf{s}^{(2)},\ \ldots,\ \mathbf{s}^{(n)}$. If $i \le n$ then the last $i$ directions have already been replaced by conjugate directions.

2. For $j = 1, 2, ..., n-1$ replace $\mathbf{s}^{(j)}$ by $\mathbf{s}^{(j+1)}$.

3. Set $\mathbf{s}^{(n)} = \mathbf{z}^{(i)} - \mathbf{x}^{(i)}$, which is a new conjugate direction. Find the minimum point $\mathbf{x}^{(i+1)} = \mathbf{z}^{(i)} + \alpha \mathbf{s}^{(n)}$ along $\mathbf{s}^{(n)}$ by a line search. For a quadratic function, when $i \le n$, $\mathbf{x}^{(i+1)}$ is a minimum point of the subspace

$$S = \left\{ \mathbf{x};\, \mathbf{x} = \mathbf{z}^{(i)} + \sum_{k=n-i}^{n} \alpha_k \mathbf{s}^{(k)} \right\}.$$

since the last $i+1$ directions are conjugate.

The first *n-i* line searches in line 1 locate a point in a parallel subspace. The last *i* line searches can be thought of as minimisation of that subspace spanned by the last *i* directions (which are conjugate).

The algorithm terminates in about $n^2$ line searches when applied to quadratic functions. This is about twice as much as Smith's method ($\frac{1}{2}n(n+1)$), but the directions are now treated equally. Pseudo-conjugate directions are retained for a general function after the *n*-th iteration and are updated from iteration to iteration. The method is therefore more effective for general functions as Smith's method. One of the problems with this method is that in some problems directions $\mathbf{s}^{(j)}$ tend to become linearly dependent. It is possible to introduce modifications which deal with this problem. One possibility is to reset the direction set every certain numbers of the cycles.

## *3.7 Restricted Step Methods*

The restricted step prototype algorithm is an alternative to the line search strategy, in an attempt to ensure global convergence of minimisation algorithms.

There is one fundamental difference between the line search and restricted step approach. As can be seen from precedent sections, the line search based algorithms rely to a great extent on a quadratic model. Directions of line searches are essentially constructed in such a way that when an algorithm is applied to a quadratic function with a positive definite Hessian matrix, termination occurs in a finite number of exact line searches. Then, using the argument that every sufficiently smooth function can be locally (i.e. in the neighbourhood of the current guess) approximated by a quadratic function, the strategy designed on a quadratic model is more or less directly transferred to algorithms for handling general functions.

The fact that even within a single line search the minimised function can deviate far from its quadratic approximation is ignored by the line search approach. On the contrary, the main idea of the restricted step approach is to make direct use of the quadratic model, but only in the limited region where this is an adequate approximation to the original function. This leads to a sub-problem of minimsation of a quadratic approximation, limited to a certain region. One of the benefits is that difficulties with a non-positive definite Hessian matrix are avoided. It is clear at first sight that among the important concerns of restricted step methods is how to define a so called trust region $\mathbf{\Omega}^{(k)}$, i.e. the neighbourhood of a current guess in which the use of a quadratic approximation is adequate.

In the view of the above discussion, consider the problem

$$\begin{aligned} &\textit{minimise} && q^{(k)}(\delta) \\ &\textit{subject to} && \|\delta\| \le h^{(k)}, \end{aligned} \tag{3.120}$$

where $q^{(k)}(\delta)$ is a second order Taylor approximation of $f$ about $\mathbf{x}^{(k)}$ and $\delta = \mathbf{x} - \mathbf{x}^{(k)}$, i.e.

$$q^{(k)}(\delta) = f^{(k)} + \mathbf{g}^{(k)T}\delta + \tfrac{1}{2}\delta^T \mathbf{G}^{(k)}\delta\,. \tag{3.121}$$

The second part of equation (3.120) defines the trust region as

$$\mathbf{\Omega}^{(k)} = \left\{\mathbf{x};\, \left\|\mathbf{x} - \mathbf{x}^{(k)}\right\| \le h^{(k)}\right\}. \tag{3.122}$$

Restricted step methods aim at keeping the step restriction $h^{(k)}$ as large as possible, subject to the restriction that a certain agreement between $q^{(k)}\left(\delta^{(k)}\right)$ and $f\left(\mathbf{x}^{(k)} + \delta^{(k)}\right)$ must be retained, where $\delta^{(k)}$ is the solution of (3.120). Some measure of agreement must be defined for this purpose. Let us define the actual reduction of *f* in the step *k* as

$$\Delta f^{(k)} = f^{(k)} - f\left(\mathbf{x}^{(k)} + \delta^{(k)}\right) \tag{3.123}$$

and the corresponding predicted reduction as

$$\Delta q^{(k)} = q^{(k)}(0) - q^{(k)}\left(\delta^{(k)}\right) = f^{(k)} - q^{(k)}\left(\delta^{(k)}\right). \tag{3.124}$$

Then the ratio

$$r^{(k)} = \frac{\Delta f^{(k)}}{\Delta q^{(k)}} \tag{3.125}$$

is a suitable measure of the agreement, where good agreement is indicated by $r^{(k)}$ being close to 1. In restricted step algorithms it is attempted to maintain a certain degree of agreement by adaptively changing $h^{(k)}$. Such a prototype algorithm is defined below.

**Algorithm 3.9:** Iteration of the restricted step prototype algorithm.

1. Given $\mathbf{x}^{(k)}$ and $h^{(k)}$, evaluate $f^{(k)}$, $\mathbf{g}^{(k)}$ and $\mathbf{G}^{(k)}$ to define $q^{(k)}(\delta)$ in (3.121).
2. Solve (3.120) for $\delta^{(k)}$.
3. Evaluate $f\left(\mathbf{x}^{(k)} + \delta^{(k)}\right)$ and $r^{(k)}$.
4. If $r^{(k)} < \tau_1$ set $h^{(k+1)} = \rho_1 \left\|\delta^{(k)}\right\|$,
   if $r^{(k)} > \tau_2$ and $\left\|\delta^{(k)}\right\| = h^{(k)}$ set $h^{(k+1)} = \rho_2 h^{(k)}$,
   else set $h^{(k+1)} = h^{(k)}$.
5. If $r^{(k)} \le 0$ set $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)}$ else set $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \delta^{(k)}$.

Constants used in the above algorithm must be chosen so that $0 < \tau_1 < \tau_2 < 1$, $0 < \rho_1 < 1$ and $1 < \rho_2$. A suitable choice is $\tau_1 = 0.25$, $\tau_2 = 0.75$, $\rho_1 = 0.25$ and $\rho_2 = 2$, but the algorithm is not very sensitive to the choice[1]. In line 4 of the algorithm the step restriction is tightened if agreement between $\Delta f^{(k)}$ and $\Delta q^{(k)}$ is bad, and relaxed if the agreement is good and at the same time the minimum of $q$ lies on the edge of the trust region. If the minimum of $q^{(k)}$ lies inside the trust region, then there is no need to further relax the step restriction because that constraint will become inactive anyway, and the algorithm will reduce to the basic Newton method with rapid convergence. In line 5 $\mathbf{x}^{(k)}$ is preserved if $f\left(\mathbf{x}^{(k)} + \delta^{(k)}\right) > f^{(k)}$, which guarantees the descent property $f^{(k+1)} \le f^{(k)}$.

The following strong convergence result holds for restricted step methods[1]:

**Theorem 3.7:**

For Algorithm 3.9, if $\mathbf{x}^{(k)} \in B \subset \mathbb{R}^n \ \forall k$, where $B$ is bounded, and if $f \in \mathbb{C}^2$ on $B$, then there exists an accumulation point $\mathbf{x}^\infty$ which satisfies the first and the second order necessary conditions for a local minimum. If $\mathbf{G}$ also satisfies the Lipshitz condition $\|\mathbf{G}(\mathbf{x})-\mathbf{G}(\mathbf{y})\| \leq \lambda\|\mathbf{x}-\mathbf{y}\|$ in some neighbourhood of $\mathbf{x}^\infty$ and if $\mathbf{G}^\infty$ is positive definite, then for the main sequence $r^{(k)} \to 1$, $\inf h^{(k)} > 0$, the constraint $\|\delta\| \leq h^{(k)}$ is inactive for sufficiently large $k$, and convergence is second order.

The existence of $B$ in the above theorem is not a strong requirement. It is implied if any level set $\{\mathbf{x};\ f(\mathbf{x} \leq f^{(k)})\}$ is bounded because $\mathbf{x}^{(k)}$ are descent.

If the $L_\infty$ norm is used in the step restriction, then the subproblem (3.120) becomes a quadratic programming problem with simple bounds, for which good algorithms for local solutions exist. It is however difficult to find global solutions, but in practice it is adequate to calculate only local solutions.

Slow progress of the method can arise if the norm is not scaled. Ideally the natural metric norm $\|\delta\|_G = \delta^T \mathbf{G} \delta$ would be chosen when $\mathbf{G}$ is positive definite, but the scaling of variables can also be an adequate approach.

The Hessian matrix in the restricted step methods can be replaced by the approximate Hessian $\mathbf{B}^{(k)}$ or its inverse $\mathbf{H}^{(k)}$, updated according to a quasi-Newton scheme. In such a case similar global convergence result holds as for the original method.

## *3.8 Basics of Constrained Optimisation*

The remainder of chapter 3 is devoted to the case in which constraints on the optimisation variables are defined (Figure 3.10). The presence of constraints introduces additional complexity in the treatment of local solutions in view of the definition of necessary and sufficient conditions, which is discussed in the present section.

Constrained optimisation problems are much more difficult to treat numerically than unconstrained problems. Many algorithms for their solution are based on transformation of the constrained problem to a sequence of unconstrained optimisation subproblems whose solutions converge to the solution of the constrained problem. A commonly used approach is the penalty function approach based on addition of weighted penalty terms to the objective function which cause high values where constraints are violated or close to be violated. In the limit when weights tend to infinity, the solutions of the unconstrained problems tend to the solution of the constrained problem. This approach is described in section 3.10.

The next important approach is the elimination of variables. Equality constraints are used to define implicit dependence (through solution of a nonlinear systems of equations defined by equality constraints) of a subset of optimisation variables on the remaining variables. The constrained problem is in this way transformed to an unconstrained problem defined on a reduced set of variables, but each evaluation requires a system of nonlinear equations to be solved for the dependent variables. When inequality constraints are involved, the active constraints are treated as equality constraints. Since it is not known in advance which constraints are active in the solution, the set of active constraints is iteratively updated. This leads to the active set type of methods, the principle of which are described in sections 3.9 and 3.11.

Algorithms for solution of constrained optimisation problems are based on quadratic models to a large extent. Some algorithms for general functions explicitly generate quadratic programming subproblems (quadratic objective function and linear constraints). These algorithms represent an alternative to the more traditional penalty function approach and seem to be superior from the point of view of efficiency. Section 3.9 covers some basic aspects of quadratic programming.

There are also solution algorithms which linearise both the objective functions and constraints about the current iterate and therefore generate a sequence of linear programming problems[14][9]. This approach seems to be popular in some fields, however only problems with some special structure can be successfully solved in this way (e.g. with the objective function close to linear), therefore attention is not devoted to the approach in this work. Linear programming (linear objective function and constraints) is also not treated in this work for the same reason.

### 3.8.1 Langrange Multipliers and First Order Conditions for Unconstrained Local Minima

Consider the problem (3.19) where constraints are present. For any point $\mathbf{x}'$ active or binding constraints are those for which the corresponding constraint function is zero at that point. A set of their indices will be denoted by

$$\mathcal{A}' = \mathcal{A}(\mathbf{x}') = \{ i; c_i(\mathbf{x}') = 0 \} \tag{3.126}$$

Any constraint is active at $\mathbf{x}'$ if that point is on the boundary of its feasible region. The set of active constraints at the solution $\mathcal{A}^*$ is of particular importance.

Constraints which are not active at the solution can be perturbed by small amounts without affecting the problem solution.

The gradient of the *i*-th constraint function $\nabla c_i$ will be denoted by $\mathbf{a}_i$ and referred to as the normal vector of the constraint $c_i$. These vectors can be arranged in a Jacobian matrix **A**, whose columns are constraint gradients.

Consider a problem with only equality constraints and a feasible incremental step $\delta$ taken from a local minimiser. By a Taylor series we have

$$c_i\left(\mathbf{x}^* + \delta\right) = c_i^* + \delta^T \mathbf{a}_i^* + o\left(\|\delta\|\right).$$

Since $\delta$ is a feasible step we have $c_i\left(\mathbf{x}^* + \delta\right) = c_i^* = 0$ and where the length of the step length is small, we have by neglecting higher order terms $\delta^T \mathbf{a}_i^* = 0$. By taking into account all constraints, we can define a feasible direction as a direction which satisfies

$$\mathbf{s}^T \mathbf{a}_i^* = 0 \quad \forall i \in E . \tag{3.127}$$

Clearly if **s** is a feasible direction then –**s** is also a feasible direction. Since $\mathbf{x}^*$ is a constrained local minimiser, there is no feasible descent direction, because otherwise *f* could be reduced by an arbitrarily small step in that direction. It follows that $\mathbf{s}^T \mathbf{g}^* = 0$ for any feasible direction **s**. Due to (3.127) this is satisfied if $\mathbf{g}^*$ is a linear combination of constraint gradients, i.e.

$$\mathbf{g}^* = \sum_{i \in E} \mathbf{a}_i^* \lambda_i^* = \mathbf{A}^* \lambda^* . \tag{3.128}$$

Multipliers $\lambda_i^*$ are referred to as Lagrange multipliers and can be arranged in the Lagrange multiplier vector (denoted by $\lambda^*$ without a subscript). The above equation is also a necessary condition for a local minimiser. If (3.128) would not hold, then $\mathbf{g}^*$ could be expressed as

$$\mathbf{g}^* = \mathbf{A}^* \lambda^* + \mu \tag{3.129}$$

where $\mu$ is a component of $\mathbf{g}^*$ orthogonal to all $\mathbf{a}_i^*$. Then $\mathbf{s} = -\mu$ would be a feasible descent direction (i.e. would satisfy both (3.128) and $\mathbf{s}^T \mathbf{g}^* < 0$). A feasible incremental step $\delta$ along **s** would reduce *f*, which contradicts the fact that $\mathbf{x}^*$ is a local minimiser. This is illustrated in Figure 3.9 for a single constraint.

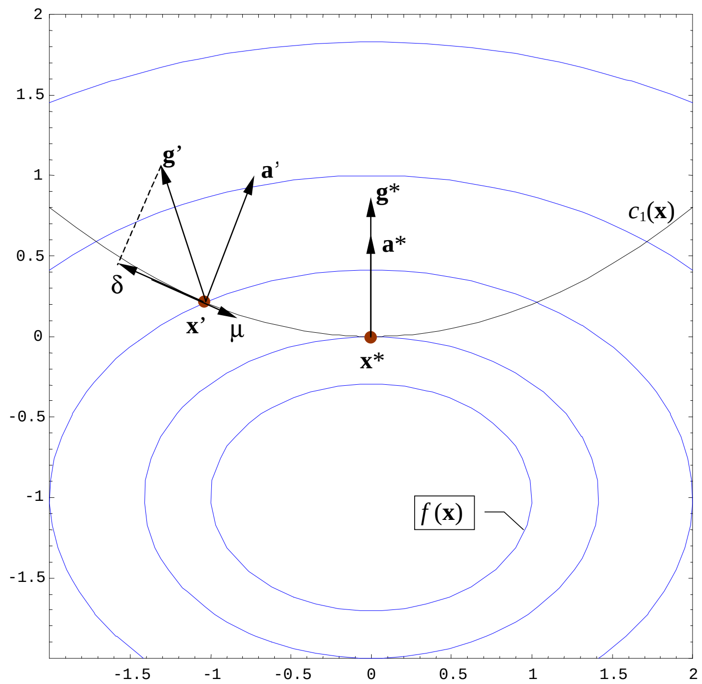



**Figure 3.9:** Illustration of necessary conditions for a constrained local minimum.

The necessary conditions are a basis of the method of Lagrange multipliers for equality constraint problems. The method searches for vectors $\mathbf{x}^*$ and $\lambda^*$, which solve the equations

$$\mathbf{g}(\mathbf{x}) = \sum_{i \in E} \lambda_i \mathbf{a}_i(\mathbf{x})$$

*and* (3.130)

$$c_i(\mathbf{x}) = 0, \; i \in E \, .$$

This approach has a similar disadvantage to the Newton method for unconstrained minimisation: the above equations are satisfied in a constrained saddle point or maximiser, since no second order information is taken into account.

The above equations can be written in a simpler form if we define the Lagrangian function

$$\mathcal{L}(\mathbf{x},\lambda) = f(\mathbf{x}) - \sum_i \lambda_i c_i(\mathbf{x}). \tag{3.131}$$

Equations (3.130) them become

$$\overline{\nabla}\mathcal{L}(\mathbf{x},\lambda) = 0, \tag{3.132}$$

where

$$\overline{\nabla} = \begin{bmatrix} \nabla_x \\ \nabla_\lambda \end{bmatrix} \tag{3.133}$$

is the gradient operator in the $n+m$ dimensional variable space ($m$ will denote the number of constraints). We see that a necessary condition for a local minimiser is that $\left(\mathbf{x}^*, \lambda^*\right)^T$ is a stationary point of the Lagrangian function.

Lagrange multipliers have a clear practical interpretation. If the Jacobian matrix of constraints has rank $m$ (linearly independent $\mathbf{a}_i$) then the multipliers in (3.128) are uniquely defined by

$$\lambda^* = \mathbf{A}^{*+}\mathbf{g}, \tag{3.134}$$

where $\mathbf{A}^{*+} = \left(\mathbf{A}^{*T}\mathbf{A}^*\right)^{-1}\mathbf{A}^T$ is a generalised inverse[2] of $\mathbf{A}^*$. Consider in such case perturbations of the right-hand sides of the constraint

$$c_i(\mathbf{x}) = \varepsilon_i, \ i \in E \tag{3.135}$$

and let $f(\varepsilon)$ and $\lambda(\varepsilon)$ denote how the solution and multipliers change with respect to perturbations. The lagrangian function for the perturbed problems is

$$\mathcal{L}(\mathbf{x},\lambda,\varepsilon) = f(\mathbf{x}) - \sum_{i \in E} \lambda_i \left(c_i(\mathbf{x}) - \varepsilon_i\right) \tag{3.136}$$

In the perturbed solution new constraints are satisfied, therefore

$$f(\mathbf{x}(\varepsilon)) = \mathcal{L}(\mathbf{x}(\varepsilon), \lambda(\varepsilon), \varepsilon).$$

Derivation of this equation with respect to $\varepsilon_i$ gives

$$\frac{d\,f}{d\,\varepsilon_i} = \frac{d\mathcal{L}}{d\varepsilon_i} = \frac{\partial \mathbf{x}^T}{\partial \varepsilon_i}\nabla_x \mathcal{L} + \frac{\partial \lambda^T}{\partial \varepsilon_i}\nabla_\lambda \mathcal{L} + \frac{\partial \mathcal{L}}{\partial \varepsilon_i}$$

By (3.132), (3.135) and (3.136) it follows that

$$\frac{df}{d\varepsilon_i} = \lambda_i\,. \tag{3.137}$$

Lagrange multipliers therefore indicate how sensitive the value of the objective function at the solution is to changes in the corresponding constraints.

Consider now a case where inequality constraints are present. Only active constraints at the solution $\mathcal{A}^*$ influence conditions for the solution. A set of active inequality constraints at the solution will be denoted by $I^*\ (= \mathcal{A}^* \cap I)$. Any feasible direction **s** must satisfy (in addition to (3.127)) the condition

$$\mathbf{s}^T \mathbf{a}_i^* \geq 0 \quad \forall i \in I^*\,. \tag{3.138}$$

Conditions for a local minimiser are

$$\mathbf{g}^* = \sum_{i \in \mathcal{A}^*} \lambda_i^* \mathbf{a}_i^* \tag{3.139}$$

and

$$\lambda_i^* \geq 0 \ \ \forall i \in I^*\,. \tag{3.140}$$

Condition (3.139) can be deduced in a similar way to (3.130). Condition (3.140), which is an extra condition with respect to the case of equality constraints, can be deduced using the result (3.137). A small perturbation of the *i*- th active inequality constraint by positive $\varepsilon_i$ induces a change $\mathbf{x}(\varepsilon)$ that is feasible with respect to the unperturbed problem. Therefore *f* must not decrease, which implies $df^* / d\varepsilon_i \geq 0$ and hence $\lambda_i^* \geq 0$.

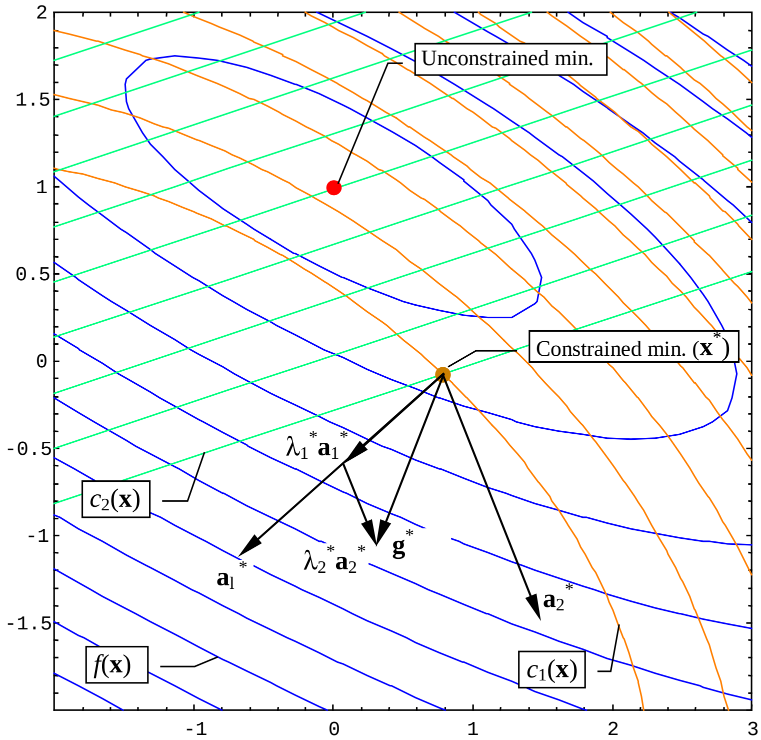



**Figure 3.10:** Constrained optimisation problem with two inequality constraints. Contours of constraint functions are drawn only in the infeasible region where their values are less or equal to zero. Since both constraints are active in the solution, the solution would remain unchanged if one or both constraints were replaced by equality constraints.

Lagrange multipliers have another important interpretation in the case of inequality constraints. Consider a point at which (3.139) is satisfied and (3.140) holds for all $i$ except for $i = p$, i.e. $\lambda_p^* < 0$, and let all $\mathbf{a}_i^*$ be linearly independent. Then it is possible to find a direction $\mathbf{s}$ for which $\mathbf{s}^T \mathbf{a}_p^* = 1$ and $\mathbf{s}^T \mathbf{a}_i^* = 0$ for all other active constraints. Such is given for example by $\mathbf{s} = \mathbf{A}^{*+T} \mathbf{e}_p$, where $\mathbf{e}_p$ is the $p$-th coordinate vector. Then $\mathbf{s}$ is a feasible direction and at the same time a descent direction since

$$\mathbf{s}^T \mathbf{g}^* = \mathbf{s}^T \mathbf{a}_p^* \lambda_p^* < 0. \qquad (3.141)$$

This first contradicts the fact that $\mathbf{x}^*$ is a local minimiser and is another proof that conditions (3.140) are necessary. Besides, it indicates that $f(\mathbf{x})$ can be reduced by moving away from the boundary of constraint $p$ for which the corresponding Lagrange multiplier is negative. This is important in the active set methods for handling inequality constraints, where constraints index $p$ with $\lambda_p^* < 0$ can be removed from the active set (section 3.9.2).

In the derivation of the first order conditions the regularity assumption that $\mathbf{a}_i^*$ are independent was made. This is not necessarily the case and an exact statement of the conditions requires more careful treatment[1].

First the notion of feasible direction must be defined more exactly. Consider a feasible point $\mathbf{x}'$ and any infinite sequence of feasible points convergent to this point $\left\{\mathbf{x}^{(k)}\right\} \to \mathbf{x}'$ where in addition $\mathbf{x}^{(k)} \neq \mathbf{x}'$ for all $k$. It is possible to write

$$\mathbf{x}^{(k)} - \mathbf{x}' = \delta^{(k)} \mathbf{s}^{(k)} \quad \forall k \tag{3.142}$$

where $\delta^{(k)} > 0$ are scalars and $\mathbf{s}^{(k)}$ are vectors of any fixed length $\sigma > 0$. A directional sequence is defined as any such sequence for which vectors $\mathbf{s}^{(k)}$ converge to some direction, i.e. $\mathbf{s}^{(k)} \to \mathbf{s}$. The limiting vector $\mathbf{s}$ is then referred to as the feasible direction. $\mathcal{F}(\mathbf{x}') = \mathcal{F}'$ will be used to denote the set of all feasible directions at $\mathbf{x}'$.

It can be seen from the previous discussion that the set of feasible directions for the linearised constraint set is

$$F(\mathbf{x}') = F' = \left\{ \mathbf{s};\, \mathbf{s} \neq 0 \wedge \begin{array}{ll} \mathbf{s}^T \mathbf{a}_i' = 0 & \forall i \in E \\ \mathbf{s}^T \mathbf{a}_i' \geq 0 & \forall i \in I' \end{array} \right\}, \tag{3.143}$$

where $I'$ is a set of active inequality constraints at $\mathbf{x}'$.

The relation $\mathcal{F}' \subseteq F'$ holds in general. $\mathcal{F}' = F'$ either if the constraints $i \in \mathcal{A}'$ are linear or vectors $\mathbf{a}_i'$, $i \in \mathcal{A}'$ are linearly independent. The assumption $\mathcal{F}' = F'$ is referred to as a constraint qualification at $\mathbf{x}'$.

The set of descent directions at $\mathbf{x}'$ is defined as

$$\mathcal{D}(\mathbf{x}') = \mathcal{D}' = \left\{ \mathbf{s};\, \mathbf{s}^T \mathbf{g}' < 0 \right\}. \tag{3.144}$$

[1] In some optimisation literature the possibility that gradients of active constraints in the solution can be linearly dependent is ignored, sometimes with an argument that this is an extremely unlikely situation.

If $\mathbf{x}^*$ is a local minimiser, then $\mathcal{F}^* \cap \mathcal{D}^* = \emptyset$, i.e. no feasible descent directions exist.

Let the following regularity assumption be made:

$$F^* \cap \mathcal{D}^* = \mathcal{F}^* \cap \mathcal{D}^* . \tag{3.145}$$

This is weaker assumption than $\mathcal{F}' \subseteq F'$. Under this assumption the following more general statement of the first order necessary conditions can be made[1],[4],[7]:

**Theorem 3.8:** Kuhn-Tucker (or KT) conditins.

If $\mathbf{x}^*$ is a local constrained minimiser and if regularity assumption(3.145) holds, then there exist Lagrange multipliers $\lambda^*$ such that $\mathbf{x}^*$ and $\lambda^*$ satisfy the following system:

$$\begin{aligned}
&\nabla_x \mathcal{L}(\mathbf{x}, \lambda) = 0 \\
&c_i(\mathbf{x}) = 0, \quad i \in E \\
&c_i(\mathbf{x}) \geq 0, \ \ i \in I \\
&\lambda_i \geq 0, \ \ i \in I \\
&\lambda_i c_i(\mathbf{x}) = 0 \ \ \forall i .
\end{aligned} \tag{3.146}$$

A point that satisfies the above conditions is referred to as a KT point. The condition $\lambda_i^* c_i^* = 0$ is referred to as the complementarity condition. It states that both $\lambda_i^*$ and $c_i^*$ can not be non-zero, which means that inactive constraints are regarded as having zero Lagrange multipliers. If there is no *i* such that $\lambda_i^* = c_i^* = 0$ then strict complementarity is said to hold. The case $\lambda_i^* = c_i^* = 0$ appears for example if an unconstrained minimiser lies on the boundary of the feasible region, which is an intermediate state between a constraint being strongly active and inactive.

### 3.8.2 Second Order Conditions

Consider first the case with only equality constraints. The second order conditions can be derived from the second order Taylor series of the Lagrangian function about the local solution. It is assumed that $\mathbf{a}_i^*$ are independent so that unique Lagrange multipliers exist. Let a feasible incremental step $\delta$ be made along any feasible direction $\mathbf{s}$. By feasibility it follows that $f(\mathbf{x}+\delta)=\mathcal{L}(\mathbf{x}+\delta,\lambda)$. We also take into account that $\mathcal{L}$ is stationary at $\mathbf{x}^*$ and $\lambda^*$ to eliminate the first derivatives. The second order Taylor expansion then gives (after neglecting higher than second order terms)

$$\begin{aligned}&f(\mathbf{x}^*+\delta)=\mathcal{L}(\mathbf{x}^*+\delta,\lambda^*)\approx\\&\mathcal{L}(\mathbf{x}^*+\delta,\lambda^*)+\delta^T\underbrace{\nabla_x\mathcal{L}(\mathbf{x}^*,\lambda^*)}_{=0}+\tfrac{1}{2}\delta^T\mathbf{W}\delta=.\\&f^*+\tfrac{1}{2}\delta^T\mathbf{W}\delta\end{aligned} \tag{3.147}$$

$\mathbf{W}$ denotes the Hessian matrix of the Lagrangian function with respect to variables $\mathbf{x}$:

$$\mathbf{W}^*=\nabla_x{}^2\mathcal{L}(\mathbf{x}^*,\lambda^*)=\nabla^2 f(\mathbf{x}^*)-\sum_i\lambda_i^*\nabla^2 c_i(\mathbf{x}^*). \tag{3.148}$$

Since $\mathbf{x}^*$ is a local minimiser, the function value taken in any feasible infinitesimal incremental step in any direction must be greater than or equal to $f^*$. It follows that

$$\mathbf{s}^T\mathbf{W}\mathbf{s}^*\geq 0 \tag{3.149}$$

for any feasible direction, i.e. for any $\mathbf{s}$ that satisfies

$$\mathbf{a}_i^{*T}\mathbf{s}=0\ \ \forall i\in E\,. \tag{3.150}$$

This is a second order necessary condition for a local minimiser, which can also be stated as a requirement that the Lagrangian function must have a non-negative curvature along any feasible direction.

A sufficient condition is that $\mathbf{x}^*$ satisfies (3.128) and

$$\mathbf{s}^T\mathbf{W}^*\mathbf{s}>0 \tag{3.151}$$

for all feasible directions $\mathbf{s}$ that satisfy (3.150) (a zero vector is not considered to be a feasible direction).

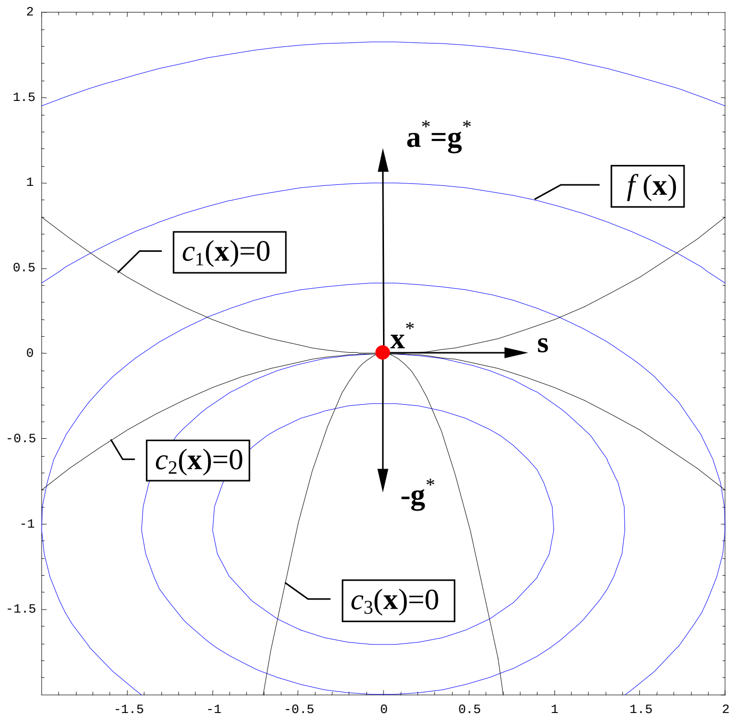



**Figure 3.11:** Illustration of the second order conditions. A problem with three different equality constraints is shown. In all three cases $\mathbf{a}^* = \mathbf{g}^*$ and $\lambda^* = 1$. The problem with constraint function $c_3$ does not match the necessary conditions for $\mathbf{x}^*$ to be a local minimiser because $\mathbf{s}^T\left(\nabla^2 f\left(\mathbf{x}^*\right)\right)\mathbf{s} < \mathbf{s}^T\left(\lambda\nabla^2 c_3\left(\mathbf{x}^*\right)\right)\mathbf{s}$ and thus $\mathbf{s}^T\mathbf{W}_3^*\mathbf{s} < 0$. The second constraint satisfies necessary but not sufficient second order conditions and therefore higher order terms of the Taylor series become significant.

When inequality constraints are present, again only active constraints affect matters. We can also realize that the second order conditions are important only along feasible stationary directions ($\mathbf{s}^T\mathbf{g}^* = 0$ with respect to constraints), but not along ascent directions where first order information is sufficient. If an inequality constraint $c_i(\mathbf{x}) \geq 0$ is present with $\lambda_i^* > 0$, then directions for which $\mathbf{s}^T\mathbf{a}_i^* > 0$ are ascent directions (according to the discussion regarding (3.141)). Stationary directions satisfy

$$\mathbf{s}^T\mathbf{a}_i^* = 0 \quad \forall i \in \mathcal{A}^* \tag{3.152}$$

The second order necessary conditions are then that (3.149) holds for all **s** that satisfy (3.152). Sufficient conditions for a strict local minimiser are that the Kuhn-Tucker conditions with strict complementarity ( $\lambda_i^* > 0\ \forall i \in I^*$ ) hold and

$$\mathbf{s}^T \mathbf{W}^* \mathbf{s} > 0 \quad \forall \mathbf{s} : \mathbf{s}^T \mathbf{a}_i^* = 0, i \in \mathcal{A}^* . \tag{3.153}$$

Exact statement of these conditions[1] again relies on some regularity assumption. Let us define a set of strictly active constraints

$$\mathcal{A}_+^* = \left\{ i; i \in E \vee \lambda_i^* > 0 \right\}. \tag{3.154}$$

Consider feasible directional sequences for which $\mathbf{x}^{(k)} \to \mathbf{x}^*$ for which

$$c_i\left(\mathbf{x}^{(k)}\right) = 0 \quad \forall i \in \mathcal{A}_+^* . \tag{3.155}$$

and define $\mathcal{G}^*$ as a set of all resulting feasible directions. A corresponding set where constraints which determine $\mathcal{G}^*$ are linearised can then be defined as

$$G^* = \left\{ \mathbf{s}; \mathbf{s} \neq 0 \wedge \begin{array}{c} \mathbf{a}_i^{*T} \mathbf{s} = 0, i \in \mathcal{A}_+^* \\ \mathbf{a}_i^{*T} \mathbf{s} \geq 0, i \in \mathcal{A}^* \setminus \mathcal{A}_+^* \end{array} \right\}. \tag{3.156}$$

$\mathcal{G}^* \subseteq G^*$ holds and in order to state the second order necessary conditions, the regularity assumption

$$\mathcal{G}^* = G^* \tag{3.157}$$

is made. The second order necessary and sufficient conditions can then be stated as below[1],[4]:

**Theorem 3.9 (second order necessary conditions):**

If $\mathbf{x}^*$ is a constrained local minimiser and if the regularity assumption (3.145) holds, then there exist multipliers $\lambda^*$ such that Theorem 3.8 is valid (i.e. $\mathbf{x}^*$ is a KT point). For any such $\lambda^*$, if also the regularity assumption (3.157) holds, it follows that

$$\mathbf{s}^T \mathbf{W}^* \mathbf{s} \geq 0 \quad \forall \mathbf{s} \in \mathbf{G}^* . \tag{3.158}$$

**Theorem 3.10 (second order sufficient conditions):**

If at $\mathbf{x}^*$ there exist multipliers $\lambda^*$ such that conditions (3.146) hold, and if

$$\mathbf{s}^T \mathbf{W}^* \mathbf{s} > 0 \quad \forall \mathbf{s} \in G^*, \tag{3.159}$$

then $\mathbf{x}^*$ is a strict local minimiser.

### 3.8.3 Convex Programming Results

Some strong theoretical results hold when the objective function is a convex function and when the feasible region is a convex set. Within the scope of this work these results are not important because of direct applicability to specific problems, but are important for treatment of subproblems that arise in some optimisation algorithms. Convex programming result are also important for statement of the duality principles, which are employed in the reasoning of some general optimisation algorithms.

By definition, a set $K$ in $\mathbb{R}^n$ is convex if for each pair of points $\mathbf{x}_0, \mathbf{x}_1 \in K$ and for each $\theta \in [0,1]$ also $\mathbf{x}_\theta \in K$ , where

$$\mathbf{x}_\theta = (1-\theta)\mathbf{x}_0 + \theta\, \mathbf{x}_1 . \tag{3.160}$$

An equivalent definition is that for any set of points $\mathbf{x}_0, \mathbf{x}_1, \ldots \mathbf{x}_m \in K$ $\mathbf{x}_\theta \in K$ where

$$\mathbf{x}_\theta = \sum_{i=0}^{m} \theta_i \mathbf{x}_i \;\; and \;\; \sum_{i=0}^{m} \theta_i = 1 \;\; \wedge \;\; \theta_i \geq 0 \;\; \forall i . \tag{3.161}$$

A convex function on a convex set $K$ is a function for which the epigraph is a convex set. The epigraph of a function is the set of points in $\mathbb{R} \times \mathbb{R}^n$ that lies on or above the graph of the function. The equivalent definition of a convex function $f(\mathbf{x})$ is that for any $\mathbf{x}_0, \mathbf{x}_1 \in K$ it follows that

$$f_\theta \leq (1-\theta) f_0 + \theta\, f_1 \quad \forall \theta \in [0,1]. \tag{3.162}$$

The definition of a strictly convex function is similar but with strict inequality in the above equation. If $-f(\mathbf{x})$ is convex then $f(\mathbf{x})$ is said to be concave.

If $f$ is convex and $\mathbb{C}^1$ on an open convex set $K$, then[1] for each pair $\mathbf{x}_0, \mathbf{x}_1 \in K$

$$f_1 \geq f_0 + (\mathbf{x}_1 - \mathbf{x}_0)^T \nabla f_0 . \tag{3.163}$$

This means that a graph of $f$ must lie above or along its linearisation about any point. It immediately follows (by interchanging $\mathbf{x}_0$ and $\mathbf{x}_1$) that

$$(\mathbf{x}_1 - \mathbf{x}_0)^T \nabla f_1 \geq f_1 - f_0 \geq (\mathbf{x}_1 - \mathbf{x}_0)^T \nabla f_0 . \tag{3.164}$$

This corresponds to a statement that the slope of a convex function $f$ is non-decreasing along any line. If $f$ is $\mathbb{C}^2$, this result implies (by taking the limit $\|\mathbf{x}_1 - \mathbf{x}_0\| \to 0$) that $\nabla^2 f(\mathbf{x})$ is positive semi-definite at each $\mathbf{x} \in K$.

A convex programming problem is a problem of minimisation of a convex function on a convex set. Such a problem is

$$\begin{array}{lll} \textit{minimise} & & f(\mathbf{x}) \\ \textit{subject to} & & \mathbf{x} \in K,\ K = \{\ \mathbf{x};\, c_i(\mathbf{x}) \geq 0,\ i = 1,2,\ldots.,m\ \}, \end{array} \tag{3.165}$$

where $f(\mathbf{x})$ is convex on $K$ and constraint functions $c_i(\mathbf{x})$ that define $K$ are concave on $\mathbb{R}^n$. Convexity of $K$ defined as above follows from the fact that an epigraph of any concave function is a convex set, and from a known theorem that intersection of convex sets is a convex set.

The following important results hold for convex programming problems[1]:

**Theorem 3.11:**

Every local solution to a convex programming problem is also a global solution, and the set of global solutions $S$ is convex. If $f(\mathbf{x})$ is also strictly convex on $K$, then the solution is unique.

**Theorem 3.12:**

In the convex programming problem (3.165), if $f(\mathbf{x})$ and $c_i(\mathbf{x})$ are $\mathbb{C}^1$ on $K$ and if the Kuhn-Tucker conditions (3.146) hold at $\mathbf{x}^*$, then $\mathbf{x}^*$ is a global solution to the problem.

### 3.8.4 Duality in Nonlinear Programming

The concept of duality provides a set of rules for transformation of one problem to another. By applying these rules alternative formulation of the problem is obtained, which is sometimes more convenient computationally or has some theoretical significance. The original problem is referred to as the primal and the transformed problem as the dual. Some duality transformations have a symmetry property that the dual of the dual is the primal (i.e. that the transformation applied twice gives the original problem).

Usually some of the variables in the dual correspond to Lagrange multipliers of the primal and take the value $\lambda^*$ at the dual solution. The dual and the primal should be related in the way that the dual has a solution from which the solution of the primal can be derived. Duality transformations of this kind are associated with the convex programming problem as the primal. A set of such duality transformations can be derived from the Wolfe dual whose statement is given in the theorem below[1].

**Theorem 3.13:**

If $\mathbf{x}^*$ solves the primal convex programming problem (3.165), if $f$ and $c_i$ are $\mathbb{C}^1$ functions, and if the regularity assumption (3.145) holds, then $\mathbf{x}^*, \lambda^*$ solve the dual problem

$$\begin{array}{ll} \underset{\mathbf{x},\lambda}{\textit{maximise}} & \mathcal{L}(\mathbf{x},\lambda) \\ \textit{subject to} & \nabla_x \mathcal{L}(\mathbf{x},\lambda) = 0, \quad \lambda \geq 0\,. \end{array} \tag{3.166}$$

The minimum primal and maximum dual function values are equal, i.e. $f^* = \mathcal{L}(\mathbf{x}^*, \lambda^*)$

The Wolfe dual is not symmetric. The dual is not necessarily even a convex programming problem. An advantageous property is that if the primal is unbounded then the dual has inconsistent constraints and therefore does not have a solution. It is possible that the primal has inconsistent constraints, but the dual still has a solution. However if the constraints are linear, then infeasible constraints in the primal imply that the dual is unbounded.

An example of application is the quadratic programming problem

$$\begin{aligned} &\textit{minimise} && \tfrac{1}{2}\mathbf{x}^T\mathbf{G}\mathbf{x}+\mathbf{g}^T\mathbf{x} \\ &\textit{subject to} && \mathbf{A}^T\mathbf{x}\ge\mathbf{b} \end{aligned} \qquad (3.167)$$

where **G** is positive definite. The Wolfe dual is

$$\begin{aligned} &\underset{\mathbf{x},\lambda}{\textit{maximise}} && \tfrac{1}{2}\mathbf{x}^T\mathbf{G}\mathbf{x}+\mathbf{g}^T\mathbf{x}-\lambda^T\left(\mathbf{A}^T\mathbf{x}-\mathbf{b}\right) \\ &\textit{subject to} && \mathbf{G}\mathbf{x}+\mathbf{g}-\mathbf{A}\lambda=0,\quad \lambda\ge 0\,. \end{aligned} \qquad (3.168)$$

The first set of constraints can be used to eliminate $\mathbf{x}$ (i.e. $\mathbf{x}=\mathbf{G}^{-1}(\mathbf{A}\lambda-\mathbf{g})$), which gives the problem

$$\begin{aligned} &\underset{\lambda}{\textit{maximise}} && -\tfrac{1}{2}\lambda^T\left(\mathbf{A}^T\mathbf{G}^{-1}\mathbf{A}\right)\lambda+\lambda^T\left(\mathbf{b}+\mathbf{A}^T\mathbf{G}^{-1}\mathbf{g}\right)-\tfrac{1}{2}\mathbf{g}^T\mathbf{G}^{-1}\mathbf{g} \\ &\textit{subject to} && \lambda\ge 0\,. \end{aligned} \qquad (3.169)$$

This is again a quadratic programming problem, but subject only to simple bounds. When the solution $\lambda^*$ is found, $\mathbf{x}^*$ is obtained by solving the equation used for elimination of $\mathbf{x}$ from (3.168).

## *3.9 Quadratic Programming*

A quadratic programming (QP) problem is an optimisation problem with quadratic objective function and linear constraint functions, i.e.

$$\textit{minimise} \qquad q(\mathbf{x})=\tfrac{1}{2}\mathbf{x}^T\mathbf{G}\mathbf{x}+\mathbf{g}^T\mathbf{x}$$

$$\begin{aligned} &\textit{subject to} && \mathbf{a}_i^T\mathbf{x} = b_i,\ i \in E \\ &\textit{and} && \mathbf{a}_i^T\mathbf{x} \geq b_i,\ i \in I\,. \end{aligned} \tag{3.170}$$

where $\mathbf{G}$ is symmetric. If $\mathbf{G}$ is positive semi-definite, a local solution $\mathbf{x}^*$ is also global, and if $\mathbf{G}$ is positive definite, it is also unique. This follows from Theorem 3.13 since such a problem is a convex programming problem. Only if $\mathbf{G}$ is indefinite can a local solution which is not global occur.

### 3.9.1 Equality Constraints Problem

The quadratic programming problem with only equality constraints can be stated as

$$\begin{aligned} &\textit{minimise} && q(\mathbf{x}) = \tfrac{1}{2}\mathbf{x}^T\mathbf{G}\mathbf{x} + \mathbf{g}^T\mathbf{x} \\ &\textit{subject to} && \mathbf{A}^T\mathbf{x} = \mathbf{b}\,. \end{aligned} \tag{3.171}$$

It will be assumed that there are $m \leq n$ constraints and that $\mathbf{A}$ has rank $m$, which ensures that unique multipliers $\lambda^*$ exist. $\mathbf{A}$ is a $n \times m$ matrix whose columns are vectors $\mathbf{a}_i$, $i \in E$ from (3.170), and $\mathbf{b} \in \mathbb{R}^m$.

The problem can be transformed to an unconstrained minimisation problem by direct elimination of variables using constraints. Let partitions

$$\mathbf{x} = \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \end{bmatrix}, \quad \mathbf{A} = \begin{bmatrix} \mathbf{A}_1 \\ \mathbf{A}_2 \end{bmatrix}, \quad \mathbf{g} = \begin{bmatrix} \mathbf{g}_1 \\ \mathbf{g}_2 \end{bmatrix}, \quad \mathbf{G} = \begin{bmatrix} \mathbf{G}_{11} & \mathbf{G}_{12} \\ \mathbf{G}_{21} & \mathbf{G}_{22} \end{bmatrix} \tag{3.172}$$

be defined, where $\mathbf{x}_1 \in \mathbb{R}^m$ and $\mathbf{x}_2 \in \mathbb{R}^{n-m}$, so that $\mathbf{A}_1$ is $m \times m$, $\mathbf{G}_{11}$ is $m \times m$, etc. Then $m$ variables in the vector $\mathbf{x}_1$ can be expressed in terms of $\mathbf{x}_2$ as

$$\mathbf{x}_1 = \mathbf{A}_1^{-T}\left(\mathbf{b} - \mathbf{A}_2^T\mathbf{x}_2\right). \tag{3.173}$$

Substituting this into $q(\mathbf{x})$ gives

$$\begin{aligned}&\psi(\mathbf{x}_2)=q(\mathbf{x}_1(\mathbf{x}_2),\mathbf{x}_2)=\\&\tfrac{1}{2}\mathbf{x}_2^T\left(\mathbf{G}_{22}-\mathbf{G}_{21}\mathbf{A}_1^{-T}\mathbf{A}_2^T-\mathbf{A}_2\mathbf{A}_1^{-1}\mathbf{G}_{12}+\mathbf{A}_2\mathbf{A}_1^{-1}\mathbf{G}_{11}\mathbf{A}_1^{-T}\mathbf{A}_2^T\right)\mathbf{x}_2+\\&\mathbf{x}_2^T\left(\mathbf{G}_{21}-\mathbf{A}_2\mathbf{A}_1^{-1}\mathbf{G}_{11}\right)\mathbf{A}_1^{-T}\mathbf{b}+\tfrac{1}{2}\mathbf{b}^T\mathbf{A}_1^{-1}\mathbf{G}_{11}\mathbf{A}_1^{-T}\mathbf{b}+\\&\mathbf{x}_2^T\left(\mathbf{g}_2-\mathbf{A}_2\mathbf{A}_1^{-1}\mathbf{g}_1\right)+\mathbf{g}_1^T\mathbf{A}_1^{-T}\mathbf{b}\end{aligned} \quad . \qquad (3.174)$$

The problem is so transformed to unconstrained minimisation of $\psi(\mathbf{x}_2)$. If the Hessian (the matrix in the round brackets in the second line) is positive definite, then a unique minimiser $\mathbf{x}_2^*$ exists and is obtained by solving the linear system of equations $\nabla\psi(\mathbf{x}_2)=0$. $\mathbf{x}_1^*$ is obtained by substitution in (3.173). The Lagrange multiplier vector is defined by $\mathbf{g}^*=\mathbf{A}\lambda^*$ where $\mathbf{g}^*=\nabla q(\mathbf{x}^*)=\mathbf{g}+\mathbf{G}\mathbf{x}^*$, and can be calculated by solving the first partition $\mathbf{g}_1^*=\mathbf{A}_1\lambda^*$.

The described approach is not the only possibility. First of all, it is possible to rearrange variables and choose some other set of variables to be independent. More generally a linear transformation of variables can be made. Such a general approach is the generalised elimination method.

Let $\mathbf{Y}$ and $\mathbf{Z}$ be $n\times m$ and $n\times(n-m)$ matrices such that $[\mathbf{Y}:\mathbf{Z}]$ is non-singular and

$$\begin{aligned}\mathbf{A}^T\mathbf{Y}&=\mathbf{I}_{m\times m},\\ \mathbf{A}^T\mathbf{Z}&=\mathbf{0}_{m\times n-m}.\end{aligned} \qquad (3.175)$$

$\mathbf{Y}^T$ can be regarded as the left generalised inverse of $\mathbf{A}$ since a solution of the system $\mathbf{A}^T\mathbf{x}=\mathbf{b}$ is given by $\mathbf{x}=\mathbf{Y}\mathbf{b}$. The solution is not unique and other solutions are given by $\mathbf{x}=\mathbf{Y}\mathbf{b}+\delta$ where $\delta$ is in the $n-m$ - dimensional null column space of **A**, i.e.

$$\mathbf{A}^T\delta=0 \qquad (3.176)$$

If the matrix **Z** has linearly independent columns $\mathbf{z}_1,\mathbf{z}_2,\ldots,\mathbf{z}_{\mathbf{n-m}}$, then these vectors form a basis of the null space of **A**[29]. At any feasible point **x** (i.e. solution of $\mathbf{A}\mathbf{x}=\mathbf{b}$) any feasible correction $\delta$ (which gives another solution) can be written as

$$\delta=\sum_{i=1}^{n-m}y_i\mathbf{z}_i\ , \qquad (3.177)$$

where $y_1, y_2, ..., y_{n-m}$ are the components in the null space of **A**, referred to also as reduced variables in this space. Any feasible point can be written as

$$\mathbf{x} = \mathbf{Y}\mathbf{b} + \mathbf{Z}\mathbf{y} \, . \tag{3.178}$$

The above equation provides a way of eliminating constraints $\mathbf{A}\mathbf{x} = \mathbf{b}$ by transformation to the $n-m$ - dimensional space of reduced variables in which constraints are always satisfied, and is therefore a generalisation of (3.173). Substituting the equation into (3.171) gives the reduced quadratic function

$$\psi(\mathbf{x}) = \tfrac{1}{2}\mathbf{y}^T\mathbf{Z}^T\mathbf{G}\mathbf{Z}\mathbf{y} + (\mathbf{g} + \mathbf{G}\mathbf{Y}\mathbf{b})^T\mathbf{Z}\mathbf{y} + \tfrac{1}{2}(\mathbf{g} + \mathbf{G}\mathbf{Y}\mathbf{b})^T\mathbf{Y}\mathbf{b} \, . \tag{3.179}$$

If the reduced Hessian matrix $\mathbf{Z}^T\mathbf{G}\mathbf{Z}$ is positive definite then a unique solution exists and can be obtained by solution of the system $\nabla\psi(\mathbf{y}) = 0$, i.e.

$$(\mathbf{Z}^T\mathbf{G}\mathbf{Z})\mathbf{y} = -\mathbf{Z}^T(\mathbf{g} + \mathbf{G}\mathbf{Y}\mathbf{b}). \tag{3.180}$$

It is convenient to solve this system by Choleski factorisation[30],[33], which also enables positive definiteness to be checked. $\mathbf{x}^*$ is then obtained from $\mathbf{y}^*$ by using (3.178). Lagrangian multipliers are obtained from $\mathbf{g}^* = \mathbf{A}\lambda^*$, which after pre-multiplying by $\mathbf{Y}^T$ gives

$$\lambda^* = \mathbf{Y}^T\mathbf{g}^* = \mathbf{Y}^T(\mathbf{G}\mathbf{x}^* + \mathbf{g}). \tag{3.181}$$

Note that $\mathbf{g}$ in this equation does not refer to the gradient vector, but is a constant vector in the definition of $q(\mathbf{x})$. The reduced gradient vector is $\mathbf{Z}^T(\mathbf{g} + \mathbf{G}\mathbf{Y}\mathbf{b})$. This shows that the reduced derivatives can be obtained by pre-multiplication by $\mathbf{Z}^T$, since $\mathbf{g} + \mathbf{G}\mathbf{Y}\mathbf{b} = \nabla q(\mathbf{Y}\mathbf{b})$ is the gradient of $q(\mathbf{x})$ at $\mathbf{x} = \mathbf{Y}\mathbf{b}$.

Different methods arise from different choice of **Y** and **Z**. It is convenient to use any orthogonal (QR) factorization[26]-[28] of **A**:

$$\mathbf{A} = \mathbf{Q}\begin{bmatrix}\mathbf{R}\\ \mathbf{0}\end{bmatrix} = [\mathbf{Q}_1 \; \mathbf{Q}_2]\begin{bmatrix}\mathbf{R}\\ \mathbf{0}\end{bmatrix} = \mathbf{Q}_1\mathbf{R} \, , \tag{3.182}$$

where **Q** is a $n \times n$ orthogonal matrix, **R** is a $m \times m$ upper triangular matrix, and $\mathbf{Q}_1$ and $\mathbf{Q}_2$ are $n \times m$ and $n \times (n-m)$ partitions of **Q**. Then we can choose

$$\mathbf{Y} = \mathbf{Q}_1\mathbf{R}^{-T}, \quad \mathbf{Z} = \mathbf{Q}_2 \, . \tag{3.183}$$

This implies that the vector $\mathbf{Yb}$ from (3.178) is orthogonal to any feasible change $\delta$ and the reduced coordinate directions $\mathbf{z}_i$ are mutually orthogonal.

The reduced system (3.180) is first solved to obtain $\mathbf{y}^*$, and then $\mathbf{x}^*$ is calculated by substitution into (3.178). Numerically it is most convenient to evaluate vector $\mathbf{Yb}$, which appears in these equations, by forward substitution in $\mathbf{R}^T\mathbf{u} = \mathbf{b}$ (since $\mathbf{R}$ is upper triangular) followed by multiplication $\mathbf{Yb} = \mathbf{Q}_1\mathbf{u}$. Multipliers $\lambda^*$ are then calculated by backward substitution in $\mathbf{R}\lambda^* = \mathbf{Q}_1^T\mathbf{g}^*$. Such a scheme is referred to as the orthogonal factorization method. Its advantage is that because of using orthogonal transformations, the method is less sensitive to round-off errors[27].

In general, **Y** and **Z** can be obtained by completion of the matrix **A** to a full-rank $n \times n$ matrix and partitioning of the inverse of that matrix. For example, we can choose any $n \times (n-m)$ matrix **V** such that the matrix $[\mathbf{A} : \mathbf{V}]$ is non-singular. **Y** and **Z** are then obtained by

$$[\mathbf{A} : \mathbf{V}]^{-1} = \begin{bmatrix} \mathbf{Y}^T \\ \mathbf{Z}^T \end{bmatrix}, \tag{3.184}$$

where **Y** and **Z** are $n \times m$ and $n \times (n-m)$ partitions respectively. They satisfy conditions (3.175) and are therefore suitable for use in the generalised elimination method. The resulting method can be interpreted as a method which makes linear transformation with the matrix $[\mathbf{A} : \mathbf{V}]$.

Different methods arise from specific choices of **V**. Choosing

$$\mathbf{V} = \begin{bmatrix} \mathbf{0} \\ \mathbf{I} \end{bmatrix} \tag{3.185}$$

results in the direct elimination method. The identity

$$\begin{bmatrix} \mathbf{A}_1 & \mathbf{0} \\ \mathbf{A}_2 & \mathbf{I} \end{bmatrix}^{-1} = \begin{bmatrix} \mathbf{A}_1^{-1} & \mathbf{0} \\ -\mathbf{A}_2\mathbf{A}_1^{-1} & \mathbf{I} \end{bmatrix} = \begin{bmatrix} \mathbf{Y}^T \\ \mathbf{Z}^T \end{bmatrix} \tag{3.186}$$

gives expressions for **Y** and **Z**. It can then be verified by substitution into (3.180) and taking into account the appropriate partitioning that the resulting method is identical to the direct elimination method.

The orthogonal factorization method is obtained by setting

$$\mathbf{V} = \mathbf{Q}_2, \tag{3.187}$$

where $\mathbf{Q}_2$ is defined by (3.182). By using the identity

$$[\mathbf{A} : \mathbf{V}]^{-1} = [\mathbf{Q}_1\mathbf{R} : \mathbf{Q}_2]^{-1} = \begin{bmatrix} \mathbf{R}^{-1}\mathbf{Q}_1^T \\ \mathbf{Q}_2^T \end{bmatrix} = \begin{bmatrix} \mathbf{Y}^T \\ \mathbf{Z}^T \end{bmatrix} \tag{3.188}$$

(3.183) is obtained, which confirms that the orthogonal factorization method was obtained. The above equation can be expressed as

$$[\mathbf{A} : \mathbf{V}]^{-1} = \begin{bmatrix} \mathbf{A}^+ \\ \mathbf{V}^+ \end{bmatrix} \tag{3.189}$$

where $\mathbf{A}^+ = \left(\mathbf{A}^T\mathbf{A}\right)^{-1}\mathbf{A}^T$ is the full rank generalised inverse, therefore $\mathbf{Y} = \mathbf{A}^{+T}$ from (3.183).

### 3.9.2 Active Set Methods

Inequality constraints can not be simply used to eliminate variables or reduce the dimension of the problem. Only those inequality constraints that are active in the solution actually affect matters. If it would be known in advance which constraints are active in the solution, these constraints could be used as equality constraints and all other constraints could be ignored. Active set methods gradually update the set of active constraints and solve the resulting equality constrained problems where constraints regarded as inactive are temporarily ignored. It is assumed that the Hessian matrix of the problem is positive definite. The basic idea is illustrated in Figure 3.12 and described below .

On the *k*-th iteration a feasible point $\mathbf{x}^{(k)}$ is known which satisfies active constraints as equalities, i.e. $\mathbf{a}_i^T\mathbf{x}^{(k)} = b_i \;\; \forall i \in \mathcal{A}$ where $\mathcal{A}$ is the index set of constraints currently regarded as active and treated as equality constraints. All equality constraints are in this set. $\mathbf{x}^{(k)}$ also satisfies $\mathbf{a}_i^T\mathbf{x}^{(k)} > b_i \;\; \forall i \notin \mathcal{A}$, so that the current active set $\mathcal{A}$ is equivalent to the set of active constraints $\mathcal{A}^{(k)}$.

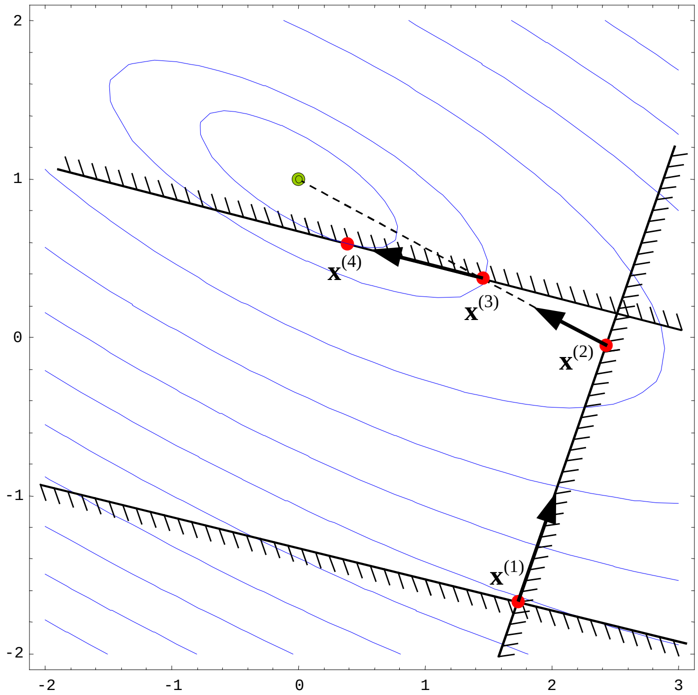



**Figure 3.12:** Progress of the active set method in a problem with three inequality constraints.

The iteration attempts to solve the equality problem where only active constraints occur. By shifting the origin to $\mathbf{x}^{(k)}$ and looking for a correction $\delta^{(k)}$ this problem is

$$\begin{aligned} &\textit{minimise} && \tfrac{1}{2}\delta^T \mathbf{G}\delta + \delta^T \mathbf{g}^{(k)} \\ &\textit{subject to} && \mathbf{a}^T \delta = 0 \quad \forall\, i \in \mathcal{A}, \end{aligned} \tag{3.190}$$

where $\mathbf{g}^{(k)} = \nabla q\left(\mathbf{x}^{(k)}\right) = \mathbf{g} + \mathbf{G}\mathbf{x}^{(k)}$.

If $\delta$ is feasible with respect to constraints not in $\mathcal{A}$, then $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \delta$ is taken. If not a line search is made in the direction $\mathbf{s}^{(k)} = \delta^{(k)}$ to find the best feasible point. $\alpha^{(k)}$ is found which solves

$$\alpha^{(k)} = \min\left(1, \min_{\substack{i \notin \mathcal{A} \wedge \\ \mathbf{a}_i^T \mathbf{s}^{(k)} < 0}} \frac{b_i - \mathbf{a}_i^T \mathbf{x}^{(k)}}{\mathbf{a}_i^T \mathbf{s}^{(k)}}\right) \tag{3.191}$$

and $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha^{(k)}\mathbf{s}^{(k)}$ is set.

If $\alpha^{(k)} < 1$ then a new constraint (with index *p*, say) which achieves the minimum in the above equation becomes active and its index *p* is added to the active set $\mathcal{A}$.

If $\mathbf{x}^{(k)}$ solves the current equality problem, then it is possible to compute multipliers $\lambda^{(k)}$ as described in the previous section. Vectors $\mathbf{x}^{(k)}$ and $\lambda^{(k)}$ satisfy all the first order conditions for the original inequality constrained problem except possibly the conditions $\lambda_i \geq 0, i \in I$. The test is therefore made if these conditions are satisfied for all inequality constraints in $\mathcal{A}$. If so, the first order conditions are satisfied and since the problem is convex (because **G** is positive definite), this is sufficient for $\mathbf{x}^{(k)}$ to be a global solution. Otherwise there exists an index *q* such that $\lambda_q^{(k)} < 0$. In this case it is possible to reduce $q(\mathbf{x})$ by allowing constraint *q* to become inactive (according to discussion around equation (3.141)). Constraint *q* is therefore removed from $\mathcal{A}$ and the algorithm continues as before. It is possible that there are more than one indices with $\lambda_j^{(k)} < 0$. Then *q* is selected so that it solves

$$\min_{i \in \mathcal{A} \cap I} \lambda_i^{(k)}. \tag{3.192}$$

The complete algorithm is outlined below.

**Algorithm 3.10:** The active set method.

A feasible point $\mathbf{x}^{(1)}$ must be given. $\mathcal{A} = \mathcal{A}^{(1)}$ is set where $\mathcal{A}^{(1)}$ contains indices of all constraints for which $c_i\left(\mathbf{x}^{(1)}\right) = 0$. The *k*-th iteration is then as follows:

1. If $\delta = \mathbf{0}$ does not solve (3.190) then go to 3.
2. Compute Lagrange multipliers $\lambda^{(k)}$ and solve (3.192). If $\lambda_q^{(k)} \geq 0$ then terminate with $\mathbf{x}^* = \mathbf{x}^{(k)}$, otherwise remove $q$ from $\mathcal{A}$.
3. Solve (3.190) for $\mathbf{s}^{(k)}$.
4. Solve (3.191) to find $\alpha^{(k)}$ and set $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha^{(k)}\mathbf{s}^{(k)}$.
5. If $\alpha^{(k)} < 1$, add $p$ to $\mathcal{A}$.
6. Set $k = k + 1$ and go to 1.

The initial feasible point can be obtained from any given point $\mathbf{x}^{(0)}$ by iteratively solving the problem

$$\begin{aligned} &\textit{minimise} && \sum_{i \in V^{(k)}} \left( b_i - \mathbf{a}_i^T \mathbf{x} \right) \\ &\textit{subject to} && \mathbf{a}_i^T \mathbf{x} \ge b_i \quad \forall\, i \notin V^{(k)}, \end{aligned} \tag{3.193}$$

where $V^{(k)}$ is the set of infeasible constraints at $\mathbf{x}^{(k)}$. Iteration is repeated until $\mathbf{x}^{(k)}$ becomes a feasible point. Minimisations are performed as line searches along edges $\mathbf{s}^{(k)} = \mathbf{a}_q^{(k)}$ where $q^{(k)}$ is the index with the least Lagrange multiplier in iteration *k*. Each search terminates with a new constraint becoming active[1].

So far it was assumed that the Hessian matrix **G** is positive definite. If **G** is indefinite then local solutions exist which are not global. For any local solution the reduced Hessian matrix $\mathbf{Z}^T\mathbf{G}\mathbf{Z}$ is positive semi-definite and this matrix is actually used when the equality problem is solved. However, when the algorithm proceeds, not necessarily all constraints that are active in the solution are in the active set. Therefore problem (3.190) with indefinite reduced Hessian can arise. In this case a solution of (3.190) $\delta^{(k)}$ is no longer a minimiser. Any feasible descent direction can be chosen for $\mathbf{s}^{(k)}$, for example the negative reduced gradient vector. $\alpha^{(k)}$ is then obtained from

$$\alpha^{(k)} = \min_{\substack{i \notin \mathcal{A},\\ \mathbf{a}_i^T \mathbf{s}^{(k)} < 0}} \frac{b_i - \mathbf{a}_i^T \mathbf{x}^{(k)}}{\mathbf{a}_i^T \mathbf{s}^{(k)}} \tag{3.194}$$

rather than from (3.191). If the above equation does not have a solution (i.e. the infimum of the right-hand side is $-\infty$), this indicates that the original QP problem is unbounded.

## *3.10 Penalty Methods*

Penalty methods[6] are a traditional and commonly used approach to constrained minimisation. The idea of penalty methods is to control constraint violations by penalizing them. The original objective function is modified by addition of penalty terms, which monotonically increase as constraint violations increase. The sum of the objective function and penalty terms is called the penalty function. Some parameter is usually associated with penalty terms to control the amount of the penalty. The minimiser of the objective function is approximated by unconstrained minimisers of the penalty function, which should converge to the constrained minimiser as the control parameter is increased (Figure 3.13).

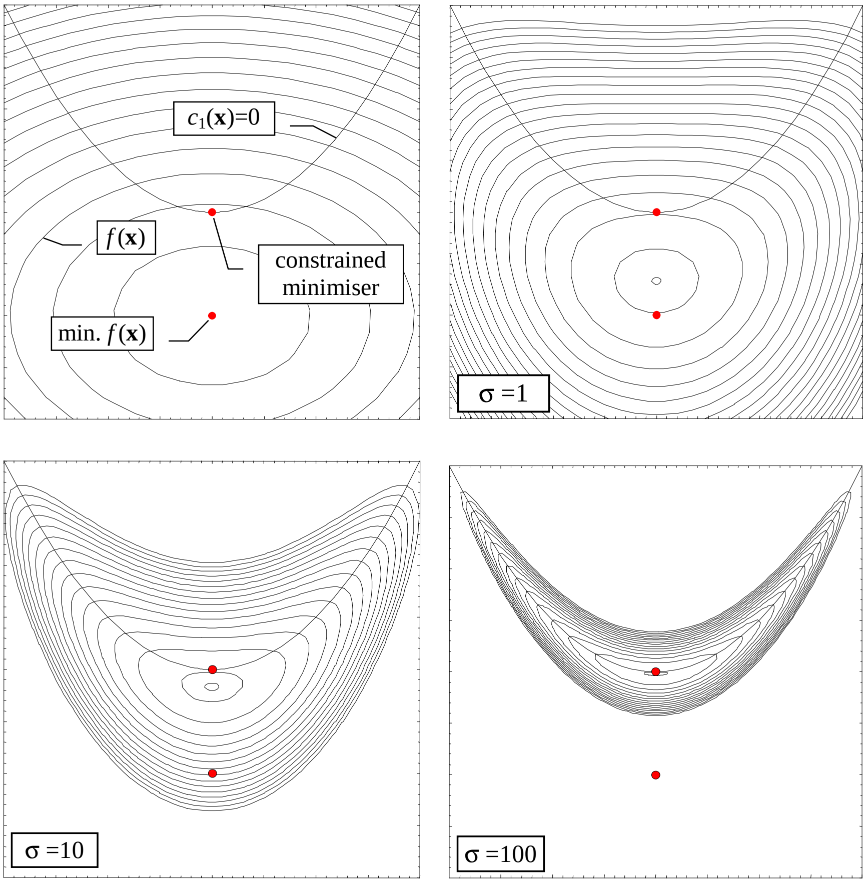



**Figure 3.13:** Use of penalty functions. The problem with one equality constraint is sketched in the first picture. The subsequent pictures show penalty function contours with increasing parameter $\sigma$. The minimiser of the penalty functions approaches the problem solution as $\sigma$ increases, but also ill-conditioning increases.

The following penalty function can be used for equality constraints:

$$\phi(\mathbf{x},\sigma) = f(\mathbf{x}) + \frac{1}{2}\sigma\sum_{i\in E}\left(c_i^{\;2}\right) = f(\mathbf{x}) + \frac{1}{2}\sigma\sum_{i\in E}\mathbf{c}(\mathbf{x})^T\mathbf{c}(\mathbf{x}). \qquad (3.195)$$

Parameter $\sigma$ determines the amount of the penalty. A simple penalty algorithm is outlined below.

**Algorithm 3.11:** The penalty algorithm.

1. Choose a fixed sequence $\{\sigma^{(k)}\} \to \infty$, e.g. $\{1,10,100,1000,...\}$.
2. Find a local minimiser $\mathbf{x}(\sigma^{(k)})$ of $\phi(\mathbf{x},\sigma^{(k)})$, using a minimiser of the previous iteration as a starting guess.
3. Terminate if $\mathbf{c}(\mathbf{x}(\sigma^{(k)}))$ is sufficiently small, otherwise go to 2.

The quantities associated with $\sigma^{(k)}$ will be denoted by upper index *k*, e.g. $\mathbf{x}(\sigma^{(k)}) = \mathbf{x}^{(k)}$, $f(\mathbf{x}(\sigma^{(k)})) = f^{(k)}$, etc. The following convergence result holds for such an algorithm[6]:

**Theorem 3.14** (penalty function convergence):

Let $f(\mathbf{x})$ be bounded below on a non-empty feasible region and let global minimisers be evaluated in step 2 of the above algorithm. If $\sigma^{(k)} \to \infty$ monotonically, then $\{\phi^{(k)}(\mathbf{x}^{(k)},\sigma^{(k)})\}$, $\{\mathbf{c}^{(k)T}\mathbf{c}^{(k)}\}$ and $\{f^{(k)}\}$ are non-decreasing, $\mathbf{c}^{(k)} \to 0$ and any accumulation point $\mathbf{x}^*$ of $\{\mathbf{x}^{(k)}\}$ solves the equality constrained problem.

The algorithm has some other limiting properties, which enable useful estimations to be made and are gathered in the theorem below.

**Theorem 3.15** (penalty function convergence):

If $\sigma^{(k)} \to \infty$, $\mathbf{x}^{(k)} \to \mathbf{x}^*$ and $rank\,\mathbf{A}^* = m$ (m is the number of constraints), then $\mathbf{x}^*$ is a KT point and the following hold:

$$\lambda^{(k)} = -\sigma^{(k)}\mathbf{c}^{(k)} = \lambda^* + o(1), \tag{3.196}$$

$$f^* = \phi^{(*)} = \phi^{(k)} + \tfrac{1}{2}\sigma^{(k)}\mathbf{c}^{(k)T}\mathbf{c}^{(k)} + o(1/\sigma), \tag{3.197}$$

$$\mathbf{h}^{(k)} = \frac{-\mathbf{T}^*\lambda^*}{\sigma^{(k)}} + o(1/\sigma), \tag{3.198}$$

where $\mathbf{T}$ is defined by

$$\begin{bmatrix} \mathbf{W}^* & -\mathbf{A}^* \\ -\mathbf{A}^{*T} & \mathbf{0} \end{bmatrix}^{-1} = \begin{bmatrix} \mathbf{H}^* & -\mathbf{T}^* \\ -\mathbf{T}^{*T} & \mathbf{U}^* \end{bmatrix}, \tag{3.199}$$

$\mathbf{W}$ is the Hessian matrix of the Lagrangian function and $\mathbf{A}$ is the Jacobian matrix of constraints. Notation $a = o(h) \Leftrightarrow a/h \to 0$ has been used.

These results enable some enhancements of the algorithm to be made. (3.196) gives an estimation of the Lagrange multipliers and (3.197) can be used as a better estimation to $f^*$ than $\phi^{(k)}$ itself. (3.199) can be used to terminate the iteration and to provide better initial approximations when minimising $\phi\left(\mathbf{x}, \sigma^{(k)}\right)$.

For inequality constraint problems the following penalty function can be used:

$$\phi(\mathbf{x}, \sigma) = f(\mathbf{x}) + \frac{1}{2}\sigma \sum_{i \in I} \left(\min(0, c_i(\mathbf{x}))\right)^2 . \tag{3.200}$$

A disadvantage of this penalty function is the jump discontinuity in second derivatives where $c_i(\mathbf{x}) = 0$. $\mathbf{x}^{(k)}$ approaches $\mathbf{x}^*$ from the infeasible side of the inequality constraints, therefore algorithms that use such a penalty function are called exterior point algorithms.

Another class of algorithms for inequality constraints are barrier function methods. Barrier terms, which are infinite on the constraint boundaries are added to the penalty function. These algorithms preserve strict constraint feasibility in all iterations and are therefore called interior point algorithms. Their use is advantageous when the objective function is not defined in the infeasible region.

Two commonly used barrier functions are the inverse barrier function

$$\phi(\mathbf{x}, r) = f(\mathbf{x}) + r \sum_{i \in I} \frac{1}{c_i(\mathbf{x})} \tag{3.201}$$

and the logarithmic barrier function

$$\phi(\mathbf{x}, r) = f(\mathbf{x}) - r \sum_{i \in I} \ln(c_i(\mathbf{x})). \tag{3.202}$$

A sequence $\{r^{(k)}\} \to 0$ is chosen, which ensures that the barrier terms become more and more negligible as compared to the objective function, except close to the constraint boundary. Sequential minimisation of the penalty functions is performed as in Algorithm 3.11.

Penalty and barrier approaches have a simple extension for problems with mixed equality and inequality constraints. Mixed penalty terms for equality constraints and penalty or barrier terms for inequality constraints are added to the objective function for corresponding constraints[6],[14].

The described algorithms are linearly convergent. A difficulty associated with the penalty and barrier approach is that when the control parameter $\sigma$ is increased (or $r$ decreased, respectively), the Hessian of the penalty (or barrier) function becomes increasingly ill-posed, which is evidently illustrated in Figure 3.13. It is therefore difficult to find accurate solutions of the subsequent unconstrained minimisation problems. The additional problem with barrier functions is that they are not defined in the infeasible region, which can be difficult to handle for minimisation algorithms.

### 3.10.1 Multiplier Penalty Functions

The multiplier penalty functions represent an attempt to use penalty functions adequately by keeping the control parameter $\sigma$ finite and thus to avoid ill-conditioning of the penalty function when $\sigma$ is large. The approach follows from the idea that the constrained minimiser $\mathbf{x}^*$ can be made an unconstrained minimiser of $\phi(\mathbf{x},\sigma)$ by changing the origin of the penalty terms. This leads to the penalty function

$$\begin{aligned}\phi(\mathbf{x},\theta,\sigma) &= f(\mathbf{x}) + \tfrac{1}{2}\sum_{i\in E}\sigma_i\left(c_i(\mathbf{x})-\theta_i\right)^2 = \\ & f(\mathbf{x}) + \tfrac{1}{2}\left(\mathbf{c}(\mathbf{x})-\theta\right)^T \mathbf{S}\left(\mathbf{c}(\mathbf{x})-\theta\right)\end{aligned}, \quad (3.203)$$

where $\theta,\sigma \in \mathbb{R}^n$ and $\mathbf{S} = diag(\sigma_i)$ is a diagonal matrix with $S_{ii} = \sigma_i$, and the equality constrained problem is considered. The aim of the algorithm is to find the optimal shift of the origin $\theta$ such that a minimiser of $\phi(\mathbf{x},\theta,\sigma)$ with respect to variables $\mathbf{x}$ will correspond to the constrained minimiser $\mathbf{x}^*$.

Let us introduce different parameters

$$\lambda_i = \theta_i \sigma_i, \quad i = 1, 2, ..., m. \tag{3.204}$$

If we ignore the term $\frac{1}{2}\sum \sigma_i \theta_i^2$, which is independent of $\mathbf{x}$ and therefore does not affect the minimiser, $\phi$ becomes

$$\phi(\mathbf{x}, \lambda, \sigma) = f(\mathbf{x}) - \lambda^T \mathbf{c}(\mathbf{x}) + \frac{1}{2}\mathbf{c}(\mathbf{x})^T \mathbf{S}\,\mathbf{c}(\mathbf{x}). \tag{3.205}$$

Because the above function is obtained from (3.195) by adding a multiplier term $-\lambda^T \mathbf{c}$, it is referred to as the multiplier penalty function[1]. There exists optimum values of multipliers $\lambda$, for which $\mathbf{x}^*$ minimises $\phi(\mathbf{x}, \lambda, \sigma)$. It turns out that these values are the Lagrange multipliers $\lambda^*$ at the solution, provided that parameters $\sigma_i$ are large enough. An exact formulation of this is given in the theorem below[1].

**Theorem 3.16:**

If second order sufficient conditions for a constrained local minimum hold at $\mathbf{x}^*$, $\lambda^*$, then there exists $\sigma' \geq \mathbf{0}$ (i.e. $\sigma_i' \geq 0 \; \forall i$) such that for any $\sigma > \sigma'$ $\mathbf{x}^*$ is an isolated local minimiser of $\phi(\mathbf{x}, \lambda^*, \sigma)$, i.e. $\mathbf{x}^* = \mathbf{x}(\lambda^*)$.

Illustration of the multiplier penalty function is shown in Figure 3.14. This is done for the same problem as in Figure 3.13, so that the multiplier penalty function can be compared to the standard penalty function. The optimal value $\lambda^*$ was used in the figure and both values of $\sigma_1$ were sufficiently large, so that the minimum of the penalty function corresponds to the solution of the original equality constrained problem.

The Lagrange multipliers at the solution of the original problem are not known in advance, therefore a method for generating a sequence $\lambda^{(k)} \to \lambda^*$ must be incorporated in the algorithm.

[1] The term augmented Lagrangian function is also used, since the function can be considered as the Lagrangian function where *f* is augmented by the term $\frac{1}{2}\mathbf{c}(\mathbf{x})^T \mathbf{S}\,\mathbf{c}(\mathbf{x})$.

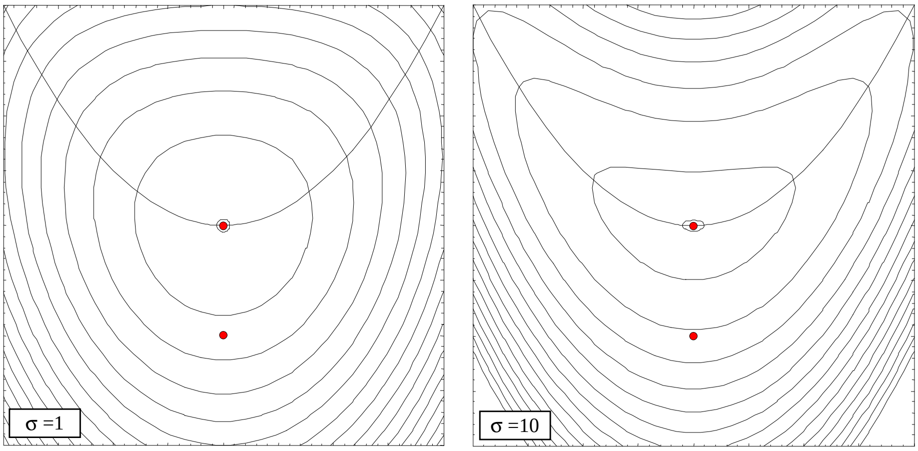



**Figure 3.14:** Multiplier penalty functions for problem illustrated in Figure 3.13. Minimiser of the multiplier penalty functions corresponds to the constrained minimiser even for a smaller value $\sigma_1 = 1$.

To construct such a method it is assumed that the second order sufficient conditions hold at $\mathbf{x}^*$ and that components of vector $\sigma$ are sufficiently large. Consider $\mathbf{x}$ being implicitly dependent on $\lambda$ in a way that $\mathbf{x}(\lambda)$ is a minimiser of $\phi(\mathbf{x},\lambda)$. Since $\mathbf{x}^* = \mathbf{x}(\lambda^*)$ is by Theorem 3.16 an isolated local minimiser of $\phi(\mathbf{x},\lambda^*)$, $\mathbf{x}(\lambda)$ is defined uniquely in some neighbourhood $\Omega_\lambda$ of $\lambda^*$. $\mathbf{x}(\lambda)$ can be determined by solving the equations

$$\nabla\phi(\mathbf{x},\lambda) = 0. \tag{3.206}$$

Consider the function

$$\psi(\lambda) = \phi(\mathbf{x}(\lambda),\lambda). \tag{3.207}$$

Since $\mathbf{x}(\lambda)$ is a local minimum of $\phi(\mathbf{x},\lambda)$, it follows that

$$\psi(\lambda) = \phi(\mathbf{x}(\lambda),\lambda) \le \phi(\mathbf{x}^*,\lambda) = \phi(\mathbf{x}^*,\lambda^*) = \psi(\lambda^*), \tag{3.208}$$

where $\phi(\mathbf{x}^*,\lambda) = \phi(\mathbf{x}^*,\lambda^*)$ is obtained by using $\mathbf{c}^* = 0$ (feasibility) in (3.205). We have $\psi(\lambda) \le \psi(\lambda^*)$, therefore $\lambda^*$ is a local unconstrained maximiser of $\psi(\lambda)$, and this is true globally if $\mathbf{x}(\lambda)$ is a global minimiser of $\phi(\mathbf{x},\lambda)$. A sequence $\lambda^{(k)} \to \lambda^*$

can be generated by applying an unconstrained minimisation method to $-\psi(\lambda)$, for which derivatives of $\psi$ with respect to $\lambda$ are needed.

Derivatives of $\phi$ with respect to $\mathbf{x}$ are obtained from (3.205):

$$\nabla\phi(\mathbf{x},\lambda,\sigma) = \mathbf{g} - \mathbf{A}\,\lambda + \mathbf{ASc} \tag{3.209}$$

and

$$\mathbf{W}_\sigma = \nabla^2\phi(\mathbf{x},\lambda,\sigma) = \nabla^2 f - \sum_{i\in E}(\lambda_i - \sigma_i c_i)\nabla^2 c_i + \mathbf{ASA}^T\,. \tag{3.210}$$

By the chain rule we have

$$\frac{d\psi}{d\lambda} = \frac{\partial\psi}{\partial\mathbf{x}}\frac{\partial\mathbf{x}}{\partial\lambda} + \frac{\partial\phi}{\partial\lambda},$$

and since $\partial\phi/\partial\mathbf{x} = \mathbf{0}$ from (3.206) and $\partial\phi/\partial\lambda_i = -c_i$ from (3.205), it follows that

$$\nabla_\lambda\psi\lambda = -\mathbf{c}(\mathbf{x}(\lambda)). \tag{3.211}$$

By the chain rule we then have

$$\frac{d\mathbf{c}}{d\lambda} = \frac{\partial\mathbf{c}}{\partial\mathbf{x}}\frac{\partial\mathbf{x}}{\partial\lambda} = \mathbf{A}^T\,\frac{\partial\mathbf{x}}{\partial\lambda}\,.$$

Applying $d/d\lambda$ to (3.206) gives

$$\frac{d(\nabla\phi)}{d\lambda} = \frac{\partial(\nabla\phi)}{\partial\mathbf{x}}\frac{\partial\mathbf{x}}{\partial\lambda} + \frac{\partial(\nabla\phi)}{\partial\lambda} = \mathbf{0}\,.$$

$\partial(\nabla\phi)/\partial\mathbf{x} = \nabla^2\phi = \mathbf{W}_\sigma$ and $\partial(\nabla\phi)/\partial\lambda = -\mathbf{A}$ from (3.209), therefore $\dfrac{\partial\mathbf{x}}{\partial\lambda} = \mathbf{W}_\sigma^{-1}\mathbf{A}$ and

$$\nabla_\lambda^2\psi(\lambda) = -\frac{d\mathbf{c}}{d\lambda} = -\mathbf{A}^T\mathbf{W}_\sigma^{-1}\mathbf{A}\Big|_{\mathbf{x}(\lambda)}\,. \tag{3.212}$$

The sequence $\lambda^{(k)} \to \lambda^*$ can be obtained by applying Newton's method from some initial estimate $\lambda^{(1)}$, which gives

$$\lambda^{(k+1)} = \lambda^{(k)} - \left(\left(\mathbf{A}^T \mathbf{W}_\sigma^{-1} \mathbf{A}\right)^{-1} \mathbf{c}\right)\Big|_{\mathbf{x}\left(\lambda^{(k)}\right)}, \tag{3.213}$$

which requires second derivatives of $f$ and **c**. When only first derivatives are available, a quasi-Newton method can be used to find $\mathbf{x}\left(\lambda^{(k)}\right)$ and the resulting **H** matrix can be used to approximate $\mathbf{W}_\sigma^{-1}$ in the above equation, i.e.

$$\lambda^{(k+1)} = \lambda^{(k)} - \left(\left(\mathbf{A}^T \mathbf{H} \mathbf{A}\right)^{-1} \mathbf{c}\right)\Big|_{\mathbf{x}\left(\lambda^{(k)}\right)}, \tag{3.214}$$

An algorithm that uses the derived results is described below.

**Algorithm 3.12:** The Multiplier penalty algorithm.

1. Set $\lambda = \lambda^{(1)}$, $\sigma = \sigma^{(1)}$, $k = 0$ and $\left\|\mathbf{c}^{(0)}\right\|_\infty = \infty$.
2. Find the minimiser $\mathbf{x}(\lambda, \sigma)$ of $\phi(\mathbf{x}, \lambda, \sigma)$ and evaluate $\mathbf{c} = \mathbf{c}(\mathbf{x}(\lambda, \sigma))$.
3. If $\|\mathbf{c}\|_\infty > \frac{1}{4}\left\|\mathbf{c}^{(k)}\right\|_\infty$ then set $\sigma_i = 10\sigma_i \ \forall i : |c_i| > \frac{1}{4}\|\mathbf{c}\|_\infty$ and go to 2.
4. Set $k = k + 1$, $\lambda^{(k)} = \lambda$, $\sigma^{(k)} = \sigma$ and $\mathbf{c}^{(k)} = \mathbf{c}$.
5. Evaluate $\lambda^{(k)}$ according to (3.214) (where **H** and **A** are known from step 2) and go to 2..

The aim of line 3 in the above algorithm is to achieve linear convergence at rate $1/4$ or better. The required rate of convergence is obtained when parameters $\sigma$ are sufficiently large. $\sigma$ remains constant then and only the parameters $\lambda$ are changed.

The multiplier penalty function for the inequality constrained case can be derived and used in a similar way[1],[2].

The use of multiplier penalty methods is a significant improvement as compared with the traditional penalty methods. High accuracy of the constrained minimum can be achieved at low values of penalty parameters $\sigma$. Ill-conditioning of the minimised penalty function, which is a serious obstacle when using the traditional penalty approach, is avoided to a great extent. An advantage inherited from the penalty approach is that any type of existing unconstrained minimisation techniques can be directly employed in the algorithm. However, the sequential nature of the penalty approach is less efficient than the more direct approach of the sequential quadratic programming approach described in the next section.

## *3.11 Sequential Quadratic Programming*

The penalty approach to constrained optimisation is based on definition of a sequence of unconstrained problems whose solutions converge to the solution of the original problem. A more direct approach is based on approximations of the objective and constraint functions. This seems to be a more efficient approach and many recent developments in optimisation algorithms are related to this approach[15]-[22].

Consider first the equality constrained problem. A system (3.132) is a stationary point condition for a local solution $\mathbf{x}^*$ and Lagrange multipliers in the solution $\lambda^*$. By applying Newton's method to solve this stationary point problem the following iteration is obtained:

$$\left[\overline{\nabla}^2 \mathcal{L}^{(k)}\right]\begin{bmatrix}\delta\mathbf{x}\\ \delta\lambda\end{bmatrix} = -\overline{\nabla}\mathcal{L}^{(k)}, \tag{3.215}$$

where $\overline{\nabla}^2 \mathcal{L}$ is the matrix of second derivatives of the Lagrangian functions with respect to variables $\mathbf{x}, \lambda$ and $\overline{\nabla}\mathcal{L}$ is defined by (3.133). The resulting method is referred to as the Lagrange-Newton method when applied to the solution of an equality constrained problem.

Expressions for the first and second order derivatives are obtained from (3.131). By taking into account these expressions, (3.215) becomes

$$\begin{bmatrix}\mathbf{W}^{(k)} & -\mathbf{A}^{(k)}\\ -\mathbf{A}^{(k)T} & \mathbf{0}\end{bmatrix}\begin{bmatrix}\delta\mathbf{x}\\ \delta\lambda\end{bmatrix} = \begin{bmatrix}-\mathbf{g}^{(k)} + \mathbf{A}^{(k)}\lambda^{(k)}\\ \mathbf{c}^{(k)}\end{bmatrix} \tag{3.216}$$

$\mathbf{A}^{(k)}$ is the Jacobian matrix of constraints evaluated at $\mathbf{x}^{(k)}$ and $\mathbf{W}^{(k)} = \nabla_x{}^2\left(\mathbf{x}^{(k)}, \lambda^{(k)}\right)$ is the Hessian matrix of the Lagrangian function with respect to variables $\mathbf{x}$, i.e.

$$\mathbf{W}^{(k)} = \nabla^2 f\left(\mathbf{x}^{(k)}\right) - \sum_{i\in E} \lambda_i^{(k)} \nabla^2 c_i\left(\mathbf{x}^{(k)}\right). \tag{3.217}$$

The system (3.216) can be rearranged to be solved for $\lambda^{(k+1)} = \lambda^{(k)} + \delta\lambda$ instead of $\delta\lambda$. If we write $\delta^{(k)} = \delta\mathbf{x}$, the system becomes

$$\begin{bmatrix} \mathbf{W}^{(k)} & -\mathbf{A}^{(k)} \\ -\mathbf{A}^{(k)T} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \delta \\ \lambda \end{bmatrix} = \begin{bmatrix} -\mathbf{g}^{(k)} \\ \mathbf{c}^{(k)} \end{bmatrix}. \tag{3.218}$$

Solution of the system gives $\lambda^{(k+1)}$ and $\delta^{(k)}$, while $\mathbf{x}^{(k+1)}$ is obtained by

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \delta^{(k)}. \tag{3.219}$$

Similarly as in the case of the Newton Method for unconstrained problems, the system of equations in the Lagrange-Newton iteration can be considered as a solution of a minimisation problem. Consider the problem

$$\begin{aligned} &\underset{\delta}{\textit{minimize}} && q^{(k)}(\delta) = \frac{1}{2}\delta^T \mathbf{W}^{(k)}\delta + \mathbf{g}^{(k)T}\delta + f^{(k)} \\ &\textit{subject to} && \mathbf{l}^{(k)}(\delta) = \mathbf{A}^{(k)T}\delta + \mathbf{c}^{(k)} = 0\,. \end{aligned} \tag{3.220}$$

This can be considered as an approximation of the original problem where the objective function is approximated by the second order Taylor approximation with the addition of constraint curvature terms in the Hessian, and constraints are approximated by the first order Taylor approximation about $\mathbf{x}^{(k)}$. The problem can be solved sequentially, which results in the sequential quadratic programming method summarised below:

**Algorithm 3.13:** Sequential quadratic programming.

For $k = 1, 2, ...$

1. Solve (3.220) for $\delta^{(k)}$. Set $\lambda^{(k+1)}$ to the vector of Lagrange multipliers of the linear constraints.
2. Set $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \delta$ .

The first order conditions $\overline{\nabla}\mathcal{L} = 0$ for this problem give (3.218), therefore the solution $\delta^{(k)}$ of the system (3.218) is a stationary point of the Lagrangian function of the problem (3.220). Following the discussion in section 3.9.1, the second order sufficient conditions for (3.220) are that the reduced Hessian matrix $\mathrm{Z}^{(k)T}\mathbf{W}^{(k)}\mathbf{Z}^{(k)}$ is positive definite. If this is true, then $\delta^{(k)}$ minimises (3.220). It follows that if unique minimisers exist in Algorithm 3.13 for each *k*, then the iteration sequence is identical to that given by the Lagrange-Newton method (3.218) and (3.219). The Lagrange-Newton method can converge to a KT point which is not a minimiser, therefore the sequential quadratic programming algorithm is preferred.

The algorithm can be generalised for solving inequality constrained problems. In this case the subproblem

$$\begin{aligned} &\underset{\delta}{minimize} && q^{(k)}(\delta) = \frac{1}{2}\delta^T \mathbf{W}^{(k)}\delta + \mathbf{g}^{(k)T}\delta + f^{(k)} \\ &subject\ to && \mathbf{l}^{(k)}(\delta) = \mathbf{A}^{(k)T}\delta + \mathbf{c}^{(k)} \geq 0\,. \end{aligned} \tag{3.221}$$

is solved instead of (3.220).

The Lagrange-Newton and SQP algorithms have good local convergence properties stated in the following theorem[1]:

**Theorem 3.17:**

If $\mathbf{x}^{(1)}$ is sufficiently close to $\mathbf{x}^*$, if the Lagrangian matrix

$$\nabla^2 \mathcal{L}^{(1)} = \begin{bmatrix} \mathbf{W}^{(1)} & -\mathbf{A}^{(1)} \\ -\mathbf{A}^{(1)T} & \mathbf{0} \end{bmatrix}$$

is non-singular and if second order sufficient conditions hold at $\mathbf{x}^*, \lambda^*$ with $rank\ \mathbf{A}^* = m$ (where $m$ is the number of constraints), then the Lagrange-Newton iteration converges with second order. If $\lambda^{(1)}$ is such that (3.220) is solved uniquely by $\delta^{(1)}$, then the same is true for the SQP method.

The Hessian matrix of the Lagrangian function $\mathbf{W}$ is required in the SQP method. It is possible to approximate $\mathbf{W}$ by using updating formulae[1],[18], analogous to those in quasi-Newton methods. For example, a matrix $\mathbf{B}^{(k)}$ that approximates $\mathbf{W}^{(k)}$ can be updated according to the DFP or BFGS formula, but with

$$\gamma^{(k)} = \nabla\mathcal{L}\left(\mathbf{x}^{(k)}, \lambda^{(k+1)}\right) - \nabla\mathcal{L}\left(\mathbf{x}^{(k)}, \lambda^{(k+1)}\right). \tag{3.222}$$

The resulting algorithms are superlinearly convergent.

The main difficulty of the SQP algorithm as stated above is lack of global convergence properties. The algorithm can fail to converge remote from the solution and it is possible that in some iteration the solution of the subproblem (3.220) or (3.221) does not even exist. The reason for this is essentially the same as for any method which constructs estimates purely on the basis of some simplified models (as for example Newton's method for unconstrained minimisation), i.e. the model is in general adequate only in a limited region which does not necessarily contain the problem solution.

While the line search strategy is a common approach to ensure global convergence of the unconstrained minimisation algorithms, this approach is less applicable in the direct methods for constrained minimisation (except those which solve a sequence of unconstrained subproblems). The reason for this is that especially when non-linear equality constraints are present, any straight line from the current iterate will typically have only one feasible point, which makes use of the line search in a standard way impossible.

The other approach for inducing global convergence is the trust region approach. By adding a step length restriction $\|\delta\| \leq h^{(k)}$ to (3.220) or (3.221) the possibility of an unbounded correction is removed. The difficulty is that if $\mathbf{x}^{(k)}$ is infeasible and $h^{(k)}$ is sufficiently small, then the resulting subproblem may not have any feasible points. Another way to ensure that the resulting subproblem is not unbounded is to add the Levenberg-Marquardt term $\nu \mathbf{I}$ to $\mathbf{W}^{(k)}$. It is possible to make $\mathbf{W}^{(k)}$ positive definite by sufficiently increasing the parameter $\nu$ [4].

A way of avoiding the difficulties with step length restriction is use of the $L_1$ exact penalty function[1],[4] in conjunction with the SQP method. An exact penalty function is a penalty function whose unconstrained local minima correspond to constrained local minima of the original problem. The $L_1$ exact penalty function for a general constrained problem is given by

$$\phi(\mathbf{x}) = \nu\, f(\mathbf{x}) + \sum_{i \in E} |c_i(\mathbf{x})| + \sum_{i \in I} \max(0, -c_i(\mathbf{x})). \tag{3.223}$$

Where $\nu$ is a control parameter that weights the relative contribution of $f(\mathbf{x})$ and the penalty terms. If $\mathbf{a}_i^*$ are linearly independent, if $\nu < 1 / \|\lambda_i\|_\infty$ and if $\mathbf{x}^*$ satisfies the second order sufficient conditions for the original problem, then $\mathbf{x}^*$ is a local minimiser of (3.223) and can be obtained by a single unconstrained minimisation. The disadvantage of such a penalty function is that it has discontinuous first derivatives on the border of the feasible region (Figure 3.15), which requires the use of special techniques for non-smooth minimisation.

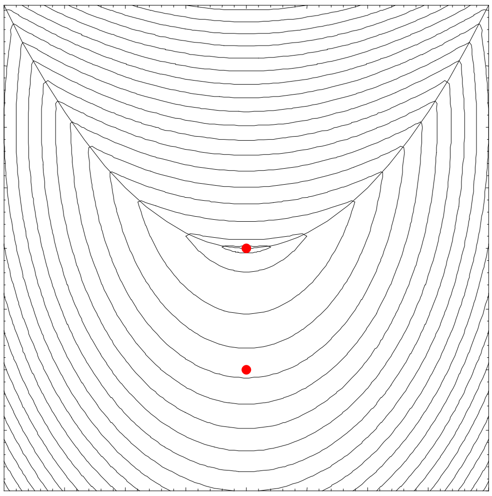


**Figure 3.15:** $L_i$ exact penalty functions for problem illustrated in Figure 3.13.

For use with the SQP method, approximations (3.220) and (3.221) are substituted in (3.223) and the step restriction (in $L_\infty$ norm) is added, which yields the subproblem

$$\begin{aligned} &\underset{\delta}{\textit{minimize}} \quad \psi^{(k)}(\delta) = \nu\, q^{(k)}(\delta) + \sum_{i \in E} \left| l_i^{(k)}(\delta) \right| + \sum_{i \in I} \max\left(0, -l_i^{(k)}(\delta)\right) \\ &\textit{subject to} \qquad \|\delta\|_\infty \le h^{(k)} \end{aligned} \tag{3.224}$$

This is an example of a so called $L_1$QP problem, for which effective algorithms exist[1]. Algorithm 3.13 that solves the subproblem (3.224) is consequently referred to as the $SL_1$QP algorithm.

The difficulties with an infeasible subproblem when using the step restriction are avoided by using the exact penalty function. The radius of the trust region $h^{(k)}$ is adjusted adaptively in a similar way as in restricted step algorithms for unconstrained minimisation.

Most of the difficulties related to use of the $L_1$QP subproblem arise from lack of smoothness. The derivative discontinuities give rise to grooves in the penalty surface, which can be difficult to follow by an algorithm. Another problem related to derivative discontinuities is the Maratos effect[1],[21], in which although $\mathbf{x}^{(k)}, \lambda^{(k)}$ may be arbitrarily close to the solution, the $SL_1$QP method fails to reduce the $L_1$ exact penalty function. To avoid the effect, the step $\delta^{(k)}$ must be recalculated after making the correction for the higher order errors that arise.

The SQP method and its variants seem to be among the most promising methods for solving general nonlinear problems. A variant of the method developed by A. Tits, E.R. Panier, J. Zhou and C. Lawrence[22],[23] is built in the optimisation shell described in the next chapter.

## *3.12 Further Remarks*

In the present chapter some of the basis of nonlinear programming is outlined. This knowledge is important for understanding the practical requirements for implementation of the algorithmic part in the optimisation shell. The literature cited in this chapter is mostly related to the mathematical and algorithmic background of optimisation and less to practical implementation (except references [3], [8] and [26]). Some implementation aspects are stressed in the next chapter within a larger framework of the optimisation shell. The need for hierarchical and modular implementation, which is stated there, is partially based on the heterogeneity of optimisation algorithms evident from the present chapter.

In practice it is not always obvious which algorithm to use in a given situation. This depends first of all on the case being solved. Although the theory can offer substantial support for making the judgment, most of the literature on optimisation methods recognize the significance of numerical experimentation alongside the theoretical development. This implies a significant aspect that was borne in mind during development of the optimisation shell. The shell should not only include a certain number of algorithms, but also provide an open framework for incorporation of new algorithms and testing them on simple model functions as well as on practical problems.

Many issues important for engineering practice were not taken into account. One of them is handling multiple conflicting optimisation criteria, i.e. solving the problem stated as

$$\begin{aligned} &\textit{minimise} && \left[ f_1(\mathbf{x}),\, f_2(\mathbf{x}), \ldots,\, f_m(\mathbf{x}) \right] \\ &\textit{subject to} && \mathbf{x} \in \Omega\,. \end{aligned} \tag{3.225}$$

A common approach is to weight the individual criteria, which leads to the problem

$$\begin{aligned} &\textit{minimise} && f(\mathbf{x}) = w_1 f_1(\mathbf{x}) + w_2 f_2(\mathbf{x}) + \ldots + w_m f_m(\mathbf{x}) \\ &\textit{subject to} && \mathbf{x} \in \Omega\,, \end{aligned} \tag{3.226}$$

where $w_1$, …, $w_m$ are positive weighting coefficients. The problem which arises is how to choose these coefficients. The choice is made either on the basis of experience or in an iterative process where optimisation is performed several times and coefficients are varied on the basis of the optimisation results.

Sometimes it is more convenient to designate one criterion as a primary objective and to constrain the magnitude of the others, e.g. in the following way:

$$\begin{aligned} &\textit{minimise} && f_1(\mathbf{x}) \\ &\textit{subject to} && f_2(\mathbf{x}) \le C_2\,, \\ & && \ldots \\ & && f_m(\mathbf{x}) \le C_m\,, \\ & && \mathbf{x} \in \Omega\,. \end{aligned} \tag{3.227}$$

This approach suffers for a similar defect as weighting criteria, i.e. the solution depends on the choice of coefficients $C_2$, …, $C_m$. Attempts to overcome this problem lead to consideration of Pareto optimality[9],[14] and solution of the min-max problem[9],[20].

Another important practical issue is optimisation in the presence of numerical noise. Most of the methods considered in this chapter are designed on the basis of certain continuity assumptions and do not perform well if the objective and constraint functions contain a considerable amount of noise. This can often not be avoided due to complexity of the applied numerical models and their discrete nature (e.g. adaptive mesh refinement in the finite element simulations).

A promising approach to optimisation in the presence of noise incorporates approximation techniques[35],[36]. In this approach successive low order approximations of the objective and constraint functions are made locally on the basis of sampled function values and/or derivatives. This leads to a sequence of approximate optimisation subproblems. They refer to minimisation of the approximate objective functions subject to the approximate constraints and to additional step restriction, which restricts the solution of the subproblem to the region where the approximate functions are adequate. The subproblems are solved by standard nonlinear programming methods. For approximations more data is usually sampled than the minimum amount necessary for determination of the coefficients of the approximate functions, which levels out the effect of noise. A suitable strategy must be defined for choosing the limits of the search region and for the choice of sampling points used for approximations (i.e. the plan of experiments)[35].

A common feature of all methods mentioned in this chapter is that they at best find a local solution of the optimisation problem. There are also methods which can (with a certain probability) find the global solution or more than one local solution at once. The most commonly used are simulated annealing[26],[9],[14] and genetic algorithms[9],[14]. Most of these methods are based on statistical search, which means that they require a large number of function evaluations in order to accurately locate the solution. This makes them less convenient for use in conjunction with expensive numerical simulations, except in cases where global solutions are highly desirable. Use of these techniques can also be suitable for finding global solutions of certain optimisation problems which arise as sub-problems in optimisation algorithms and in which the objective and constraint functions are not defined implicitly through a numerical simulation.

## *References:*


[1] R. Fletcher, *Practical Methods of Optimization (second edition)*, John Wiley & Sons, New York, 1996).

[2] E. J. Beltrami, *An Algorithmic Approach to Nonlinear Analysis and Optimization*, Academic Press, New York, 1970

[3] J. E. Dennis (Jr.), R. B. Schnabel, *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*, SIAM, Philadelphia, 1996.

[4] D. P. Bertsekas, *Nonlinear Programming (second edition),* Athena Scientific, Belmont, 1999.

[5] *Mathematical Optimization*, electronic book at http://csep1.phy.ornl.gov/CSEP/MO/MO.html , Computational Science Education Project, 1996.

[6] A. V. Fiacco, G. P. McCormick, *Nonlinear Programming – Sequential Unconstrained Minimisation Techniques,* Society for Industrial and Applied Mathematics, Philadelphia, 1990.

[7] D. P. Bertsekas, *Constrained Optimization and Lagrange Multiplier Methods*, Athena Scientific, Belmont, 1996.

[8] J. L. Nazareth, *The Newton – Cauchy Framework – A Unified Approach to Unconstrained Nonlinear Minimisation,* Springer – Verlag, Berlin, 1994.

[9] A. D. Belgundu, T. R. Chandrupatla: *Optimization Concepts and Applications in Engineering*, Prentice Hall, New Jersy, 1999.

[10] P. E. Gill, W. Murray, M. H. Wright, *Practical Optimization*, Academic Press, London, 1981.

[11] M. J. D. Powell (editor), *Nonlinear Optimization – Proceedings of the NATO Advanced Research Institute, Cambridge, July 1981*, Academic Press, London, 1982.

[12] M. H. Wright, *Direct Search Methods: Once Scorned, Now Respectable*, in D. F. Griffiths and G. A. Watson (eds.), Numerical Analysis 1995 (Proceedings of the 1995 Dundee Biennial Conference in Numerical Analysis, p.p. 191 – 208, Addison Wesley Longman, Harlow, 1996.

[13] K.G. Murty, *Linear Complementarity, Linear and Nonlinear Programming*, Helderman-Verlag, 1988.

[14] S.R. Singiresu, *Engineering Optimization – Theory and Practice (third edition)*, John Wiley & Sons, New York, 1996.

[15] E. Panier, A. L. Tits, *On Combining Feasibility, Descent and Superlinear Convergence in Inequality Constrained Optimization*, Mathematical Programming, Vol. 59 (1993), p.p. 261 - 276.

[16] C. T. Lawrence, A. L. Tits, *Nonlinear Equality Constraints in Feasible Sequential Quadratic Programming*, Optimization Methods and Software, Vol. 6, 1996, pp. 265 - 282.

[17] J. L. Zhou, A. L. Tits, *An SQP Algorithm for Finely Discretized Continuous Minimax Problems and Other Minimax Problems With Many Objective Functions*, SIAM Journal on Optimization, Vol. 6, No. 2, 1996, pp. 461 - 487.

[18] P. Armand, J. C. Gilbert, *A piecewise Line Search Technique for Maintaining the Positive Definiteness of the Matrices in the SQP Method*, Research Report No. 2615 of the “Institut national de recherche en informatique et en automatique”, Rocquencourt, 1995.

[19] C. T. Lawrence, A. L. Tits, *Feasible Sequential Quadratic Programming for Finely Discretized Problems from SIP*, in R. Reemtsen, J.-J. Ruckmann (eds.): Semi-Infinite Programming, in the series Nonconcex Optimization and its Applications. Kluwer Academic Publishers, 1998.

[20] J. L. Zhou, A. L. Tits, *Nonmonotone Line Search for Minimax Problems*, Journal of Optimization Theory and Applications, Vol. 76, No. 3, 1993, pp. 455 - 476.

[21] J. F. Bonnans, E. Panier, A. L. Tits, J. L. Zhou, *Avoiding the Maratos Effect by Means of a Nonmonotone Line search: II. Inequality Problems - Feasible Iterates*, SIAM Journal on Numerical Analysis, Vol. 29, No. 4, 1992, pp. 1187-1202.

[22] C. T. Lawrence, J. L. Zhou, A. L. Tits, *User’s Guide for CFSQP Version 2.5: A C Code for Solving (Large Scale) Constrained Nonlinear (Minimax) Optimization Problems, Generating Iterates Satisfying all Inequality Constraints*, Institute for Systems Research, University of Maryland, Technical Report TR-94-16r1, 1997.

[23] *The FSQP Home page*, electronic document at http://www.isr.umd.edu/Labs/CACSE/FSQP/fsqp.html , maintained by the Institute for Systems Research, University of Maryland.

[24] H. J. Greenberg, *Mathematical Programming Glossary*, electronic document at http://www.cudenver.edu/~hgreenbe/glossary/glossary.html , 1999.

[25] *Optimization Frequently Asked Questions*, electronic document at http://www-unix.mcs.anl.gov/otc/Guide/faq/ , maintained by Robert Fourer, The Optimization Technology Center.

[26] W.H. Press, S.S. Teukolsky, V.T. Vetterling, B.P. Flannery, *Numerical Recipies in C – the Art of Scientific Computing,* Cambridge University Press, Cambridge, 1992.

[27] J. W. Demmel, *Applied Numerical Linear Algebra,* SIAM, Philadelphia, 1997.

[28] L. N. Trefethen, D. Bau, *Numerical Linear Algebra,* SIAM, Philadelphia, 1997.

[29] B. Jacob, *Linear Algebra*, W. H. Freeman and Company, New York, 1990.

[30] I. N. Bronstein, K. A. Smendljajew, G. Musiol, H. Mühlig, *Taschenbuch des Mathematik (second edition - in German),* Verlag Harri Deutsch, Frankfurt am Main, 1995.

[31] I. Kuščer, A. Kodre, H. Neunzert, *Mathematik in Physik und Technik (in German),* Springer - Verlag, Heidelberg, 1993.

[32] E. Kreyszig, *Advanced Engineering Mathematics (second edition)*, John Wiley & Sons, New York, 1993.

[33] Z. Bohte, *Numerične metode*, Društvo matematikov, fizikov in astronomov SRS, Ljubljana, 1987.

[34] K. J. Bathe, *Finite Element Procedures*, p.p. 697-745, Prentice Hall, New Jersey, 1996.

[35] F. van Keulen, V. V. Toropov, *Multipoint Approximations for Structural Optimization Problems with Noisy Response Functions*, electronic document at http://www-tm.wbmt.tudelft.nl/~wbtmavk/issmo/paper/mam_nois2.htm.

[36] J. F. Rodriguez, J. E. Renaud, *Convergence of Trust Region Augmented Lagrangian Methods Using Variable Fidelity Approximation Data*, In: WCSMO-2 : proceedings of the Second World Congress of Structural and Multidisciplinary Optimization, Zakopane, Poland, May 26-30, 1997. Vol. 1, Witold Gutkowski, Zenon Mroz (editors), 1st ed., Lublin, Poland, Wydawnictwo ekoincynieria (WE), 1997, pp. 149-154.

# 4 OPTIMISATION SHELL "INVERSE"

## *4.1 Aims and Basic Structure of the Shell*

### 4.1.1 Basic Ideas

As mentioned in the introductory part, the main purpose of the optimisation shell "Inverse" is to utilize a finite element method based simulation code for solving inverse and optimisation problems. The philosophy of the shell[4]-[6] follows the idea that two naturally distinct parts can be recognized in the solution scheme of optimisation problems (Figure 4.1).

One part of the solution procedure is the solution of a direct problem at given optimisation parameters. This comprises numerical solution of the equations that govern the system of interest. In the scope of this work, this is performed by a finite element simulation, referenced in chapter 2 and schematically shown in a dashed frame on the right-hand side of Figure 4.1.

It is regarded that when a set of optimisation parameters is given, the system is completely determined in the sense that any quantity required by the optimisation algorithm can be evaluated. This usually refers to the value of the objective and constrained function and possibly their derivatives for a given set of parameters. Evaluation of these quantities is referred to as direct analysis[1] and is shown in the larger dashed frame in Figure 4.1.

[1]The notion of direct analysis is not used in a uniform way in the literature. Some authors use this term to denote merely the numerical simulation and sometimes the term is not defined strictly. In the present work the term "direct analysis" refers strictly to evaluation of the relevant quantities (e.g. the value of the objective and constraint functions) at a given set of optimisation parameters and includes all tasks that are performed as a part of this evaluation.

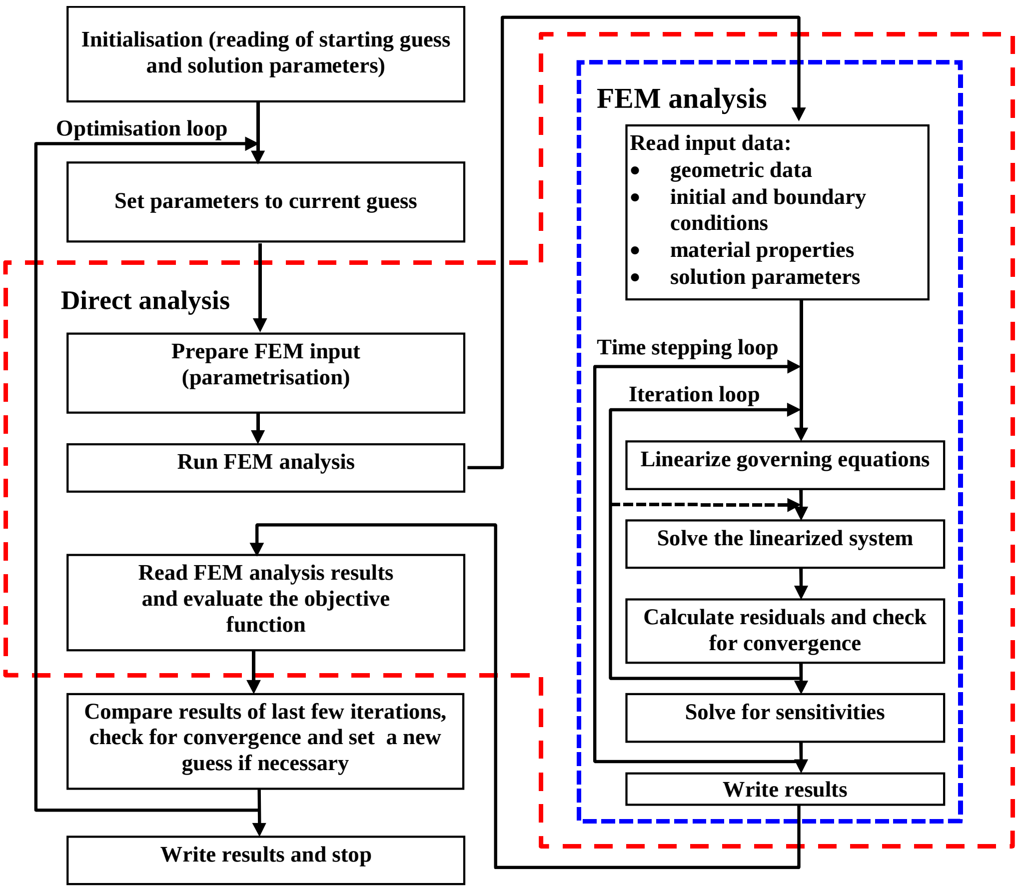



**Figure 4.1:** Typical solution scheme for an optimisation problem.

In order to utilise an existing simulation environment for solution of inverse and optimisation problems, the optimisation shell performs tasks on the left-hand side of Figure 4.1. Taking into consideration merely the solution scheme as shown in this figure, these tasks can be further divided into two parts. The part not included in the larger frame obviously represents an optimisation algorithm in its most basic sense (i.e. as was treated in the previous chapter). The part included in the frame represents those tasks of the direct analysis that are not performed by the simulation environment. This part can be viewed as an interface between the optimisation algorithm and the simulation.

The above discussion indicates that two basic elements of an optimisation system, i.e. optimisation algorithms and simulation tools, can be implemented as physically separate parts. This is one of the key ideas followed by the present work and is clearly reflected in separate and independent treatment in chapters 2 and 3. It must be emphasised that the above statements do not exclude dependency between optimisation algorithms and solution algorithms for the direct problem applied in specific cases. It is evident that close correlation between different numerical algorithms applied in the solution scheme of a specific problem is not excluded and was actually stressed at the end of the previous chapter. For example, whether analytical derivatives are provided by the simulation module or not usually plays a crucial role in defining the optimisation algorithm whose use will result in the most effective overall solution of the problem. This however does not affect physically separate treatment or implementation of either algorithm.

The scheme in Figure 4.1 is restricted to a solution process of a specific problem. When an optimisation system is treated, it is also important how the problem is defined and which solution strategies can be applied by using the system. In this respect it is significant to consider individual tools and algorithms implemented within the system, operational relations and interfaces between the parts of the system, which enable synchronous function, and finally the user interface.

Figure 4.2 outlines the operation of the presented optimisation system. It consists of the optimisation shell and the finite element simulation environment. In the solution scheme, the shell performs the tasks on the left-hand side of Figure 4.1, while the simulation environment is employed in solution of the direct problem, which corresponds to the tasks shown in the right-hand side of the figure.

The direct problem solved within the optimisation loop is determined by the values of the optimisation parameters. The problem is determined when boundary conditions, geometry, constitutive relations, etc. are known. These represent input data for numerical simulation. A transformation between optimisation parameters and the input data must therefore be defined. This transformation is referred to as parametrisation and is shown in Figure 4.1 as the first task of a direct analysis. This task is typically performed by the shell.

Optimisation parameters usually affect only a part of the simulation input data[1]. It is therefore advantageous to prepare a skeleton of the direct problem in advance and use it as a template for parametrisation. Most conveniently this skeleton is a definition of a direct problem at a specific set of optimisation parameters. It is usually created using pre-processing facilities of the simulation environment as shown in Figure 4.2.

The optimisation shell changes the affected input data according to the values of optimisation parameters. In the figure, this is done by updating the simulation input file, but can also be done directly by manipulating the data structure of the

[1] In this respect we talk more specifically about parametrisation of individual components, e.g. parametrisation of shape (or domain), parametrisation of material models, etc.

simulation programme, provided that the shell and the simulation unit can share data structures in the memory. After the simulation is performed, the shell reads the results and evaluates the quantities required by the optimisation algorithm. The shell must therefore be able to access the necessary simulation results. In the figure, results are accessed through the simulation output file, but a direct access can also be implemented.

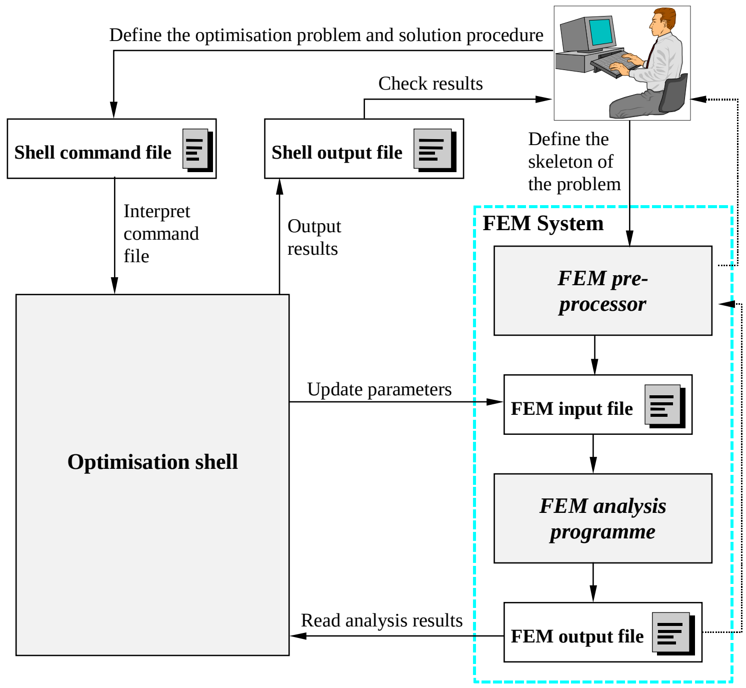



**Figure 4.2:** Operation scheme of the optimisation system.

Changing input data for numerical simulation, running the simulation and reading its results are performed by interface utilities of the shell. These are direct analysis tasks performed by the shell and are shown within the left-hand side of the larger frame in Figure 4.1. The shell can also perform a certain amount of processing of simulation results. It is not strictly defined which parametrisation and result processing tasks are performed by the shell and which by the simulation environment. This depends mostly on the capabilities of the simulation pre- and post-processing modules. The shell should permit employment of available capabilities in the simulation environment if these are convenient for performing the relevant tasks.

The optimisation shell includes implementation of various optimisation algorithms and other tools which can serve in the solution of optimisation problems. These tools are accessible through the shell user interface, which is separated from the user interface of the simulation environment. The current user interface is implemented through the shell input file in which the user defines the problem, and the shell output file, where the shell writes its results (Figure 4.2). Unlike traditional simulation input files, which consist of various sets of data written in some prescribed format, the shell input file consists of commands that are interpreted by its built-in interpreter. It is therefore commonly referred to as the shell command file.

### 4.1.2 Requirements Taken into Consideration

Before continuing with description of the optimisation shell, it is appropriate to mention some requirements which affect its design[1]-[3],[7]. These requirements will be referenced later in the text in order to justify certain design aspects.

The basic demand for a good optimisation system is flexibility. It must be possible to apply the system to a large variety of problems that can possibly appear. This concerns definition of the problem itself as well as definition of the solution strategy. On one hand this flexibility is determined by the set of tools for solution of different subproblems, which are offered by the system. On the other hand the conceptual structure of the system should not impose any fundamental restrictions on the way how different tools can be combined to solve complex problems.

Somehow conflicting with flexibility is the demand for simplicity of use. Logical structure of the system is a prerequisite for avoiding conflicts induced by these two demands. A system is easy to be applied for certain types of problems if the user is required to provide only that information necessary to define the particular problem and if the requirements are set by the system in a clear way. It is obvious that this can be achieved only on a case to case basis, so that all particularities can be taken into account. The system must be structured hierarchically, so that high level easy-to-use tools for particular problems can be implemented by templates built on the lower-level basis. These tools can introduce additional concepts, but these should apply locally and should not affect the underlying system, which can be still applied independently and should retain generality.

A group of requirements is related to the economy of the system development. This essentially means the ability of achieving the best possible effect with limited development resources. The effect is measured in terms of applicability of the system to various problems the potential user can be faced with and in terms of effort needed for problem definition and computer time needed for problem solution. Beside logical structure of the system, economy of development is mostly related to its portability, modularity and openness.

Portability means that the system can be easily transferred from one computational platform to another. A portable system can be developed in any homogeneous or heterogeneous computer environment so that its transfer to a different environment requires minimal additional development effort. It is most easily achieved by using portable development tools. Due to portability reasons most of the system was developed in ANSI C[25],[26]. Compilers based on the ANSI C standard are available on most existing computer architectures and are usually implemented in a strict enough manner that it is possible to transfer programmes between different platforms without major modifications. Use of non-standard libraries has been avoided as much as possible in the shell development. Where system dependent details could not be completely avoided, they were captured in isolated and clearly distinguished locations which are easy to identify and modify when transfer to a new platform is performed. One of the development principles that were taken into account is also that turning off system dependent details should affect as little functionality as possible. In this way detrimental effects of non-standard behaviour of any system part can usually be avoided to a great extent.

The modular structure of the system also has beneficial effect on economy of development. In a modular system, tools that constitute its functionality are implemented in separate units. Development of these units must be as independent as possible, so that development and change of specific tools does not affect and is not affected by existent structure and functionality. Modularity is best achieved by imposing a limited number of clear general rules on the system and providing a simple implementation interface for development of new modules. This interface must be such that it does not restrict the range of tools which can potentially be added to the system. It must be possible to apply this interface to existing general tools which were not primarily developed to be included in the system. Optimisation algorithms provide a good example of these principles. The main concern of an optimisation algorithm is to effectively locate a constrained local minimiser to a given accuracy, i. e. with as few function evaluations and housekeeping operations as possible. If the algorithm is used as a part of a complex optimisation system, a number of additional implementation details must be solved such as interaction with the simulation environment. However, this should not affect development of the algorithm itself, because the aim of the algorithm remains the same. Additional requirements such as interaction with simulation environments must be overcome by the implementation interface, which is used at the final stage when the already implemented algorithm is built into the system.

Openness of the system includes two aspects. The first aspect regards the definition of the problem. In this respect openness means that the user can easily access various built-in utilities and employ external programmes using the available interfacing utilities when defining a solution strategy for the problem to be solved. This has strong impact on the flexibility of the system. The development aspect of openness means that existing functionality can be directly employed when developing additional tools in the system. The shell can therefore be easily integrated with other programmes and different modules can be developed independently and merged together. This facilitates development of higher level and more case specific tools on the basis of lower-level utilities. Openness of the system is to a large extent conditional on its modularity.

### 4.1.3 Operation Synopsis

The optimisation shell "Inverse" operates on the basis of an input (or command) file, in which the user defines what the shell should do. The problem to be solved is not defined in a descriptive way as is usually the case with simulation programmes, but as a set of instructions for the shell, which ensures sufficient flexibility of the user interface[5],[6].

Figure 4.3 shows how parts of the optimisation system interact in the solution procedure. Any action of the optimisation shell is triggered by the corresponding command in the command file. The shell file interpreter[15] reads commands one by one and runs internal interpreter functions that correspond to them. The built-in optimisation algorithms and other utilities such as mathematical tools (function approximation, matrix operations etc.) or interfacing with the simulation, are accessed through the interpreter functions. Each command in the command file has its own argument block through which arguments can be passed to corresponding functions. The argument block is enclosed in curly brackets that follow the command.

The shell includes a general built-in function that performs the direct analysis (Figure 4.3). Optimisation algorithms and some other utilities such as tabulating functions call this function for evaluation of necessary quantities such as values of the objective and constraint functions. Actually there is an additional interface function between this function and any calling algorithm (not shown in the figure). This function covers specificity of the algorithm regarding input and output of the direct analysis. This includes formats of function arguments prescribed by the specific algorithm. It also concerns the fact that different algorithms require different data to be evaluated, e.g. some of them require derivatives and the others do not.

The general analysis function runs interpretation of a specific block in the shell command file, referred to as the analysis block. This block is so interpreted every time the direct analysis is performed and is therefore used for user definition of the direct analysis. Since the complete interface with the simulation environment is accessible through interpreter commands, the user can precisely define how the numerical simulation at specific values of optimisation parameters is performed and how data is transferred between the shell and simulation environment. The analysis block is physically an argument block of the *analysis* command. When this command is encountered by the interpreter, the position of its argument block is stored so that the block can be interpreted any time by the general analysis function.

There must exist a data link for transferring input and output parameters of the direct analysis between the calling algorithm and user definition in the analysis block. The data is passed through function arguments between the algorithm and the direct analysis function called by that algorithm. The data link between this function and user definition is established through pre-defined variables. These variables uniquely define the place where particular input and output data of the direct analysis is stored. The internal analysis functions automatically update input data obtained by the algorithm (e.g. values of optimisation parameters) on the appropriate pre-defined location. After interpretation of the analysis block it retrieves the data to be returned to the calling algorithm (e.g. values of the objective and constraint functions) from the locations defined for this purpose. The user can access these locations through interpreter functions for accessing variables. The user must ensure in the analysis definition that analysis results are correctly evaluated and stored to the appropriate locations, where they can be retrieved by the analysis function and returned to the algorithm[18].

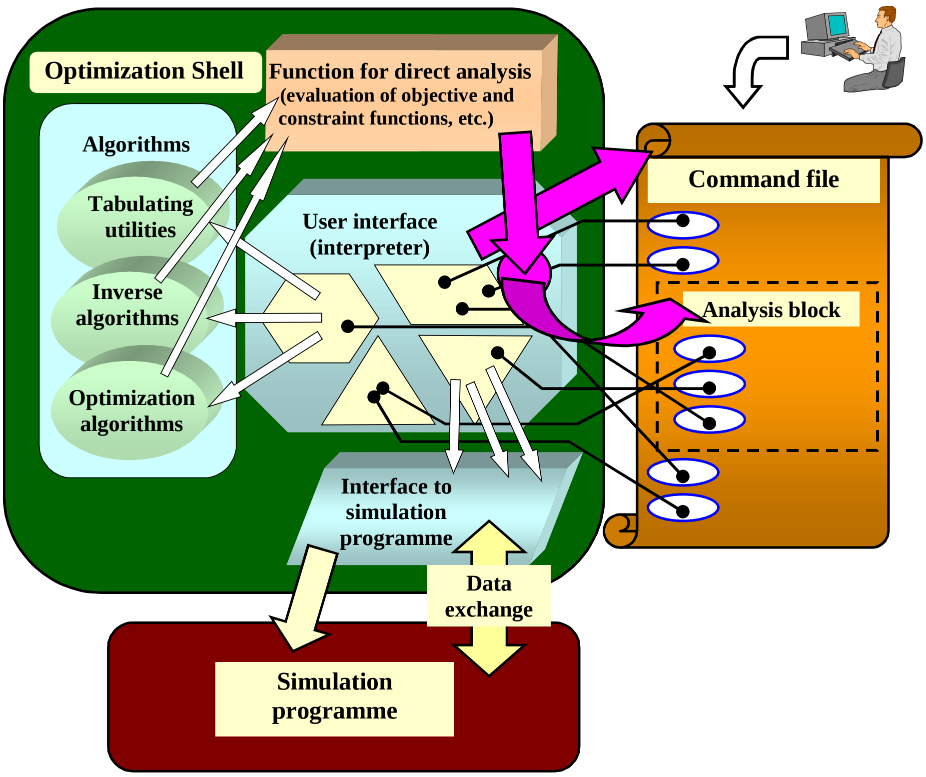



**Figure 4.3:** Structure and operation of the optimisation shell.

Beside interpreter commands for accessing various built-in tools, there are also commands for controlling the flow of interpretation, such as branching and looping commands, for example. Programming capabilities of the interpreter are supplemented by a system of user-defined variables[17] and a system for evaluation of mathematical expressions[16]. The shell can therefore be programmed in a way that resembles a high level programming language[1].

The shell design allows the user to interact with the solution process at several levels. All tools and algorithms provided by the shell are run from the command file. Their output results can be stored in user-defined variables, and so used as input for other built-in algorithms. The available utilities can therefore be easily combined as necessary when applied to the solution of more complex problems. This feature is further enhanced by programming capabilities of the interpreter, which makes the shell very flexible and applicable to a large variety of problems.

At first sight the consequence of such a flexible user interface implemented through the interpreter is that the optimisation system can not be made easy to use. It might seem that solution of any optimisation problem requires detailed programming of the solution process, which is only assisted by built-in utilities.

This is not an entirely correct impression. For any set of similar problems it is possible to implement a high level interface in such a way that definition of the problem and the solution procedure require a minimum amount of user interference. Such an interface can be built by templates written for the shell interpreter. User interaction can be reduced merely to insertion of input data, which can eventually be assisted by an external user interface.

Such high level interfaces inevitably restrict the range of problems that can be solved by the system. Their use is adequate when the optimisation system is used for highly specialised purposes. Another way of making the system easier to use is to introduce high level functions that perform complex tasks or combination of groups of tasks, which appear steadily in a larger group of related problems. This can result in a hierarchical structure of utilities where the user can decide which level to use. High level tools make the use of the system easier without imposing a priori restrictions on flexibility. New higher level commands can be created by combination of existing lower level utilities using the shell interpreter. In this way interventions in the shell source code are avoided, which reduces the level of skill necessary for implementation of such tasks.

By implementation of hierarchically structured sets of lower and higher level commands, the two fundamentally conflicting demands for flexibility and simplicity of use can be compromised. Currently the most urgent problem with the shell is that many necessary sets of high level specialised commands are not as yet implemented, which is especially expressive at interfacing the simulation environments.

[1] In this respect file interpreter commands are also referred to as functions. This is sometimes better to be avoided in order to avoid ambiguity and confusion of interpreter commands and internal functions of which the shell consists.

## *4.2 Function of the Shell*

The discussion in the previous sections was centered around the basic concepts of the shell. In order to make these things less abstract, a few more details regarding the shell function are given in the present section. Some details will be cleared from the user point of view. The intention of this section is however not to serve as a user reference, but merely to give more insight into how previously described concepts are reflected when the shell is applied to the solution of problems. Detailed reference of the existing functionality of the shell exist in the form of manuals available on the Internet[12],[14]-[22].

### 4.2.1 Basic File Interpreter Syntax

The basic file interpreter syntax is simple:

```
command1 { arguments } command2 { arguments } …
```

When a shell command file is interpreted, the interpreter simply searches for commands, locates their argument blocks and passes control to the appropriate functions that are in charge of execution. For each interpreter command there exists an appropriate function installed in the file interpreter system. These functions are usually just an interface between commands in the command file and those functions which really do the job, and are referred to as interpreter functions in this text. Interpreter functions take care of the correct transfer of arguments from the argument block in the command file and for imposing additional syntax and other rules imposed by the optimisation shell. Such two stage arrangement makes it possible to easily incorporate functions and modules that were not primarily developed for use in the shell. Separation of the concepts imposed by the shell and those implied by a specifically incorporated function or module is achieved in this way. The two stage calling arrangement is evident from Figure 4.3.

The file interpreter is supported by the system for evaluation of mathematical expressions or expression evaluator. This is an independent system of the shell. Its capabilities are accessed by the file interpreter functions for treatment of their arguments. The basic functionality offered by the expression evaluator is evaluation of mathematical expression with the ability of defining new variables and functions. For the interpreter itself the most important use of the expression evaluator is evaluation of conditions in branching and looping commands. This enables the interpreter to be used as a programming language.

Control of the interpretation flow[13][15] is implemented through branching and looping interpreter commands and through the function definition utility. All related functionality is treated as a part of the file interpreter. The syntax of these utilities is described below.

The *if* command interprets a block of code in the basis of the value of a condition. Its syntax is the following:

```
if { (condition) [ block1 ] else [ block 2 ] }
```

If the condition in round brackets is true (that is non-zero), then the block of code *block1* is interpreted, otherwise the block *block2* is interpreted. The condition is evaluated by the expression evaluator.

The *while* commands repeatedly interprets a block of code. The block is being interpreted as long as the condition remains satisfied (i.e. the value of the condition expression is non-zero). The syntax is the following:

```
while { ( condition ) [ block ] }
```

The condition in the round bracket is evaluated by the expression evaluator in each iteration of the loop before the code block in square brackets is interpreted. The first evaluation of the condition expression as zero causes exit of the loop and interpretation is continued after the *while* command argument block. Typically the code block contains commands that affect the value of the condition expression, so that after a certain number of iterations the value of the expression becomes zero and the loop exits.

Similar to the while command is the *do* command, except that the condition expression is evaluated after interpretation of the code block and therefore the block is interpreted at least once. Its syntax is the following:

```
do { [ block ] while (condition) }
```

Beside standard branching and looping commands, the ability of defining new interpreter commands is relevant. This is referred to as the function definition utility and is also implemented through an interpreter command. This utility enables implementation of commands that perform combined tasks by employing existing commands. Higher level commands can therefore be implemented without interference in the shell source code. Commands defined by the function definition utility behave in a similar manner to the built-in commands.

New interpreter commands are defined by using the *function* command. Its syntax is the following:

```
function { funcname ( arg1 arg2 … ) [ defblock ] }
```

*funcname* is the name of the new function, *arg1*, *arg2*, etc. are names of function arguments, and *defblock* is the definition block of the new command. The *function* command makes the interpreter install a new command, which includes storage of the function argument list and position of the definition block. When the newly defined command is encountered later during interpretation, its definition block is interpreted. Occurrences of arguments in the definition block are replaced by actual arguments prior to interpretation. The replacement is made on a string basis, so that the meaning of arguments is not prescribed by the definition of the new function[15]. In the definition block, the arguments must be marked by argument names preceeded by the hash sign ('#'). The interpreter can in this way recognise occurrences of arguments and replace them by actual arguments stated in the argument block of the called command.

A clear example[13] of how the function definition utility can be used is given by the implementation of the for loop through the interpreter. This can be done in the following way[1]:

```
1. function { for ( begin condition end body )
2. [
3.   #begin
4.   while { ( #condition )
5.   [
6.     #body
7.     #end
8.   ] }
9. ] }
```

The function requires four arguments. *begin* is the code block interpreted before the loop is entered. *condition* is the looping condition that is checked before every iteration of the loop. It must be an expression that can be evaluated in the expression evaluator. *body* is the code segment that is interpreted in the loop, and *end* is the code segment that is interpreted after the loop. Using the newly defined function *for*, the following code will print numbers from 1 to 5 to the standard output:

```
for { ={i:1}   i<=5   ={i:i+1}
  { write { $i "\n" } }
}
```

When the interpreter encounters the command for, it replaces formal arguments in the definition block (lines 2 to 8 in the definition segment of the code) with actual arguments and interprets the definition block. The resulting code that is actually interpreted is then as follows:

[1]Line numbers simply enable referencing portions of code. In the command file lines are not numbered.

```
={i:1}
while { ( i<=5 )
[
  write { $i "\n" }
  ={i:i+1}
] }
```

### 4.2.2 Expression Evaluator

The expression evaluator (succinctly referred to as the calculator)[16] is an independent shell module. Support to control of the interpretation flow is one of its basic tasks, therefore the interpreter and the expression evaluator are inseparably connected.

The calculator contains a set of built-in mathematical functions and operators, which can be arbitrarily combined with variables and numbers to form expressions. The calculator system currently supports only scalar variables. The syntax for forming mathematical expressions is standard and is described in detail in [16].

The file interpreter commands = and *$* serve for user interaction with the expression evaluator.

The syntax of the = command is the following:

```
= { varname: expression }
```

The expression is first evaluated by the calculator and its value is assigned to the calculator variable named *varname*.

The *$* commands calculator variables and functions in terms of expressions. Definition of a variable has the following syntax:

```
$ { varname : expression }
```

The expression is not evaluated at execution of this command. It is assigned to the variable as an expression that defines how the value of the variable is calculated. If the value of any part of the defining expression is changed, this affects the value of the variable. It is not even necessary that all calculator variables and functions that appear in the expression are defined at the time the *$* command is interpreted. The value of the variable becomes defined as soon as all variables and functions appearing in the expression are defined.

Definition of new expression evaluator functions have the following syntax:

$ { funcname [ arg1, arg2, … ] : expression }

The defining expression contains formal arguments *arg1*, *arg2*. After the definition, the function can be used in the same way as built-in expression evaluator functions. Function evaluation consists of evaluation of the defining expression after replacement of formal arguments by the values of actual arguments.

The definition of new calculator functions may, as definition of variables, include variables and functions that are not yet defined. The use of functions = and *$* is illustrated by the following example:

```
1. ${ a: cubesum[b,c] }
2. ${cubesum[x,y]: (x+y)^3 }
3. ={ b: 1 }
4. ={ c: 2 }
5. write { $a }
```

The first line defines a new calculator variable *a* as *cubesum[b,c]*. Neither the variables *b* and *c* nor the function *cubesum* are defined at the point of execution of this line. The function of two variables *cubesum* is defined in line 2 as the third power of the sum of its arguments. In line 3 the variable *b* is defined and assigned the value 1, and in line 4 the variable *c* is defined and assigned the value 2. The value of the variable *a* is defined after this because the values of all terms of its defining expression are defined. The last line writes the value of the expression evaluator variable *a*, which is $cubesum(b,c) = (b+c)^3 = (1+2)^3 = 27$.

New expression evaluator functions can also be defined using the interpreter by the *definefunction* command[15] with the following syntax:

```
definefunction { funcname [defblock] }
```

Evaluation of a function defined in this way includes interpretation of its definition block *defblock*. Additional calculator and interpreter functions, which can be used in the definition block, facilitate definition of how the function is evaluated. The file interpreter function *return* is used for final specification of the value which the function evaluates. Its syntax is

```
return { expression }
```

The function defined by the *definefunction* command is evaluated to the value of the *expression* given at interpretation of the *return* command in the function definition block.

The function definition is also facilitated by two pre-defined expression evaluator functions. *numargs* is evaluated to the number of actual arguments passed at function evaluation while *argument* is evaluated to values of specific arguments that are passed.

Use of the *definefunction* can be illustrated by the following example, where an expression evaluator function *Sumation*, which evaluates to the sum of its arguments, is defined[13]:

```
definefunction { Sumation
[
  ={retsum:0}
  ={indsum:1}
  while { (indsum<=numargs[ ])}
  [
    ={retsum: retsum+argument[indsum] }
    ={indsum: indsum+1 }
  ] }
  return{retsum}
] }
```

Beside evaluation of condition expressions in branching and looping interpreter commands, the expression evaluator can be used for evaluation of numerical arguments of file interpreter commands. Any numerical argument can be given, stated either in the direct form by specifying its value, or by an expression evaluator variable in the form

```
$ varname
```

or by a mathematical expression of the form

```
$ { expression }
```

The interpreter function that corresponds to the command called by such arguments, use the expression evaluators to evaluate the appropriate values that replace variables or expressions before arguments are used.

### 4.2.3 User Defined Variables

The shell uses a system of user defined variables (also referred to as shell variables) for data storage. Individual algorithms and other utilities usually have their own local data storage, but input and output data of built-in utilities should be transferable to or from the system of user defined variables. In this way results of any algorithm can be used in other algorithms. Since running of any algorithm or utility as well as access to user variables is arranged through a common user interface (i.e. the command file interpreter), the necessary data transfer between different utilities is easily achieved.

The system of user-defined variables is considered as an individual module of the shell, which provides data storage and transfer services. These services are accessible through special interpreter and calculator commands for manipulating variables and through the possibility of using variables of various types as input arguments of interpreter commands. It must be noted that the calculator variables are a part of a separate system and are not treated as shell variables. Transfer between both systems is completely supported and in some cases calculator variables are used for the same purpose as the user-defined variables.

The shell variables hold objects (elements) of different types: options, counters, scalars, vectors, matrices, strings and files. Each variable can hold a multidimensional table of elements of a specific type (Figure 4.4). The number of dimensions of this table is referred to as the rank of the variable.

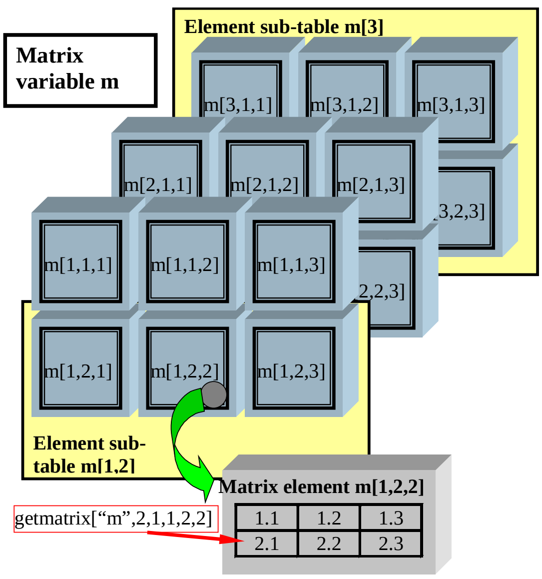



**Figure 4.4:** Example of a matrix variable that holds a $3\times2\times3$-dimensional table of 2 by 3 matrices.

For each type of variable there is a set of interpreter and expression evaluator functions for their manipulation, i.e. creation, initialisation, copying, moving, etc. The following example shows how to create a matrix variable as shown in Figure 4.4:

```
newmatrix { m [ 3 2 3 ] }
setmatrix { m [ 1 2 2 ]
  2 3 { { 1: 1.1 1.2 1.3 } { 2: 2.1 2.2 2.3 } }
}
```

The *newmatrix* command creates a matrix variable with a $3\times 2\times 3$ - dimensional table of elements, each of which is a matrix. Indices in square brackets specify dimensions of the variable element table (note that these dimensions are not related to dimensions of matrix elements of the variable). After creation, the matrix elements are not initialised and contain no data. Values of specific elements are set by the *setmatrix* command. Indices in square brackets in this case specify the matrix in the variable element table whose values are set. In the above case, the element with indices $[1,2,2]$ is set to the following $2\times 3$ matrix:

$$\begin{bmatrix} 1.1 & 1.2 & 1.3 \\ 2.1 & 2.2 & 2.3 \end{bmatrix}. \tag{4.1}$$

If the rank of the matrix variable (which is three in this case) was zero, then a call to the *newmatrix* command would not be necessary since the *setmatrix* function creates a zero rank variable automatically if it does not yet exist.

The expression evaluator functions can be used for accessing data stored in variables. For each variable type there exist functions, which are evaluated as dimensions of variable element tables or as components of variable elements. For example, the *getmatrixdim* function evaluates a specific dimension of the variable element table. The expression

```
getmatrixdim["m",2]
```

evaluates to the second dimension of the element table of the matrix variable *m*, which is 2 in the case that *m* is defined as above (Figure 4.4).

The *getmatrix* function evaluates to the value of a specific component of a specific matrix element. The expression

```
getmatrix["m",2,1,1,2,2]
```

evaluates the component 2-1 of the element $[1,2,2]$ of the matrix variable *m*, which is 2.1 if the variable is defined as above. The first two indices specify the component (row and column number, respectively) and the last indices specify the matrix element of the variable element table.

There are several other functions for manipulation of matrix variables, and analogous functions are implemented for other types of variables. A complete list of these functions can be found in the corresponding manual[17].

Some interpreter commands can operate on whole sub-tables of variable elements. The notion of a sub-table is also illustrated in Figure 4.4. The matrix variable *m* shown in the figure contains an element table of rank 3 and of dimensions $3\times2\times3$. This table consists of two sub-tables of rank 2 and of dimensions $2\times3$, each of which further consists of two sub-tables of rank 1 and dimension 3, each of which contains 3 matrix elements.

Some shell variables are used for carrying specific data relevant for optimisation. Such variables are referred to as variables with pre-defined meaning or briefly pre-defined variables.

Of particular importance are those variables which are responsible for data links between user definition of the direct analysis in the *analysis* block of the command file and optimisation algorithms[18] (Figure 4.3 and the surrounding discussion). A list of these variables is shown in Table 4.1.

The pre-defined variables are a part of the user-defined variables, therefore all functions for manipulating variables are applicable to these variables. Some additional commands are designed especially for easier handling of these variables. Some general functions of the pre-defined variables operate in a slightly different way on pre-defined variables. This is especially true for creation and initialisation commands, which take into account known information regarding dimensions. Dimensions of the pre-defined variables are often related to the characteristics of the optimisation problem being solved, therefore the same dimensions can be shared with more than one variable (Table 4.2). These characteristic dimensions have a special storage space that is not a part of the variable system. Their values are however directly accessible through the interpreter and expression evaluator functions[17].

There are some other variables with pre-defined meaning[17], which support common tasks related to the solution of optimisation problems. For example, variables in Table 4.1 have equivalents with the suffix "opt" instead of "mom", which store optimum values of the corresponding quantities so that they can be retained for further use. Vector variables *meas* and *sigma* are used for holding input data for inverse problems, namely the measurements and their estimated errors. Pre-defined file variables for holding commonly used files are also defined, i.e. *infile* for shell input file, *outfile* for shell output file, *aninfile* for simulation input file and *anoutfie* for simulation output file. There are groups of interpreter and calculator functions, which operate specifically on these variables. A set of output functions operate on *outpfile*, and a set of general interfacing functions operate on *infiile*[19].

Some functions of the interface module with the simulation programme operate on *aninfile* and *anoutfile*[22].

**Table 4.1:** Variables with pre-defined meaning, which are used for transfer of input and output arguments of direct analysis between user definition and the calling algorithm. The meaning of dimensions is shown in Table 4.2.

| **Variable name** [ element table dim. ] ( element dim. ) | **Meaning** |
|---|---|
| **Scalar variables** | |
| **objectivemom** [] < [numobjectives] > | Value(s) of the objective function(s) at the current parameter values. |
| **constraintmom** [numconstraints] | Values of the constraint functions at the current parameter values. |
| **Vector variables** | |
| **parammom** [] (numparam) | Current values of parameters. |
| **measmom** [] (nummeas) | Current values of simulated measurements. |
| **gradobjectivemom** [] < [numobjectives] > (numparam) | Gradient of objective function(s) at the current parameter values |
| **gradconstraintmom** [numconstraints] (numparm) | Gradients of constraint functions at the current parameter values. |
| **gradmeasmom** [nummeas] (numparam) | Gradients of the simulated measurements at the current parameter values. |
| **Matrix variables** | |
| **der2objectivemom** [] < [numobjectives] > (numparam,numparam) | Second derivatives (Hessian) of the objective function(s) at the current parameter values. |
| **der2constraintmom** [numconstraints] (numparam,numparam) | Second derivatives (Hessian) of the constraint functions at the current parameter values. |
| **der2measmom** [nummeas] (numparam,numparam) | Second derivatives (Hessian) of the simulated measurements at the current parameter values. |

**Table 4.2:** Characteristic dimensions of variables with a pre-defined meaning.

| Dimension | Meaning |
|---|---|
| *numparam* | Number of optimization parameters |
| *numconstraints* | Number of constraint functions |
| *Numobjectives* | Number of objective functions (usually equals 1) |
| *Nummeas* | Number of measurements (applicable for inverse problems) |

### 4.2.4 Argument Passing Conventions

The file interpreter itself has nothing to do with interpretation of command arguments. It only passes positions of command argument blocks to the corresponding functions. Each individual function (shown in the user interface area in Figure 4.3) is responsible for interpretation and treatment of its arguments. The shell provides some general rules about argument passing, which represent a non-obligatory recommendation and may be overridden by individual functions. This freedom allows implementation of interpreter commands with arguments of types and format adapted to specific tasks and not common for the shell. The shell user interface can therefore be customised to a great extent.

The shell provides a set of functions for interpretation of specific supported types of arguments. Within functions installed in the file interpreter system these functions can be used for interpretation of arguments. Shell functions for interpretation of arguments can be used as library functions and provide an implementation interface, which enables new utilities to be built into the shell in accordance with standard rules that apply for the shell. Such an implementation interface plays an important role in ensuring openness and flexibility discussed in section 4.1.2. Argument passing rules supported by the shell are briefly described below, while a complete description can be found in [15].

Multiple arguments may be separated either by spaces or by commas. Because some arguments (respectively strings that represent them) themselves contain commas and spaces, it must be unambiguous for each argument to which position it extends, which is a basic requirement for formatting conventions.

Objects of all types defined by the shell can be passed as arguments. Each type has its own formatting conventions. For example, a matrix object given by (4.1) can be given in one of the following forms:

```
1. 2 3 { { 1: 1.1 1.2 1.3 } { 2: 2.1 2.2 2.3 } }

2. 2 3 { { 1 1: 1.1 } { 1 2: 1.2 } { 1 3: 1.3 } { 2 1: 2.1 } { 2 2:
   2.2 } { 2 3: 2.3 }  }


3. 2 3 { { 1.1 1.2 1.3 2.1 2.2 2.3 } }
```

If we just want to specify a matrix with a given number of rows and columns without specifying components, only dimensions need to be given followed by empty curly brackets, e.g.

```
2 3 {  }
```

In commands that assign a matrix to an existing object, we can specify an arbitrary number of its components without dimensions, e.g.

```
{ { 1 1: 1.1 } { 2 2: 2.2 } { 2 3: 2.3 }  }
```

The *setmatrix* command mentioned in the previous section is an example of an interpreter command that takes a matrix argument. For this function, the last format can be used if the matrix element, which is being initialised by the *setmatrix* function, already exists and is of correct dimensions. Since in all possible formats specification of a matrix object is concluded by curly brackets (usually containing components), it is unambiguous where the argument ends and where to expect the next argument.

Other types of objects have similarly logical format conventions. String objects must be specified in double quotes if they contain spaces. Special characters that can not be represented in text files or can not be specified directly because of formatting rules, can be specified by two character sequences consisting of a backslash and specification character. For example, the newline character can be represented by the sequence "\n".

Variable arguments are specified by variable names not included in quotes. Variables can contain more than one object of a specific type. Commands that operate on individual objects therefore do not take variable arguments. Variable arguments are typical for commands which create, copy or move a whole variable, like for example the *newmatrix* command for creation of matrix variables, mentioned in section 4.2.3.

Arguments that refer to variable elements are specified by a variable name followed by element indices in square brackets. Indices specify element position in the variable element table. Elements of zero-rank variables can be in some cases specified only by variable name, but the name followed by empty square brackets is always acceptable. Sub-tables of variable elements are specified in the same way as individual elements. For example, the $[2,3]$-th element of a matrix variable *mat1* is specified by

```
mat1 [2 3]
```

Numerical arguments can be specified by numbers, expression evaluator variables (variable name following the dollar sign) or mathematical expressions (stated in curly brackets following the dollar sign), as has been described in section 4.2.2. Indices in element specifications are also regarded as numerical arguments, therefore this rule applies. If an expression evaluator variable *a* is defined and has a value 2, the above specification of a matrix element can be written equivalently as

```
mat1 [$a ${a+1}]
```

Objects of any type can also be specified by a reference to an existing object of the same type in the shell variable system. A copy of that object is created in this case and passed as an argument. Object specification must be included in curly brackets following the hash sign, e.g.

```
# { mat1 [4 1] }
```

specifies a matrix that is a copy of the element with indices $[4, 1]$ of the matrix variable named *mat1*.

Finally, variable names in specification of variables, elements or sub-tables of elements can be replaced by a reference to an existing string element. For example, if a zero rank string variable *str* is defined and its only element is the string *"mat1"*, then the following specification of the $[2,3]$-th element of a matrix variable *mat1* is adequate:

```
# { str [ ] } mat1 [2 3]
```

### 4.2.5 Summary of Modules and Utilities

A brief survey of modules that provide basic functionality needed for solution of inverse and optimisation problems is given in this section. Figure 4.3 and the surrounding discussion provides a basic explanation of the importance of individual modules. A basic functionality of the shell is listed in Table 4.1.

The core of the shell is optimisation algorithms. Other utilities provide the functionality needed for the definition of the problems to which optimisation algorithms are applied. In this respect utilities that enable the definition of the direct analysis are the most important, which especially refers to interfacing with the simulation environment. Open structure of the shell enables interfacing with any simulation environment. An interface module[22] for a finite element system Elfen[30],[31] has already been implemented.

A general file interface[19] enables interfacing with any programme for which a special interface module is not implemented, through its input and output files. These modules provide a set of basic utilities for manipulating text files, such as searching for data, reading and updating data, copying data, etc.

Additional support for the definition of the direct analysis is offered by the expression evaluator. It can be used in combination with the file interpreter capabilities to specify how the quantities required by an optimisation algorithm are derived from basic results obtained by a numerical simulation. It can be regarded in this respect that additional post-processing of results, which is not provided by the simulation environment but is needed for formation of information required by algorithms, is taken over by the shell.

**Table 4.3:** Principal modules of the optimisation shell inverse.

| |
|---|
| **Optimisation module**[18] includes optimisation algorithms and other tools (e.g. tabulating utilities, support for Monte Carlo simulations, etc.). It also includes utilities for definition of direct analysis, including organisation of data transfer between analysis definition and optimisation algorithms. |
| **File interpreter**[15] represents the shell user interface. |
| **Flow control** module includes implementation of branches and loops, a function definition utility, and some other flow control utilities. |
| **Syntax checker**[20] enables checking command file syntax before running it. Some troublesome errors such as parenthesis mismatches can be detected by this tool. Arguments are also checked for some basic interpreter commands (e.g. for flow control commands). |
| **Debugger**[20] allows step-by-step execution of commands, execution of arbitrary portions of code, checking and changing values of variables between execution, etc. |
| **Expression evaluator** (**calculator**)[16] evaluates mathematical expressions which appear in argument blocks of file interpreter commands. |
| **Variable handling module**[17] includes basic operations on variables such as creation and deleting, copying, initialisation, etc. |
| **General file interface**[19] provides a set of functions for interfacing simulation and other programmes. |
| **Interfacing** modules provide tools for interfacing specific simulation programmes, which includes execution control and data exchange functions. |
| **Miscellaneous utilities module**[21] include various auxiliary utilities, for example utilities for interaction with the operating system. |

Various auxiliary utilities[21] can be used to control the solution process or provide additional support to the interfacing module. The most important are output commands, which enable the user to output any information of interest to the terminal or shell output file. Other utilities enable control of execution and CPU time and interaction with the file system (changing directories, deleting files, etc.).

The file interpreter[15] represents a user interface, which provides access to the shell utilities. Accessibility of the shell functionality through the file interpreter is the basis of flexibility, which enables the shell to be applied to a large variety of problems. The ability of installing new file interpreter commands is a basis of openness of the shell as regards the possibility of implementing new tools that interact with the existing functionality. An open library provides an implementation interface for building in new tools in accordance with the shell concepts. A part of this library consists for example of functions for interpretation of arguments of file interpreter commands.

Instruments for checking the shell execution and correctness of user definition of the problem are a significant part of the shell. A user interface implemented as a file interpreter imposes a high level of flexibility on one hand, but on the other hand definition of problems with such an interface is prone to errors. This is especially true when a lack of high level commands is experienced and must be overcome by using programming capabilities of the shell user interface to a large extent.

**Table 4.4:** A list of debugger commands with brief descriptions.

| | |
|---|---|
| **?** prints a short help.<br>**q** finishes the debugging process.<br>**s** executes the next file interpreter's command.<br>**S** executes the next file interpreter's command; commands that execute code blocks are executed as single commands.<br>**n** *num*. Executes the next *num* commands.<br>**N** *num* executes the next *num* commands; functions that contain code blocks are executed as single commands.<br>**x** *num* executes the code until *num* levels lower lever of execution is reached. Default value for *num* is 1. | **e** *expr* evaluates the expression *expr* by the expression evaluator. If *expr* is not specified the user can input expression in several lines, ending with an empty line.<br>**w** *expr* adds expression e*xpr* to the watch table. Without the argument, values of all expressions in the watch table are printed.<br>**dw** *num* removes the expression with serial number *num* from the watch table.<br>**aw** *switch* with *switch* equal to zero turns automatic watching off; otherwise it turns it on.<br>**pw** prints all expressions in the watch table. |
| **c** executes the code until the next active **break** command is reached.<br>**ab** *id* activates all breaks with the identification number *id* ("*" means all identification numbers).<br>**sb** *id* suspends all breaks with the identification number *id* ("*" means all identification numbers).<br>**pb** prints information about active breaks.<br>**tb** *id* prints status of breaks with identification number *id*. | **r** *comblock* interprets *comblock* by the file interpreter. If *comblock* is not specified the user can input commands in several lines, ending with an empty line.<br>**rd** *comblock* does the same as ***r***, except that the code is also debugged.<br>**rf** *filename* sets the name of the file into which the user's commands will be written, to *filename*. |
| **v** *shift* prints a segment of code around the current viewing position shifted for *shift* lines.<br>**vr** *shift* prints a segment of code around the line of interpretation shifted for *shift* lines.<br>**va** *linenum* prints a segment of code in the interpreted file around the line *linenum*.<br>**nv** *num1 num2* sets the number of printed lines before and after the centerline when the code is viewed. | **Breaks** are set in the command file by function **break**, whose argument (optional) is break identification number, e.g.<br><br>`break { 3 }` |

The shell contains two tools, which facilitate location of errors[20]. The syntax checker detects some common and obvious syntax errors such as misspelling of commands and mismatched brackets. It can be applied to check the command file before the shell is run. The debugger (Table 4.4) enables tracing an execution of the shell. It enables step by step interpretation of the command file between which the state of the calculator and shell variables can be inspected or changed. The debugger is a useful tool not only for detection of logical errors in the command file, but also for detection of unexpected results of various built-in tools or stand-alone programs which are employed in problem solution.

### 4.2.6 A Simple Example

A simple example[13] is shown in order to highlight the shell function discussed in previous sections. The example shows how the following problem can be solved by the shell:

$$\begin{aligned} &\underset{x,y}{\textit{minimise}} && x^2 + y^4 \\ &\textit{subject to} && y \geq (x-3)^6 \wedge y \geq 17 - x^2 \end{aligned} \tag{4.2}$$

The objective and the two inequality constraint functions are

$$\begin{aligned} f(x,y) &= x^2 + y^4 \\ c_1(x,y) &= y - (x-3)^6 \\ c_2(x,y) &= x^2 + y - 17 \end{aligned} \tag{4.3}$$

The command file which makes the shell solve this problem is the following:

```
setfile{outfile quick.ct}

*{ Objective and constraint functions: }
${f[x,y]: x^2+y^4 }
${g1[x,y]: -((x-3)^6-y) }
${g2[x,y]: -(17-x^2-y) }
*{ Objective function derivatives: }
${dfdx[x,y]: 2*x }
${dfdy[x,y]: 4*y^3 }
*{ First constraint function derivatives: }
${dg1dx[x,y]: -(6*(x-3)^5) }
${dg1dy[x,y]: 1 }
*{ Second constraint function derivatives: }
${dg2dx[x,y]: 2*x }
${dg2dy[x,y]: 1 }

setvector{parammom 2 {} }
newscalar{objectivemom}
newscalar{constraintmom[2]}

analysis
{
```

Analysis block

```
20.      ={x:getvector["parammom",1]}
21.      ={y:getvector["parammom",2]}
22.      setscalar{objectivemom ${f[x,y]} }
23.      setvector{ gradobjectivemom
24.        {  ${dfdx[x,y]}  ${dfdy[x,y]} }
25.      }
26.      setscalar{constraintmom[1] ${g1[x,y]} }
27.      setvector{ gradconstraintmom[1]
28.        {  ${dg1dx[x,y]} ${dg1dy[x,y]}  }
29.      }
30.      setscalar{constraintmom[2] ${g2[x,y]} }
31.      setvector{ gradconstraintmom[2]
32.        {  ${dg2dx[x,y]} ${dg2dy[x,y]}  }
33.      }
34.    }

35.    setvector{parammom { 0 0 } }
36.    analyse{}

37.    optfsqp0{ 1 2 0 0 0 0.00001 0.00001 300 1
38.      { 2 { 15 -3 } }
39.      { 2 {} }
40.      { 2 {} }
41.    }
```

The *setfile* command in line 1 creates the shell output file *outfile* where functions will write their reports and error reports, and connects this file with the physical file named "quick.ct".

Lines 3 to 14 contain some preliminary definitions of new expression evaluator functions, which will be used later in the analysis block. These are the objective (line 3) and both constraint functions (lines 4 and 5), derivatives of the objective function with respect to the first (line 7) and the second (line 8) parameter, and derivatives of the first (lines 10 and 11) and the second (lines 13 and 14) constraint function with respect to both parameters.

In lines 15 to 16 we create variables with pre-defined meaning *parammom*, *objectivemom* and *constraintmom*. The aim of this is merely to specify the relevant characteristic dimensions of the problem. These are stored in internal variables of the shell and are used when creating pre-defined variables whose dimensions are by definition equal to these characteristic dimensions. By creating vector *parammom*, the number of parameters *numparam* is defined, by creating scalar *objectivemom* the number of objective functions *numobjectives* is defined and by creating scalar variable *constraintmom* the number of constraints *numconstraints* is defined. No values are assigned to these variables. The same effect as creating *parammom* would have been obtained for example by creating *paramopt*, and creating vector *gradconstraintmom* could replace both creating vector *parammom* and scalar *constraintmom*, since both *numconstraints* and *numparam* are relevant for this variable.

Lines 20 to 33 form the analysis block, which represents the definition of the direct analysis and is interpreted at every analysis run. This block specifies how relevant quantities such as the objective and constraint functions and their derivatives are evaluated at a specific set of optimization parameters.

In lines 20 and 21 the current values of parameters are stored in expression evaluator variables x and y. These values are obtained during optimisation from vector *parammom* where they are put by the general analysis function, called by the algorithm that requests execution of a direct analysis.

In lines 22 to 33 the relevant quantities are evaluated and stored into the appropriate pre-defined variables where the calling algorithm can obtain them. The value of the objective function is stored into scalar *objectivemom* (line 22), its gradient is stored into vector *gradobjectivemom* (lines 23 to 25), values of the constraint functions are stored into scalar variable *constraintmom* (lines 26 and 30), and their gradients to vector variable *gradconstraintmom* (lines 27 to 29 and 31 to 33) Auxiliary functions, which were defined in lines 3 to 14 of the initialisation part are used, called with the current parameters stored in calculator variables x and y (lines 20 and 21). In more realistic cases this part would include running some numerical simulation at the current parameters, the necessary interfacing with the simulation *programme* (for updating simulation input and reading results) and possibly some housekeeping for deriving final values from the simulation results.

A test analysis at parameters $[0,0]^T$ is run in line 36 by the *analyse* command. This command takes parameter values from the pre-defined vector *parammom*; which is set in line 35.

Finally, the problem is solved using the command *fsqp0*, which runs the feasible sequential quadratic programming optimization algorithm (lines 37 to 41). This function requires nine numerical arguments, namely the number of objective functions, the number of non-linear inequality constraints, the number of linear inequality constraints, the number of non-linear equality constraints, the number of linear equality constraints, the final norm requirement for the Newton direction, the maximum allowed violation of nonlinear equality constraints at an optimal point, the the maximum number of iterations, information on whether gradients are provided or not, and three vector arguments, namely the initial guess and lower and upper parameter bounds.

Suppose that the above command file has been saved as "quick.cm" and that the shell programme is named "inverse". We can run the shell by

```
inverse quick.cm
```

which solves the problem (4.2). The report including final results can then be checked in the file "quick.ct".

## *4.3 Selected Implementation Issues*

### 4.3.1 Programming Language and Style

The present section discusses some basic implementation issues, which influence the shell function, effectiveness and economy of its development.

As regards computer programming, the same operations can usually be implemented in a number of different ways. It is the function of a programme that matters the most, however there can be big differences in programme efficiency and final development cost between different implementations of the same system. Awkward implementation usually results in unexpected bugs and unnecessary problems with inefficiency, which can never be completely disclosed at the testing stage.

Careless programming frequently results in a rigid system, which serves its purpose well, but it is hard to introduce changes and add new functionality. For complex systems it is impossible to predict all requirements that can possibly arise from application needs. System design can therefore not be completely planned in advance and it is particularly important that it is easy to introduce changes in the system and expand its functionality.

Since the implementation style has a strong influence on the overall quality of the system and economy of its development, it deserves special attention. The desired system properties often impose conflicting implementation demands, therefore it is difficult to set generally applicable implementation rules. What is appropriate depends on what should be achieved.

A common conflicting situation in programming is induced when the need for maximum efficiency arises simultaneously with the requirement to make the system open and accessible to different developers. The last demand is achieved if the system is logically structured and its function implemented through small closed units. Such implementation leads to a certain overhead of function calls and allocation of data, which could otherwise be reused, which to some extent affects the programme efficiency. The loss of efficiency is in some cases negligible and can be sacrificed without hesitation. In other cases a compromise must be accepted after consideration of solutions that are acceptable in view of openness and still have a lesser impact on efficiency.

Knowing programming rules means being aware of the effects of different programming approaches. Implementation of complex systems is to a large extent the art of making good compromises. This requires a significant amount of planning, sometimes in a very abstract sense because situations in which the system might be exposed can be foreseen only to a limited extent.

The optimisation shell Inverse is programmed in ANSI C[24]-[26]. This is an extension of the traditional Ritchie’s C[23]. It is implemented in more or less an invariant way on all modern platforms and is currently one of the most portable programming languages[1]. C is a terse and logical language with a small set of keywords and powerful set of operators which support low level access to computer capabilities. It still provides all facilities typical for high level languages, such as structured data types.

Unlike many other programming languages C does not impose unnecessary restrictions which do not arise from the computer architecture. Many of these restrictions imposed by other languages are in some cases extremely difficult to overcome. C completely supports dynamic memory allocation. Arrays can have variable length. This is for example not the case in Pascal, which makes it difficult to handle matrix operations in a modular way. Function addresses are treated in a natural way in C and can be assigned to variables. This enables the fully dynamic treatment of function calls, which can in some languages be achieved only by passing function addresses through function arguments (as in FORTRAN) or not at all (as in Pascal). A favourable feature for programmers who wish to have a complete understanding of code function is that function arguments are always passed by value. Effects equivalent to passing by reference in some other languages are achieved by passing a pointer to a variable instead of the variable itself. In many other languages this is done implicitly, so that the language rules hide to a great extent what is actually happening. Having complete insight in the code function therefore requires a deeper knowledge of the programming language, which is not true for C. A similar example is pointer arithmetic, which closely follows native computer logic.

An often heard argument against the use of C in numerical applications is that programmes written in C are slower than for example programmes written in FORTRAN. The author of this text is not aware of any theoretical arguments or

[1] Experience show that in practice there are minor differences between various implementations of ANSI C. However, the number of particularities is small and they can be kept under control.

comparable tests which would support such statements[1]. It is however possible to use C for specific tasks in an inefficient way, which is less usual in other programming languages because of their restrictions. This simply means that it is less likely in some languages that an unskilled programmer would implement specific tasks in an inefficient way. A typical example is unnecessary overuse of dereference inside iterative parts of the code, for example in matrix operations.

C does not directly support object oriented programming in a way as C++ does[28]. Object oriented programming was introduced to support more human-like formulation of ideas in programming languages and a more open structure of programmed modules. An especially strong feature of object oriented programming is that commonality between different ideas can be made explicit by using inheritance. Therefore it is possible to relate similar ideas in a natural way, which makes the code clearer. There are some examples in the shell where advantages of object oriented programming could be used. Such situations are however not typical and use of plain C does probably not represent a great loss in terms of development efficiency.

The shell consists of a number of hierarchically arranged modules. Modules represent closed units, which provide a given kind of functionality and make it available to other parts of the programme. The design of a module includes definition of related data types and a complete set of operations that can be performed on these types. This leads to a given functionality, which represents realisation of some a given idea in the programming sense.

Programming in a modular way allows concentration on one type of problems at a time. Solutions are provided independently of other problems. An important gain of such an approach is that functionality can be tested independently for small and well defined units, which significantly reduces testing complexity. Modular programming in C significantly reduces the possibility of memory handling errors, which are among the most problematic and common errors in C. Such errors can be avoided if all memory allocation and deallocation is performed by functions provided by the appropriate modules, which are designed so that they exclude the possibility of common errors such as accessing memory through bad pointers, releasing the same pointer several times, etc.

#### 4.3.1.1 Example: the Stack Module

A typical example is the stack module. This module introduces stacks of objects and provides a complete set of operations on such data structures. Objects on

[1] It is possible that this common opinion was influenced by inefficiency of the C compilers at the early stage of the language development.

stacks are represented by their pointers, which are of type *void **, so that any type of objects can be stored on such stacks.

The module does actually not provide only the push and pop utility, which are typical for stacks in a common sense, but also insertion of an object to a given position, deletion of an object on a given position, sorting of objects according to a specified criterion, etc.

The stack type is defined in the following way:

```
typedef struct{
  int n,r,ex;
   void **s;
} _stack;

typedef _stack *stack;
```

Type *stack* is defined as a pointer to a structure of the type *_stack*. It is common in the shell that objects are presented by pointers of a given type. Another commonly accepted rule is that all pointers are initialised to NULL. This pre-defined value (which is essentially 0 on all modern systems) is used as an unambiguous indication that the specific object is not allocated.

Structure member (or field) *s* is the array of pointers, which holds objects that are on the stack. The structure also contains three integers. *n* is the number of objects that are on the stack, *r* is a number of allocated pointers in the array, and *ex* is an auxiliary field which defines the excess of memory allocated for the table *s* when it runs short of space to hold objects that are added to the stack. The possibility of allocating more memory that is currently needed allows that the array *s* is not reallocated every time a new objects is added to the stack.

All possible operations on stacks are provided by the functions which are defined in the module. These functions take care that the *stack* type is always used in a prescribed way, which excludes the possibility of errors. Internal rules that ensure proper function are hidden to the user of the module, which will be shown on some specific functions provided by the module. The user must only be aware of module functionality and some general rules for using the module.

The basic operations on any type of complex objects are creation and deletion. Stack objects are created by function *newstack*, which is declared as

```
stack newstack (int excess);
```

The function creates a new stack object and returns a pointer to it. It takes an integer argument, which specifies the value of the field *ex* of the created stack. The function also allocates the table of pointers *s* so that it can hold *ex* pointers. *r* is set to *excess* and *n* to 0.

Deletion of a stack object is achieved by the function *deletestack*, which is declared as

```
void dispstack(stack *st);
```

The function requires a pointer to stack, which must be the address of the stack to be deleted. This enables the function to set the value of the deleted stack to NULL after performing other necessary operations on it (note that arguments in C are passed by value). In this way other functions that would operate on the same stack can detect that the stack no longer exists. The function *dispstack* first releases the memory used by the field *s* (if it is allocated). Then it releases the memory used by the structure itself. This memory was dynamically allocated by the *newstack* function and can be used by other objects after it is released. The *dispstack* function checks if the memory to be released is allocated (this is detected through pre-defined value NULL, which is generally used for pointers that are not initialised). This excludes the possibility of trying to release the same pointer twice, which is an error that on most systems causes abnormal programme termination. The user of the stack module does not need to take care of the possibility of such errors, because all necessary mechanisms are built into functions provided by the stack module. The only concern is that all pointers to objects are initialised to NULL before they are used. This is a general rule of safe C programming and is strictly obeyed in shell development.

The *dispstack* functions does not affect objects that are on the stack. If pointers to these objects reside only on the stack that is being deleted, these objects must be deleted prior to deletion of the stack, otherwise the memory occupied by them is not accessible any more and is a permanent waste untill the end of programme execution. Deletion of objects on the stack can be performed explicitly by deleting objects one by one. The module also provides a function for deletion of all objects at a time, which is declared as

```
void dispstackvalspec(stack st,void (*disp) (void **));
```

This function requires two arguments. The first argument is the stack (a pointer) whose elements will be deleted. The second argument (*disp*) is the function which is used for deletion of each individual object. The function deletes all objects on the stack by calling function *disp* with their addresses as arguments. *disp* must be a function that is equivalent to *dispstack* for the type of objects that reside on the stack. It is also supposed that the function sets the pointer value to NULL after the object pointed to by that pointer is deleted.

There also exists the function *dispstackallspec* which does deletion of objects contained in a stack and the stack itself. Its declaration differs from declaration of *dispstackvalspec* in that the address of the stack must be passed as the first argument rather than the stack itself, because this is required for deletion of the stack. The declaration of the function is the following:

```
void dispstackallspec(stack *st,void (*disp) (void **));
```

It is appropriate to give an implementation note at this stage. The *dispstackallspec* function could be implemented in the following very simple way:

```
void dispstackallspec(stack *st,void (*disp) (void **))
{
  dispstackvalspec(*st,disp);
  dispstack(st);
}
```

Functions *dispstackvalspec* and *dispstack* perform all that is necessary for the function *dispstackallspec*. However, by implementing the function this way we have two additional function calls inside the function. From the point of view of efficiency it is better that bodies of both functions are explicitly repeated within the function, so that these two calls are avoided while other code that is executed remains the same. This will make the appearance of the function more complex, which is not important since the user of the stack module will typically not interfere with the function definition but only with its declaration. Also the compiled programme that uses the module will be of a slightly larger size, but this is not problematic because the code of the function body appears only in one place within the programme, while the function will be typically called many times. The effect on efficiency is in this case more important than the effect on the code size.

Basic operations on stacks are push and pop. Push adds an objects on the top of the stack, while pop takes an object from the top. Their declarations are

```
void pushstack(stack st, void *el);
```

and

```
void *popstack(stack st);
```

Both functions require the stack on which the operation is performed as the first argument. The second argument of the *push* function is the object (a pointer), which is pushed to the stack. The function adds this pointer to the table *s* of the stack after the last object on it and increments the field *n* which holds the number of occupied places. If the table of pointers *s* does not contain enough space to hold a new element, it is reallocated. Existing objects that are already on the stack are kept in the new table. The number of elements for which *s* is allocated is always written in the field *r*, which excludes the possibility of mistakes. Whenever the table is reallocated or deleted by any function of the module, the *r* field is updated.

Function *popstack* picks the last object on the stack and returns it as a pointer of undefined type (void *). This pointer can be assigned to any variable which is of the same type as the object obtained from the stack. It is the user's responsibility to ensure that the types match. The type agreement could be more easily ensured in object oriented languages such as C++, in a sense that errors regarding type compatibility would be detected during compilation time. In practice this has not shown to be a serious problem, because stack objects are usually used in a limited scope where it is not hard to mix up pointer types.

The *popstack* function also decrements the value of the *n* field of the stack. It does not set the value of the last pointer in the array *s* to NULL, because since *n* is reduced, that pointer is out of the range and can never be accessed. When an object is picked from the stack, the array of pointers is larger than necessary. If the difference between the number of pointers which the array can hold (indicated by the field *r*) and the actual number of pointers on the stack (indicated by the field *n*) is larger than the value of the field *ex*, *s* is reallocated so that its physical size matches the number of objects which it holds.

Beside the above mentioned functions, the stack module contains many other functions, which allow the user of the module to perform the necessary operations. Among important functions provided are searching for an object with certain properties and sorting of objects on the stack according to a provided comparison operator.

The aim of this brief description of a module is to show the role of such closed modules in construction of the programme. A module consists of data types and a full set of operations on these types, which together represent implementation of some idea. In the case of the stack module the underlying idea is abstract and very general. It implies that multiple objects of a given type can be arranged in a stack, so that objects can be taken from the top or added to the top of the stack, sorted, searched for, etc.

The module hides all implementation details that are not relevant for use of the module. In other words, its user does not need to know much more about the module than what it is used for and how it is used. For a programmer who wants to use stack functionality it is not important what the structure of the stack object is. It is important for example that there exists a function for pushing a new object to a stack, that the objects on the stack can be sorted and that searching for objects on a sorted stack is much quicker than searching on unsorted stacks. The user must be aware of the underlying ideas, but on an abstract level which has nothing to do with implementation inside the module. Implementation of the module functions can be changed (e.g. in order to improve efficiency or eliminate bugs) without changing function declarations, which represent the implementation interface and the only interaction point of the user with the module. Introducing changes inside a well designed module therefore does not affect any portion of code where this module is used. The same reasoning as for function applies to data types, which can be extended without affecting any portion of the code outside the module. The only consequence of such changes is that the programme must be recompiled.

The advantage of such closed modules is also that their function can be tested in a very limited scope, which makes it easier to find bugs before the module is actually used. Modules prevent unnecessary code replication. When similar ideas arise on different places, the same implementation is used everywhere. This implies also that bugs are likely to show on more than one place, which makes their detection easier. When for example a bug in the stack module is detected through errors in a part of the code where the idea of stacks is used, the bug is eliminated once (in the module) and this corrects the function of any portion of code where the same idea is used. This can be especially beneficial for bugs which show in rare and random occasions.

The idea of stacks as described above is very basic. The stack module has therefore a low rank in programme hierarchy and is used in many other modules. For example, stacks are used to hold shell variables, calculator variables, operators and functions, interpreter functions, etc. Stacks are often used for holding multiple input or output arguments of the same type, but variable number. For example, a function that finds all occurrences of a given string in a file returns string positions on a stack.

In many modules the idea of stacks arises within a broader context, which form the basis of the module. More complex derived data types therefore include stack objects as fields in their structure. All operations provided by the stack module can be performed on these fields, which means that an existing idea already implemented in a closed module is incorporated as a part of a broader idea. The new type inherits properties of stacks in a way, which resembles some concepts of object oriented programming.

The concept of inheritance can also be observed in a reverse way. We can have stacks of objects of different types, e.g. a stack of matrices or a stack of vectors. Various objects are implemented through completely different structures and sets of functions that can operate on them. Different objects are also deleted in different ways, and when we want to delete a whole stack of objects, this procedure must inherit specifics of deletion of the objects of a given data type. The mechanism of deletion of a stack of objects has already been described above in connection with the function *dispstackvalspec* and *dispstackallspec*.

Figure 4.5 shows the organisation of the main shell modules.

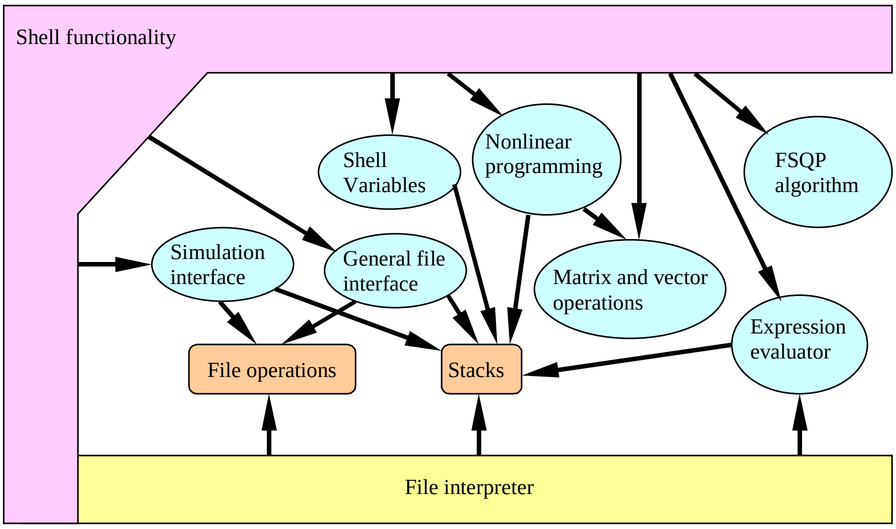



**Figure 4.5:** Organisation of shell modules. Arrows show dependencies. The scheme is not complete.

### 4.3.2 File Interpreter and Open Library

All shell functionality is accessed through file interpreter commands. The file interpreter therefore deserves special attention. Core data type of the file interpreter is the *_ficom* type, which is fundamentally defined as follows:

```
typedef struct{

  . . . /* syntax definition */

  char stopint;

  . . . /* support to flow control */

  stack functions;
  FILE *fp;
  char *filename;
  long from,to;

  . . . /* support to user defined functions */

  int which;
  fifunction func;
  long pos,begin,end,argpos;

  . . . /* temporary files */
```

```
   FILE *in;
   FILE *out;
   ssyst syst;

   . . . /* support to definition of calculator functins */

   . . . /* support to tracing of calling sequence */

   char debug, check;
   licom lint;

   . . . /* support to debugging */

} _ficom;
```

Only the most important fields are written. Comments indicate the missing groups of fields. File interpreter is an object of type *ficom*, which is a pointer to the above structure and is defined as

```
typedef _ficom *ficom;
```

Interpretation is performed by the function fileinterpret, which is declared as

```
void fileinterpret(ficom com);
```

Its only argument is an object of the type *ficom*, which actually represents the interpretation and is also referred to as the interpretation object. More than one interpretation can be independently performed within a programme, provided that there is more than one interpretation object. The corresponding object must be initialised before the interpretation begins. Memory for its structure must be allocated and the file name (field *filename*) and scope of interpretation (fields *from* and *to*) must be set.

An object of the type *ficom* holds all data relevant for interpretation. Beside the state of interpretation (e.g. current position of interpretation and information about the executed command) it also holds data which instructs the interpreter how to perform interpretation (e.g. in debugging mode or not). Function *fileinterpret*, which performs the interpretation, therefore does not need any local variables. This function is sometimes referred to simply as the interpreter, although this term is in its broadest meaning used for the whole interpreter module.

The field *stopint* tells the interpreter to exit interpretation if the field value is different from zero. This field is set by some functions or automatically if the interpreter hits the end of the interpreted code.

The field *fp* is the file pointer of the interpreted file. The interpreter and functions called by the interpreter accesses the interpreted file through this pointer.

The name of the interpreted file is stored in the field *filename*. It is used by the interpreter to open the file if it is not yet open. It is also used by some other functions, for example by those which report errors.

The interpreter can be used for interpretation of a part of a command file. Fields *from* and *to* hold the beginning and the end of the block which should be interpreted (zero values indicate that the whole file should be interpreted).

Interpreter commands are installed on the field *functions*, which is a stack. Objects on this stack are of the type *fifunction*. Such an object holds a command name and address of the function that corresponds to the command. When the interpretation object is initialised, its own commands such as looping and branching commands are installed on this stack. The shell installs additional commands through which its functionality can be accessed before it runs the interpreter. The *fifunction* type is defined in the following way:

```
typedef struct{
  char *name;
  void (*action) (ficom);
  void (*checkaction) (ficom);
} _fifunction;

typedef _fifunction *fifunction;
```

Field *name* is the name of the installed interpreter command, *action* is the function that corresponds to the command, and *checkaction* is the function that checks the syntax of command arguments. The last function can be executed only when the syntax checker is run.

Fields *which*, *func*, *pos*, *begin*, *end* and *argpos* are set by the interpreter and contain information about the command that is currently being interpreted. *which* is the position of the interpreted command on the stack *functions*, *func* is the corresponding object on this stack, and other fields are positions in the interpreted file. *pos* is the position of the command, *begin* and *end* define the position of the command argument block and *argpos* is an auxiliary field used by functions which correspond to interpreter commands. These functions use the field at interpretation of command arguments. *argpos* is set to the same value as *begin* by the interpreter.

While interpretation takes place, the interpreter searches for commands in the command file. When another command is found, the interpreter sets the field *pos* to its position and finds the corresponding object (i.e. the object with the command name) on the stack *functions*. Such an object represents the definition of the command and contains the address of the function that corresponds to the command (see the declaration of *fifunction* above). The interpreter sets the field *which* to the position of the object on the stack and the field *func* to the object itself, or reports an error if the corresponding object does not exist. It then finds the position of the command argument block (which is enclosed in curly brackets) and sets the fields *begin*, *end* and *argpos* to the appropriate values. Finally it calls the function, which corresponds to the command and whose address can be accessed through the field *func* (i.e. *func->action*). This function takes the interpretation object as argument. This gives the function access to relevant information concerning the interpretation, especially information about command argument block, which is needed for interpretation of arguments.

The interpreter module provides some basic command such as commands for controlling execution flow and some input and output commands. Fields *in* and *out* hold input and output files of the interpreter. It is not necessary that these files are defined during interpretation. The shell installs its output file to the field *out* when that file becomes defined (which is on user command in the command file). This way the output commands which are defined in the interpreter module use the same file as the commands which are additionally installed by the shell.

The expression evaluator offers a similar example. This module is similar to the interpreter module in that several independent expression evaluators (with their own private set of user defined variables and functions) can exist in a programme at the same time. This is however not the case in the optimisation shell because there is no need to have more evaluators. Some basic file interpreter commands need the expression evaluator for evaluation of branching and looping conditions. It is represented by the field *syst*. The commands that need the expression evaluator access its function through this field. It is of the type *ssyst*, which has a similar meaning for an expression evaluator as the type *ficom* has for an interpreter. The optimisation shell initialises the expression evaluator and installs it in the file interpreter before it starts interpretation of the command file. Other functions of the shell that use the expression evaluator therefore use the same object as the file interpreter.

Initialisation of the expression evaluator prior to file interpretation is not compulsory. If none of the commands that need the expression evaluator is ever used, then the interpreter can run without it. Such a situation can actually occur if an interpreter is used for some specific purpose where the expression evaluator is not needed. This illustrates the flexibility of modules such as the file interpreter. The expression evaluator is a complex system that needs substantial memory in order to function, therefore it is beneficial if the interpreter may not use it when this is not necessary.

Fields *check* and *debug* hold instructions for the file interpreter. If *check* is nonzero, then the command file is just checked for syntax errors rather than interpreted. The interpreter can only find errors which concern its function and rules. This excludes errors in command arguments since the interpreter itself has nothing to do with interpretation of arguments. The interpreter allows functions to be installed that check argument syntax for commands that are installed on its system. These functions are specified in the *checkaction* field of the object of the type *fifunction* (see its declaration above). Such objects are a representation of installed command on the stack field *functions* of the file interpreter object. If for a specific command the field *checkaction* is not NULL then the interpreter runs this function to check command arguments. The only argument of the function is the interpreter object itself, through which the checking function can find all necessary data, including the position of the argument block and the file pointer through which the command file is accessed.

If the *debug* field is nonzero, the interpretation is performed in the debug mode. A number of control parameters are used for telling the interpreter how to function, e.g. how many commands to interpret at a time or how many interpretation levels to exit. After interpretation of a specified portion of the code, control is passed to the user. A line interpreter is run in which the user inputs instructions for the interpreters through a command line. Basic debugger commands that can be used in this place are summarised in Table 4.4. Functions that carry out debugger commands are installed on the line interpreter system. Some of these functions give instructions to the interpreter through the appropriate fields on the interpreter objects. Such instructions are, for example, that only one command or a group of commands must be interpreted, or that a certain number of interpretation levels must be left. Other functions perform concrete actions, for example change the definition of the expression evaluator variables and functions or print variable values. In the debugger, the user can also run arbitrary interpreter commands. In debugger it is therefore possible to check directly the effect of changes in the command file on function of the shell.

Shell functions that correspond to interpreter commands usually just extract arguments and call other functions to perform algorithms and other tasks. Such a two stage arrangement is evident from Figure 4.3. Its advantage is that the shell side of the implementation is separated from the module which is a source of a specific kind of functionality. Different modules can therefore be developed independently. The functions that correspond to interpreter commands take care of proper incorporation of functionality provided in the shell system. The most basic task in this respect is data exchange between the shell and module functions. This is done through arguments of interpreter commands in accordance with argument passing conventions, which were described in section 4.2.4.

Argument passing mechanisms are facilitated by a set of shell functions for interpretation of command arguments and for interaction with the shell variable system. These functions are a part of the shell open library[1], which represents an implementation interface for incorporation of any kind of functionality within the shell.

Functions of the shell open library are mostly used for implementation of interpreter functions, which correspond to interpreter commands and provide interface between the shell and the incorporated modules (Figure 4.6). Library functions hide unnecessary implementation details. They also provide a certain level of invariability with respect to changes in the shell structure and especially with respect to changes in incorporated modules. This implies that the shell itself can be treated as a module when new functionality is incorporated.

Figure 4.6 schematically shows the operation of an algorithm incorporated in the shell. A part of this scheme can be recognized in Figure 4.3. Relations between the incorporated algorithm, shell interpreter, interpreter function that provides an interface between the shell and the algorithm, and library functions which facilitate implementation of this interpreter function, are shown in more detail.

The implementation interface facilitates incorporation of new utilities in compliance with the shell conventions, but does not impose these rules a priori. Designers of new modules are given freedom to introduce individual rules for use of specific utilities. Interaction with the shell structure, function and philosophy is possible on different levels. For example, it is not necessary to use shell functions for interpretation of command arguments of specific types. Low level library functions enable more basic interaction with the file interpreter, such as direct access to the command file and position of the argument block. For specific commands individual rules can be set for interpretation of their arguments. Such low level interaction enables the introduction of additional concepts in the shell if they are necessary for a given functionality. For example, a new data type can be introduced together with complete support as offered for existing data types. Rules for passing objects of that type through command arguments can be imposed through functions that interpret arguments of that type and are added to the shell library.

[1] The term open library is used because functions in this library are designed for broader use. Many functions of modules which constitute the shell have global linkage, but only functions of the open library are designed for use outside the shell development team. These functions are designed with special emphasis on simplicity of use and invariability with respect to changes in the structure of the shell and its constitutive modules.

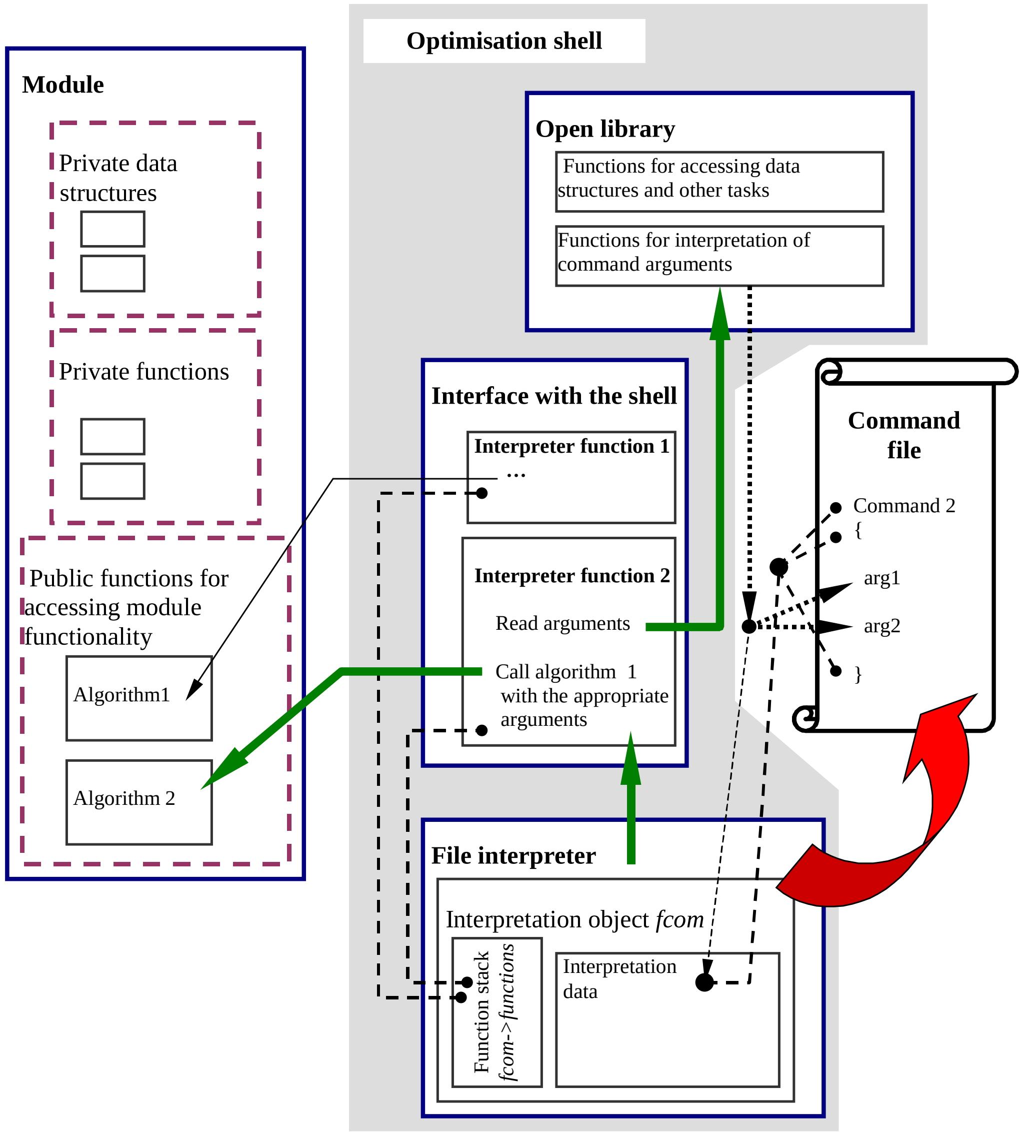



**Figure 4.6:** Incorporation of module functionality in the shell. Dashed lines show data connections. Continuous arrows indicate calling sequences.

The design of the implementation interface with the properties described requires some effort, especially careful planning, which sometimes includes formulation of ideas in a rather abstract way. The effort required for ensuring openness and flexibility can be justified if the shell is viewed as a general system designed for use and development by a broader community. Application to practical problems gives rise to many subproblems of a heterogeneous nature. Implementation of appropriate optimisation tools therefore by far exceeds merely the scope of implementation of effective optimisation algorithms. It requires an integrated approach to the development of an optimisation system, which facilitates incorporation of expertise from different fields. Support to distributed development with sufficient implementation freedom and ability of introducing new concepts in the shell is essential from this point of view.

### 4.3.3 Incorporation of Optimisation Algorithms

Optimisation algorithms are the principal constituent of the shell functionality. Their incorporation in the shell follows similar rules as incorporation of other utilities, which was described in the final part of the previous section. There are some specifics, some of which are supported by functions in the shell open library, which are used specifically for incorporation of optimisation algorithms.

Each incorporated optimisation algorithm has a corresponding file interpreter function, which is installed on the function stack (field *functions*) of the interpretation object at the initialisation of the optimisation shell. Installation is performed by calling the open library function *instfintfunc* in a special portion of code, which is compiled and linked with the shell. Function *instfintfunc* takes the name of the installed file interpreter command and the address of the installed file interpreter function as arguments.

The interpreter function is an interface between the shell and the incorporated algorithm. This function is executed by the shell interpreter whenever it encounters the command installed together with the function (Figure 4.6). It reads command arguments and passes them to the optimisation algorithm which it calls. Reading of command arguments is performed by the appropriate open library functions.

The interpreter functions that correspond to optimisation algorithms normally provide a few additional things. It is a shell convention that the results of an optimisation algorithm are stored in specific pre-defined variables[1] (section 4.2.3). For example, optimal parameters are stored in the vector variable *paramopt* and optimal value of the objective function is stored in the vector variable *objectiveopt*. These values are normally output argument of the function which represents a given optimisation algorithm. It is the job of the appropriate file interpreter function to store these values in the appropriate places. This is performed by using the shell open library functions, which are designed for setting specific pre-defined variables.

Support to the above mentioned convention provides an evident example of what is needed for incorporation of a given functionality in the shell. The open

[1] This is a non-obligatory rule, which makes use of algorithm results by other utilities possible. A general rule is that if an algorithm does not follow this convention, this must be indicated in the appropriate manual.

library should in principle provide all necessary utilities, therefore a specific knowledge regarding the open library is a prerequisite. This requires some knowledge about the shell function, which however does not exceed the knowledge required for using the shell. In some particular cases additional knowledge is needed in order to incorporate the functionality in compliance with standard conventions. This knowledge does also not exceed the user level. The shell open library as an implementation interface complies with the requirements for interaction with the shell on a modular basis. The functions through which the shell functionality is accessed hide the implementation details and provide an interface which is invariant with respect to changes in the shell.

The above considerations are also instructive from the point of view of the implementation freedom. The shell and its implementation interface does not strictly imply the convention regarding storing algorithm results to specific pre-defined variables. Besides, implementation of the algorithm allows introduction of new rules that concern use of the incorporated functionality. For example, the optimisation algorithm might return the number of performed iterations in addition other results. The appropriate interpreter function could be implemented so that it would automatically store this result to some specific shell variable. This would introduce a new rule and would actually assign a meaning to that particular variable. The example is not characteristic because it would be much more elegant to assign the number of iterations to a variable specified by a command argument. However, it is possible that incorporation of some important sets of functionality (e.g. shape parametrisation) will be most conveniently implemented by introducing some additional general rules and new groups of pre-defined variables. This is fully supported by the shell implementation interface.

Optimisation algorithms iteratively require performance of the direct analysis, which include evaluation of the objective function and other quantities. Algorithms are usually implemented as functions, which take the function that performs a direct analysis as an argument. The file interpreter function that correspond to an optimisation algorithm must call such a function and provide the analysis function (i.e. its address) as an argument. Figure 4.3 and the surrounding discussion indicate how the optimisation algorithm in connection with the shell internal analysis function operates in practice, while section 4.3.2 and especially the discussion around Figure 4.6 explain how the algorithm is invoked through the shell interpreter command. An open question remains how different analysis functions with different sets of arguments are provided. It is clear that an open library can not contain all possible analysis functions, since the range of possible variants is practically unlimited.

Solution of this problem follows from the general arrangement regarding transfer of current optimisation parameters and results of the direct analysis between an optimisation algorithm and the direct analysis (section 4.1.3). There is a common function which performs a direct analysis for any kind of algorithm. It performs interpretation of the analysis block of the command file, which contains user definition of the direct analysis. The common analysis function takes no arguments. A specific analysis function must be defined for each optimisation algorithm. This function is actually called by the algorithm and is specified as an argument at algorithm call in the corresponding file interpreter function. It calls the common analysis function and besides writes optimisation parameters and reads analysis output data from the appropriate pre-defined shell variables. The common analysis function is provided by the open shell library as well as functions for setting and reading the pre-defined variables.

The shell library includes some most common analysis functions with different argument lists. These functions can sometimes be used directly in a call to an algorithm or are called in another intermediate function which covers particularities of the algorithm. Particularities arise in various ways. Some algorithms require derivatives and others do not. Some algorithms solve unconstrained problems and therefore do not require values of constraint functions. Different algorithms require analysis data in different form. Some of them use derived structures for representation of matrices and vectors, while some of them use arrays of numbers with separate arguments for specification of array dimensions. Algorithms are also programmed in different languages with different calling and argument passing conventions, which must be accommodated by using intermediate functions.

The most difficult example is when an algorithm uses two or more separate direct analysis functions, e.g. one for evaluation of the objective function and its derivatives and the other for evaluation of constraint functions and their derivatives. Such example is the FSQP algorithm, which is currently the principal optimisation algorithm of the shell. The shell typically calls one common analysis function for a given set of optimisation parameters. The algorithm calls first the function for evaluation of the objective function and its derivatives and then the function for evaluation of constraint functions and their derivatives. The first function must execute the common analysis function, store the constraints related quantities to a temporary location and return the objective function and its derivatives to the algorithm. The second function must simply pick the values and derivatives of constraint functions from the temporary location and return these quantities to the algorithm.

For a general optimisation system it is essential that it includes various types of optimisation algorithms, since any algorithm can handle effectively only a given set of problems. Various mathematical fields relevant for the development of optimisation algorithms are still developing. It is therefore important that different algorithms developed in different environments can be easily incorporated in the system.

On the other hand such distributed development is not economic because it leads to unnecessary replication of work. In practice this also means replication of code and therefore a large executable programme. Another consequence is rigidity of code where individual solutions can not be easily combined and applied in more complex algorithms because of obstacles related to different programming methodologies.

In the shell development several negative effects of disconnected development of algorithms have already appeared. By now several algorithms have been implemented such as the simplex method, Powell’s direction set method, the Fletcher-Reeves conjugate direction method, and some variants of the penalty methods (refer to chapter 3). These algorithms were programmed in a disconnected way and were many times incorporated in the shell in a nonmodular way. The disadvantage which soon showed was that changes in the shell affected interface between the shell and algorithms. It was difficult to follow permanent changes with a selection of algorithms, which showed the need for a more modular approach.

Unsystematic approach to development has usually a direct impact on code flexibility. In a rigid code it is difficult to change particular details and experiment with different variants of algorithms, which plays an important role in development of efficient algorithms. An evident example in this respect is offered by use of different line search algorithms, especially with respect to termination criteria. Exactness of line searches effects the overall efficiency of different methods in different ways, therefore it is important to allow different termination criteria. The basic idea of the line search algorithm is however independent of this and the algorithm can be programmed in such a way that its core is not affected much by application of different termination criteria. From the view of preventing code replication a good idea would be to implement line search algorithms in two levels. The first level function would contain actual implementation of a line search and would permit choosing between different termination criteria. Two or more higher level functions would be just interfaces for use of the first level function with one or another termination criterion. Algorithms would use these functions with respect to criteria which are better for a specific algorithm. Implementation of higher level functions contributes to clarity of code because their declarations are terse and do not show functionality which is not used in a given place. Implementing common operations only once on a lower level makes it easier to maintain the system and to introduce improvements. In the present example the interpolation can be improved only in the common low level line search function. The improvement affects all derived higher level functions without affecting the way how these functions are used in algorithms. Such a hierarchical system of functions must however be implemented with care. It causes some excess in function calls, therefore it must be checked if this has a significant effect on code efficiency.

Because of the reasons specified above, a more systematic approach to development of a modular optimisation library for the shell has been initiated. Stress is placed on the hierarchical structure of algorithms and utilities for the solution of various sub-problems, on invariant and easy to use implementation interfaces for these utilities and on exploitation of commonality on the lower level as much as possible. This should enable easier derivation and testing of different variants of algorithms while keeping complexity under control and preventing unnecessary code replication.

Generality of such a library was also taken into consideration as an important design aspect. In this respect it is important to accept some compromises with taking into account the purpose for which the library will be used. For example, it seems unlikely that in problems to which the shell will be applied, any special structure of linear algebra sub-problems that arise could play a crucial role. When the outline of the module for matrix operations was set, dealing with matrices of a special structure was not taken into account. Much stress was placed on the design of a clear and easy to use implementation interface and on dealing with characteristics which often play an important role in optimisation algorithms (e.g. testing positive definiteness of matrices). The basic design of the module for matrix operations is outlined in [29].

For an optimisation library designed for incorporation in the shell, many aspects which are not directly related to algorithms are relevant. This includes some general aspects of synchronous function within a broader system, such as error handling and output control. The library will provide flexible access to these functions with possibility of their adjustment to the system in which the library functions will be used. Considering straightforward incorporation of developed algorithms in the shell is important also from the point of view that the shell can provide a good testing environment.

#### 4.3.3.1 Example: Incorporation of an Optimisation Algorithm

A complete procedure of incorporating an optimisation algorithm in the shell is shown in an example. This should more evidently illustrate the properties of the shell implementation interface described above. The Nelder-Mead simplex algorithm was chosen to illustrate the procedure. It is implemented by a C function declared as

```
void minsimp(matrix simp, vector val, double tol, int *it, int
maxit, vector *paropt, double *valopt, double func(vector x),
FILE *fp)
```

Matrix argument *simp* is the matrix of simplex apices (input and output), vector argument *val* is the vector of function values in the simplex apices, *tol* is the tolerance in function value (convergence criterion – input), *it* is the number of iterations (output), *maxit* is the maximum allowed number of iterations (input), *paropt* is the vector of optimal parameters (output), *valopt* is the optimal value of the objective function (output), *func* is the address of the function that performs the direct analysis, and file argument *fp* is the file in which results are written. *matrix* and *vector* are derived types which represent matrices and vectors. Similarly as there exists a module for manipulating *stack* objects described in section 4.3.1.1, corresponding modules exist for manipulation of objects of these two types.

First the analysis function, which will be passed as argument *func* of the algorithm function, must be defined. The function must take a vector argument (optimisation parameter) and return a floating point number (value of the objective function). It can be defined in the following way:

```
double ansimp(vector x)
{
setparammom(x);
analysegen( );
return getobjectivemom(0);
}
```

The function first sets the shell vector variable *parammom* (the current parameter values by convention) to its argument *x*, which is passed by the calling algorithm. This is done by the open library function *setparammom*. Then the common analysis function *analysegen* is called, which performs interpretation of the analysis block of the command file, which is the user definition of the direct analysis. The function then returns the value of the scalar variable *objectivemom* (value of the objective function by convention), which is obtained by the library function *getobjectivemom*. This value is set at interpretation of the analysis block.

The interpreter function which will run the algorithm can now be defined. It is assumed that the algorithm will be run by the interpreter command *optsimp* with the following argument list:

```
optsimp(simp val tol maxit)
```

where *simp* is a matrix argument (initial simplex), *val* is a vector argument (function values in the initial simplex), and *tol* (tolerance) and *maxit* (maximum number of iterations) are scalar arguments. It is assumed that the initial simplex and function values in its apices are provided by the user and passed as command arguments. The interpreter function can be defined as follows:

```
double fi_optsimp(ficom fcom)
{
matrix simp=NULL;
vector val=NULL, paropt=NULL;
double valopt, tol, maxit;
int it;
/* Extract interpreter command argumetns: */
readmatarg(fcom,&simp);
readvecarg(fcom,&val);
readscalarg(fcom,&tol);
readscalarg(fcom,&maxit);
/* Run the algorithm: */
```

```
minsimp(simp,val,tol,&it,maxit,&paropt,&valopt,ansimp,fcom-
>out)
/* Set pre-defined shell variables paramopt and objectiveopt,
which represent optimal parameters and optimal value of the
objective function, respectively: */
setparamopt(paropt);
setobjectiveopt(valopt);
/* release local variables: */
dispmatrix(&simp);
dispvector(&val);
dispvector(&paropt);
}
```

Comments in the above code explain its function to a large extent. The function takes a single argument, which is the interpretation object described in section 4.3.2 and through which all information regarding command file interpretation can be accessed. The function first extracts arguments which are passed through the argument block of the corresponding interpreter commands. Open library functions of extraction of different types of command arguments are used. These functions take the interpretation object as the first argument, since the command argument block is accessed through it. Each of these functions sets the field *fcom->argpos* to the position after the last extracted argument, so that the next function can begin argument extraction on the right place. The second argument of these functions is always the address of the variable in which the argument value is stored. All interpreter command arguments are in this case input arguments of the algorithm. In general there could also be other types of arguments, for example specifications of the shell variables where specific algorithm data should be stored.

The simplex algorithm is then called. The last argument of the call but one is the direct analysis function *ansimp*, which was defined above. The last argument is the file interpreter output file, which in the shell coincides with the shell output file.

The algorithm performs minimisation of the objective function and returns optimal parameters in the vector *paropt* and optimal value of the objective function in the number *valopt* (both are local variables). These values are then copied to the appropriate pre-defined shell variables, so that they are accessible for further use by other algorithms and utilities. This is done by the appropriate library functions. Finally, the dynamic storage which was allocated within the function is released.

After the file interpreter function is defined, the appropriate command can be installed in the file interpreter system. This is done by the following line of code:

```
instfintfunc ( "optsimp", fi_ optsimp );
```

*instfintfunc* is another function of the shell open library, which installs a file interpreter command on its function definition stack. Its first argument is the name of the command, which will be used in the shell command file. The second argument is the address of the function which is run by the file interpreter when the corresponding command is encountered. The above code must be added in the specific source file, which is compiled and linked with the optimisation shell. This code is executed at shell initialisation.

### 4.3.4 Parallelisation

Parallel execution of code represents an attempt at exceeding practical limits imposed by the capability of available computers, which are often very restrictive for demanding numerical applications. Parallelisation of the optimisation shell[8],[9] is the final implementation issue discussed in this text. It is instructive because parallelisation usually requires considerable rearrangement in the code structure. In the shell a different philosophy of parallel execution is transferred to the algorithmic level, which enables continuity of the described shell concepts and coexistence of the parallel and sequential schemes. The shell provides support for straightforward incorporation of parallel algorithms in a similar way as sequential algorithms.

The parallel interface has been built using the MPI (Message Passing Interface)[32] library and the LAM (Local Area Multicomputer)[33] environment. This enables a system to be implemented on a wide range of architectures, including shared and distributed memory multiprocessors as well as arbitrarily heterogeneous clusters of computers connected in a LAN (Local Area Network).

Direct analyses are treated as basic operations that can be executed simultaneously. The parallel scheme is implemented as a master-slave architecture (Figure 4.7). The master process controls execution of optimisation algorithms while slave processes perform simulations. Several slave processes run simultaneously on different processing units performing simulations for different sets of optimisation parameters. Both master and slave processes are actually optimisation shells with complete functionality, but their execution differs due to different functions.

The master process is responsible for the process management and keeps track of the slave processes status. All actions of the slave processes are triggered by the master, so every change of the slave status is conditional on the master request. The master process also registers the execution times of direct analyses. This information is used for the so-called load balancing in the case when the master can choose between several slaves that are available for execution of a new task.

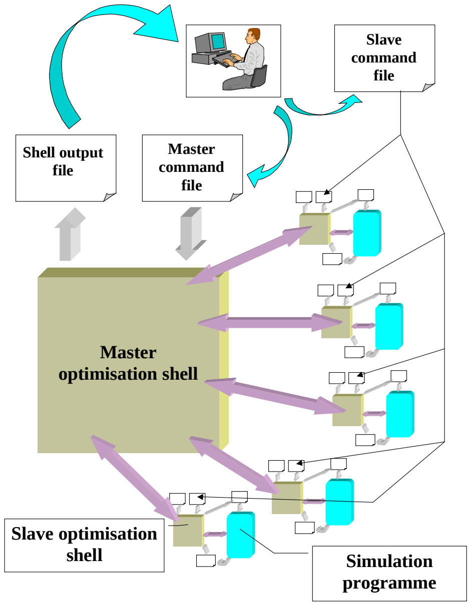



**Figure 4.7:** Parallel optimisation scheme[9]. The main optimisation shell controls the optimisation process while the slave shells interact with simulation programs and execute direct analyses.

The direct analysis is performed in three parts. The first part is executed by the master process, which sends the analysis request and appropriate data (e.g. parameter values) to a slave. The second part is executed by a slave process, which runs the simulation, collects results and evaluates appropriate quantities, which are then sent back to the master. The third part is executed by the master process. The quantities sent by the slave process are received and transferred to the calling optimisation algorithm.

Each part of the function evaluation is performed by its own function, which interprets a specific pre-defined block in the master or slave command file. This enables not only automatic exchange of parameter values and analysis results, but also arbitrary data exchange between the master and slave processes. The parallel interface provides file interpreter commands for this task.

Optimisation algorithms can use the first and third functions, which transfer data between algorithms and function definition in the appropriate command files. These two functions are an equivalent to the common analysis function in the sequential scheme. In order to enable proper task distribution, the first function must not only accept the parameter values, but also return to the algorithm identification of the slave process to which the analysis request has been sent. Similarly, the third function must return the identification of the process that performed the task. The first function must check whether there is any slave process ready to accept the task. The third one must check if slave processes have finished any task and, if necessary, wait for the next available results.

The master process interprets a command file in the same way as a sequential scheme. Every action is a consequence of a function call in the command file. The behaviour of slave processes is different since these processes only respond to master requests. When a slave process is spawned, it interprets its command file without exiting and then waits for the master process requests. The interpretation of the command file is a part of initialisation, while later on every action is triggered by a master process request. The communication between the master and slave processes is synchronised[8],[9]. To make the described functionality more clear, the course of a direct analysis is desctibed below.

Figure 4.8 shows how a direct analysis is executed on the master (the left-hand side of the figure) and a slave (the right-hand side) process. Times of characteristic events are marked by $t_0$ through $t_{26}$ in the order in which these events follow each other. The time scale is not proportional.

When the algorithm requires execution of a direct analysis and a slave process which is ready to accept a task exists, the master process sends a task request to that process ($t_1$ to $t_9$). The slave is in the waiting state at that time, which is known to the master because it keeps track of slave processes status. The slave reestablishes such a state every time after it completes an action requested by the master ($t_0$ and $t_{25}$ in the figure).

The master notifies the slave of the request by sending it the "BEGIN_DATA" message. After the receipt of this message the slave accepts the data sent by the master and stores it to the appropriate location. The master is sending the contents of its variables to the slave. The slave stores these contents in the variables of the same names (note that both master and slave are actually complete optimisation shells). Data packages carry complete information regarding the position of the data in the system of shell variables. Interpretation of the "analysis" block of the master command file ($t_3$ to $t_4$) is a part of sending a task request to the slave. In this block the user can send arbitrary data from the master system of user defined variables by using appropriate file interpreter commands. After that, standard data, i.e. the vector of parameter values *parammom*, is sent automatically ($t_5$ to $t_6$). The "END_DATA" message is then sent to the slave. After its receipt ($t_7$) the slave stops accepting data from the master and expects the "START_AN" message. Its receipt invokes the slave process analysis function ($t_8$ to $t_{14}$), which includes interpretation of the "analysis" block of the slave command file ($t_{10}$ to $t_{12}$). Normally this part corresponds to the actual performance of the direct analysis, while other parts simply take care of the proper data transfer between the master and the slave.

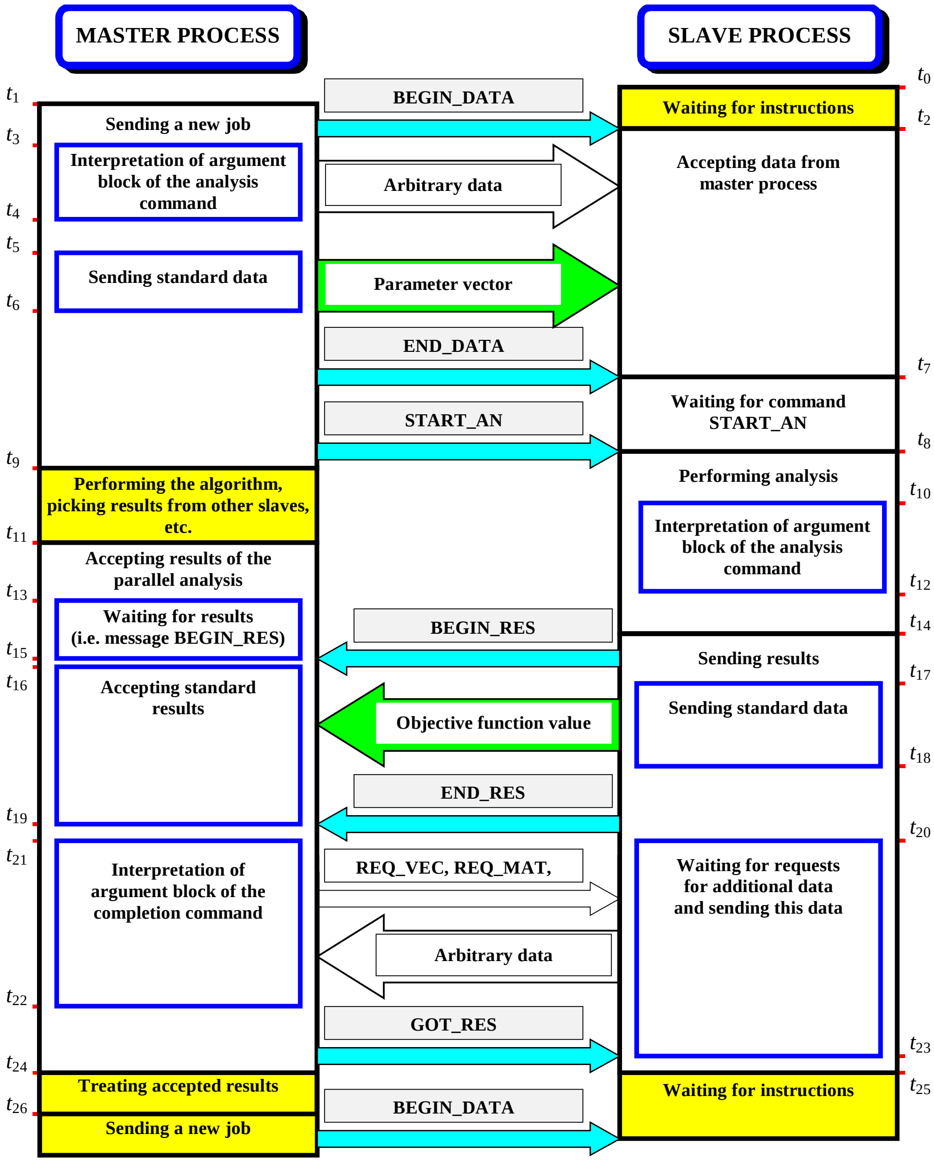



**Figure 4.8:** Course of a direct analysis in the parallel scheme[9].

After sending the task, the master can continue to perform the algorithm with available data, send parallel tasks to other slaves and accept results from them. The slave sends the "BEGIN_RES" message to the master after it finishes the analysis ($t_{14}$). The message is buffered until the master is ready to accept results ($t_{11}$). Depending on the state of the algorithm this can happen in two ways. The master can just check if such message has been sent by any of the slaves. If this is the case, it accepts the results ($t_{16}$ to $t_{24}$); otherwise it continues to do other operations and repeats the check later. The other situation occurs when the master can not do anything until it obtains results of at least one direct analysis. This typically happens when all available slaves are busy and the master has already treated all results which have arrived. In this case the master blocks its execution until the message “BEGIN_RES” is received from any of the slaves.

Receipt of results is somehow a reversed process to that of sending a task. After a receipt of the “BEGIN_RES” message the master accepts data sent by the slave until the receipt of the “END_RES” message. The slave sends this message after sending the standard data ($t_{17}$ to $t_{18}$), e.g. the value of the objective and constraint functions and their gradients. After that, the master interprets the “completion” block of its command file ($t_{21}$ to $t_{22}$) where the user can send requests for additional data to the slave. The slave waits for such requests and sends back the requested data ($t_{20}$ to $t_{23}$) until it receives the “GOT_RES” message. The master sends this message after interpretation of the “completion” block. For the slave its receipt means that it has finishes the task. It reestablishes the waiting state ($t_{25}$) and is able to accept further task requests.

The shell provides various mechanisms for process management and load balancing. Processes can be controlled, enabled and disabled by the user during the runtime[34]. This enables better performance to be achieved for a given kind of algorithm and controlled use of computational resources.

The parallel scheme does not significantly affect the concepts of the shell, neither from the user point of view nor from the view of the implementation interface for incorporation of new tools.

Parallel algorithms are incorporated in the shell in a similar way as sequential algorithms. There are two fundamental differences. The direct analysis now consists of two parts, executed at different times while other analyses can be run in between. A special function executes each part. These functions are called by the algorithm and must be provided as arguments at the call to the algorithm in the appropriate file interpreter function. They are referred to as the calling analysis function and the returning analysis function. They are specific for each algorithm, while each of them calls the appropriate common analysis function (calling or returning, respectively). The second difference is that calling and returning analysis functions return the identification number of the process which performs the appropriate analysis. This is necessary for the algorithm in order to connect specific analysis results (which are received in unpredictable order) with the corresponding optimisation parameters. Both differences are fundamentally conditioned by the nature of parallel execution and therefore do not represent an unnecessary excess in complexity.

## *References:*


[1] I. Grešovnik, J. Korelc, T. Rodič, T. Sustar, *An approach to more efficient development of computational systems for solution of inverse and optimisation problems in material forming*, In: SciTools'98 : book of abstracts, International Workshop on Modern Software Tools for Scientific Computing, Oslo, Norway, September 14-16, 1998, Oslo, SINTEF Applied Mathematics, 1998, pp. 31, Accessible on the Internet (URL): http://www.oslo.sintef.no/SciTool s98.

[2] T. Rodič, J. Korelc, M. Dutko, D.R.J. Owen, *Automatic optimisation of perform and tool design in forging*, ESAFORM bulletin, vol. 1, 1999.

[3] T. Rodič, I. Grešovnik, D.R.J. Owen, *Symbolic computations in inverse identification of constitutive constants*, In: WCCM III, Extended abstracts, Third World Congress on Computational Mechanics, Chiba, Japan, August 1-5, 1994, [S. l.], International Association for Computational Mechanics (IACM), 1994, vol. II, pp. 978-979.

[4] I. Grešovnik, T. Rodič, *Optimization system utilizing a general purpose finite element system*, In: Second World Congress of Structural and Multidisciplinary Optimization, Zakopane, Poland, May 26-30, 1997, Extended abstracts, [S. l.], ISSMO, 1997, pp. 368-369.

[5] I. Grešovnik, T. Rodič, *Optimization system utilizing a general purpose finite element system*, In: WCSMO-2 : proceedings of the Second World Congress of Structural and Multidisciplinary Optimization, Zakopane, Poland, May 26-30, 1997. Vol. 1, Witold Gutkowski, Zenon Mroz (editors), 1st ed., Lublin, Poland, Wydawnictwo ekoincynieria (WE), 1997, pp. 61-66.

[6] T. Rodič, I. Grešovnik, *A computer system for solving inverse and optimization problems*, Eng. comput., vol. 15, no. 7, pp. 893-907, 1998.

[7] I. Grešovnik, T. Rodič: *A general-purpose shell for solving inverse and optimisation problems in material forming*. In: COVAS, José Antonio (editor). Proceedings of the 2nd ESAFORM Conference on Material Forming, Guimaraes, Portugal, [13-17 April 1999]. Guimaraes, Portugal: Universidade do Minho, Departamento de Engenharia de Polímeros, 1999, pp. 497-500.

[8] I. Grešovnik, T. Šuštar, T. Rodič, *Paralelizacija v lupini za resevanje optimizacijskih in inverznih problemov* (in Slovene), In: Zbornik del, Kuhljevi dnevi '98, Logarska dolina, Slovenija, 1.-2. oktober 1998, Boris Štok (editor), Ljubljana, Slovensko društvo za mehaniko, 1998, pp. 169-176.

[9] I. Grešovnik., T. Šuštar, T. Rodič: *Parallelization of an optimization shell*. In: 3rd World congress of structural and multidisciplinary optimization : Buffalo, NY, May 17-21, 1999. Buffalo: WCSMO, 1999, pp. 221-223.

[10] *Optimization Shell Inverse*, electronic document at http://www.c3m.si/inverse/ , maintained by the Centre for Computational Continuum Mechanics, Ljubljana.

[11] I. Grešovnik et al., *What Inverse Is*, electronic document at http://www.c3m.si/inverse/basic/what/ , maintained by the Centre for Computational Continuum Mechanics, Ljubljana.

[12] *Inverse Manuals*, electronic document at http://www.c3m.si/inverse/man/ , maintained by the Centre for Computational Continuum Mechanics, Ljubljana.

[13] I. Grešovnik, *Quick Introduction to Optimization Shell "Inverse"*, electronic document at http://www.c3m.si/inverse/doc/other/quick/ , maintained by the Centre for Computational Continuum Mechanics, Ljubljana, 1999.

[14] I. Grešovnik, *A Short Guide to the Optimisation Shell Inverse – for version 3.5*, electronic document at http://www.c3m.si/inverse/doc/man/3.6/short/index.html , Ljubljana, 1999.

[15] I. Grešovnik, *Programme Flow Control in the Optimisation Shell Inverse*, electronic document at http://www.c3m.si/inverse/doc/man/3.6/flow/index.html , Ljubljana, 2000.

[16] I. Grešovnik, *The Expression Evaluator of the Optimisation Shell Inverse*, electronic document at http://www.c3m.si/inverse/doc/man/3.6/calc/index.html , Ljubljana, 2000.

[17] I. Grešovnik, *User-Defined Variables in the Optimisation Shell INVERSE*, electronic document at http://www.c3m.si/inverse/doc/man/3.6/var/index.html , Ljubljana, 2000.

[18] I. Grešovnik, *Solving Optimization Problems by the Optimization Shell Inverse*, electronic document at

http://www.c3m.si/inverse/doc/man/3.6/opt/index.html , Ljubljana, 2000.

[19] I. Grešovnik, *A General File Interface for the Optimisation Shell Inverse*, electronic document at http://www.c3m.si/inverse/doc/man/3.6/int/index.html , Ljubljana, 2000.

[20] I. Grešovnik, *Syntax Checker and Debugger for Programme INVERSE,* electronic document at http://www.c3m.si/inverse/doc/man/3.6/debug/index.html , Ljubljana, 2000.

[21] I. Grešovnik, *Miscellaneous Utilities of the Optimisation Shell INVERSE*, electronic document at http://www.c3m.si/inverse/doc/man/3.6/misc/index.html , Ljubljana, 2000.

[22] I. Grešovnik, D. Jelovšek, *Interfaces Between the Optimisation Shell INVERSE and Simulation Programmes*, electronic document at http://www.c3m.si/inverse/doc/man/3.6/sim/index.html , Ljubljana, 2000.

[23] B. W. Kernigham, D. M. Ritchie, *The C Programming Language (second edition)*, Prentice – Hall, 1988.

[24] A. Kelley, I. Pohl, *A Book on C - an Introduction to Programming in C,* Menlo Park, 1984.

[25] A. Kelley, I. Pohl, *A Book on C: Programming in C (fourth edition)*, Addison – Wessley Publishing, New York, 1998.

[26] M. Banahan, D. Brady, M. Doran, *The C Book – Featuring the ANSI C Standard (second edition),* Addison – Wesley Publications, Wokingham, 1991.

[27] J. Valley, *C Programming for UNIX*, SAMS Publishing, Carmel, 1992.

[28] B. Stroustrup, *The C++ Language (second edition),* Addison – Wesley Publishing, New York, 1992.

[29] I. Grešovnik, *Library of Matrix and Vector Operations*, C3M internal report, Ljubljana, 1999.

[30] *Elfen – Product Information,* electronic document at http://rsazure.swan.ac.uk/products/elfen/elfen.html , maintained by Rockfield Software Ltd., Swansea.

[31] *Elfen Implicit User Manual – version 2.7,* Rockfield Software Limited, Swansea, 1997.

[32] *MPI, A Message-Passing Interface Standard*, electronic document at http://www.mcs.anl.gov/mpi/mpi-report-1.1/mpi-report.html, University of Tennessee, Knoxville, Tennessee, 1995.

[33] *MPI Primer / Developing With LAM*, electronic document in PostScript at ftp://ftp.osc.edu/pub/lam/lam61.doc.ps.Z , Ohio Supercomputing Center, the Ohio State University, 1996

[34] Igor Grešovnik, *Paralelizacija v lupini Inverse* (in Slovene), C3M internal report, Ljubljana, 1997.

# 5 ILLUSTRATIVE EXAMPLES

## *5.1 Inverse Estimation of the Hardening Curve from the Tension Test*

### 5.1.1 The Tension Test

The tension test (Figure 5.1) is widely used for the mechanical testing of materials. However, accurate estimation of plastic material properties is difficult due to the non-uniform stress and strain distribution in the necking zone (Figure 5.2). Because of this phenomenon, it is not possible to determine the hardening parameters directly by measuring elongations at different loads. In order to determine true stress the Bridgeman correction is often applied which requires additional measurements of contractions at the narrowest part of the deformed sample and curvature of the neck[1]. The approach is based on the assumptions that the contour of the neck is the arc of a circle and that strains are constant over the cross section of the neck.

In the present section an inverse approach to estimation of hardening parameters is considered[2],[3]. This approach does not incorporate idealisations in the form of a priori assumptions on the stress or strain field. The material behaviour is modeled by a von Mises elasto-plastic material model[1].

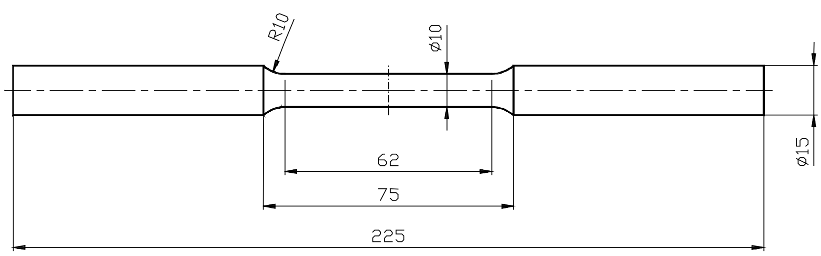



**Figure 5.1:** Sample geometry.

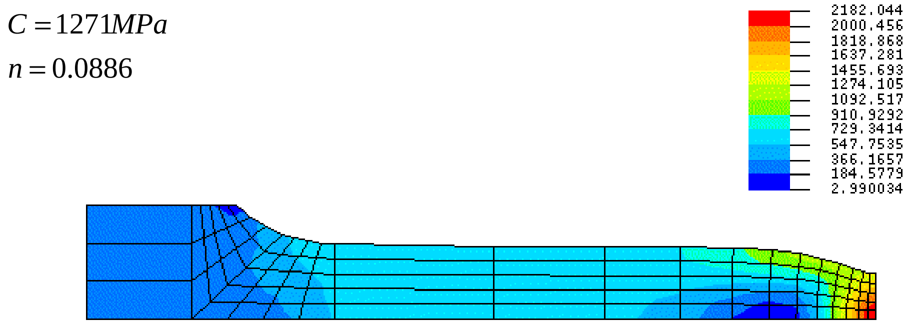



**Figure 5.2:** Longitudinal stress distribution obtained by numerical simulation. Exponential hardening law with $C = 1271MPa$ and $n = 0.0886$ was assumed. The elongation is $8\,mm$.

### 5.1.2 Estimation of an Exponential Approximation

An exponential hardening law is assumed to approximate the relationship between the effective stress and effective strain:

$$\bar{\sigma} = C\bar{\varepsilon}^n \; . \tag{5.1}$$

The unknown parameters $C$ and $n$ need to be derived from measured forces at certain elongations of the samples. Two series of measurements were performed for two different steel grades. The geometry of the samples is shown in Figure 5.1, while the experimental data are given in Table 5.1 and Table 5.2 for each series. Graphic presentation of the same data for the first sample of each series is given in Figure 5.3.

**Table 5.1:** Experimental data for the first series.

| Elongation [mm] | Force [N], sample 1 | Force [N], sample 2 | Force [N], sample 3 |
|---|---|---|---|
| 3 | 65900 | 68800 | 66800 |
| 4 | 67800 | 69900 | 67800 |
| 5 | 68650 | 70600 | 68700 |
| 6 | 68900 | 70600 | 68700 |
| 7 | 68850 | 69200 | 68400 |
| 8 | 68000 | 66600 | 68200 |

| | | | |
|---|---|---|---|
| 9 | 65800 | 61300 | 65100 |
| 10 | 61800 | 54100 | 59300 |

**Table 5.2:** Experimental data for the second series.

| Elongation [mm] | Force [N], sample 1 | Force [N], sample 2 | Force [N], sample 3 |
|---|---|---|---|
| 3 | 86000 | 85800 | 84700 |
| 4 | 87500 | 86300 | 85600 |
| 5 | 87800 | 86500 | 86400 |
| 6 | 86500 | 85900 | 84500 |
| 7 | 81700 | 84600 | 80900 |
| 8 | 74800 | 78200 | 72600 |

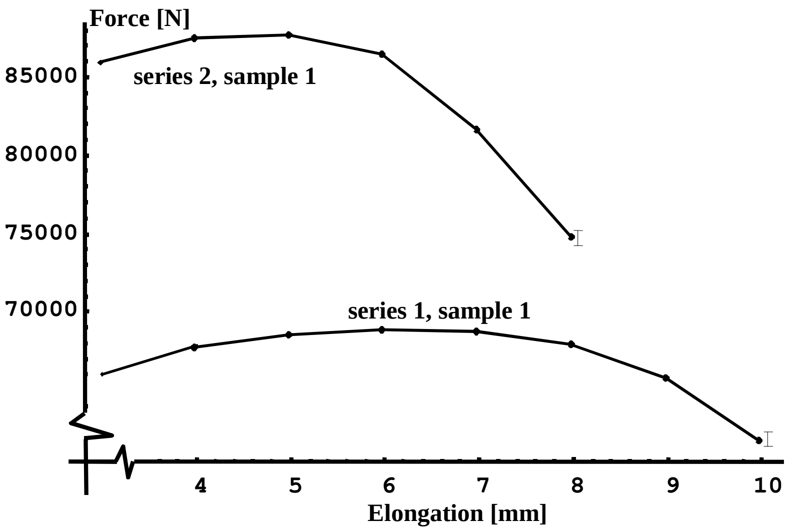



**Figure 5.3:** Measured data for the first sample of each series.

Solution of the problem was found by searching for the parameters which give the best agreement between measured and respective numerically calculated quantities. The agreement can be defined in different ways, but most commonly used is the least-square concept, mostly because of its statistical background[21]-[23]. The problem is solved by minimising the function

$$\chi^2(C,n)=\sum_{i=1}^{N}\frac{\left(F_i^{(m)}-F_i(C,n)\right)^2}{\sigma_i^2}\ , \tag{5.2}$$

where $F_i^{(m)}$ are measured forces at different elongations, $F_i(C,n)$ are the respective quantities calculated with the finite element model by assuming trial values of parameters *C* and *n*, $\sigma_i$ are the expected errors of appropriate measurements and $N$ is the number of measurements.

The scatter of experimental data for the same series which is evident from Table 5.1 and Table 5.2 is mainly due to differences in samples rather than experimental errors. This has an effect on the estimated parameters *C* and *n.* The results are summarized in Table 5.3 and Table 5.4.

**Table 5.3:** Calculated parameters C and n for the first series.

| | sample 1 | sample 2 | sample 3 |
|---|---|---|---|
| $C[MPa]$ | 1271 | 1250 | 1258 |
| $n$ | 0.1186 | 0.1010 | 0.1132 |

**Table 5.4:** Calculated parameters C and n for the second series.

| | sample 1 | sample 2 | sample 3 |
|---|---|---|---|
| $C[MPa]$ | 1492 | 1511 | 1462 |
| $n$ | 0.08422 | 0.09269 | 0.08318 |

It seems that the applied numerical model simulates the bahaviour of the investigated material adequately. This is indicated[22],[24] by the fact that the obtained minimal values of the function $\chi^2(C,n)$ were never much greater than one, assuming that the measurement errors ($\sigma_i$ in (5.2)) are one percent of the related measured values.

### 5.1.3 Estimation of a Piece-wise Linear Approximation

The flow stress of the material is a result of different hardening and softening phenomena which interact during plastic deformation. This interaction is often so complex that it is difficult to predict the form of the hardening curve $\overline{\sigma}(\overline{\varepsilon})$. In such cases it would be desirable to find an approximation of the hardeining curve without making any preassumptions regarding its form. This can be done in several ways. In this work, an approach where points of the hardening curve defining a piece-wise linear approximation are sought is considered.

The experimental measurements used for estimation of the piece-wise linear approximations are summarized in Table 5.5 and Figure 5.4. The data are for the first sample of the first series, but with 16 measurements instead of eight used for evaluation of exponential approximation.

**Table 5.5:** Experimental data used to obtain a piece-wise linear approximation of the hardening curve.

| Elongation [mm] | Force [N] |
|---|---|
| 2 | 62200 |
| 2.5 | 64400 |
| 3 | 65900 |
| 3.5 | 67000 |
| 4 | 67800 |
| 4.5 | 68200 |
| 5 | 68650 |
| 5.5 | 68800 |
| 6 | 68900 |
| 6.5 | 69000 |
| 7 | 68850 |
| 7.5 | 68600 |
| 8 | 68000 |
| 8.5 | 67100 |
| 9 | 65800 |
| 9.5 | 64000 |

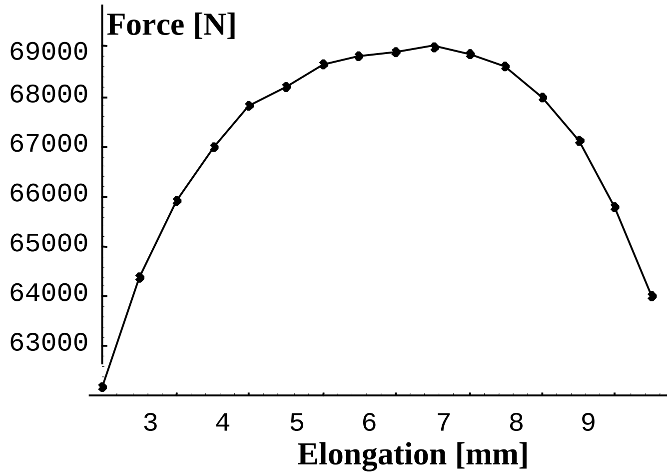



**Figure 5.4:** Measurements used for calculating a piecewise approximation of the hardening curve (measurements are for the first sample of the first series).

The points on the hardening curve were obtained by minimising the function

$$\chi^2(\bar{\sigma}_1,\bar{\sigma}_2,...,\bar{\sigma}_M) = \sum_{i=1}^{N} \frac{\left(F_i^{(m)} - F_i(\bar{\sigma}_1,\bar{\sigma}_2,...,\bar{\sigma}_M)\right)^2}{\sigma_i^2}, \tag{5.3}$$

where parameters $\bar{\sigma}_i$ are values of the curve $\bar{\sigma}(\bar{\varepsilon})$ at arbitrary equivalent strains $\bar{\varepsilon}_i$. The yield stress was known from experiments.

Approximations of the hardening curve with 4, 6, 8 and 10 points were calculated. The results are shown in Figure 5.5 to Figure 5.8. The exponential hardening curve with parameters $C = 1271 MPa$ and $n = 0.1186$ (as obtained by the inverse analysis assuming the exponential hardening law) is drawn in each figure for comparison. It is evident from these graphs that calculated piecewise linear approximations are in relatively good agreement with the calculated exponential approximation.

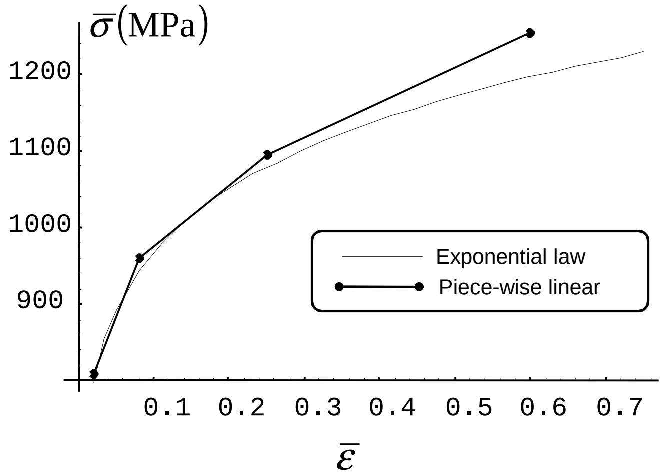



**Figure 5.5:** Comparison between exponential and piece-wise linear (4 points) approximations of the hardening curve.

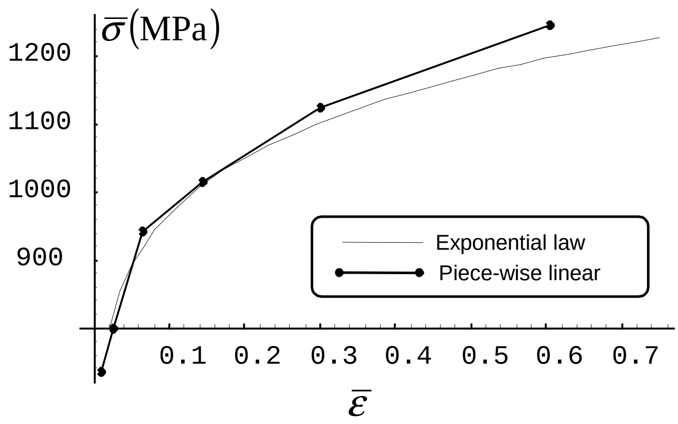



**Figure 5.6:** Comparison between exponential and piece-wise linear (6 points) approximations of the hardening curve.

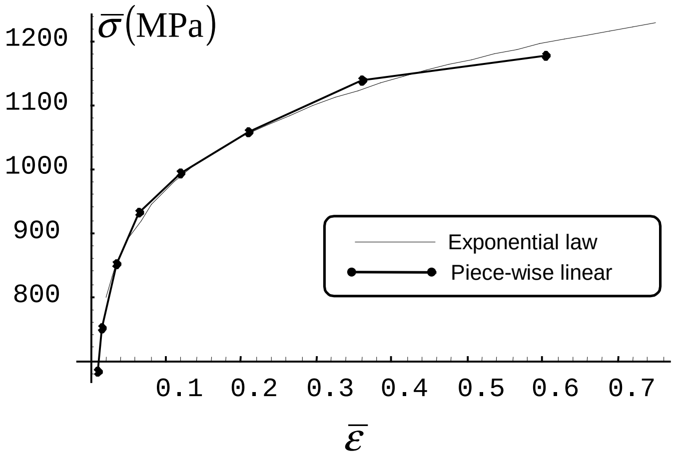



**Figure 5.7:** Comparison between exponential and piece-wise linear (8 points) approximations of the hardening curve.

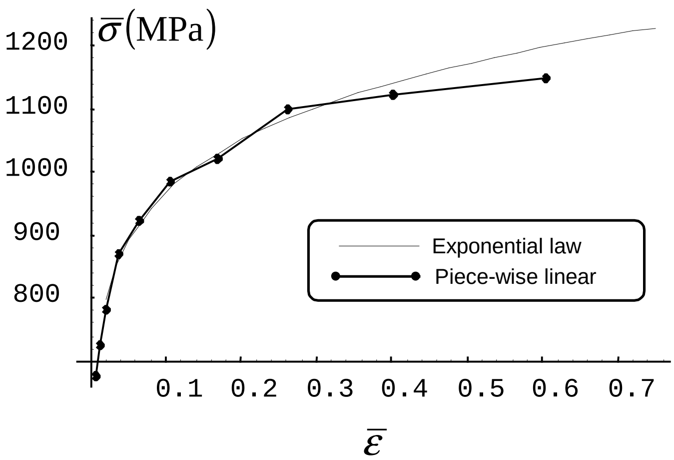



**Figure 5.8:** Comparison between exponential and piece-wise linear (10 points) approximations of the hardening curve.

### 5.1.4 Numerical Tests

A number of numerical tests were performed to investigate the stability and uniqueness of the inverse solutions for the exponential approximation of the hardening curve.

Several inverse analyses were performed with very different initial guesses and they always converged to the same results. This is the first indication that the problem is not ill-posed. Further examination was made by plotting the $\chi^2$ function (Figure 5.9 and Figure 5.10). A distinctive minimum can be recognised in these figures without indication of possible existence of several local minima.

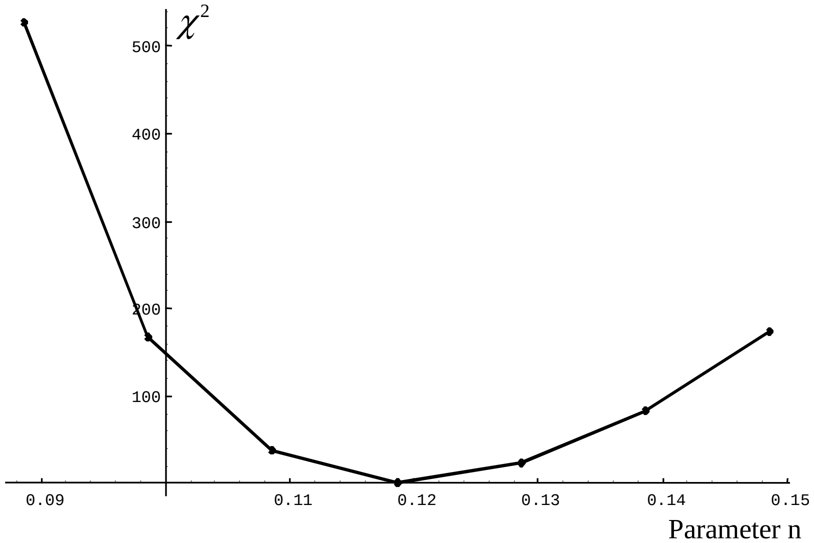



**Figure 5.9:** Dependence of function $\chi^2$ on parameter n at measured data for sample 1 of series 1. Parameter C is set to 1271 MPa.

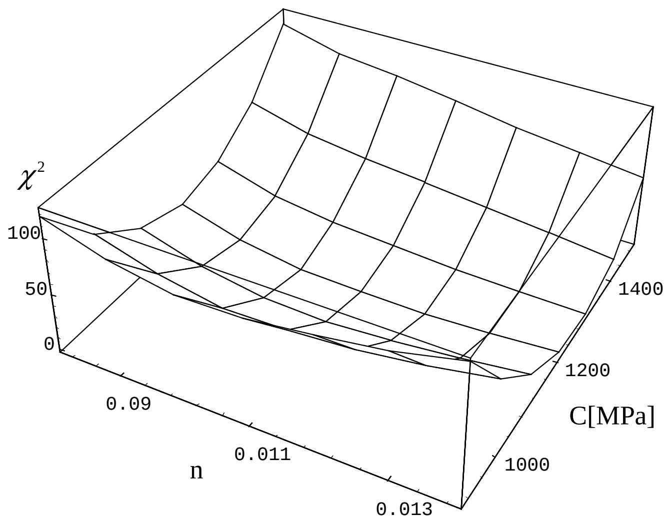



**Figure 5.10:** Dependence of function $\chi^2$ on both parameters for the same measured data as in Figure 5.3.

To test the stability of the solutions, a Monte Carlo simulation[11] was performed. It was assumed that the correct values of both parameters were known. For this purpose, the previously calculated values for the first sample of the first series were taken, namely $C = 1271\ MPa$ and $n = 0{,}1186$ (see Table 5.1). With these values the so called "exact measurements" $F_i^{(0)}$ were obtained with the same finite element model used for the inverse analysis of the real measurements. The "simulated measurements" $F_i^{(m)}$ were successively obtained by adding random errors $r_i$ to the "exact measurements". Errors were distributed normally as

$$\frac{d\,P}{d\,r_i} = \frac{1}{\sqrt{2\,\pi}} exp\left( -\frac{r_i^2}{2\,s_i^2} \right) , \tag{5.4}$$

where $s_i$ is the standard deviation of distribution. This distribution is often used to simulate measurement errors which do not have a clearly defined origin[22].

For each set of "simulated measurements" parameters *C* and *n* were calculated. Three different sets of $s_i$ were chosen so that ratios

$$R_i = \frac{s_i}{\left| F_i^0 \right|} \tag{5.5}$$

were uniform within each set. Fifty numerical experiments were performed for $R = 0.01$, twenty for $R = 0.1$ and twenty for $R = 0.001$. Then average values $\bar{z}$ and dispersions $S_z$ of the searched parameters were calculated for each set, according to

$$\bar{z} = \frac{1}{k} \sum_{i=1}^{k} z_i \tag{5.6}$$

and

$$S_z^2 = \frac{1}{k-1} \sum_{i=1}^{k} \left( z_i - \bar{z} \right)^2 \ . \tag{5.7}$$

The results are summarized in Table 5.6. Figure 5.11 shows the distribution of calculated parameters at $R = 0.01$.

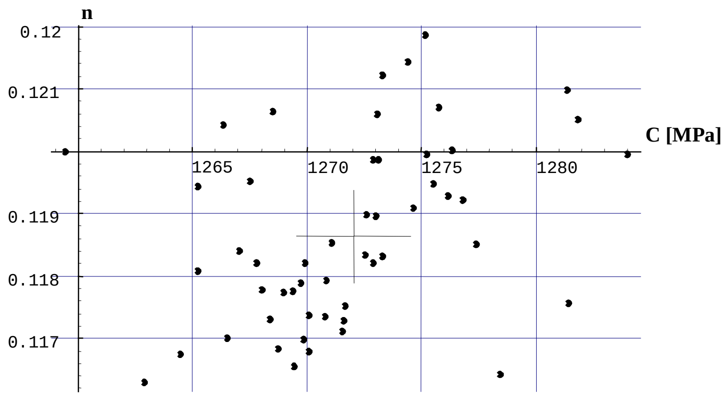



**Figure 5.11:** Results of the Monte carlo simulation for $R = 0.01$.

**Table 5.6:** Results of Monte Carlo simulations: Average values and dispersions of calculated parameters at different $R_i$

| | $R = 0{,}001$ | $R = 0{,}01$ | $R = 0{,}1$ |
|---|---|---|---|
| $\overline{C}$ | 1271.4 | 1271.8 | 1287 |
| $S_C$ | 0.58 | 4.9 | 69 |
| $\overline{n}$ | 0.118628 | 0.11867 | 0.1163 |
| $S_n$ | 0.00016 | 0.0015 | 0.014 |

### 5.1.5 Concluding Remarks

The above example illustrates the applicability of the inverse approach in parameter identification. Inverse identification can become a useful tool for estimation of those parameters which are difficult to obtain with analytical treatment of experimental results due to the complexity of the phenomena involved. An important advantage of the approach is that parameters are derived by using the same numerical model which is then applied in direct simulations.

It is necessary to take the appropriate precautions when the approach is used. It is especially necessary to make sure that the inverse problems is well conditioned and has a unique solution. If measurement errors can be estimated, then statistical tests can be used to verify the adequacy of the applied model and estimated parameters. It is wise to use the estimated parameters in additional tests where simulation results are compared with measurements.

## *5.2 Shape Optimisation of Cold Forging Processes*

Two simple examples are included to illustrate the applicability of the optimisation techniques in the design of forming processes. In the first example a pre-form of a cold forged workpiece is optimised in order to obtain the desired final form. In the second example the shape of the tool in the first stage of a two stage cold forging process is optimised with the same objective. These examples explain the methodology which could be applied in the optimisation of real processes.

### 5.2.1 Optimisation of Preform Shape

In the present example an axisymmetric workpiece is upset by flat tools (Figure 5.1) with a constant stroke. The optimisation objective is to achieve a prescribed shape of the free boundary of the workpiece by changing the initial shape of this boundary. The Von Mises elasto-plastic material model was used for the workpiece while the tool was modelled as a rigid body. The coulomb friction law was used to model the contact condition between the die and the tool. Because of symmetry the process was modelled in two dimensions.

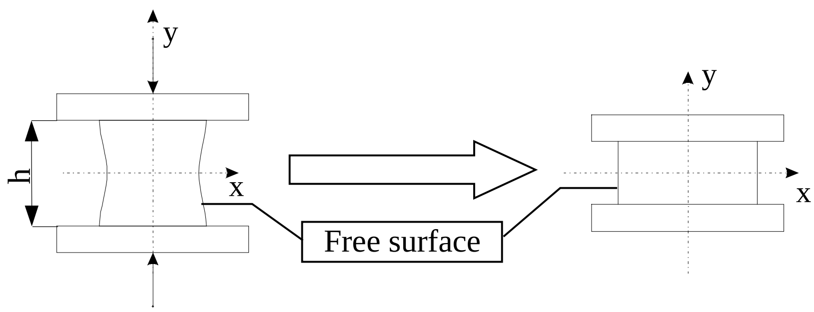



**Figure 5.12:** The forming Process.

The initial height ($h$=100 mm) of the workpiece was reduced by 40 per cent. The desired shape of the free edge was specified by a function prescribing the dependency of x position on y position of the nodes on the free boundary, i.e.

$$x_i = f(y_i) \quad i = 1,.., N, \tag{5.8}$$

where $N$ is the number of nodes on the free boundary. Two different final shapes were prescribed:

$$f(y_i) = R = 1.3\ h \tag{5.9}$$

and

$$f(y_i) = R + \cos\left(\frac{\pi y_i}{40\,mm}\right)\cos\left(\frac{\pi y_i}{150\,mm}\right) \tag{5.10}$$

The problem was solved by minimisation of the following objective function:

$$\chi^2(\mathbf{p}) = \sum_{i=1}^{N}\left((x_i(\mathbf{p}) - f(y_i(\mathbf{p})))\right)^2. \tag{5.11}$$

**p** is the vector of optimisation parameters which describe the initial free boundary shape and $x_i$ and $y_i$ are the final positions of these nodes.

The initial free boundary shape was parametrised by polynomial Lagrange interpolation[23]

$$x = P(\mathbf{p}, y) \tag{5.12}$$

on a given number of control points equidistantly distributed in the y direction between the lowest and the highest point of the workpiece (including the extreme points). The x coordinates of the control points represented optimisation parameters $p_i$. Optimisation parameters (i.e. coefficients of the interpolation polynomial) determined the initial positions of nodes on the free boundary:

$$x_i^0 = P(\mathbf{p}, y_i^0),\ i = 1,..., N, \tag{5.13}$$

The Lagrangian interpolation is defined with[23]

$$L(p_1, p_2, ..., p_M, y) = \sum_{k=1}^{M} p_k L_k(y), \tag{5.14}$$

where

$$L_k(p_1, ..., p_M, y) = \frac{(y - p_1)...(y - p_{k-1})(y - p_{k+1})...(y - p_M)}{(p_k - p_1)...(p_k - p_{k-1})(p_k - p_{k+1})...(p_k - p_M)} \tag{5.15}$$

and $M$ is the number of parameters. The control points lie on the interpolation polynomal, therefore parameters which are the x coordinates of these points have a geometrical interpretation.

Table 5.1 summarises the optimisation results. The results are shown graphically in Figure 5.13 and Figure 5.14. The quantity

$$< \Delta x >= \frac{\sqrt{\chi^2}}{N} \tag{5.16}$$

was introduced in order to make comparison of the results more evident.

**Table 5.7:** Results of optimisation for required flat boundary defined by (5.9) and for the required curved boundary defined by (5.10) for different numbers of parameters.

| | Flat boundary required | | Curved boundary required | |
|---|---|---|---|---|
| $M$ | $\chi^2$[mm$^2$] | $<\Delta x>$[mm] | $\chi^2$[mm$^2$] | $<\Delta x>$[mm] |
| 2 | 0,1072244 | 0,0125943 | 24,6340628 | 0,1908951 |
| 3 | 0,0114715 | 0,0041194 | 0,4628830 | 0,0261675 |
| 4 | 0,0000310 | 0,0002143 | 0,4060472 | 0,0245084 |
| 5 | 0,0000033 | 0,0000700 | 0,2874089 | 0,0206194 |
| 6 | | | 0,1152899 | 0,0130594 |
| 7 | | | 0,0209804 | 0,0055710 |

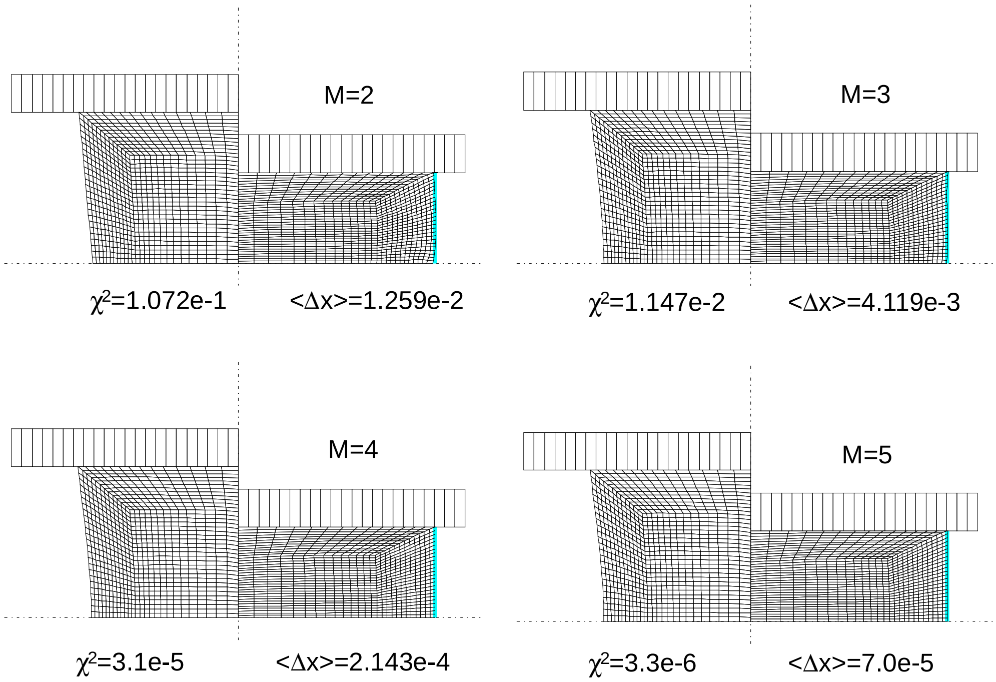
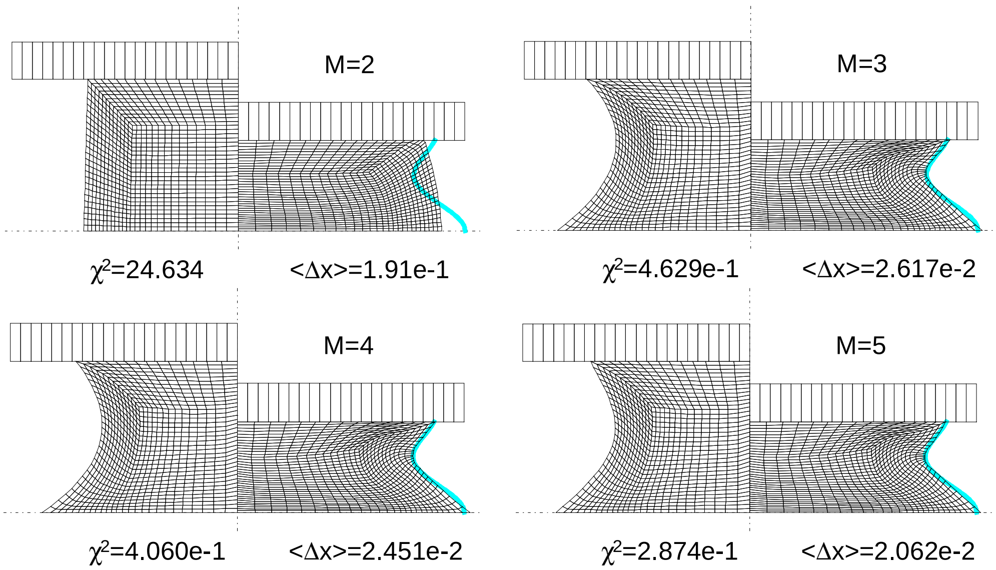



**Figure 5.13:** Comparison of solutions for the required flat free boundary defined by (5.9). The initial finite element mesh is depicted on the left hand side and the mesh after the forming process on the right hand side of each graph.



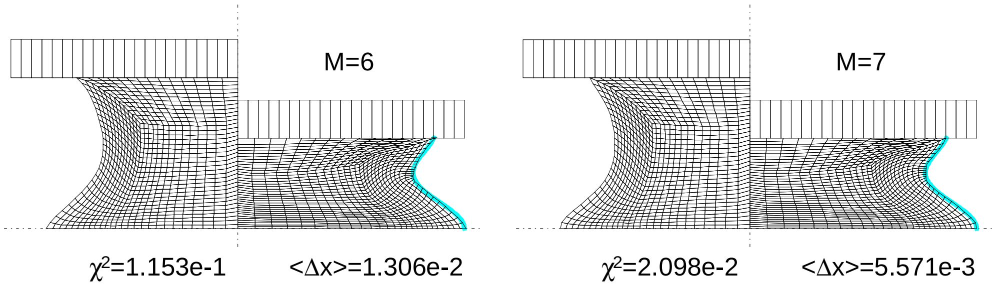



**Figure 5.14:** Comparison of solutions for the required curved free boundary defined by (5.10).

The above results were obtained with a friction coefficient $\mu = 0.1$ between the tool and the workpiece. The friction coefficient plays an important role in the process, therefore its influence was analysed. Figure 5.15 shows the simulation of the process with different friction coefficients, where the optimal initial shape of the optimisation problem with a required flat boundary was adopted as the initial geometry of the workpiece. Figure 5.16 shows the optimal initial shapes for this problem with different values of the friction coefficient.

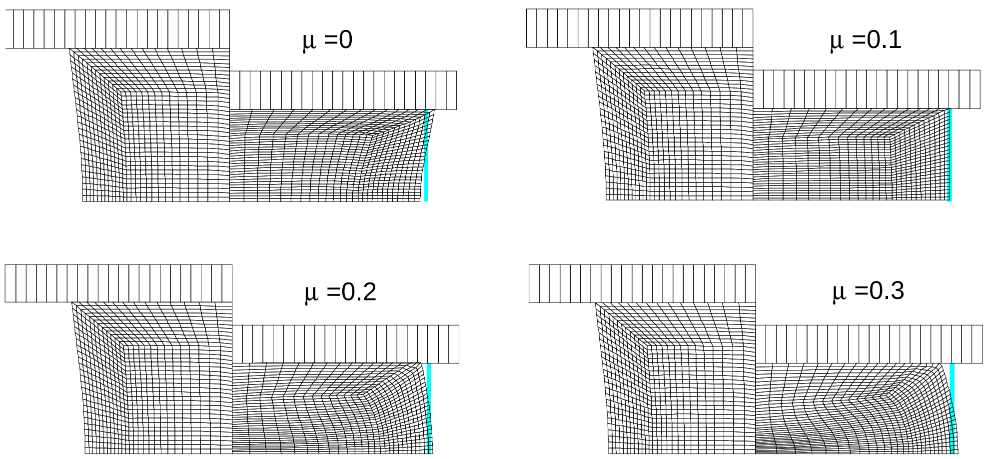



**Figure 5.15:** Simulation of the process with different values of friction coefficient $\mu$. The process starts with the optimal initial shape for $\mu = 0.1$ and required flat free boundary.

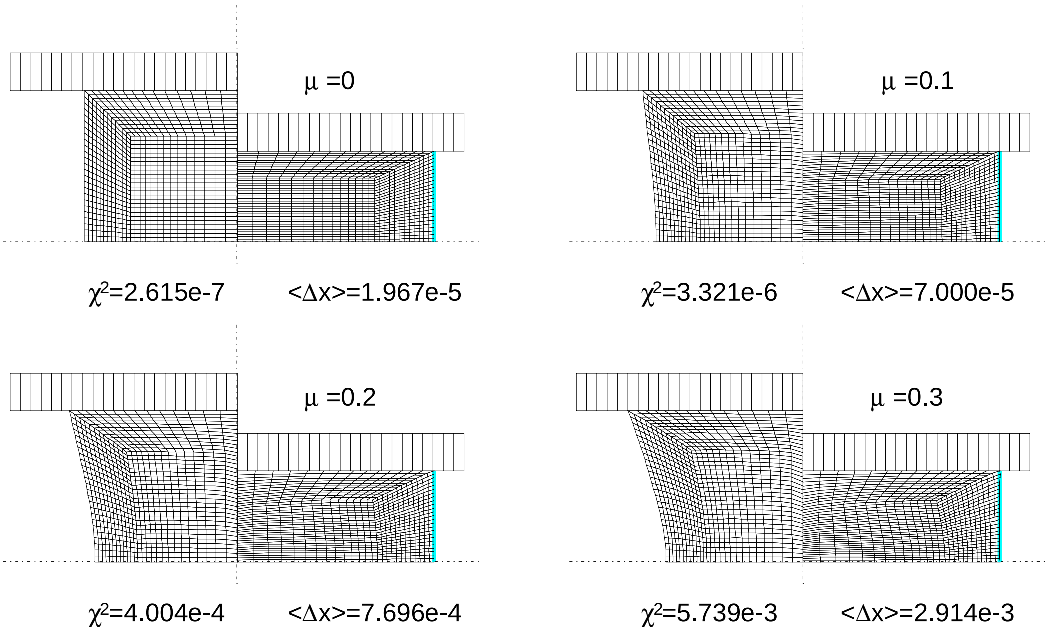
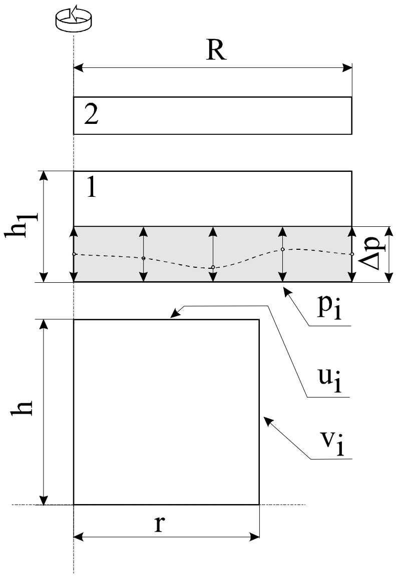



**Figure 5.16:** Solution of the optimisation problem with different friction coefficients $\mu$.

### 5.2.2 Shape Optimisation of a Tool in a Two Stage Forging

In the previous example the shape of the forged workpiece was optimised with the aim of achieving the desired final shape. In multi stage forging processes the shape of the workpiece is obtained by the preceeding operation in the forging sequence. This situation is illustrated by the following example where the tool shape for the first of the two forging operations is optimised in order to achieve the desired final shape of the workpiece.

The optimised process is outlined in Figure 5.17. In the first stage the axisymmetric workpiece is forged by a tool with curved boundary. In the second stage the workpiece is forged by a flat tool with the stroke corresponding to the final height reduction of 40 per cent. The same material model as in the previous example was assumed.




| | |
|---|---|
| cylinder radius | r=50 mm |
| cylinder height: | h=50 mm |
| tools radius | R=75 mm |
| tool 1 height: | $h_1$=30 mm |
| allowed parameter range: | Δp=15 mm |
| | |
| $u_i$: contact surface control point y coordinates | |
| $v_i$: free surface control point x coordinates | |
| $p_i$: tool 1 control point x coordinates (correspond to optimisation parameters) | |

**Figure 5.17:** Outline of the optimised process.

The objective of optimisation was to find such a shape of the first tool that both the contact surface and free boundary of the workpiece are flat after the second stage. This was achieved by minimisation of the following objective function:

$$D(\mathbf{p}) = D(\mathbf{u}(\mathbf{p}), \mathbf{v}(\mathbf{p})) = D_u + D_v = \sum_{i=1}^{M} \frac{(u_i - \overline{u})^2}{\overline{u}^2} + \sum_{i=1}^{N} \frac{(v_i - \overline{v})^2}{\overline{v}^2}, \qquad (5.17)$$

where

$$\overline{u} = \sum_{j=1}^{M} \frac{u_j}{M}, \qquad (5.18)$$

$$\overline{\mathrm{v}} = \sum_{j=1}^{N} \frac{v_j}{N}, \qquad (5.19)$$

*M* is the number of control points (nodes) on the contact surface of the workpiece, *N* is the number of control points on the free surface of the workpiece, and the meaning of other quantities is evident from Figure 5.17. The dependence of $u_i$ and $v_i$ on optimisation parameters **p** was suppressed for clarity.

The shape of the contact surface of the first tool was parametrised with 3rd order splines in such a way that optimisation parameters correspond to y coordinates of the equidistant control points defining the splines.

In order to avoid physically infeasible situations, the range of parameters $p_i$ was limited to an interval of the height $\Delta p$ (Figure 5.17). This was achieved by introducing new variables[4] $t_i$ and defining the transformation $F$ that maps $t_i$ to $p_i$, in the following way:

$$p_i = F(t_i) = \tfrac{1}{2}(p_{\min} + p_{\max}) + \tfrac{1}{\pi}(p_{\max} - p_{\min})\, arctg(t_i), \qquad (5.20)$$

where $p_{\min}$ and $p_{\max}$ are lower and upper bounds for parameters $p_i$, respectively. The transformation $F$ maps the interval $(-\infty, \infty)$ to $(p_{\min}, p_{\max})$. The objective function defined in the space of parameters $t_i$, i.e.

$$\tilde{D}(\mathbf{t}) = D(\mathbf{p}(\mathbf{t})), \qquad (5.21)$$

was minimised with respect to these parameters. Results are summarised in Table 5.8. Parameters $p_i$ in the table are scaled with respect to $h_1 = 30\,mm$. Five shape parameters were used, while $p_6$ denotes the tool stroke. The first row of the table contains results for the process performed with the flat tool in the first stage. The second row contains results for the optimised shape with a constant stroke in the first stage (20 mm). The third row contains results for the case where the stroke of the tool in the first stage is taken as the sixth parameter. The corresponding forging sequences are shown in Figure 5.18 to Figure 5.20

**Table 5.8:** Optimal parameters and values of the objective function.

| | $p_1$ | $p_2$ | $p_3$ | $p_4$ | $p_5$ | $p_6$ | $D_u$ | $D_v$ | $D$ |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 1 | 2/3 | 1.063E-4 | 7.927E-3 | 8.033E-3 |
| 2 | 0.506 | 0.992 | 0.991 | 0.539 | 0.509 | 2/3 | 1.67E-04 | 2.61E-04 | 4.28E-04 |
| 3 | 0.506 | 0.991 | 0.992 | 0.832 | 0.51 | 0.832 | 1.30E-04 | 1.49E-05 | 1.44E-04 |

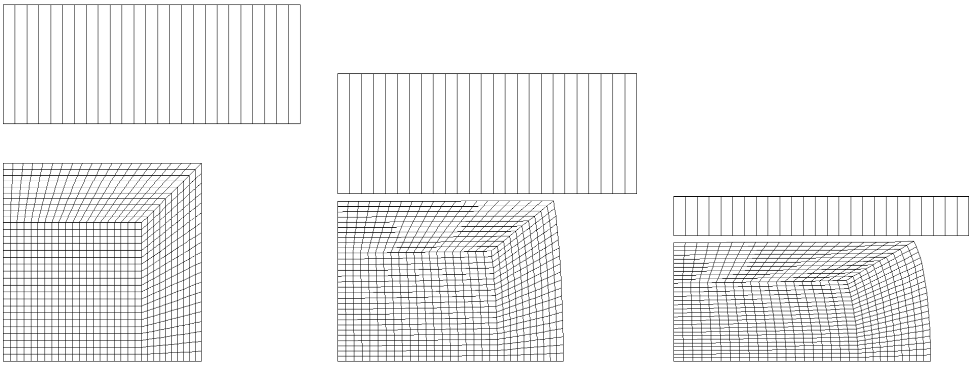


**Figure 5.18:** Forging process with a flat tool.

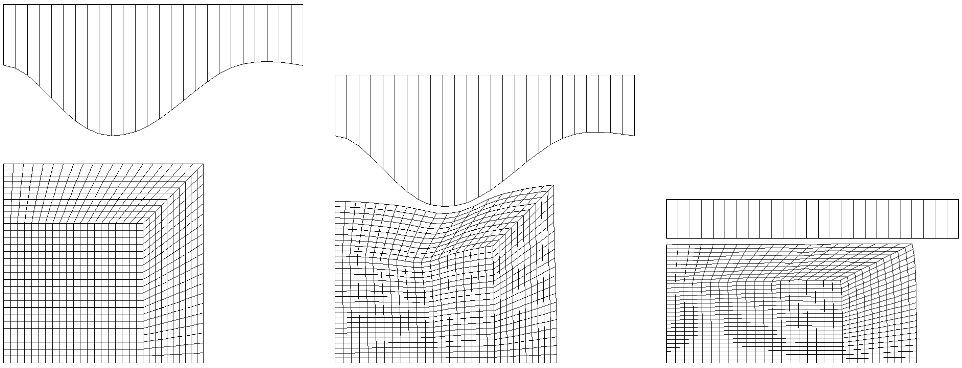


**Figure 5.19:** Forging process with an optimised tool shape and a prescribed stroke.

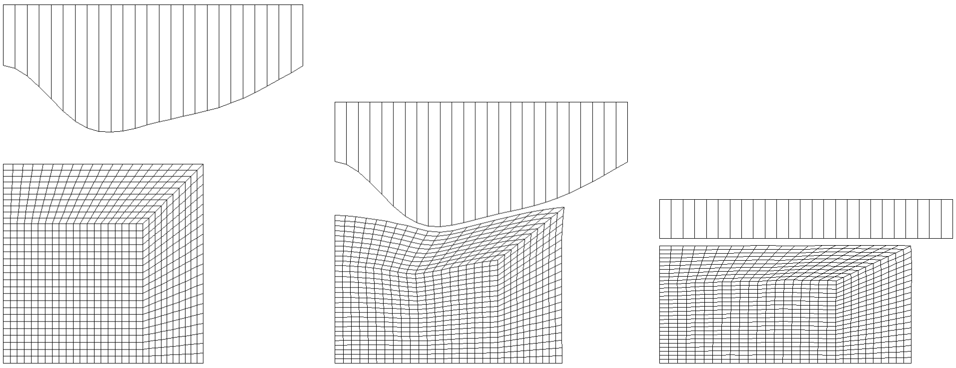


**Figure 5.20:** Forging process with an optimised tool shape, where the stroke was taken as a parameter.

### 5.2.3 Optimisation of Heating Parameters for Hot Forming Operation

In the present example a two stage forming process is considered. In the first stage (heating, Figure 5.21) an axisymmetric billet with length $L$=30 *mm* and radius $r$=10 *mm* with initial temperature 20℃ is heated at $x=L$ with a heat flux $F_o$ for time $t_h$ while the surface at $x=0$ is kept at the initial temperature. Other surfaces are insulated.

In the second stage (forming, Figure 5.22) the billet is deformed in such a way that the prescribed displacement at $x=L$ is $u_x=-2$ *mm* and at $x=0$ is $u_x=0$. Other surfaces are free. The deformation is modelled by an ideal elasto-plastic material model where the flow stress depends on temperature. No heat transfer is assumed in the second stage.

Material properties are summarised in Table 5.9 and Table 5.10. Figure 5.21 and Figure 5.22 show the discretised configurations of the specimen before and after heating and forming. Only half of the domain is represented due to symmetry.

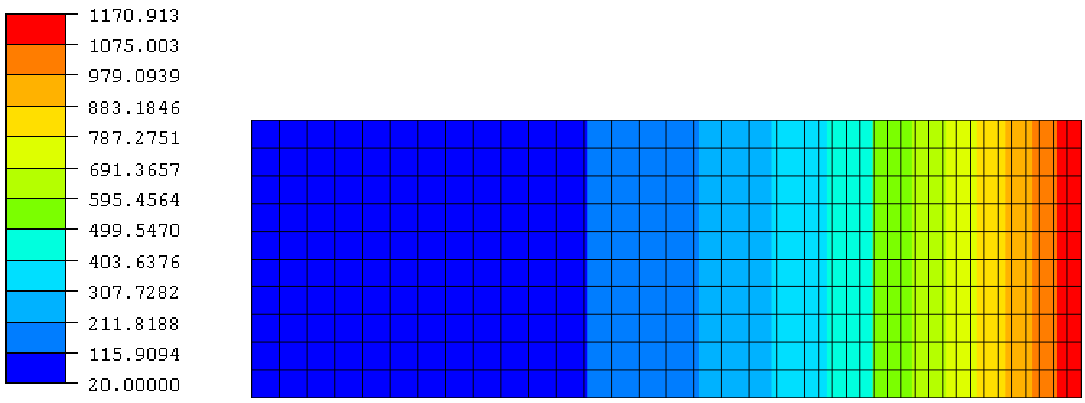



**Figure 5.21:** Temperature distribution after heating. The heat flux $F_o$=3.500 W/m$^2$ is applied for $t_h$=10 s.

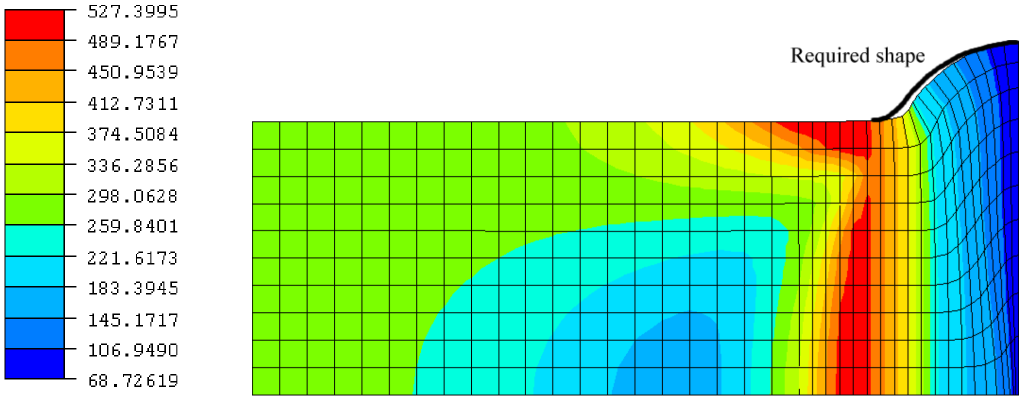



**Figure 5.22:** Effective stress distribution in the deformed specimen after forming. Displacement of the right surface is $u_x$=-2 mm.

**Table 5.9:** Material properties.

| | |
|---|---|
| Thermal conductivity (K) | 30 W/mK |
| Heat capacity (C) | 500 J/kg K |
| Density (ρ) | 7850 kg/m$^3$ |
| Youngs modulus (E) | 210000 MPa |
| Poissons ratio (ν) | 0.3 |

**Table 5.10:** Yield stress as a function of temperature.

| Temperature [$^o$C] | Yield stress [MPa] |
|---|---|
| 20 | 700 |
| 100 | 620 |
| 200 | 560 |
| 300 | 540 |
| 400 | 510 |
| 500 | 500 |
| 600 | 390 |
| 700 | 200 |
| 800 | 180 |
| 900 | 150 |
| 1000 | 120 |
| 1100 | 90 |
| 1200 | 60 |

The aim is to find optimal heating parameters defined by heat flux $F_o$ and heating time $t_h$ so that the difference between the required and computed shapes of the specimen after forming is minimal. The objective function to be minimised is expressed in terms of the differences of the required and computed node coordinates in the interval $20mm \leq x \leq 28mm$. The choice of heating parameters is constrained by a maximum permissible temperature of the specimen $T_{max}=1200^{o}C$. The constraint is presented in Figure 5.23.

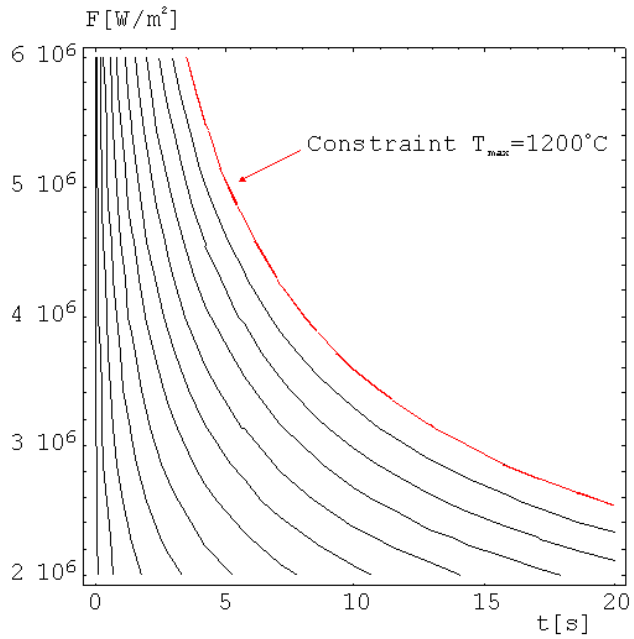



**Figure 5.23:** Constraint imposed on the choice of heating parameters. Contours show maximum temperature of the specimen as a function of applied heat flux and heating time. Spacing between contours is $100^{o}C$

The constraint is implied by adding a penalty term in the objective function:

$$D(\mathbf{p}) = \sum_{i=1}^{N} \left(y_i^p - y_i^m(\mathbf{p})\right)^2 + C \exp\left(T(\mathbf{p}) - T_{\max}\right) \tag{5.22}$$

$\mathbf{p} = [F_0, t_h]^t$ is the vector of optimisation parameters. The nodal coordinates prescribed by the required final shape are denoted by upper index $p$ and the nodal coordinates calculated at given values of optimisation parameters are denoted by upper index $m$. The temperature at the right-hand end of the billet after heating is denoted by $T(\mathbf{p})$.

The heat flux and heating time can only have positive values. This is ensured by applying a transformation function that maps parameters from [-∞,∞] to [0,∞]:

$$p_i = G(q_i) = A\exp(Bq_i),\ i = 1,2\,. \tag{5.23}$$

The billet deformation after the forming depends on the billet temperature distribution, which is calculated according to the following equation[20],[22]:

$$T(x,F,t) = \frac{Fx}{K} - \frac{8Fl}{K\pi^2}\sum_{n=1}^{\infty}\left(\frac{(-1)^n}{(2n+1)^2}\exp(-\frac{K(2n+1)^2\pi^2 t}{\rho C 4l^2})\right)\sin\left(\frac{(2n+1)\pi x}{2l}\right) \tag{5.24}$$

Temperature distributions for three different heating regimes are presented in Figure 5.24.

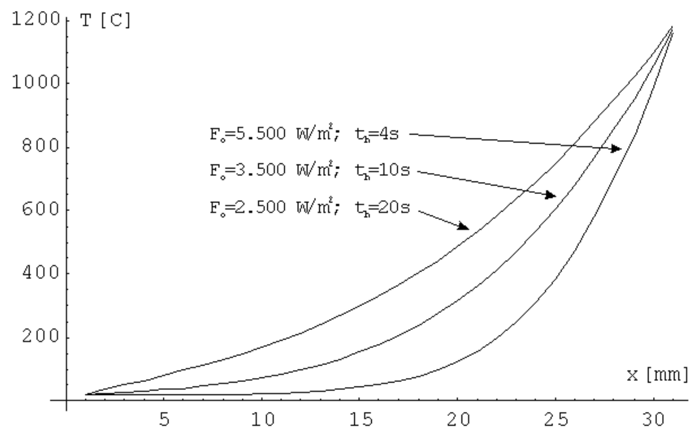



**Figure 5.24:** Temperature distributions along the x axis for three different heating regimes.

Different temperature distributions result in different shapes of the billet after the forming. Shapes that correspond to temperature distributions from the Figure 5.24 are presented in Figure 5.25.

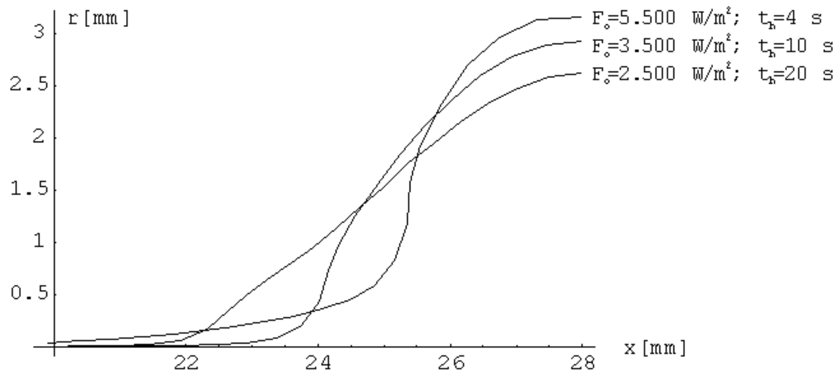



**Figure 5.25:** Deformed shapes of the specimen after forming for three different heating regimes from Figure 5.24.

The required shape of the billet after both stages of forming was prescribed by the following function:

$$y^p(x) = r + \frac{3}{1 + 25000\exp(-2(x-20))}. \tag{5.25}$$

Optimisation was done by the inverse shell using the nonlinear simplex method. Optimal solution was found in 34 iterations. The required shape and the shape achieved with optimal parameters are shown in Figure 5.26.

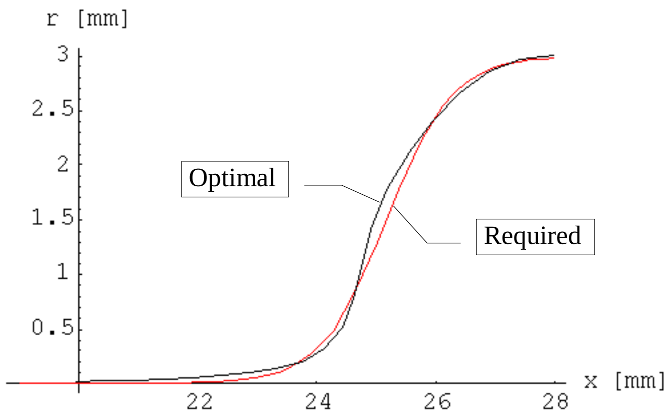



**Figure 5.26:** Required and optimal shape of the billet at optimal parameter values $F_{opt}$ = 3628674.9 W/m$^2$ and $t_{opt}$ = 7.99 s. Value of the objective function at these parameters was D($\mathbf{p}_{\mathrm{opt}}$) = 0.1448.

The optimisation shell tabulating utilities were employed to sample the objective function in the neighbourhood of optimal parameters. Sampled data was plotted by *Mathematica*. Figure 5.27 shows 19x17 points diagram of the objective function without the penalty term. In Figure 5.28 the constraints, the optimisation path and the optimal solution are also shown.

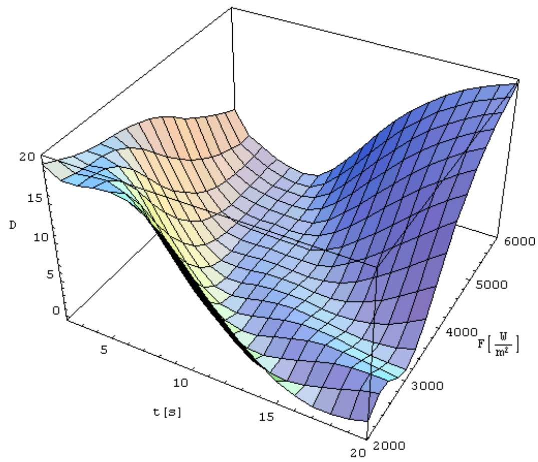



**Figure 5.27:** Objective function without the penalty term.

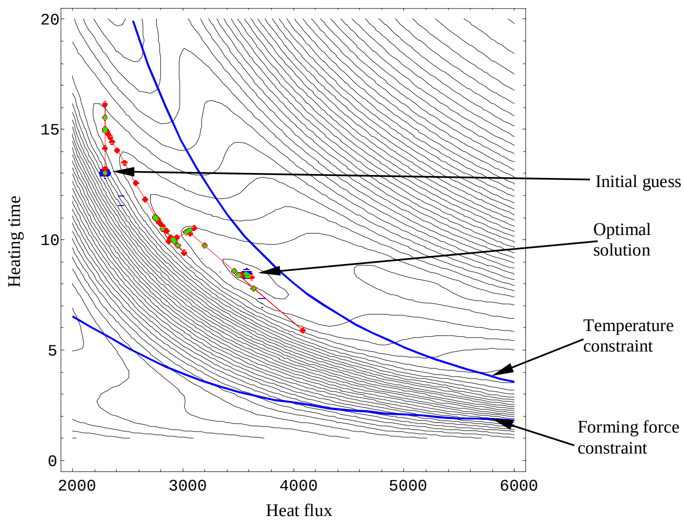



**Figure 5.28:** Contours of the objective function with temperature constraint and optimal solution.

## *5.3 Optimal Prestressing of Cold Forging Dies*

The final example considered is an industrial case and is related to prestressing of cold forging dies which are subject to low cycle fatigue.

Cold forging dies are subjected to high operational loads which often lead to fatigue failure. To prevent or reduce excessive growth of fatigue cracks the dies are used in a prestressed condition which must be designed in such a way that plastic cycling and tensile stress concentrations in the die are minimised[5],[6]. This can be achieved by optimising the geometry of the interface between the stress rings and die inserts. Prestressing of an axisymmetric die[7]-[9] which can be simulated in two dimensions is considered first. Then a three dimensional prestressing example[13],[14] is presented.

### 5.3.1 Optimisation of the Geometry of the Outer Surface of an Axisymmetric Die

Prestressing is used in cold forging technology to increase the service life of tooling systems. In conventional approaches the interference between the stress ring and the die insert is uniform resulting in small variations of fitting pressure distributions. However, the introduction of high strength stripwound containers[5] allows relatively high variations of the fitting pressure which can be optimised.

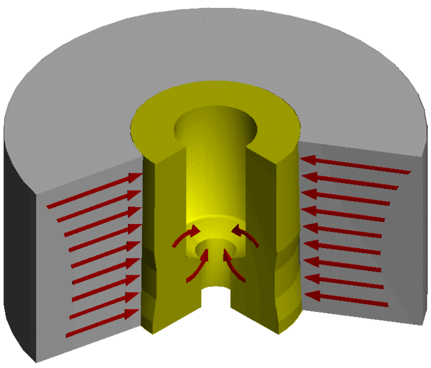


**Figure 5.29:** Prestressing of the die insert.

In the approach proposed in [6] a non-uniform fitting pressure distribution is obtained by modifying the geometry of the interference which is parametrised as presented in Figure 5.30. By optimising the position and geometry of the groove it is possible to achieve high compressive stresses in the inlet radius (Figure 5.29) and therefore reduce damage and eventual crack propagation in this critical part of the die insert during use.

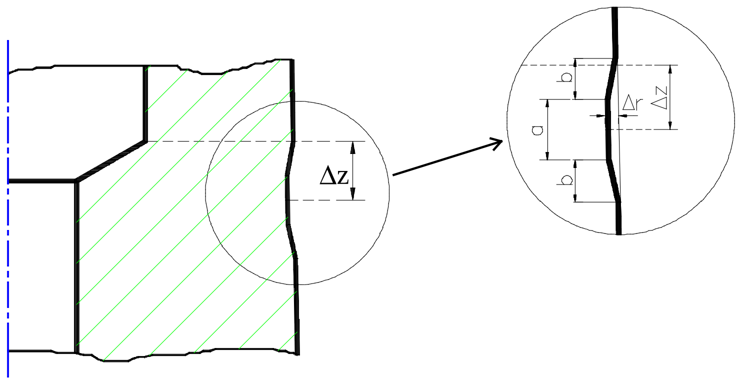



**Figure 5.30:** Geometric design of the interference at the die-ring interface.

The tooling system was discretised by the finite element method as presented in Figure 5.31. Both the tool and the ring were considered elastic and Coulomb's friction law was assumed at their interface. The prestressed conditions were analysed so that the die insert and the ring overlap at the beginning of the computation. Equilibrium is then achieved by an incremental-iterative procedure by updating the contact penalty coefficient.

Two objectives of the optimisation procedure were applied: to position the minimum of the axial stress acting in the inlet radius close to node 6 (see Figure 5.31) and to make this minimum as numerically large as possible. Optimisation was performed as minimisation of the objective function which was designed to measure the violation of our objectives in the following way:

$$F(a,b,\Delta r,\Delta z) = K\left(f_m(a,b,\Delta r,\Delta z)\right)^2 + \sigma_{zz}^{(6)}(a,b,\Delta r,\Delta z), \qquad (5.26)$$

where $f_m(a,b,\Delta r,\Delta z)$ is a measure of the distance between node 6 and the point on the inlet radius where minimum axial stress is calculated, $\sigma_{zz}^{(6)}(a,b,\Delta r,\Delta z)$ is the axial stress at node 6 and $K$ is a weighting factor which weights the importance of the two objectives. If the second term in (5.26) is omitted, the problem does not have a unique solution. There is more than one set of parameters for which the minimum axial stress appears exactly in node 6. Therefore the objective function was designed with both terms. The results obtained with $K = 1000$ and $K = 100$ are given in Table 5.11 and Table 5.12, respectively.

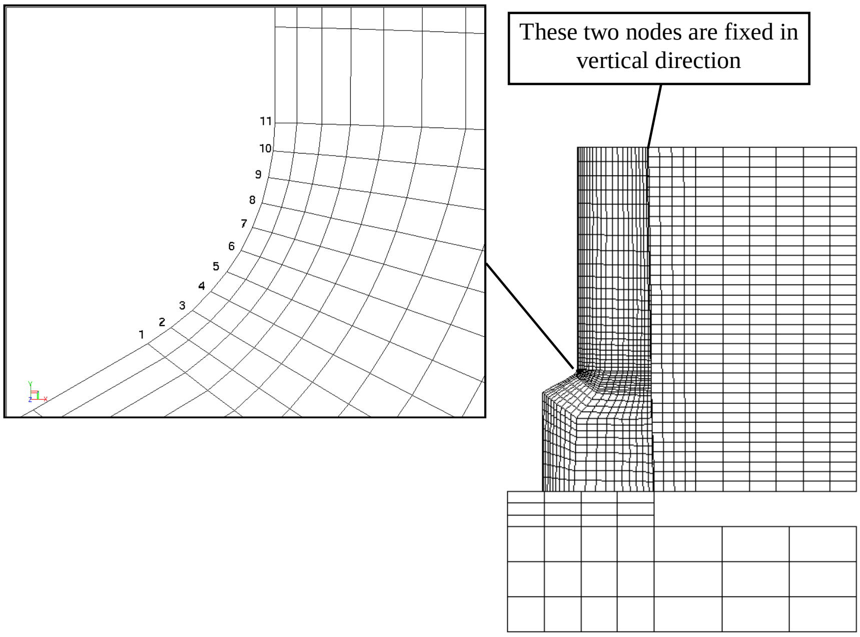



**Figure 5.31:** Numerical discretisation of the tooling system.

After the optimisation a parameter study has been performed to assess the stability of the problem. It has been found that the problem is well posed, so that the optimisation approach presented can be applied. The only restriction is the initial guess which should be chosen so that the axial stresses in the inlet radius are compressive.

**Table 5.11:** Results of optimisation with $K = 1000$.

| | $a$ $[mm]$ | $b$ $[mm]$ | $\Delta r$ $[mm]$ | $\Delta z$ $[mm]$ |
|---|---|---|---|---|
| Initial guess | 6 | 8 | 0.3 | -4 |
| Final solution | 4.97 | 6.15 | 0.319 | -7.01 |
| Final value of $F$ | -1511 | | | |

**Table 5.12:** Results of optimisation with $K$ = 100.

| | $a$ [$mm$] | $b$ [$mm$] | $\Delta r$ [$mm$] | $\Delta z$ [$mm$] |
|---|---|---|---|---|
| Initial guess | 6 | 8 | 0.3 | -4 |
| Final solution | 5.79 | 7.11 | 0.308 | -5.28 |
| Final value of $F$ | -1855 | | | |

## 5.3.2 Evaluation of Optimal Fitting Pressure on the Outer Die Surface

In the present example the fitting pressure distribution at the interface between the die insert and the stress ring (Figure 5.32) was optimised.

In order to reduce the appearance of cracks, the spherical part of the stress tensor at the critical locations is to be minimised by varying the fitting pressure distribution at the interface between the die insert and the stress ring. The following two constraints were taken into account:

- The normal contact stress distribution at the interface between die insert and stress ring must be compressive.
- The effective stress distribution at all points within the prestressed die must be below the yield stress.

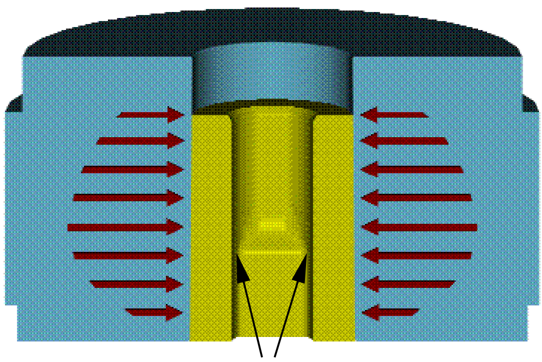



**Figure 5.32:** A prestressed cold forging die with indicated critical locations where cracks tend to occur.

The fitting pressure distribution was represented by 220 parameters corresponding to a subdivision of the contact surface into 20 vertical and 11 circumferential units $A_{ij}$ (Figure 5.33). Index $k$ which associated the optimisation parameter (i.e. the pressure) $p_k$ with the corresponding $A_{ij}$ is computed as $k=(j-1)\cdot 11+i$.

Because of symmetry only one half of the die was simulated (Figure 5.33). The objective function was defined as the spherical part of the stress tensor at the critical location, i.e.

$$\theta(\mathbf{p})=\frac{1}{3}\sigma_{kk}^{crit.}(\mathbf{p}). \tag{5.27}$$

The first constraint was enforced by using transformations where instead of optimisation parameters $\mathbf{p}$ a new set of variables $\mathbf{t}$ is introduced. The following transformations are applied:

$$p_k=\alpha\frac{g_k}{\sqrt{g_i g_i}}; \qquad g_k=e^{t_k}. \tag{5.28}$$

In the above equation $\alpha$ is a scalar variable which satisfies the second constraint. Once the optimisation problem is solved for $\mathbf{t}$ the optimal set of parameters $\mathbf{p}$ is derived by using equation (5.28).

Sensitivities of the objective function with respect to optimisation parameters were calculated according to the adjoint method (chapter 3) in the finite element environment, as well as the objective function. They are shown in Figure 5.33. These calculations were used in the optimisation procedure. The obtained optimal pressure distribution is shown in Table 5.13 and in Figure 5.34. Figure 5.35 shows the prestressing conditions and the effective stress distribution for the optimally distributed fitting pressure.

**Table 5.13:** Optimal set of parameters $\mathbf{p}^{\mathrm{opt}}$ defining the fitting pressure distribution.

| j \ i | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **1** | 93.22 | 89.60 | 85.38 | 82.11 | 80.36 | 83.31 | 92.85 | 105.13 | 121.88 | 136.91 | 147.27 |
| **2** | 101.00 | 97.53 | 90.23 | 82.95 | 81.88 | 87.34 | 102.07 | 121.50 | 151.92 | 174.00 | 185.89 |
| **3** | 116.63 | 103.89 | 91.26 | 81.00 | 77.90 | 90.44 | 119.19 | 160.50 | 209.39 | 246.51 | 270.45 |
| **4** | 129.90 | 108.10 | 86.25 | 66.13 | 60.13 | 79.96 | 135.44 | 209.61 | 297.44 | 357.58 | 398.65 |
| **5** | 138.26 | 103.64 | 64.15 | 27.40 | 7.03 | 43.68 | 124.26 | 257.59 | 409.57 | 526.52 | 600.05 |
| **6** | 129.25 | 85.72 | 27.50 | 0.55 | 0.30 | 0.58 | 102.36 | 328.85 | 582.39 | 764.12 | 859.46 |

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **7** | 99.56 | 46.61 | 0.71 | 0.19 | 0.13 | 0.20 | 104.45 | 409.26 | 733.05 | 977.29 | 1109.89 |
| **8** | 64.44 | 4.84 | 0.28 | 0.13 | 0.10 | 0.19 | 128.25 | 468.18 | 830.35 | 1113.63 | 1261.72 |
| **9** | 24.71 | 0.77 | 0.24 | 0.15 | 0.14 | 0.58 | 208.79 | 532.54 | 851.79 | 1117.16 | 1259.67 |
| **10** | 0.75 | 0.47 | 0.30 | 0.25 | 0.38 | 73.55 | 274.47 | 556.57 | 807.61 | 1007.96 | 1121.13 |
| **11** | 0.28 | 0.29 | 0.28 | 0.48 | 19.04 | 138.51 | 302.58 | 506.80 | 687.32 | 843.22 | 915.33 |
| **12** | 0.19 | 0.21 | 0.27 | 0.59 | 41.78 | 148.50 | 282.74 | 415.76 | 560.36 | 661.28 | 716.32 |
| **13** | 0.16 | 0.18 | 0.24 | 0.51 | 27.46 | 118.75 | 225.14 | 332.13 | 424.23 | 500.06 | 542.03 |
| **14** | 0.15 | 0.17 | 0.22 | 0.43 | 6.14 | 84.16 | 164.29 | 246.59 | 320.54 | 374.65 | 403.52 |
| **15** | 0.16 | 0.18 | 0.23 | 0.38 | 1.75 | 53.06 | 116.07 | 176.89 | 231.37 | 270.39 | 289.37 |
| **16** | 0.18 | 0.20 | 0.25 | 0.39 | 0.99 | 27.31 | 74.64 | 118.57 | 156.76 | 187.19 | 200.39 |
| **17** | 0.21 | 0.23 | 0.28 | 0.40 | 0.80 | 5.40 | 44.14 | 73.12 | 99.67 | 119.70 | 132.83 |
| **18** | 0.27 | 0.29 | 0.34 | 0.45 | 0.71 | 1.86 | 12.72 | 29.81 | 49.39 | 65.89 | 72.32 |
| **19** | 0.36 | 0.38 | 0.43 | 0.51 | 0.66 | 0.96 | 1.54 | 2.83 | 6.18 | 15.27 | 23.07 |
| **20** | 0.51 | 0.52 | 0.54 | 0.56 | 0.57 | 0.62 | 0.66 | 0.68 | 0.69 | 0.72 | 0.72 |

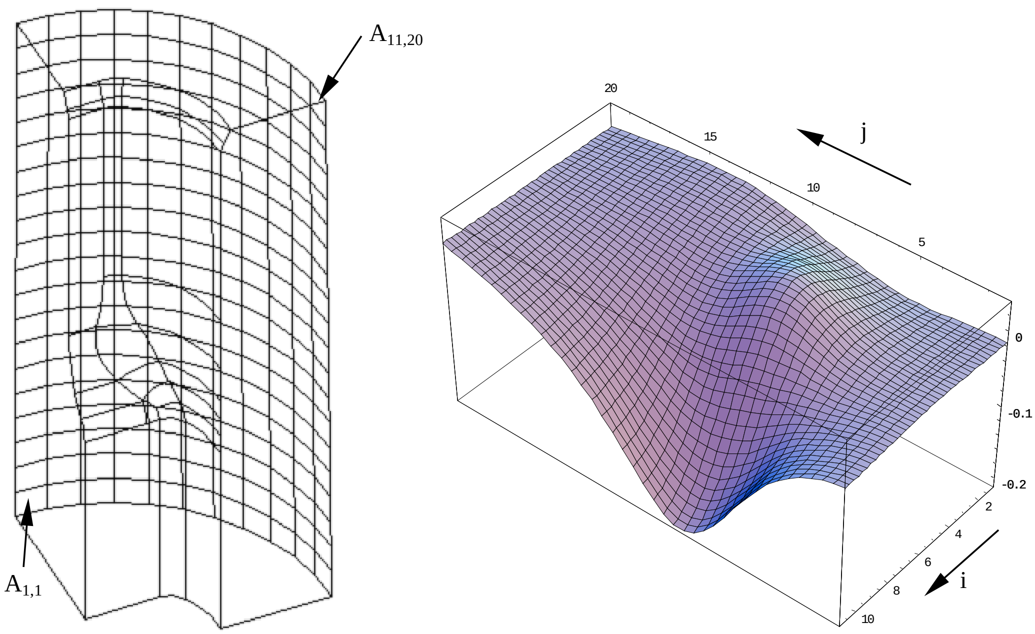



**Figure 5.33:** Subdivision of the outer surface of the die and sensitivities $D\theta/Dp_k$ .

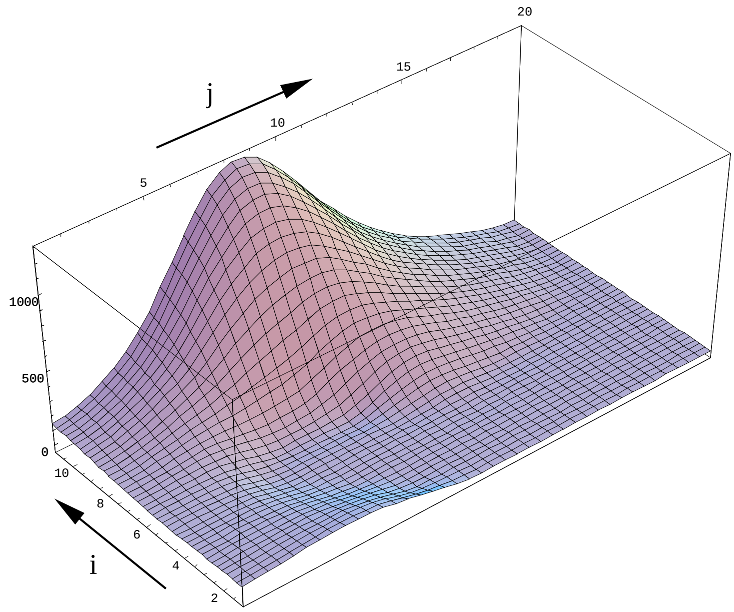



**Figure 5.34:** Optimal fitting pressure distribution.

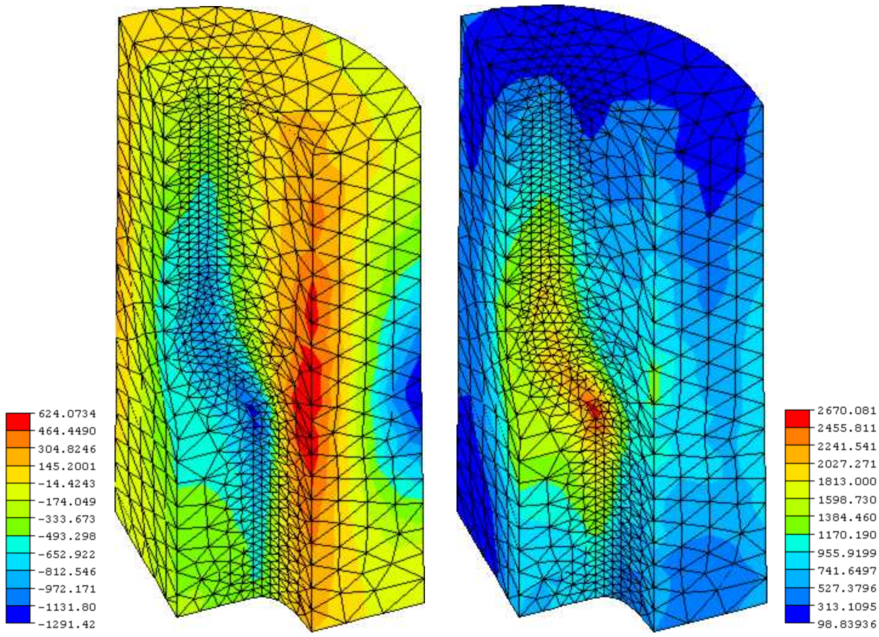



**Figure 5.35:** Prestressing conditions $\sigma_{kk}/3$ and effective stress distribution for $\mathbf{p}^{\text{opt}}$

## 5.4 Further Examples

A number of other problems were solved by the presented optimisation shell. In the work done by Musil[15] friction parameters were estimated from the results of the spike forming test (Figure 5.36). A block sample was pressed between two cylinders and a plate. Torque and the two components of the force acting on the cylinders were measured at different stages to provide input for inverse analysis.

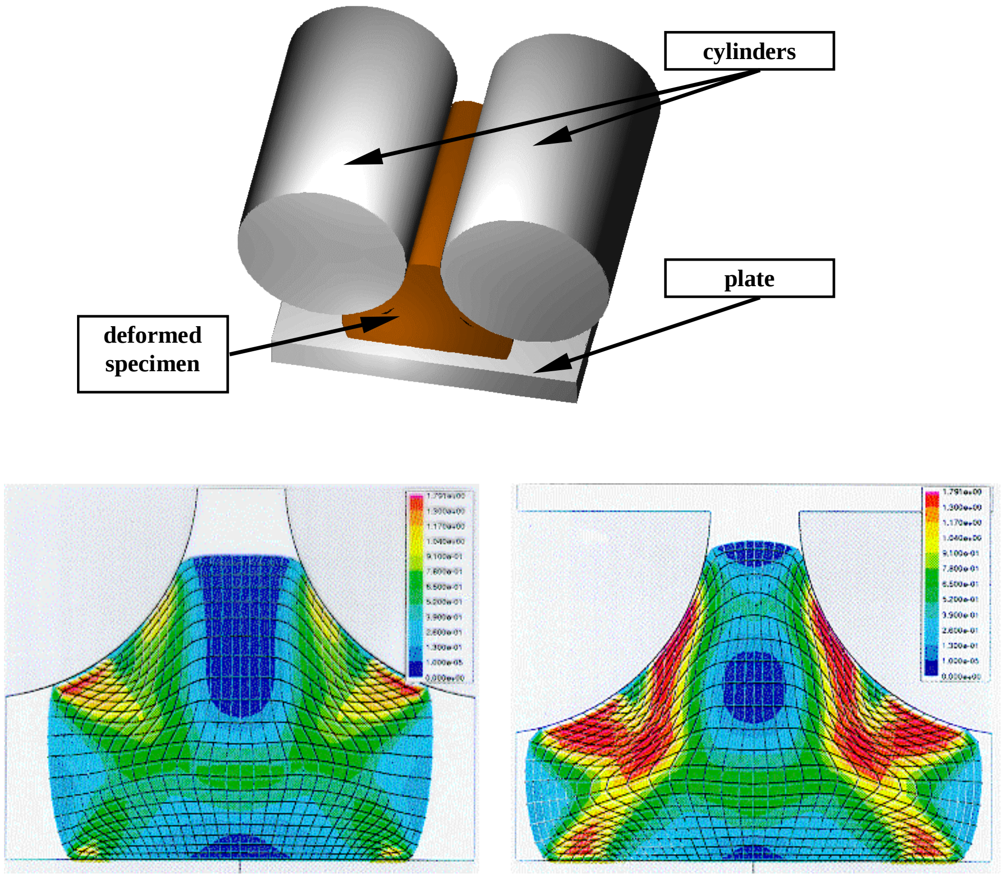



**Figure 5.36:** Experimental set-up and numerical simulation of the spike forming test.

Another inverse parameter identification is described in the work done by Goran Kugler[16] where the dependence of the heat transfer coefficient between two bodies in contact on the normal contact stress is estimated. A hotter cylindrical specimen was symmetrically pressed by cooler dies. The temperature in a few sampling points within the dies was measured at different times.

These two examples illustrate the applicability of the inverse approach to identification of model parameters which are difficult to estimate by other methods. The parameters of physical models that describe contact phenomena which take place during hot working operations are especially difficult to quantify. Due to high contact stresses and temperatures it is hard to make in situ measurements. It is however possible to design experiments in which conditions similar to those in the real process are reestablished and where accurate indirect measurements can be performed. The estimated parameters can be used to calibrate numerical simulation of the real process.

A shape optimisation example which is close to real-life problems in metal forming was solved by Damijan Markovič[19]. A pre-form shape was optimised in order to obtain optimal die filling and material flow. The problem is outlined in Figure 5.37, while the results of a numerical simulation are shown in Figure 5.38.

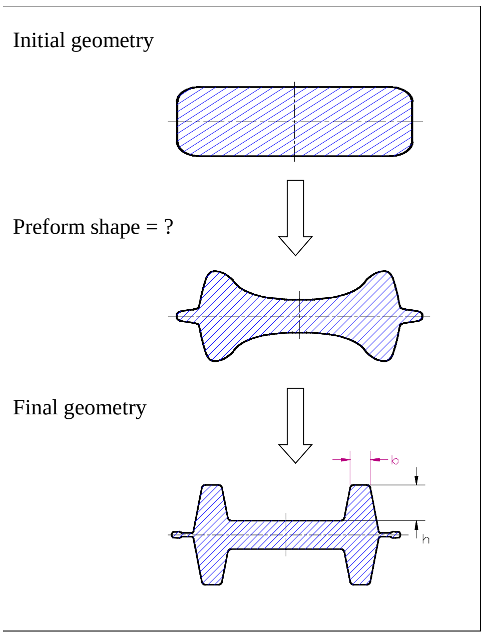



**Figure 5.37:** Two stage forging process where the pre-form shape is to be optimised.

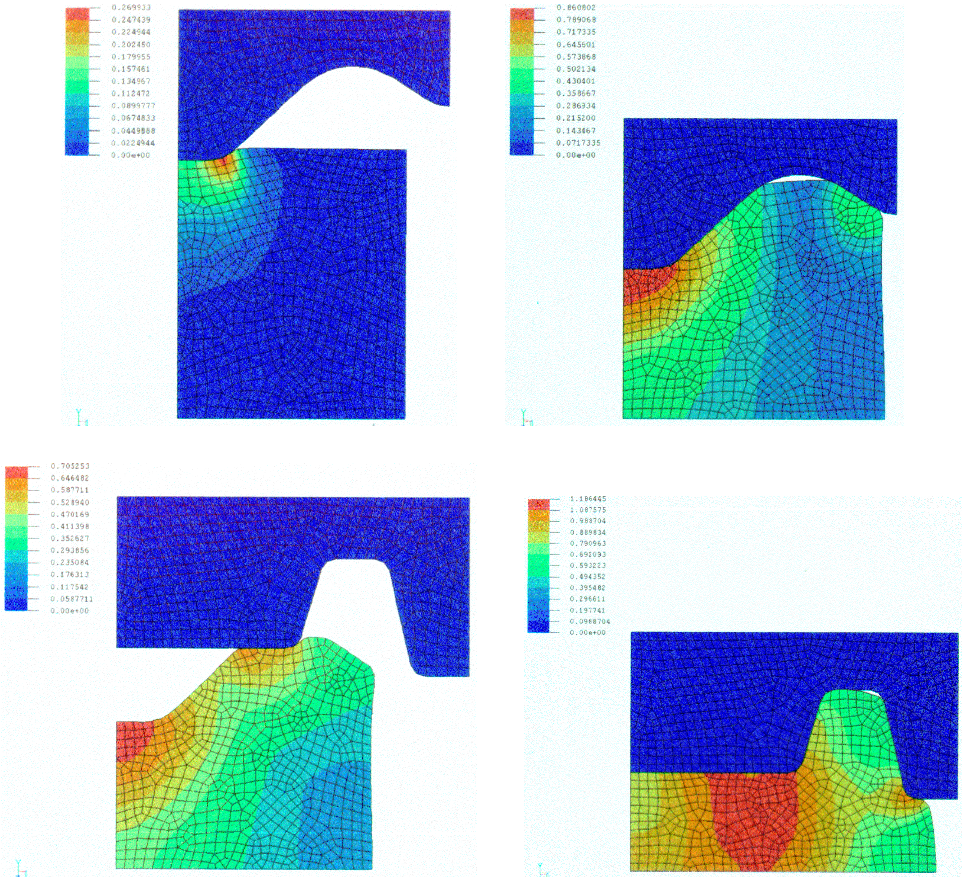


**Figure 5.38:** Numerical simulation of a two stage forging process.

An interesting example which demonstrates a stand-alone use of the optimisation shell was designed by Jelovšek[18]. Equilibrium arrangements of a given number of equally charged particles in a planar circular region were obtained by minimisation of the total potential energy with respect to particle positions. Different arrangements were obtained by different initial guesses. Expressions defining the objective and constraint functions and their derivatives can be expressed analytically and were evaluated by the optimisation shell. Sample results are shown in Figure 5.39.

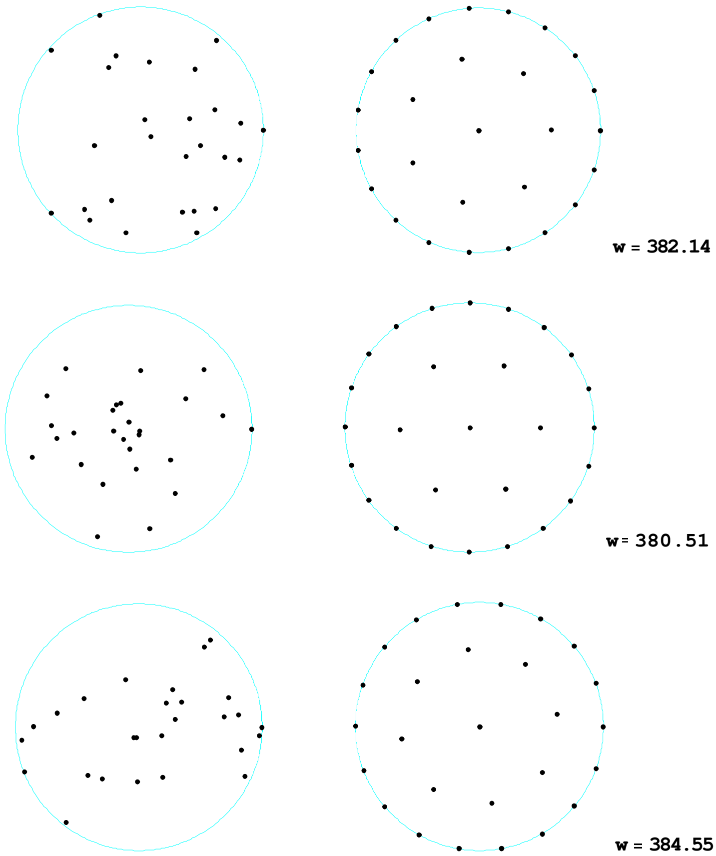



**Figure 5.39:** Three different equilibrium arrangements of 27 charged particles in a circular region[18]. The corresponding random initial configurations are shown on the left. Relative potential energy of the equilibrium states is indicated on the right.

## *References:*


[1] G. Dieter, *Mechanical Metallurgy*, McGraw-Hill, Singapore, 1986.

[2] T. Rodič, I. Gresovnik, D.R.J. Owen, *Application of error minimization concept to estimation of hardening parameters in the tension test*, In: Computational plasticity : fundamentals and applications : proceedings of the Fourth International conference held in Barcelona, Spain, 3rd-6th, April, 1995, Swansea, Pineridge Press, 1995, part 1, pp. 779-786.

[3] I. Grešovnik, T. Rodič, *Določanje parametrov plastičnega utrjevanja z inverzno analizo nateznega preizkusa* (in Slovene), In: Zbornik del, Kuhljevi dnevi '94, Smarjeske Toplice, 22. - 23. september 1994, Maks Oblak, ur., V Ljubljani, Slovensko drustvo za mehaniko, 1994, pp. 193-198.

[4] T. Rodič, I. Grešovnik, *Omejevanje zalog vrednosti pri reševanju inverznih problemov z uporabo transformacij* (in Slovene), In: Zbornik del : Kuhljevi dnevi '95, Kuhljevi dnevi '95, Radenci, 21. - 22. september 1995, Maks Oblak, ur., V Ljubljani, Slovensko drustvo za mehaniko, 1995, pp. 231-238.

[5] J. Gronbaek, *Stripwound Cold-forging Tools - a Technical and Economical Alternative*, Journal of Materials Processing Technology, Vol. 35, pp. 483-493, Elsevier, 1992.

[6] E. B. Nielsen, *Strip winding - Combined Radial and Axial Prestressing of Cold Forging Die Inserts*, Ph.D. Thesis, Technical University of Denmark and STRECON Technology, 1994.

[7] T. Rodič, I. Grešovnik, *Optimization of the prestressing of a cold forging tooling system*, In: Inverse problems in engineering : theory and practice : preliminary proceedings of the 2nd International conference on Inverse problems in engineering : theory and practice, Le Croisic, France, 9-14 June 1996. Vol. 2, New York, Engineering Foundation, 1996.

[8] T. Rodič, I. Grešovnik, *Constraint optimization of the prestressing of a cold forging tooling system*, In: Inverse problems in engineering : theory and practice, 2nd International Conference on Inverse Problems in Engineering, Le Croisic, France, June 9-14, 1996, Didier Delaunay, ur., Yvon Jarny, ur., Keith A. Woodbury, ur., New York, ASME, cop. 1998, pp. 45-52.

[9] T. Rodič, I. Grešovnik, Anton Pristovsek, Joze Korelc, *Optimiranje prednapetja orodij za preoblikovanje v hladnem* (in Slovene), In: Zbornik del, Kuhljevi dnevi '98, Logarska dolina, Slovenija, 1.-2. oktober 1998, Boris Stok, ur., Ljubljana, Slovensko drustvo za mehaniko, 1998, pp. 145-151.

[10] T. Rodič, J. Korelc, I. Grešovnik, *Inverse analyses and optimisation of cold forging processes*, In: First ESAFORM conference on material forming, Sophia-Antipolis (France), 17-20 March 1998, [Paris], [S. n.], 1998, pp. 255-258.

[11] T. Rodič, I. Grešovnik, *Analiza občutljivosti napetostnega polja v prednapetem orodju za preoblikovanje v hladnem* (in Slovene), RMZ-mater. geoenviron., vol. 46, no. 1, pp. 89-94, 1999.

[12] T. Rodič, I. Grešovnik, *Application of inverse and optimisation methods in cold forging technology* , In: 29th ICFG : plenary meeting, Gyur, Hungary 1996, 8-11th September 1996, Györ, Metal forming group of scientific soc. of Hungarian mech. engineers, 1996, pp. 8.1-10.

[13] T. Rodič, I. Grešovnik, M. Hänsel, M. Heidert, *Optimal Prestressing of Cold Forging Dies*, in M. Geiger (editor), Advanced Technology of Plasticity 1999: Proceedings of the 6th International Conference on Technology of Plasticity (held in Nuremberg, Germany), vol. 1, pp.. 253-258, Springer, Berlin, 1999.

[14] T. Rodič, I. Grešovnik, D. Jelovšek, J. Korelc, *Optimisation of Prestressing of a Cold Forging Die by Using Symbolic Templates*, in ECCM '99 : European Conference on Computational Mechanics (held in Munich, Germany), pp. 362-372, Technische Universität München, 1999.

[15] M. Musil, *Določitev koeficienta trenja z inverzno metodo* (in Slovene), diplomsko delo, University of Ljubljana, 1998.

[16] G. Kugler, *Določitev koeficienta toplotne prestopnosti z inverzno analizo* (in Slovene), diplomsko delo, University of Ljubljana, 1998.

[17] D. Jelovšek, *Določitev predoblike pri plastičnem preoblikovanju* (in Slovene), diplomsko delo, University of Ljubljana, 1998.

[18] D. Jelovšek, *Equilibrium States of Charged Particles in a Two-dimensional Circular Region*, electronic document at http://www.c3m.si/inverse/examples/charges/index.html, Ljubljana, 2000.

[19] Damijan Markovič, *Optimizacija predoblike pri kovanju* (in Slovene), diplomsko delo, University of Ljubljana, in press.

[20] H. S. Carslaw, J. C. Jaeger, *Conduction of Heat in Solids*, Clarendon Press, Oxford, 1959.

[21] A. Likar, *Osnove fizikalnih merjenj in merilnih sistemov (in Slovene)*, DMFA Slovenije, Ljubljana, 1992.

[22] I. Kuščer, A. Kodre, H. Neunzert, *Mathematik in Physik und Technik (in German)*, Springer - Verlag, Heidelberg, 1993.

[23] E. Kreyszig, *Advanced Engineering Mathematics (second edition)*, John Wiley & Sons, New York, 1993.

[24] W.H. Press, S.S. Teukolsky, V.T. Vetterling, B.P. Flannery, *Numerical Recipies in C – the Art of Scientific Computing*, Cambridge University Press, Cambridge, 1992.

# 6 CONCLUSIONS AND FURTHER WORK

In the present work a shell for solution of optimisation and inverse problems in conjunction with simulation programmes was presented. Emphasis was placed on the open and flexible structure of the shell, which makes it general with respect to the variety of problems to which it can be applied as well as the simulation systems with which it can be used.

The shell was combined with a general finite element system *Elfen* and applied to selected problems related to metal forming. This provided a good test for the adequacy of the shell concepts from the point of view that a complex simulation system was successfully utilised for solution of optimisation problems involving non-linear and path dependent responses, coupling of phenomena, frictional contact and large deformations. Experience justified the initial idea of the optimisation system consisting of a set of independent tools for solution of individual subproblems. The shell offers a framework for connecting such tools in a common system where they can be combined in the solution of complex optimisation problems. Once these tools are linked with the shell, the necessary interaction between them is established and they can be employed for the solution of optimisation problems as a part of a synchronised solution environment.

One of the basic guidelines in shell design was that it should not impose any a priori restrictions regarding the type of optimisation problems to which it is applicable. It was however assumed throughout that the shell will be applied to problems where evaluation of the objective or constraint functions (and eventually their derivatives) includes an expensive numerical simulation. This assumption allowed the file interpreter to be used as a user interface, thus the focus was on openness and flexibility of the interface rather than its speed.

The assumption regarding computationally demanding numerical simulations also dictates demands for the optimisation algorithms incorporated in the shell. In most cases the time needed for algorithm housekeeping operations is insignificant and the number of direct analyses necessary for finding the solution should therefore be regarded as principal the measure of effectiveness of the algorithm.

The shell design aims at building a general optimisation system applicable to a wide variety of problems. To make this goal achievable, the shell must provide a flexible framework for implementation and testing of new utilities, which must be accompanied by use in practice. Such a broad scope implies a number of demands, which can not be met instantly but are a matter of systematic long term development. The last section in this text is therefore devoted to assessment of possible further development of the shell. This assessment is based on appreciation of the current state and practical experience, which gives indication regarding which development tasks will be among the most important in the future.

## *6.1 Further Work Related to the Optimisation Shell*

Development tasks can be divided in two groups. The first group includes developments related to the shell structure and concepts, while the second group includes development and incorporation of modules with given functionality.

Development of a complete open library is currently regarded as a primary development task. Such a library will enable incorporation of modules developed in different places. It must provide a condensed standard set of simple to use functions, which still enable sufficient interaction with constituent units of the shell. A large portion of the library has already been implemented and must be equipped by appropriate documentation. Other parts of the library will be developed simultaneously with introduction of additional functionality and final definition of additional concepts. Another important task is definition of rules for adding functions for direct access to module functionality to the shell library. The current arrangement anticipates access to module functionality through the user interface, while the framework for making this functionality available for direct use in other modules has not yet been set up.

The file interpreter will probably undergo substantial changes. Experience has shown that use of the current implementation is sometimes difficult and prone to errors. In order to suppress this deficiency, the syntax will have to be modified, probably towards the syntax of some common high level language such as C. This will require partial revision of the interaction between the interpreter, the expression evaluator and the variable system. This will also affect the syntax checker and debugger, which alone need some improvements to become a more reliable tool for detection and elimination of errors in command files.

A great deficiency of the shell is that it does not have a sufficiently general common system for processing and presentation of results. Such a system should enable, for example, the storing of the complete information about the optimisation path for later presentation in a standard way. This should be accompanied by appropriate visualisation tools. Currently the optimisation path can be written to a file by using the appropriate output command in the analysis block of the command file.

The need for a modular and hierarchical library of optimisation algorithms has been indicated by practical experience. The basis of such a library is currently being set up, while its development is a long term task which will be strongly affected by simultaneous experience gained by use in practice. One argument for development of such a library is the observation that more complex optimisation methods often incorporate more basic algorithms for the solution of subproblems. A well structured hierarchical library can therefore significantly facilitate development of increasingly sophisticated algorithms developed for special purposes. The need for development of special purpose algorithms is always present in an optimisation system such as the shell. A readily available example which supports this statement is establishing a recurrent interaction between an optimisation algorithm and a finite element simulation in such a way that a finer mesh is used in simulation as the solution of the optimisation problem is approached. This can save time since a coarse mesh is used far from the solution where high accuracy of simulation results is not crucial.

There is another argument which supports development of optimisation algorithms simultaneously with development of the shell. The resulting library will include some facilities which are usually not a part of existing optimisation libraries, but are important for incorporation in the shell system in compliance with its concepts. Such facilities will enable e.g. use of a common system for reporting errors and a common system for presentation of results.

Development of the optimisation library will induce the need for a suitable testing environment. By now algorithms were tested either on practical examples or on test problems defined through the command file. In the future this should be supplemented by a system of standard test functions pre-defined in the shell. Such a testing suite will make comparison of different algorithms for similar problems easier. The test problems will be designed with features which are expected to be difficult for specific algorithms.

Development of a general shape optimisation module will begin in the near future. It will increase the applicability of the shell because shape plays an important role in almost all branches of engineering design. This module will include tools for definition of parameter dependent transformations of shape and appropriate functions for transformation of discrete sets of points as well as continuous domains. Module functions will act on the geometrical level, therefore it will be possible to combine module functionality with existing functionality already implemented in individual simulation systems.

More specific tasks will be related to further effective utilisation of simulation systems. A direct interface with the finite element system *Elfen* has now been implemented. This interface is currently on a very basic level and includes access to programme data structures and basic control over its execution. Higher level functions will be added in the future, which will enable e.g. direct use of built-in post-processing capabilities such as integration of derived quantities over surfaces and volumes. Use of the programme for solution of optimisation problems will be facilitated by introduction of optimisation entities. These are objects, which include the definition of geometrical entities that are involved in definition of the optimisation problems, specification of data needed by the shell and specification of operations which will be performed on this data.

Use of the shell for practical purposes will in certain cases impose stronger requirements with respect to simplicity of use. Such requirements will be met by building templates, which will utilise the shell for solution of specific sets of problems. These templates will represent upgrades of the shell user interface by trading a certain level of generality for the required user friendliness. Various facilities will be employed in building such templates, e.g. high level special purpose commands implemented in the shell, portions of code for the shell file interpreter prepared for accomplishing pre-defined combinations of tasks, and pre and post processing utilities integrated in the simulation environment which will be utilised for the solution of problems.