Université Libre de Bruxelles
Faculté des Sciences Appliquées
Service: IRIDIA

Année académique
2003-2004

CONDOR: a constrained, non-linear, derivative-free parallel optimizer for continuous, high computing load, noisy objective functions.

Directeur de Thèse :
Pr. H. Bersini

Thèse de doctorat présenté par Frank Vanden Berghen en vue de l'obtention du grade de Docteur en Sciences Appliquées.

“Everything should be made as simple as possible, but not simpler.”

– Albert Einstein

“Anyone who invokes authors in discussion is not using his intelligence but his memory.”

– Leonardo da Vinci

Summary

This thesis is about parallel, direct and constrained optimization of high-computing-load objective functions.

The main result is a new original algorithm: CONDOR ("*COnstrained, Non-linear, Direct, parallel Optimization using trust Region method for high-computing load, noisy functions*"). Other original results are described at the beginning of Chapter 1. The aim of this algorithm is to find the minimum $x^* \in \Re^n$ of an objective function $\mathcal{F}(x) \in \Re$ using the least number of function evaluations. It is assumed that the dominant computing cost of the optimization process is the time needed to evaluate the objective function $\mathcal{F}(x)$ (One evaluation can range from 2 minutes to 2 days). The algorithm will try to minimize the number of evaluations of $\mathcal{F}$, at the cost of a huge amount of routine work.

CONDOR has been designed with one application in mind: the METHOD project. (METHOD stands for Achievement Of Maximum Efficiency For Process Centrifugal Compressors THrough New Techniques Of Design). The goal of this project is to optimize the shape of the blades inside a Centrifugal Compressor (see illustration of the compressor's blades in Figure 11.1).The objective function is based on a CFD (computation fluid dynamic) code which simulates the flow of the gas inside the compressor. The shape of the blades in the compressor is described by 31 parameters ($n = 31$). We extract from the numerical simulation the outlet pressure, the outlet velocity, the energy transmit to the gas at stationary conditions. We aggregate all these indices in one general overall number representing the quality of the turbine. We are trying to find the optimal set of 31 parameters for which this quality is maximum. The evaluations of the objective function are very noisy and often take more than one hour to complete (the CFD code needs time to "converge").

CONDOR is a *direct* optimization tool (i.e., that the derivatives of $\mathcal{F}$ are not required). The only information needed about the objective function is a simple method (written in Fortran, C++,...) or a program (a Unix, Windows, Solaris,... executable) which can evaluate the objective function $\mathcal{F}(x)$ at a given point x. The algorithm has been specially developed to be very robust against noise inside the evaluation of the objective function $\mathcal{F}(x)$. This situation is very general, the algorithm can thus be applied on a vast number of situations.

CONDOR is able to use several CPU's in a cluster of computers. Different computer architectures can be mixed together and used simultaneously to deliver a huge computing power. The optimizer will make simultaneous evaluations of the objective function $\mathcal{F}(x)$ on the available CPU's to speed up the optimization process.

The experimental results are very encouraging and validate the quality of the approach: CONDOR outperforms many commercial, high-end optimizer and it might be the fastest optimizer in its category (fastest in terms of number of function evaluations). When several CPU's are used, the performances of CONDOR are unmatched.

The experimental results open wide possibilities in the field of noisy and high-computing-load objective functions optimization (from two minutes to several days) like, for instance, industrial shape optimization based on CFD (computation fluid dynamic) codes (see [CAVDB01, PVdB98, Pol00, PMM$^+$03]) or PDE (partial differential equations) solvers.

Finally, we propose a new, original, easily comprehensible, free and fully stand-alone implementation in C/C++ of the optimization algorithm: the CONDOR optimizer. There is no call to fortran, external, unavailable, expensive, copyrighted libraries. You can compile the code under Unix, Windows, Solaris,etc. The only library needed is the standard TCP/IP network transmission library based on sockets (only in the case of the parallel version of the code).

The algorithms used inside CONDOR are part of the Gradient-Based optimization family. The algorithms implemented are Dennis-Moré Trust Region steps calculation (It's a restricted Newton's Step), Sequential Quadratic Programming (SQP), Quadratic Programming(QP), Second Order Corrections steps (SOC), Constrained Step length computation using L_1 merit function and Wolf condition, Active Set method for active constraints identification, BFGS update, Multivariate Lagrange Polynomial Interpolation, Cholesky factorization, QR factorization and more!

Many ideas implemented inside CONDOR are from Powell's UOBYQA (Unconstrained Optimization BY quadratical approximation) [Pow00] for unconstrained, direct optimization. The main contribution of Powell is Equation 6.6 which allows to construct a full quadratical model of the objective function in very few function evaluations (at a *low* price). This equation is very successful in that and having a full quadratical model allows us to reach unmatched convergence speed.

A full comprehension of all the aspects of the algorithm was, for me, one of the most important point. So I started from scratch and recoded everything (even the linear algebra tools). This allowed me to have a fully stand-alone implementation. Full comprehension and full re-implementation is needed to be able to extend the ideas to the parallel and constrained cases.

Acknowledgments - Remerciements

Je remercie en premier lieu mes parents, ma fiancée, Sabrina, et mon frère, David, qui m'ont soutenu durant toutes ces années. Sabrina, en particulier, pour avoir supporté mon humeur massacrante lorsque trop de difficultés s'amassaient sur mes épaules, après de longues heures de développement infructueux. Sans eux, rien n'aurait pu se faire.

Je remercie les membres du Jury de thèse pour toute l'attention qu'ils ont mise dans la lecture de mon travail.

Tous les chercheurs de IRIDIA méritent mes remerciements pour la bonne humeur et le cadre agréable de travail qu'ils m'ont fait partager. Je remercie plus particulièrement Hugues Bersini, le chef du laboratoire IRIDIA et coordinateur de mes travaux, qui m'a fait découvrir le monde de l'intelligence artificielle et les techniques d'optimisation liées à l'apprentissage des régulateurs flous (Fuzzy Controller). Ces régulateurs utilisaient une forme primitive de rétro-propagation du gradient inspirée des équations utilisées pour l'apprentissage des réseaux de neurones multicouches [BDBVB00]. Ce premier contact avec le monde de l'optimisation m'a donné l'envie de m'investir dans une thèse portant sur l'optimisation en général et je l'en remercie. Je remercie aussi tout spécialement Antoine Duchâteau avec lequel j'ai fait mes premiers pas en optimisation lors du dévellopement conjoint de régulateurs auto-adaptatifs flous. Je remercie aussi, pêle-mêle, Colin Molter, Pierre Philippe, Pierre Sener, Bruno Marchal, Robert Kennes, Mohamed Ben Haddou, Muriel Decreton, Mauro Birattari, Carlotta Piscopo, Utku Salihoglu, David Venet, Philippe Smets, Marco Saerens, Christophe Philemotte, Stefka Fidanova, et Christian Blum pour les nombreuses et sympatiques discussions que nous avons eues ensembles. Mes remerciement vont également à Marco Dorigo, Gianluca Bontempi, Nathanäel Ackerman, Eddy Bertolissi, Thomas Stuetzle, Joshua Knowles, Tom Lenaerts, Hussain Saleh, Michael Sampels, Roderich Groß, Thomas Halva Labella, Max Manfrin pour leur présence dynamique et stimulante.

Je tiens aussi à remercier Maxime Vanhoren, Dominique Lechat, Olivier Van Damme, Frederic Schoepps, Bernard Vonckx, Dominique Gilleman, Christophe Mortier, Jean-Pierre Norguet et tous les amis avec lesquels je passe régulièrement mon temps libre.

Les derniers remerciements sont adressés au Professeur Raymond Hanus pour l'aide précieuse qu'il m'a apportée durant ma thèse, au Professeur Philippe Van Ham pour sa pédagogie excellente, au Professeur Guy Gignez qui m'a donné goût aux sciences et à l'informatique et à Madame Lydia Chalkevitch pour son important soutien moral.

Contents

Chapter 1

Introduction

Abstract

We will present an algorithm which optimizes a non-linear function as follows:

$$y = \mathcal{F}(x),\ x \in \Re^n\ y \in \Re \qquad \text{Subject to: } \begin{cases} b_l \leq x \leq b_u, & b_l, b_u \in \Re^n \\ Ax \geq b, & A \in \Re^{m \times n}, b \in \Re^m \\ c_i(x) \geq 0, & i = 1, \ldots, l \end{cases} \tag{1.1}$$

This algorithm will, hopefully, find the value of x for which y is the lowest and for which all the constraints are respected. The dimension n of the search space must be lower than 50. We do NOT have to know the derivatives of $\mathcal{F}(x)$. We must only have a code which evaluates $\mathcal{F}(x)$ for a given value of x. The algorithm is particularly well suited when the code for the evaluation of $\mathcal{F}(x)$ is computationally demanding (e.g. demands more than 1 hour of processing). We also assume that the time needed for an evaluation of a non-linear constraint $c_i(x)$ is negligible. There can be a limited noise on the evaluation of $\mathcal{F}(x)$. Each component of the vector x must be a continuous real parameter of $\mathcal{F}(x)$.

The original contributions of this work are:

- The *free, stand-alone* implementation in C++ of the whole algorithm. Most optimizer are expensive piece of software. Usually "free" optimizers are using expensive, external, unavailable libraries (such libraries are usually the Harwell Libraries, the NAG tools, the NPSOL non-linear solver,...). In CONDOR, there is no call to expensive, external libraries.
- The algorithm used for the parallelization of the optimization procedure.
- The algorithm used for the constrained case.
- The assembly of different well-known algorithms in one code (In particular, the code of the Chapter 4 is a completely new implementation of the Moré and Sorensen algorithm which has never been used in conjunction with quadratical model build by interpolation).
- The bibliographical work needed to:
  - Understand and implement all the parts of the algorithm.

  - Acquire a huge amount of background knowledge in constrained and unconstrained continuous optimization. This is necessary to be able to review all the available techniques and choose the best promising one. However, I will not present here a full review of state-of-the-art techniques (for a starting point, see the excellent book "Trust-region Methods" by Andrew R. Conn, Nicholas I.M.Gould, and Philippe L.Toint [CGT00a]). A very short review for the constrained case is given in Chapter 8. In particular, this Section contains:
    * A short, concise and original introduction to Primal-Dual methods for non-linear optimization. The link between Primal-Dual methods and Barrier methods is motivated and intuitively explained.
    * A short, complete, intuitive and original description and justification of the Second Order Step (SOC) used inside SQP algorithms.

- The redaction of the complete description of the CONDOR algorithm ("*COnstrained, Non-linear, Direct, parallel Optimization using trust Region method for high-computing load function*"). This description is intended for graduate school level and requires no a priori knowledge in the field of continuous optimization. Most optimization books are very formal and give no insight to the reader. This thesis is written in a informal way, trying to give an intuition to the reader of what is going on, rather than giving pure formal demonstration which are usually hiding the intuition behind many formalism. The thesis is a complete start from scratch (or nearly) and is thus a good introduction to the optimization theory for the beginner.

- The comparison of the new algorithm CONDOR with other famous algorithms: CFSQP, DFO, UOBYQA, PDS,etc. The experimental results of this comparison shows very good performance of CONDOR compared to the other algorithms. On nearly all the unconstrained problems, CONDOR finds the optimal point of the objective function in substantially fewer evaluation of the objective function than its competitor, especially when more than one CPU is used. In the constrained case, the performances of CONDOR are comparable to the best algorithms. Preliminary results indicates that on box and linear constraints only, CONDOR also outperforms its competitors. However, more numerical results are needed to assert definitively this last statement.

- The application of a gradient-based optimizer on a objective function based on CFD code is unusual. This approach is usually rejected and is considered by many researcher as a "dead-end". The validation of the usefulness of the gradient-based approach is a primordial result.

- The algorithm to solve the secondary Trust-Region subproblem described in Section 5 is slightly different that the algorithm proposed by Powell. Numerical results exhibit better performances for the CONDOR algorithm.

The other ideas used inside CONDOR are mostly coming from recent work of M.J.D.Powell [Pow00].

1.1 Motivations

We find very often in the industry simulators of huge chemical reactors, simulators of huge turbo-compressors, simulators of the path of a satellite in low orbit around earth,... These simulators were written to allow the design engineer to correctly estimate the consequences of the adjustment of one (or many) design variables (or parameters of the problem). Such codes very often demands a great deal of computing power. One run of the simulator can take as much as one or two hours to finish. Some extreme simulations take a day to complete.

These kinds of code can be used to optimize "in batch" the design variables: The research engineer can aggregate the results of the simulation in one unique number which represents the "goodness" of the current design. This final number y can be seen as the result of the evaluation of an objective function $y = \mathcal{F}(x)$ where x is the vector of design variables and $\mathcal{F}$ is the simulator. We can run an optimization program which find x^*, the optimum of $\mathcal{F}(x)$.

Most optimization algorithms require the derivatives of $\mathcal{F}(x)$ to be available. Unfortunately, we usually don't have them. Very often, there is also some noises on $\mathcal{F}(x)$ due to rounding errors. To overcome these limitations, I present here a new optimizer called "CONDOR".

Here are the assumptions needed to use this new optimizer:

- The dimension n of the search space must be lower than 50. For larger dimension the time consumed by this algorithm will be so long and the number of function evaluations will be so huge that I don't advice you to use it.
- No derivatives of $\mathcal{F}(x)$ are required. However, the algorithm assumes that they exists. If the function is not continuous, the algorithm can still converge but in a greater time.
- The algorithm tries to minimize the number of evaluations of $\mathcal{F}(x)$, at the cost of a huge amount of routine work that occurs during the decision of the next value of x to try. Therefore, the algorithm is particularly well suited for high computing load objective function.
- The algorithm will only find a local minimum of $\mathcal{F}(x)$.
- There can be a limited noise on the evaluation of $\mathcal{F}(x)$.
- All the design variables must be continuous.
- The non-linear constraints are "cheap" to evaluate.

1.2 Formal description

This thesis about optimization of non-linear **continuous** functions subject to box, linear and non-linear constraints. We want to find $x^* \in \mathcal{R}^n$ which satisfies:

$$\mathcal{F}(x^*) = \min_x \mathcal{F}(x) \quad \text{Subject to:} \begin{cases} b_l \leq x \leq b_u, & b_l, b_u \in \Re^n \\ Ax \geq b, & A \in \Re^{m \times n}, b \in \Re^m \\ c_i(x) \geq 0, & i = 1, \ldots, l \end{cases} \tag{1.2}$$

$\mathcal{F}(x) : \Re^n \rightarrow \Re$ is called the objective function.
b_l and b_u are the box-constraints.
$Ax \geq b$ are the linear constraints.

$c_i(x)$ are the non-linear constraints.

The following notation will be used:

$g_i = \frac{\partial \mathcal{F}}{\partial x_i}$	g is the gradient of $\mathcal{F}$.
$H_{i,j} = \frac{\partial^2 \mathcal{F}}{\partial x_i \partial x_j}$	H is the Hessian matrix of $\mathcal{F}$.

The choice of the algorithm to solve an optimization problem mainly depends on:

- The dimension n of the search space.
- Whether or not the derivatives of $\mathcal{F}(x)$ are available.
- The time needed for one evaluation of $\mathcal{F}(x)$ for a given x.
- The necessity to find a global or a local minimum of $\mathcal{F}(x)$.
- The noise on the evaluation of $\mathcal{F}(x)$.
- Whether the Objective function is smooth or not.
- Whether the search space is continuous (there is no discrete variable like variable which can take the following values: red, green, blue.).
- The presence of (non-linear) constraints.

If there are lots of noise on $\mathcal{F}$, or if a global minima is needed, or if we have discrete variables we will use evolutionary algorithm (like genetic algorithms). These kind of algorithms can usually only have box constraints.

In the rest of the thesis, we will make the following assumptions:

- The objective function is smooth
- The non-linear constraints are "cheap" to evaluate
- We only want a local minimum.

An optimization (minimization) algorithm is nearly always based on this simple principle:

1. Build an approximation (also called "local model") of the objective function around the current point.
2. Find the minimum of this model and move the current point to this minimum. Go back to step 1.

Like most optimization algorithms, CONDOR uses, as local model, a polynomial of degree two. There are several ways of building this quadratic. CONDOR uses multivariate lagrange interpolation technique to build its model. This techniques is particularly well-suited when the dimension of the search space is low. When there is no noise on the objective function, we can use another, cheaper technique called "BFGS update" to construct the quadratic. It allows us to build local models at very low CPU cost (it's very fast).

We have made the assumption that an evaluation of the objective function is very expensive (in term of computing time). If this is not the case, we must construct a local model in a very short time. Indeed, it serves no point to construct a perfect local model using many computer resources to carefully choose the direction to explore. It's best to use an approximate, cheap to obtain, search direction. This will lead to a little more function evaluations but, since they are cheap, the overall process will be faster. An example of such algorithm is the "Rosenbrock's method". If the objective function is very cheap to evaluate, it's a good choice. You will find in the annexe (section 13.9) a personal implementation in C++ of this method. Algorithms based on the "BFGS update" are also able to construct a good model in a very short time. This time can still become not negligible when the dimension of the search space is high (greater than 1000). For higher dimension, the choice of the algorithm is not clear but, if an approximation of the Hessian matrix H of the objective function is directly available, a good choice will be a Conjugate-gradient/Lanczos method.

Currently, most of the researches in optimization algorithms are oriented to huge dimensional search-space. In these algorithms, we construct approximate search direction. CONDOR is one of the very few algorithm which adopts the opposite point of view. CONDOR build the most precise local model of the objective function and tries to reduce at all cost the number of function evaluations.

One of the goals of this thesis is to give a detailed explanation of the CONDOR algorithm. The thesis is structured as follows:

- **Unconstrained Optimization:** We will describe the algorithm in the case when there are no constraints. The parallelization of the algorithm will also be explained.
  - **Chapter 2:** A basic description of the CONDOR algorithm.
  - **Chapter 3:** How to construct the local model of the objective function? How to assess its validity?
  - **Chapter 4:** How to compute the minimum of the local model?
  - **Chapter 5:** When we want to check the validity (the precision) of our local model, we need to solve approximately $d_k = \min_{d \in \Re^n} |q(x_k + d)|$ subject to $\|d\|_2 < \rho$. How do we do?
  - **Chapter 6:** The precise description of the CONDOR algorithm.
  - **Chapter 7:** Numerical results of the CONDOR algorithm on unconstrained problems.
- **Constrained Optimization:**
  - **Chapter 8:** We will make a short review of algorithms available for constrained optimization and motivate the choice of our algorithm.
  - **Chapter 9:** Detailed discussion about the chosen and implemented algorithm.
  - **Chapter 10:** Numerical Results for constrained problems.
- **The METHOD project** (chapter 11)
  The goal of this project is to optimize the shape of the blades inside a Centrifugal Compressor (see illustration of the compressor's blades in Figure 11.1). This is a concrete, industrial example of use of CONDOR.

- **Conclusion** (chapter 12)
- **Annexes** (chapter 13)
- **Code** (chapter 14)

Part I

Unconstrained Optimization

Chapter 2

An introduction to the CONDOR algorithm.

The material of this chapter is based on the following references: [Fle87, PT95, BT96, Noc92, CGT99, DS96, CGT00a, Pow00].

2.1 Trust Region and Line-search Methods.

In continuous optimization, you have basically the choice between two kind of algorithm:

- Line-search methods
- Trust region methods.

In this section we will motivate the choice of a trust region algorithm for unconstrained optimization. We will see that trust region algorithms are a natural evolution of the Line-search algorithms.

2.1.1 Conventions

$\mathcal{F}(x) : \Re^n \rightarrow \Re$ is the objective function. We search for the minimum x^* of it.

x^*	The optimum. We search for it.
$g_i = \dfrac{\partial \mathcal{F}}{\partial x_i}$	g is the gradient of $\mathcal{F}$.
$H_{i,j} = \dfrac{\partial^2 \mathcal{F}}{\partial x_i \partial x_j}$	H is the Hessian matrix of $\mathcal{F}$.
$H_k = H(x_k)$	The Hessian Matrix of F at point x_k
$B_k = B(x_k)$	The current approximation of the Hessian Matrix of F at point x_k If not stated explicitly, we will always assume $B = H$.
$B^* = B(x^*)$	The Hessian Matrix at the optimum point.
$\mathcal{F}(x+\delta) \approx \mathcal{Q}(\delta) = \mathcal{F}(x) + g^t\delta + \frac{1}{2}\delta^t B \delta$	$\mathcal{Q}$ is the quadratical approximation of $\mathcal{F}$ around x.

All vectors are column vectors.

In the following section we will also use the following convention.

k	k is the iteration index of the algorithm.
s_k	s_k is the direction of research. Conceptually, it's only a direction not a length.
δ_k	$\delta_k = \alpha_k s_k = x_{k+1} - x_k$ is the step performed at iteration k.
α_k	α_k is the length of the step preformed at iteration k.
$\mathcal{F}_k$	$\mathcal{F}_k = \mathcal{F}(x_k)$
g_k	$g_k = g(x_k)$
h_k	$\|h_k\| = \|x_k - x^*\|$ the distance from the current point to the optimum.

2.1.2 General principle

The outline of a simple optimization algorithm is:

1. Search for a descent direction s_k around x_k (x_k is the current position).
2. In the direction s_k, search the length α_k of the step.
3. Make a step in this direction: $x_{k+1} = x_k + \delta_k$ (with $\delta_k = \alpha_k s_k$)
4. Increment k. Stop if $g_k \approx 0$ otherwise, go to step 1.

A simple algorithm based on this canvas is the "steepest descent algorithm":

$$s_k = -g_k \tag{2.1}$$

We choose $\alpha = 1 \Longrightarrow \delta_k = s_k = -g_k$ and therefore:

$$x_{k+1} = x_k - g_k \tag{2.2}$$

This is a very slow algorithm: it converges linearly (see next section about convergence speed).

2.1.3 Notion of Speed of convergence.

linear convergence	$\|x_{k+1} - x^*\| < \epsilon\ \|x_k - x^*\|$
superlinear convergence	$\|x_{k+1} - x^*\| < \epsilon_k \|x_k - x^*\|$ with $\epsilon_k \to 0$
quadratic convergence	$\|x_{k+1} - x^*\| < \epsilon\ \|x_k - x^*\|^2$

with $0 \leq \epsilon < 1$

These convergence criterias are sorted in function of their speed, the slowest first.

Reaching quadratic convergence speed is very difficult.

Superlinear convergence can also be written in the following way:

$$\lim_{k \to \infty} \frac{\|x_{k+1} - x^*\|}{\|x_k - x^*\|} = 0 \tag{2.3}$$

2.1.4 A simple line-search method: The Newton's method.

We will use $\alpha = 1 \Longrightarrow \delta_k = s_k$.

We will use the curvature information contained inside the Hessian matrix B of $\mathcal{F}$ to find the descent direction. Let us write the Taylor development limited to the degree 2 of $\mathcal{F}$ around x:

$$\mathcal{F}(x+\delta) \approx \mathcal{Q}(\delta) = \mathcal{F}(x) + g^t\delta + \frac{1}{2}\delta^t B\delta$$

The unconstrained minimum of $\mathcal{Q}(\delta)$ is:

$$\begin{aligned} \nabla \mathcal{Q}(\delta_k) = g_k + B_k\delta_k &= 0 \\ \Longleftrightarrow B_k\delta_k &= -g_k \end{aligned} \tag{2.4}$$

Equation 2.4 is called the equation of the *Newton's step* δ_k.

So, the Newton's method is:

1. solve $B_k\delta_k = -g_k$ (go to the minimum of the current quadratical approximation of $\mathcal{F}$).
2. set $x_{k+1} = x_k + \delta_k$
3. Increment k. Stop if $g_k \approx 0$ otherwise, go to step 1.

In more complex line-search methods, we will run a one-dimensional search ($n = 1$) in the direction $\delta_k = s_k$ to find a value of $\alpha_k > 0$ which "reduces sufficiently" the objective function. (see Section 13.1.2 for conditions on α: the Wolf conditions. A sufficient reduction is obtained if you respect these conditions.)

Near the optimum, we must always take $\alpha = 1$, to allow a "full step of one" to occur: see Chapter 2.1.6 for more information.

Newton's method has quadratic convergence: see Section 13.1.1 for proof.

2.1.5 B_k must be positive definite for Line-search methods.

In the Line-search methods, we want the search direction δ_k to be a descent direction.

$$\Longrightarrow \delta^T g < 0 \tag{2.5}$$

Taking the value of g from 2.4 and putting it in 2.5, we have:

$$\begin{aligned} -\delta^T B\delta &< 0 \\ \Leftrightarrow \quad \delta^T B\delta &> 0 \end{aligned} \tag{2.6}$$

The Equation 2.6 says that B_k must always be positive definite.

In line-search methods, we must always construct the B matrix so that it is positive definite.

One possibility is to take $B = I$ (I=identity matrix), which is a very bad approximation of the Hessian H but which is always positive definite. We retrieve the "steepest descent algorithm".

Another possibility, if we don't have B positive definite, is to use instead $B_{new} = B + \lambda I$ with λ being a very big number, such that B_{new} is positive definite. Then we solve, as usual, the Newton's step equation (see 2.4) $B_{new}\delta_k = -g_k$. Choosing a high value for λ has 2 effects:

1. B will become negligible and we will find, as search direction, "the steepest descent step".
2. The step size $\|\delta_k\|$ is reduced.

In fact, only the above second point is important. It can be proved that, if we impose a proper limitation on the step size $\|\delta_k\| < \Delta_k$, we maintain global convergence even if B is an indefinite matrix. Trust region algorithms are based on this principle. (Δ_k is called the trust region radius).

The old Levenberg-Marquardt algorithm uses a technique which adapts the value of λ during the optimization. If the iteration was successful, we decrease λ to exploit more the curvature information contained inside B. If the previous iteration was unsuccessful, the quadratic model don't fit properly the real function. We must then only use the "basic" gradient information. We will increase λ in order to follow closely the gradient (”steepest descent algorithm”).

For intermediate value of λ, we will thus follow a direction which is a mixture of the "steepest descent step" and the "Newton step". This direction is based on a perturbated Hessian matrix and can sometime be disastrous (There is no geometrical meaning of the perturbation λI on B).

This old algorithm is the base for the explanation of the update of the trust region radius in Trust Region Algorithms. However, in Trust Region Algorithms, the direction δ_k of the next point to evaluate is perfectly controlled.

To summarize:

- **Line search methods:**
  We search for the step δ_k with $\nabla \mathcal{Q}(\delta_k) = 0$ and we impose $B_k \geq 0$
- **Trust Region methods:**
  The step δ_k is the solution of the following constrained optimization problem:

  $$\mathcal{Q}(\delta_k) = \min_{\delta} \mathcal{Q}(\delta) \quad \text{subject to } \|\delta\| < \Delta_k$$

  B_k can be any matrix. We can have $\nabla \mathcal{Q}(\delta_k) \neq 0$.

2.1.6 Why is Newton's method crucial: Dennis-Moré theorem

The Dennis-Moré theroem is a very important theorem. It says that a non-linear optimization algorithm will converge superlinearly at the condition that, asymptotically, the steps made are equals to the Newton's steps ($A_k\delta_k = -g_k$ (A_k is not the Hessian matrix of $\mathcal{F}$; We must have: $A_k \rightarrow B_k$ (and $B_k \rightarrow H_k$)). This is a very general result, applicable also to constrained optimization, non-smooth optimization, ...

Dennis-moré characterization theorem:
The optimization algorithm converges superlinearly and $g(x^*) = 0$ iff $H(x)$ is "Lipchitz continuous" and the steps $\delta_k = x_{k+1} - x_k$ satisfies

$$A_k \delta_k = -g_k \tag{2.7}$$

where

$$\lim_{k \to \infty} \frac{\|(A_k - H^*)\delta_k\|}{\|\delta_k\|} = 0 \tag{2.8}$$

definition:
A function $g(x)$ is said to be Lipchitz continuous if there exists a constant γ such that:

$$\|g(x) - g(y)\| \leq \gamma \|x - y\| \tag{2.9}$$

To make the proof, we first see two **lemmas**.

Lemma 1.

We will prove that if H is Lipchitz continuous then we have:

$$\|g(v) - g(u) - H(x)(v - u)\| \leq \frac{\gamma}{2}\|v - u\|(\|v - x\| + \|u - x\|) \tag{2.10}$$

The well-known Riemann integral is:

$$\mathcal{F}(x + \delta) - \mathcal{F}(x) = \int_x^{x+\delta} g(z)dz$$

The extension to the multivariate case is straight forward:

$$g(x + \delta) - g(x) = \int_x^{x+\delta} H(z)dz$$

After a change in the integration variable: $z = x + \delta t \Rightarrow dz = \delta dt$, we obtain:

$$g(x + \delta) - g(x) = \int_0^1 H(x + t\delta)\delta dt$$

We substract on the left side and on the right side $H(x)\delta$:

$$g(x + \delta) - g(x) - H(x)\delta = \int_0^1 (H(x + t\delta) - H(x))\delta dt \tag{2.11}$$

Using the fact that $\| \int_0^1 H(t)dt\| \leq \int_0^1 \|H(t)\|dt$, and the cauchy-swartz inequality $\|a\ b\| < \|a\|\|b\|$ we can write:

$$\|g(x+\delta)-g(x)-H(x)\delta\| \quad \leq \quad \int_0^1 \|H(x+t\delta)-H(x)\|\|\delta\|dt$$

Using the fact that the Hessian H is Lipchitz continuous (see Equation 2.9):

$$\begin{aligned}\|g(x+\delta)-g(x)-H(x)\delta\| &\leq \int_0^1 \gamma\|t\delta\|\|\delta\|dt \\ &\leq \gamma\|\delta\|^2 \int_0^1 t\ dt \\ &\leq \frac{\gamma}{2}\|\delta\|^2 \qquad (2.12)\end{aligned}$$

The lemma 2 can be directly deduced from Equation 2.12.

Lemma 2.

If H is Lipchitz continuous, if H^{-1} exists, then there exists $\epsilon > 0, \alpha > 0$ such that:

$$\alpha\|v-u\| \leq \|g(v)-g(u)\| \qquad (2.13)$$

hold for all u, v which respect $\max(\|v-x\|, \|u-x\|) \leq \epsilon$.

If we write the triangle inequality $\|a+b\| < \|a\|+\|b\|$ with $a = g(v)-g(u)$ and $b = H(x)(v-u)+g(u)-g(v)$ we obtain:

$$\|g(v)-g(u)\| \geq \|H(x)(v-u)\| - \|g(v)-g(u)-H(x)(v-u)\|$$

Using the cauchy-swartz inequality $\|a\ b\| < \|a\|\|b\| \Leftrightarrow \|b\| > \dfrac{\|a\ b\|}{\|a\|}$ with $a = H^{-1}$ and $b = H(v-u)$, and using Equation 2.10:

$$\|g(v)-g(u)\| \geq \left[\frac{1}{\|H(x)^{-1}\|} - \frac{\gamma}{2}(\|v-x\|+\|u-x\|)\right]\|v-u\|$$

Using the hypothesis that $\max(\|v-x\|, \|u-x\|) \leq \epsilon$:

$$\|g(v)-g(u)\| \geq \left[\frac{1}{\|H(x)^{-1}\|} - \gamma\epsilon\right]\|v-u\|$$

Thus if $\epsilon < \dfrac{1}{\|H(x)^{-1}\|\gamma}$ the lemma is proven with $\alpha = \dfrac{1}{\|H(x)^{-1}\|} - \gamma\epsilon$.

Proof of Dennis-moré theorem.

We first write the "step" Equation 2.7:

$$\begin{aligned}
A_k\delta_k &= -g_k \\
0 &= A_k\delta_k + g_k \\
0 &= (A_k - H^*)\delta_k + g_k + H^*\delta_k \\
-g_{k+1} &= (A_k - H^*)\delta_k + [-g_{k+1} + g_k + H^*\delta_k] \\
\frac{\|g_{k+1}\|}{\|\delta_k\|} &\leq \frac{\|(A_k - H^*)\delta_k\|}{\|\delta_k\|} + \frac{\| - g_{k+1} + g_k + H^*\delta_k\|}{\|\delta_k\|}
\end{aligned}$$

Using lemma 2: Equation 2.10, and defining $e_k = x_k - x^*$, we obtain:

$$\frac{\|g_{k+1}\|}{\|\delta_k\|} \leq \frac{\|(A_k - H^*)\delta_k\|}{\|\delta_k\|} + \frac{\gamma}{2}(\|e_k\| + \|e_{k+1}\|)$$

Using $\lim_{k\to\infty} \|e_k\| = 0$ and Equation 2.8:

$$\lim_{k\to\infty} \frac{\|g_{k+1}\|}{\|\delta_k\|} = 0 \tag{2.14}$$

Using lemma 3: Equation 2.13, there exists $\alpha > 0, k_0 \geq 0$, such that, for all $k > k_0$, we have (using $g(x^*) = 0$):

$$\|g_{k+1}\| = \|g_{k+1} - g(x^*)\| \geq \alpha\|e_{k+1}\| \tag{2.15}$$

Combing Equation 2.14 and 2.15,

$$\begin{aligned}
0 &= \lim_{k\to\infty} \frac{\|g_{k+1}\|}{\|\delta_k\|} \geq \lim_{k\to\infty} \alpha\frac{\|e_{k+1}\|}{\|\delta_k\|} \\
&\geq \lim_{k\to\infty} \alpha\frac{\|e_{k+1}\|}{\|e_k\| + \|e_{k+1}\|} = \lim_{k\to\infty} \frac{\alpha r_k}{1 + r_k}
\end{aligned} \tag{2.16}$$

where we have defined $r_k = \frac{\|e_{k+1}\|}{\|e_k\|}$. This implies that:

$$\lim_{k\to\infty} r_k = 0 \tag{2.17}$$

which completes the proof of superlinear convergence.

Since $g(x)$ is Lipschitz continuous, it's easy to show that the Dennis-moré theorem remains true if Equation 2.8 is replaced by:

$$\lim_{k\to\infty} \frac{\|(A_k - H_k)\delta_k\|}{\|\delta_k\|} = \lim_{k\to\infty} \frac{\|(A_k - B_k)\delta_k\|}{\|\delta_k\|} = 0 \tag{2.18}$$

This means that, if $A_k \to B_k$, then we must have α_k (the length of the steps) $\to 1$ to have superlinear convergence. In other words, to have superlinar convergence, the "steps" of a secant method must converge in magnitude and direction to the Newton steps (see Equation 2.4) of the same points.

A step with $\alpha_k = 1$ is called a "full step of one". It's necessary to allow a "full step of one" to take place when we are near the optimum to have superlinear convergence.

The Wolf conditions (see Equations 13.4 and 13.5 in Section 13.1.2) always allow a "full step of one".

When we will deal with constraints, it's sometime not possible to have a "full step of one" because we "bump" into the frontier of the feasible space. In such cases, algorithms like FSQP will try to "bend" or "modify" slightly the search direction to allow a "full step of one" to occur.

This is also why the "trust region radius" Δ_k must be large near the optimum to allow a "full step of one" to occur.

2.2 A simple trust-region algorithm.

In all trust-region algorithms, we always choose $\alpha_k = 1$. The length of the steps will be adjusted using Δ, the trust region radius.

Recall that $\mathcal{Q}_k(\delta) = f(x_k) + < g_k, \delta > + \frac{1}{2} < \delta, B_k \delta >$ is the quadratical approximation of $\mathcal{F}(x)$ around x_k.

A simple trust-region algorithm is:

1. solve $B_k \delta_k = -g_k$ subject to $\|\delta_k\| < \Delta_k$.
2. Compute the "degree of agreement" r_k between $\mathcal{F}$ and $\mathcal{Q}$:
$$r_k = \frac{f(x_k) - f(x_k + \delta_k)}{\mathcal{Q}_k(0) - \mathcal{Q}_k(\delta_k)} \tag{2.19}$$
3. update x_k and Δ_k :

$r_k < 0.01$ (bad iteration)	$0.01 \leq r_k < 0.9$ (good iteration)	$0.9 \leq r_k$ (very good iteration)
$x_{k+1} = x_k$ $\Delta_{k+1} = \dfrac{\Delta_k}{2}$	$x_{k+1} = x_k + \delta_k$ $\Delta_{k+1} = \Delta_k$	$x_{k+1} = x_k + \delta_k$ $\Delta_{k+1} = 2\Delta_k$

(2.20)

4. Increment k. Stop if $g_k \approx 0$ otherwise, go to step 1.

The main idea in this update is: only increase Δ_k when the local approximator $\mathcal{Q}$ reflects well the real function $\mathcal{F}$.

At each iteration of the algorithm, we need to have B_k and g_k to compute δ_k.

There are different ways for obtaining g_k:

- Ask the user to provide a function which computes explicitly $g(x_k)$. The analytic form of the function to optimize should be known in order to be able to derivate it.

- Use an "Automatic differentiation tool" (like "ODYSSEE"). These tools take, as input, the (fortran) code of the objective function and generate, as output, the (fortran) code which computes the function AND the derivatives of the function. The generated code is called the "adjoint code". Usually this approach is very efficient in terms of CPU consumption.

  If the time needed for one evaluation of f is 1 hour, than the evaluation of $f(x_k)$ AND $g(x_k)$ using the adjoint code will take at most 3 hours (independently of the value of n: the dimension of the space). This result is very remarkable. One drawback is the memory consumption of such methods which is very high. For example, this limitation prevents to use such tools in domain of "Computational Fluid Dynamics" code.

- Compute the derivatives of $\mathcal{F}$ using forward finite differences:

$$\frac{\partial \mathcal{F}}{\partial x_i} = g_i = \frac{\mathcal{F}(\overline{x} + \epsilon_i \overline{e_i}) - \mathcal{F}(\overline{x})}{\epsilon_i} \qquad i = 1, \ldots, n \tag{2.21}$$

  If the time needed for one evaluation of f is 1 hour, then the evaluation of $f(x_k)$ AND $g(x_k)$ using this formulae will take $n+1$ hours. This is indeed very bad. One advantage, is, if we have $n+1$ CPU's available, we can distribute the computing load easily and obtain the results in 1 hour.

  One major drawback is that ϵ_i must be a very small number in order to approximate correctly the gradient. If there is a noise (even a small one) on the function evaluation, there is a high risk that $g(x)$ will be completely un-useful.

- Extract the derivatives from a (quadratic) polynomial which interpolates the function at points close to x_k. This approach has been chosen for CONDOR.

  When there is noise on the objective function, we must choose the interpolation sites very carefully. If we take points too close from each other, we will get a poor model: it's destroyed by the noise. If we take points very far from each other, we don't have enough information about the local shape of the objective function to guide correctly the search.

  Beside, we need $N = (n+1)(n+2)/2$ points to build a quadratic polynomial. We cannot compute N points at each iteration of the algorithm. We will see in chapter 3 how to cope with all these difficulties.

There are different ways for obtaining B_k. Many are unpractical. Here are some reasonable ones:

- Use a "BFGS-update". This update scheme uses the gradient computed at each iteration to progressively construct the Hessian matrix H of f. Initially, we set $B_0 = I$ (the identity matrix). If the objective function is quadratic, we will have after n update, $B_n = H$ exactly (Since f is a quadratic polynomial, H is constant over the whole space). If the objective function is not a quadratical polynomial, $B(x_n)$ is constructed using $g(x_0), g(x_1), \ldots, g(x_{n-1})$ and is thus a mixture of $H(x_0), H(x_1), \ldots, H(x_{n-1})$. This can lead to poor approximation of $H(x_n)$, especially if the curvature is changing fast.

  Another drawback is that B_k will always be positive definite. This is very useful if we are using Line-search techniques but is not appropriate in the case of trust-region method. In fact, $\mathcal{Q}_k(\delta) = f(x_k) + < g_k, \delta > + \frac{1}{2} < \delta, B_k \delta >$ can be a very poor approximation of the real shape of the objective function if, locally, H_k is indefinite or is negative definite. This can lead to a poor search direction δ_k.

If there is noise on the objective function and if we are using a finite difference approximation of the gradient, we will get a poor estimate of the gradient of the objective function. Since B_k is constructed using only the gradients, it will also be a very poor estimate of H_k.

- Extract B_k from a (quadratic) polynomial which interpolates the function at points close to x_k. This approach has been chosen in CONDOR.

  The point are chosen close to x_k. B_k is thus never perturbed by old, far evaluations.

  The points are "far from each others" to be less sensitive to the noise on the objective function.

  B_k can also be positive, negative definite or indefinite. It reflects exactly the actual shape of f.

2.3 The basic trust-region algorithm (BTR).

Defnition: The trust region $\mathcal{B}_k$ is the set of all points such that

$$\mathcal{B}_k = \{x \in \Re^n | \|x - x_k\|_k \leq \Delta_k\} \tag{2.22}$$

The simple algorithm described in the Section 2.2 can be generalized as follows:

1. **Initialization** An initial point x_0 and an initial trust region radius Δ_0 are given. The constants η_1, η_2, γ_1 and γ_2 are also given and satisfy:

   $$0 < \eta_1 \leq \eta_2 < 1 \text{ and } 0 < \gamma_1 \leq \gamma_2 < 1 \tag{2.23}$$

   Compute $f(x_0)$ and set $k = 0$

2. **Model definition** Choose the norm $\|\cdot\|_k$ and define a model m_k in $\mathcal{B}_k$

3. **Step computation** Compute a step s_k that "sufficiently reduces the model" m_k and such that $x_k + s_k \in \mathcal{B}_k$

4. **Acceptance of the trial point.** Compute $f(x_k + s_k)$ and define:

   $$r_k = \frac{f(x_k) - f(x_k + s_k)}{m_k(x_k) - m_k(x_k + s_k)} \tag{2.24}$$

   If $r_k \geq \eta_1$, then define $x_{k+1} = x_k + s_k$; otherwise define $x_{k+1} = x_k$.

5. Trust region radius update. Set

$$\Delta_{k+1} \in \begin{cases} [\Delta_k, \infty) & \text{if } r_k \geq \eta_2, \\ [\gamma_2 \Delta_k, \Delta_k) & \text{if } r_k \in [\eta_1, \eta_2), \\ [\gamma_1 \Delta_k, \gamma_2 \Delta_k) & \text{if } r_k < \eta_1. \end{cases} \tag{2.25}$$

   Increment k by 1 and go to step 2.

Under some very weak assumptions, it can be proven that this algorithm is globally convergent to a local optimum [CGT00a]. The proof will be skipped.

2.4 About the CONDOR algorithm.

To start the unconstrained version of the CONDOR algorithm, we will basically need:

- A starting point x_{start}
- A length ρ_{start} which represents, basically, the initial distance between the points where the objective function will be sampled.
- A length ρ_{end} which represents, basically, the final distance between the interpolation points when the algorithm stops.

DEFINITION: The local approximation $q_k(s)$ of $f(x)$ is *valid* in $\mathcal{B}_k(\rho)$ (a ball of radius ρ around x_k) when $|f(x_k + s) - q_k(s)| \leq \kappa \rho^2 \quad \forall \|s\| \leq \rho$ where κ is a given constant independent of x.

We will approximatively use the following algorithm (for a complete description, see chapter 6):

1. Create an interpolation polynomial $q_0(s)$ of degree 2 which intercepts the objective function around x_{start}. All the points in the interpolation set $\mathcal{Y}$ (used to build $q(x)$) are separated by a distance of approximatively ρ_{start}. Set $x_k =$ *the best point of the objective function known so far.* Set $\rho_0 = \rho_{start}$. In the following algorithm, $q_k(s)$ is the quadratical approximation of $f(x)$ around x_k, built by interpolation using $\mathcal{Y}$ (see chapter 3). $q_k(s) = f(x_k) + g_k^t s + s^t H_k s$ where g_k is the approximate gradient of $f(x)$ evaluated at x_k and H_k is the approximate Hessian matrix of $f(x)$ evaluated at x_k.
2. Set $\Delta_k = \rho_k$
3. Inner loop: solve the problem for a given precision of ρ_k.
   (a) i. Solve $s_k = \min_{s \in \Re^n} q_k(s)$ subject to $\|s\|_2 < \Delta_k$.
   ii. If $\|s_k\| < \frac{1}{2}\rho_k$, then break and go to step 3(b) because, in order to do such a small step, we need to be sure that the model is valid.
   iii. Evaluate the function $f(x)$ at the new position $x_k + s$. Update the trust region radius Δ_k and the current best point x_k using classical trust region technique (following a scheme similar to Equation 2.20). Include the new x_k inside the interpolation set $\mathcal{Y}$. Update $q_k(s)$ to interpolate on the new $\mathcal{Y}$.

   iv. If some progress has been achieved (for example, $\|s_k\| > 2\rho$ or there was a reduction $f(x_{k+1}) < f(x_k)$), increment k and return to step **i**, otherwise continue.

(b) Test the validity of $q_k(x)$ in $\mathcal{B}_k(\rho)$, like described in chapter 3.

- **Model is invalid:**
  Improve the quality of the model $q_k(s)$: Remove the worst point of the interpolation set $\mathcal{Y}$ and replace it (one evaluation required!) with a new point x_{new} such that: $\|x_{new} - x_k\| < \rho$ and the precision of $q_k(s)$ is substantially increased.
- **Model is valid:**
  If $\|s_k\| > \rho_k$ go back to step 3(a), otherwise continue.

4. **Reduce** ρ since the optimization steps are becoming very small, the accuracy needs to be raised.

5. If $\rho = \rho_{end}$ stop, otherwise increment k and go back to step **2**.

From this description, we can say that ρ and Δ_k are two trust region radius (global and local).

Basically, ρ is the distance (Euclidian distance) which separates the points where the function is sampled. When the iterations are unsuccessful, the trust region radius Δ_k decreases, preventing the algorithm to achieve more progress. At this point, loop 3(a)i to 3(a)iv is exited and a function evaluation is required to increase the quality of the model (step 3(b)). When the algorithm comes close to an optimum, the step size becomes small. Thus, the inner loop (steps 3(a)i. to 3(a)iv.) is usually exited from step 3(a)ii, allowing to skip step 3(b) (hoping the model is *valid*), and directly reducing ρ in step 4.

The most inner loop (steps 3(a)i. to 3(a)iv.) tries to get from $q_k(s)$ good search directions without doing any evaluation to maintain the quality of $q_k(s)$ (The evaluations that are performed on step 3(a)i) have another goal). Only inside step 3(b), evaluations are performed to increase this quality (called a "model step") and only at the condition that the model has been proven to be invalid (to spare evaluations!).

Notice the update mechanism of ρ in step 4. This update occurs only when the model has been validated in the trust region $\mathcal{B}_k(\rho)$ (when the loop 3(a) to 3(b) is exited). The function cannot be sampled at point too close to the current point x_k without being assured that the model is *valid* in $\mathcal{B}_k(\rho)$.

The different evaluations of $f(x)$ are used to:

(a) guide the search to the minimum of $f(x)$ (see inner loop in the steps 3(a)i. to 3(a)iv.). To guide the search, the information gathered until now and available in $q_k(s)$ is *exploited.*

(b) increase the quality of the approximator $q_k(x)$ (see step 3(b)). To avoid the degeneration of $q_k(s)$, the search space needs to be additionally *explored.*

(a) and (b) are antagonist objectives like usually the case in the *exploitation/exploration* paradigm. The main idea of the parallelization of the algorithm is to perform the *exploration* on distributed CPU's. Consequently, the algorithm will have better models $q_k(s)$ of $f(x)$ at disposal and choose better search direction, leading to a faster convergence.

CONDOR falls inside the class of algorithm which are proven to be globally convergent to a local (maybe global) optimum [CST97, CGT00b].

In the next chapters, we will see more precisely:

- **Chapter 3:** How to construct and use $q(x)$? Inside the CONDOR algorithm we need a polynomial approximation of the objective function. How do we build it? How do we use and validate it?
- **Chapter 4:** How to solve $\delta_k = \min_{\delta \in \Re^n} q(x_k + \delta)$ subject to $\|\delta\|_2 < \Delta_k$? We need to know how to solve this problem because we encounter it at step (3)(a)i. of the CONDOR algorithm.
- **Chapter 5:** How to solve approximately $d_k = \min_{d \in \Re^n} |q(x_k + d)|$ subject to $\|d\|_2 < \rho$? We need to know how to solve this problem because we encounter it when we want to check the validity (the precision) of the polynomial approximation.
- **Chapter 6:** The precise description of the CONDOR unconstrained algorithm.

Chapter 3

Multivariate Lagrange Interpolation

The material of this chapter is based on the following references: [DBAR98, DB98, DBAR90, SX95, Sau95, SP99, Lor00, PTVF99, RP63, Pow02].

3.1 Introduction

One way to generate the local approximation $\mathcal{Q}_k(\delta) = f(x_k) + < g_k, \delta > + \frac{1}{2} < \delta, B_k \delta >$ of the objective function $\mathcal{F}(x), \quad x \in \Re^n$ around x_k is to make Multivariate Lagrange Interpolation.

We will sample $\mathcal{F}$ at different points and construct a quadratic polynomial which interpolates these samples.

The position and the number N of the points are not random. For example, if we try to construct a polynomial $L : \Re^2 - > \Re : z = c_1 + c_2 x + c_3 y$ of degree 1 (a plane), which interpolates locally a function $F : \Re^2 \rightarrow \Re$, we need **exactly** 3 points A, B and C. Why do we need **exactly** 3 points (apart from the fact that 3 points in 3D determines a plane)? Because we need to solve for c_1, c_2, c_3 the following linear system:

$$\begin{pmatrix} 1 & A_x & A_y \\ 1 & B_x & B_y \\ 1 & C_x & C_y \end{pmatrix} \begin{pmatrix} c_1 \\ c_2 \\ c_3 \end{pmatrix} = \begin{pmatrix} f(A) \\ f(B) \\ f(C) \end{pmatrix} \tag{3.1}$$

The matrix above is called the “Vandermonde Matrix”.

We can say even more: What happens if these three points are on the same line? There is a simple infinity of planes which passes through three aligned points. The determinant of the Vandermonde Matrix (called here after the “Vandermonde determinant”) will be null. The interpolation problem is not solvable. We will say that ”the problem is NOT poised”.

In opposition to the univariate polynomial interpolation (where we can take a random number of point, at random different places), the multivariate polynomial interpolation imposes a precise number of interpolation points at precise places.

In fact, if we want to interpolate by a polynomial of degree d a function $F : \Re^n \rightarrow \Re$, we will need $N = r_n^d = C_n^{d+n}$ points (with $C_k^n = \dfrac{n!}{k!(n-k)!}$). If the Vandermonde determinant is not null for this set of points, the problem is "well poised".

Example: If we want to construct a polynomial $\mathcal{Q} : \Re^2 -> \Re : z = c_1 + c_2 x + c_3 y + c_4 x^2 + c_5 xy + c_6 y^2$ of degree 2, which interpolates locally a function $F : \Re^2 \rightarrow \Re$, at points $\{A, B, C, D, E, F\}$we will have the following Vandermonde system:

$$\begin{pmatrix} 1 & A_x & A_y & A_x^2 & A_x A_y & A_y^2 \\ 1 & B_x & B_y & B_x^2 & B_x B_y & B_y^2 \\ 1 & C_x & C_y & C_x^2 & C_x C_y & C_y^2 \\ 1 & D_x & D_y & D_x^2 & D_x D_y & D_y^2 \\ 1 & E_x & E_y & E_x^2 & E_x E_y & E_y^2 \\ 1 & F_x & F_y & F_x^2 & F_x F_y & F_y^2 \end{pmatrix} \begin{pmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \\ c_5 \\ c_6 \end{pmatrix} = \begin{pmatrix} f(A) \\ f(B) \\ f(C) \\ f(D) \\ f(E) \\ f(F) \end{pmatrix}$$

Beware! Never try to resolve directly Vandermonde systems. These kind of systems are very often badly conditioned (determinant near zero) and can't be resolved directly.

If we already have a polynomial of degree d and want to use information contained in new points, we will need a block of exactly C_{n-1}^{d+n-1} new points. The new interpolating polynomial will have a degree of $d + 1$. This is called **"interpolation in block"**.

3.2 A small reminder about univariate interpolation.

We want to interpolate a simple curve $y = f(x), x \in \Re$ in the plane (in $\Re^2$). We have a set of N interpolation points $(\boldsymbol{x}_{(i)}, f(\boldsymbol{x}_{(i)})), \quad i = 1, \ldots, N \quad \boldsymbol{x}_{(i)} \in \Re$ on the curve. We can choose N as we want. We must have $\boldsymbol{x}_{(i)} \neq \boldsymbol{x}_{(j)}$ if $i \neq j$.

3.2.1 Lagrange interpolation

We define a Lagrange polynomial $P_i(x)$ as

$$P_i(x) := \prod_{\substack{j=1 \\ j \neq i}}^{N} \frac{x - \boldsymbol{x}_{(j)}}{\boldsymbol{x}_{(i)} - \boldsymbol{x}_{(j)}} \tag{3.2}$$

We will have the following property: $P_i(x_j) = \delta_{(i,j)}$ where $\delta_{(i,j)}$ is the Kronecker delta:

$$\delta_{(i,j)} = \begin{cases} 0, & i \neq j; \\ 1, & i = j. \end{cases} \tag{3.3}$$

Then, the interpolating polynomial $L(x)$ is:

$$L(x) = \sum_{i=1}^{N} f(\boldsymbol{x}_{(i)})\ P_i(x) \tag{3.4}$$

This way to construct an interpolating polynomial is not very effective because:

- The Lagrange polynomials $P_i(x)$ are all of degree N and thus require lots of computing time for creation, evaluation and addition (during the computation of $L(x)$)...
- We must know in advance all the N points. An iterative procedure would be better.

The solution to these problems : The Newton interpolation.

3.2.2 Newton interpolation

The Newton algorithm is iterative. We use the polynomial $P_k(x)$ of degree $k-1$ which already interpolates k points and transform it into a polynomial P_{k+1} of degree k which interpolates $k+1$ points of the function $f(x)$. We have:

$$P_{k+1}(x) = P_k(x) + \quad (x - \boldsymbol{x}_{(1)}) \cdots (x - \boldsymbol{x}_{(k)}) \quad [\boldsymbol{x}_{(1)}, \ldots, \boldsymbol{x}_{(k+1)}]f \tag{3.5}$$

The term $(x - \boldsymbol{x}_{(1)}) \cdots (x - \boldsymbol{x}_{(k)})$ assures that the second term of P_{k+1} will vanish at all the points $\boldsymbol{x}_{(i)} \quad i = 1, \ldots, k$.

Definition: $[\boldsymbol{x}_{(1)}, \ldots, \boldsymbol{x}_{(k+1)}]f$ is called a "divided difference". It's the unique leading co-efficient (that is the coefficient of x^k) of the polynomial of degree k that agree with f at the sequence $\{\boldsymbol{x}_{(1)}, \ldots, \boldsymbol{x}_{(k+1)}\}$.

The final Newton interpolating polynomial is thus:

$$P(x) = P_N(x) = \sum_{k=1}^{N} \quad (x - \boldsymbol{x}_{(1)}) \cdots (x - \boldsymbol{x}_{(k-1)}) \quad [\boldsymbol{x}_{(1)}, \ldots, \boldsymbol{x}_{(k)}]f \tag{3.6}$$

The final interpolation polynomial is the sum of polynomials of degree varying from 0 to $N-1$ (see Equation 3.5). The manipulation of the Newton polynomials (of Equation 3.5) is faster than the Lagrange polynomials (of Equation 3.2) and thus, is more efficient in term of computing time. Unfortunately, with Newton polynomials, we don't have the nice property that $P_i(x_j) = \delta_{(i,j)}$.

We can already write two basic properties of the divided difference:

$$[\boldsymbol{x}_{(k)}]f = f(\boldsymbol{x}_{(k)}) \tag{3.7}$$

$$[\boldsymbol{x}_{(1)}, \boldsymbol{x}_{(2)}]f = \frac{f(\boldsymbol{x}_{(2)}) - f(\boldsymbol{x}_{(1)})}{\boldsymbol{x}_{(2)} - \boldsymbol{x}_{(1)}} \tag{3.8}$$

The error between $f(x)$ and $P_N(x)$ is:

$$f(x) - P_N(x) = (x - \boldsymbol{x}_{(1)}) \cdots (x - \boldsymbol{x}_{(N)}) \quad [\boldsymbol{x}_{(1)}, \dots, \boldsymbol{x}_{(N)}, x]f \tag{3.9}$$

3.2.3 The divided difference for the Newton form.

Lemma 1

We will proof this by induction. First, we rewrite Equation 3.9, for N=1, using 3.7:

$$f(x) = f(\boldsymbol{x}_{(1)}) + (x - \boldsymbol{x}_{(1)})[\boldsymbol{x}_{(1)}, x]f \tag{3.10}$$

Using Equation 3.8 inside 3.10, we obtain:

$$f(x) = f(\boldsymbol{x}_{(1)}) + (x - \boldsymbol{x}_{(1)})\frac{f(x) - f(\boldsymbol{x}_{(1)})}{x - \boldsymbol{x}_{(1)}} \tag{3.11}$$

This equation is readily verified. The case for $N = 1$ is solved. Suppose the Equation 3.9 verified for $N = k$ and proof that it will also be true for $N = k + 1$. First, let us rewrite equation 3.10, replacing $\boldsymbol{x}_{(1)}$ by $\boldsymbol{x}_{(k+1)}$ and replacing $f(x)$ by $[\boldsymbol{x}_{(1)}, \dots, \boldsymbol{x}_{(k)}, x]f$ *as a function of x.* (In other word, we will interpolate the function $f(x) \equiv [\boldsymbol{x}_{(1)}, \dots, \boldsymbol{x}_{(k)}, x]f$ at the point $\boldsymbol{x}_{(k+1)}$ using 3.10.) We obtain:

$$[\boldsymbol{x}_{(1)}, \dots, \boldsymbol{x}_{(k)}, x]f = [\boldsymbol{x}_{(1)}, \dots, \boldsymbol{x}_{(k+1)}]f + (x - \boldsymbol{x}_{(k+1)})[\boldsymbol{x}_{(k+1)}, x]\Big([\boldsymbol{x}_{(1)}, \dots, \boldsymbol{x}_{(k)}, \cdot]f\Big) \tag{3.12}$$

Let us rewrite Equation 3.9 with $N = k$.

$$f(x) = P_k(x) + (x - \boldsymbol{x}_{(1)}) \cdots (x - \boldsymbol{x}_{(k)}) \quad [x_1, \dots, x_k, x]f \tag{3.13}$$

Using Equation 3.12 inside Equation 3.13:

$$f(x) = P_{k+1}(x) + (x - \boldsymbol{x}_{(1)}) \cdots (x - \boldsymbol{x}_{(k+1)})[\boldsymbol{x}_{(k+1)}, x]\Big([\boldsymbol{x}_{(1)}, \dots, \boldsymbol{x}_{(k)}, \cdot]f\Big) \tag{3.14}$$

Let us rewrite Equation 3.5 changing index $k + 1$ to $k + 2$.

$$P_{k+2}(x) = P_{k+1}(x) + \quad (x - \boldsymbol{x}_{(1)}) \cdots (x - \boldsymbol{x}_{(k+1)}) \quad [\boldsymbol{x}_{(1)}, \dots, \boldsymbol{x}_{(k+2)}]f \tag{3.15}$$

Recalling the definition of the divided difference: $[x_1, \dots, x_{k+2}]f$ is the unique leading coefficient of the polynomial of degree $k + 1$ that agree with f at the sequence $\{\boldsymbol{x}_{(1)}, \dots, \boldsymbol{x}_{(k+2)}\}$. Because of the uniqueness and comparing equation 3.14 and 3.15, we see that (replacing x by $\boldsymbol{x}_{(k+2)}$ in 3.14):

$$[\boldsymbol{x}_{(k+1)}, \boldsymbol{x}_{(k+2)}]\Big([\boldsymbol{x}_{(1)}, \dots, \boldsymbol{x}_{(k)}, \cdot]f\Big) = [\boldsymbol{x}_{(1)}, \dots, \boldsymbol{x}_{(k+2)}]f \tag{3.16}$$

Using Equation 3.16 inside 3.14:

$$f(x) = P_{k+1}(x) + (x - \boldsymbol{x}_{(1)}) \cdots (x - \boldsymbol{x}_{(k+1)})[\boldsymbol{x}_{(1)}, \dots, \boldsymbol{x}_{(k+2)}]f \tag{3.17}$$

Recalling the discussion of the paragraph after Equation 3.11, we can say then this last equation complete the proof for $N = k + 1$. The lemma 1 is now proved.

Lemma 2
This is clear from the definition of $[\boldsymbol{x}_{(1)}, \dots, \boldsymbol{x}_{(k)}]f$.

$[\boldsymbol{x}_{(1)}, \dots, \boldsymbol{x}_{(k)}]f$ is a symmetric function of its argument $\boldsymbol{x}_{(1)}, \dots, \boldsymbol{x}_{(k)}$, that is, it depends only on the numbers $\boldsymbol{x}_{(1)}, \dots, \boldsymbol{x}_{(k)}$ and not on the order in which they occur in the argument list.

Lemma 3

A useful formula to compute finite differences.

$$[\boldsymbol{x}_{(i)}, \dots, \boldsymbol{x}_{(i+k)}]f = \frac{[\boldsymbol{x}_{(i+1)}, \dots, \boldsymbol{x}_{(i+k)}]f - [\boldsymbol{x}_{(i)}, \dots, \boldsymbol{x}_{(i+k-1)}]f}{\boldsymbol{x}_{(i+k)} - \boldsymbol{x}_{(i)}} \tag{3.18}$$

Combining Equation 3.16 and Equation 3.8, we obtain:

$$\frac{[\boldsymbol{x}_{(1)}, \dots, \boldsymbol{x}_{(k)}, \boldsymbol{x}_{(k+2)}]f - [\boldsymbol{x}_{(1)}, \dots, \boldsymbol{x}_{(k)}, \boldsymbol{x}_{(k+1)}]f}{\boldsymbol{x}_{(k+2)} - \boldsymbol{x}_{(k+1)}} = [\boldsymbol{x}_{(1)}, \dots, \boldsymbol{x}_{(k+2)}]f \tag{3.19}$$

Using Equation 3.19 and lemma 2, we obtain directly 3.18. The lemma is proved.

Equation 3.18 has suggested the name "divided difference".

interp. site	value	first div. diff.	second div. diff.	...	$(N-2)^{\text{nd}}$ divided diff.	$(N-1)^{\text{st}}$ divided diff.
$\boldsymbol{x}_{(1)}$	$f(\boldsymbol{x}_{(1)})$					
		$[\boldsymbol{x}_{(1)}, \boldsymbol{x}_{(2)}]f$				
$\boldsymbol{x}_{(2)}$	$f(\boldsymbol{x}_{(2)})$		$[\boldsymbol{x}_{(1)}, \boldsymbol{x}_{(2)}, \boldsymbol{x}_{(3)}]f$			
		$[\boldsymbol{x}_{(2)}, \boldsymbol{x}_{(3)}]f$				
$\boldsymbol{x}_{(3)}$	$f(\boldsymbol{x}_{(3)})$		$[\boldsymbol{x}_{(2)}, \boldsymbol{x}_{(3)}, \boldsymbol{x}_{(4)}]f$	·	$[\boldsymbol{x}_{(1)}, \dots \boldsymbol{x}_{(N-1)}]f$	
		$[\boldsymbol{x}_{(3)}, \boldsymbol{x}_{(4)}]f$				$[\boldsymbol{x}_{(1)}, \dots \boldsymbol{x}_{(N)}]f$
$\boldsymbol{x}_{(4)}$			·	·	$[\boldsymbol{x}_{(2)}, \dots \boldsymbol{x}_{(N)}]f$	
⋮	⋮	·				
$\boldsymbol{x}_{(N-1)}$	$f(\boldsymbol{x}_{(N-1)})$		$[\boldsymbol{x}_{(N-2)}, \boldsymbol{x}_{(N-1)}, \boldsymbol{x}_{(N)}]f$			
		$[\boldsymbol{x}_{(N-1)}, \boldsymbol{x}_{(N)}]f$				
$\boldsymbol{x}_{(N)}$	$f(\boldsymbol{x}_{(N)})$					

We can generate the entries of the divided difference table *column by column* from the given dataset using Equation 3.18. The top diagonal then contains the desired coefficients $[\boldsymbol{x}_{(1)}]f$, $[\boldsymbol{x}_{(1)}, \boldsymbol{x}_{(2)}]f, [\boldsymbol{x}_{(1)}, \boldsymbol{x}_{(2)}, \boldsymbol{x}_{(3)}]f, \dots, [\boldsymbol{x}_{(1)}, \dots, \boldsymbol{x}_{(N)}]f$ of the final Newton form of Equation 3.6.

3.2.4 The Horner scheme

Suppose we want to evaluate the following polynomial:

$$P(x) = c_0 + c_1 x + c_2 x^2 + c_3 x^3 + c_4 x^4 \tag{3.20}$$

We will certainly NOT use the following algorithm:

1. **Initialisation** $r = c_0$

```
2. For $k = 2, \dots, N$
    (a) Set $r := r + c_k x^k$
3. Return $r$
```

This algorithm is slow (lots of multiplications in x^k) and leads to poor precision in the result (due to rounding errors).

The Horner scheme uses the following representation of the polynomial 3.20: $P(x) = c_0 + x(c_1 + x(c_2 + x(c_3 + xc_4)))$, to construct a very efficient evaluation algorithm:

```
1. Initialisation $r = c_N$
2. For $k = N - 1, \dots, 0$
    (a) Set $r := c_k + x\ r$
3. Return $r$
```

There is only $N - 1$ multiplication in this algorithm. It's thus very fast and accurate.

3.3 Multivariate Lagrange interpolation.

We want to interpolate an hypersurface $y = f(x), x \in \Re^n$ in the dimension $n + 1$. We have a set of interpolation points $(\boldsymbol{x}_{(i)}, f(\boldsymbol{x}_{(i)})), \quad i = 1, \dots, N \quad \boldsymbol{x}_{(i)} \in \Re^n$ on the surface. We must be sure that the problem is well poised: The number N of points is $N = r_n^d$ and the Vandermonde determinant is not null.

3.3.1 The Lagrange polynomial basis $\{P_1(x), \dots, P_N(x)\}$.

We will construct our polynomial basis $P_i(x) \quad i = 1, \dots, N$ iteratively. Assuming that we already have a polynomial $\mathcal{Q}_k = \sum_{i=1}^k f(\boldsymbol{x}_{(i)})P_i(x)$ interpolating k points, we will add to it a new polynomial P_{k+1} which doesn't destruct all what we have already done. That is the value of $P_{k+1}(x)$ must be zero for $x = \boldsymbol{x}_{(1)}, \boldsymbol{x}_{(2)}, \dots, \boldsymbol{x}_{(k)}$. In other words, $P_i(\boldsymbol{x}_{(j)}) = \delta_{(i,j)}$. This is easily done in the univariate case: $P_k(x) = (x - \boldsymbol{x}_{(1)}) \dots (x - \boldsymbol{x}_{(k)})$, but in the multivariate case, it becomes difficult.

We must find a new polynomial P_{k+1} which is somewhat "perpendicular" to the $P_i \quad i = 1, \dots, k$ with respect to the k points $\boldsymbol{x}_{(1)}, \dots, \boldsymbol{x}_{(k)}$. Any multiple of P_{k+1} added to the previous P_i must leave the value of this P_i unchanged at the k points $\boldsymbol{x}_{(1)}, \dots, \boldsymbol{x}_{(k)}$. We can see the polynomials P_i as "vectors", we search for a new "vector" P_{k+1} which is "perpendicular" to all the P_i. We will use a version of the Gram-Schmidt othogonalisation procedure adapted for the polynomials. The original Gram-Schmidt procedure for vectors is described in the annexes in Section 13.2.

We define the scalar product with respect to the dataset $\mathcal{K}$ of points $\{\boldsymbol{x}_{(1)}, \dots, \boldsymbol{x}_{(k)}\}$ between the two polynomials P and Q to be:

$$< P, Q >_{\mathcal{K}} = \sum_{j=1}^{k} P(\boldsymbol{x}_{(j)})Q(\boldsymbol{x}_{(j)}) \tag{3.21}$$

We have a set of independent polynomials $\{P_{1\_old}, P_{2\_old}, \ldots, P_{N\_old}\}$. We want to convert this set into a set of orthonormal vectors $\{P_1, P_2, \ldots, P_N\}$ with respect to the dataset of points $\{\boldsymbol{x}_{(1)}, \ldots, \boldsymbol{x}_{(N)}\}$ by the Gram-Schmidt process:

1. **Initialization** k=1;
2. **Normalisation**

   $$P_k(x) = \frac{P_{k\_old}(x)}{|P_{k\_old}(\boldsymbol{x}_{(k)})|} \tag{3.22}$$

3. **Orthogonalisation**
   For $j = 1$ to N , $j \neq k$ do:

   $$P_{j\_old}(x) = P_{j\_old}(x) - P_{j\_old}(\boldsymbol{x}_{(k)}) P_k(x) \tag{3.23}$$

   We will take each P_{j_old} and remove from it the component parallel to the current polynomial P_k.

4. **Loop** increment k. If $k < N$ go to step 2.

After completion of the algorithm, we discard all the P_{j_old}'s and replace them with the P_j's for the next iteration of the global optimization algorithm.

The initial set of polynomials $\{P_1, \ldots, P_N\}$ can simply by initialized with the monomials of a polynomial of dimension n. For example, if $n = 2$, we obtain: $P_1(x) = 1$, $P_2(x) = x_1$, $P_3(x) = x_2$, $P_4(x) = x_1^2$, $P_5(x) = x_1\, x_2$, $P_6(x) = x_2^2$, $P_7(x) = x_2^3$, $P_8(x) = x_2^2\, x_1$, ...

In the Equation 3.22, there is a division. To improve the stability of the algorithm, we must do "pivoting". That is: select a salubrious pivot element for the division in Equation 3.22). We should choose the $\boldsymbol{x}_{(k)}$ (among the points which are still left inside the dataset) so that the denominator of Equation 3.22 is far from zero:

$$|P_{k\_old}(x)| \text{ as great as possible.} \tag{3.24}$$

If we don't manage to find a point $\boldsymbol{x}$ such that $Q_k(\boldsymbol{x}) \neq 0$, it means the dataset is NOT poised and the algorithm fails.

After completion of the algorithm, we have:

$$P_i(\boldsymbol{x}_{(j)}) = \delta_{(i,j)} \quad i, j = 1, \ldots, N \tag{3.25}$$

3.3.2 The Lagrange interpolation polynomial $L(x)$.

Using Equation 3.25, we can write:

$$L(x) = \sum_{j=1}^{N} f(\boldsymbol{x}_{(j)}) P_j(x) \tag{3.26}$$

3.3.3 The multivariate Horner scheme

Lets us rewrite a polynomial of degree d and dimension n using the following notation ($N = r_n^d$):

$$P(x) = \sum_{i=1}^{N} c_i \prod_{j=1}^{n} x_j^{\alpha(i,j)} \quad \text{with } \max_{i,j} \alpha(i,j) = d \tag{3.27}$$

α is a matrix which represents the way the monomials are ordered inside the polynomial. Inside our program, we always use the "order by degree" type. For example, for $n = 2, d = 2$, we have: $P(x) = c_1 + c_2x_1 + c_3x_2 + c_4x_1^2 + c_5x_1x_2 + c_6x_2^2$. We have the following matrix α:

$$\alpha = \begin{pmatrix} 0 & 0 \\ 1 & 0 \\ 0 & 1 \\ 2 & 0 \\ 1 & 1 \\ 0 & 2 \end{pmatrix}$$

We can also use the "inverse lexical order". For example, for $n = 2, d = 2$, we have: $P(x) = c_1x_1^2 + c_2x_1x_2 + c_3x_1 + c_4x_2^2 + c_5x_2 + c_6$. We have the following matrix α' (the $'$ is to indicate that we are in "inverse lexical order") :

$$\alpha'(\text{ for } n = 2 \text{ and } d = 2) = \begin{pmatrix} 2 & 0 \\ 1 & 1 \\ 1 & 0 \\ 0 & 2 \\ 0 & 1 \\ 0 & 0 \end{pmatrix} \qquad \alpha'(\text{ for } n = 3 \text{ and } d = 3) = \begin{pmatrix} 3 & 0 & 0 \\ 2 & 1 & 0 \\ 2 & 0 & 1 \\ 2 & 0 & 0 \\ 1 & 2 & 0 \\ 1 & 1 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 2 \\ 1 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 3 & 0 \\ 0 & 2 & 1 \\ 0 & 2 & 0 \\ 0 & 1 & 2 \\ 0 & 1 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 3 \\ 0 & 0 & 2 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix}$$

This matrix is constructed using the following property of the "inverse lexical order":

$$\exists j : \alpha'(i+1, j) < \alpha'(i, j) \text{ and } \forall k < j \quad \alpha'(i+1, k) = \alpha'(i, k)$$

The "inverse lexical order" is easier to transform in a multivariate horner scheme. For example: for the polynomial $P(x) = (c_1x_1 + c_2x_2 + c_3)x_1 + (c_4x_2 + c_5)x_2 + c_6$, we have:

- Set $r_1 := c_1; \quad r_2 = 0;$
- Set $r_2 := c_2;$
- Set $r_1 := c_3 + x_1r_1 + x_2r_2; \quad r_2 := 0;$

- Set $r_2 := c_4$;
- Set $r_2 := c_5 + x_2 r_2$;
- Return $c_6 + x_1 r_1 + x_2 r_2$

You can see that we retrieve inside this decomposition of the algorithm for the evaluation of the polynomial $P(x)$, the coefficient c_i in the same order that they appear when ordered in “inverse lexical order”.

Let us define the function $TR(i') = i$. This function takes, as input, the index of a monomial inside a polynomial ordered by ”inverse lexical order” and gives, as output, the index of the same monomial but, this time, placed inside a polynomial ordered ”by degree”. In other words, This function makes the **TR**ansformation between index in inverse lexical order and index ordered by degree.

We can now define an algorithm which computes the value of a multivariate polynomial ordered by degree by multivariate horner scheme:

```
1. Declaration
   $n$ : dimension of the space
   $N = r_n^d$ : number of monomial inside a polynomial of dimension $n$ and degree $d$.
   $r_0, \dots, r_n$ : registers for summation ($\in \Re$)
   $a_1, \dots, a_n$ : counters ($\in N_0$)
   $c_i, \quad i = 1, \dots, N$ : the coefficients of the polynomial ordered by degree.
2. Initialization
   Set $r_0 := c_{Tr(1)}$
   set $a_j := \alpha'(1, j) \quad j = 1, \dots, n$
3. For $i = 2, \dots, N$
   (a) Determine $k = \max_j \{1 \leq j \leq n : \alpha'(i, j) \neq \alpha'(i-1, j)\}$
   (b) Set $a_k := a_k - 1$
   (c) Set $r_k := x_k(r_0 + r_1 + \ldots + r_k)$
   (d) Set $r_0 := c_{Tr(i)}, r1 := \ldots := r_{k-1} := 0$
   (e) Set $a_j := \alpha(i, j) \quad j = 1, \dots, k-1$
4. Return $r0 + \ldots + r_n$
```

In the program, we are caching the values of k and the function TR, for a given n and d. Thus, we compute these values once, and use pre-computed values during the rest of the time. This lead to a great efficiency in speed and in precision for polynomial evaluations.

3.4 The Lagrange Interpolation inside the optimization loop.

Inside the optimization program, we only use polynomials of degree lower or equal to 2. Therefore, we will always assume in the end of this chapter that $d = 2$. We have thus $N = r_n^2 = (n+1)(n+2)/2$: the maximum number of monomials inside all the polynomials of the optimization loop.

3.4.1 A bound on the interpolation error.

We assume in this section that the objective function $f(x), x \in \Re^n$, has its third derivatives that are bounded and continuous. Therefore, if y is any point int $\Re^n$, and if $\bar{d}$ is any vector int $\Re^n$ that has $\|\bar{d}\| = 1$, then the function of one variable

$$\phi(\alpha) = f(y + \alpha\bar{d}), \quad \alpha \in \Re, \tag{3.28}$$

also has bounded and continuous third derivatives. Further there is a least non-negative number M, independent of y and $\bar{d}$, such that every functions of this form have the property

$$|\phi'''(\alpha)| \leq M, \quad \alpha \in \Re \tag{3.29}$$

This value of M is suitable for the following bound on the interpolation error of f(x):

$$\textit{Interpolation Error} = |L(x) - f(x)| < \frac{1}{6} M \sum_{j=1}^{N} |P_j(x)| \|x - \boldsymbol{x}_{(j)}\|^3 \tag{3.30}$$

Proof

We make any choice of y. We regard y as fixed for the moment, and we derive a bound on $|L(y) - f(y)|$. The Taylor series expansion of $f(x)$ around the point y is important. Specifically, we let $T(x), x \in \Re^n$, be the quadratic polynomial that contains all the zero order, first order and second order terms of this expansion, and we consider the possibility of replacing the objective function f by $f - T$. The replacement would preserve all the third derivatives of the objective function, because T is a quadratic polynomial. Therefore, the given choice of M would remain valid. Further more, the quadratic model $f - T$ that is defined by the interpolation method would be $L - T$. It follows that the error on the new quadratic model of the new objective function is $f - L$ as before. Therefore, when seeking for a bound on $|L(y) - f(y)|$ in terms of third derivatives of the objective function, we can assume without loss of generality, that the function value $f(y)$, the gradient vector $g(y) = f'(y)$ and the second derivative matrix $H(y) = f''(y)$ are all zero.

Let j be an integer from $1, \dots, N$ such that $\boldsymbol{x}_{(j)} \neq y$, let $\bar{d}$ be the vector:

$$\bar{d} = \frac{\boldsymbol{x}_{(j)} - y}{\|\boldsymbol{x}_{(j)} - y\|} \tag{3.31}$$

and let $\phi(\alpha), \alpha \in \Re$, be the function 3.28. The Taylor series with explicit remainder formula gives:

$$\phi(\alpha) = \phi(0) + \alpha\phi'(0) + \frac{1}{3}\alpha^2\phi''(0) + \frac{1}{6}\alpha^3\phi'''(\varepsilon), \qquad \alpha \geq 0, \tag{3.32}$$

where ε depends on α and is in the interval $[0, \alpha]$. The values of $\phi(0)$, $\phi'(0)$ and $\phi''(0)$ are all zero due to the assumptions of the previous paragraph, and we pick $\alpha = \|y - \boldsymbol{x}_{(j)}\|$. Thus expressions 3.31, 3.28, 3.32, 3.29 provide the bound

$$|f(\boldsymbol{x}_{(j)})| = \frac{1}{6}\alpha^3|\phi'''(\varepsilon)| \leq \frac{1}{6}M\|y - \boldsymbol{x}_{(j)}\|^3, \tag{3.33}$$

which also holds without the middle part in the case $\boldsymbol{x}_{(j)} = y$, because of the assumption $f(y) = 0$. Using $f(y) = 0$ again, we deduce from Equation 3.26 and from inequality 3.33, that the error $L(y) - f(y)$ has the property:

$$\begin{aligned} |L(y) - f(y)| = |L(y)| &= |\sum_{j=1}^{N} f(x_j) P_j(y)| \\ &\leq \frac{1}{6} M \sum_{j=1}^{N} |P_j(y)| \|y - \boldsymbol{x}_{(j)}\|^3. \end{aligned}$$

Therefore, because y is arbitrary, the bound of equation 3.30 is true.

In the optimization loop, each time we evaluate the objective function f, at a point x, we adjust the value of an estimation of M using:

$$M_{new} = \max \left[M_{old}, \frac{|L(x) - f(x)|}{\frac{1}{6} \sum_{j=1}^{N} |P_j(x)| \|x - \boldsymbol{x}_{(j)}\|^3} \right] \tag{3.34}$$

3.4.2 Validity of the interpolation in a radius of ρ around $\boldsymbol{x}_{(k)}$.

We will test the validity of the interpolation around $\boldsymbol{x}_{(k)}$. If the model (=the polynomial) is too bad around $\boldsymbol{x}_{(k)}$, we will replace the "worst" point $\boldsymbol{x}_{(j)}$ of the model by a new, better, point in order to improve the accuracy of the model.

First, we must determine x_j. We select among the initial dataset $\{\boldsymbol{x}_{(1)}, \ldots, \boldsymbol{x}_{(N)}\}$ a new dataset $\mathcal{J}$which contains all the points $\boldsymbol{x}_{(i)}$ for which $\|\boldsymbol{x}_{(i)} - \boldsymbol{x}_{(k)}\| > 2\rho$. If $\mathcal{J}$is empty, the model is valid and we exit.

We will check all the points inside $\mathcal{J}$, one by one.

We will begin by, hopefully, the worst point in $\mathcal{J}$: Among all the points in $\mathcal{J}$, choose the point the further away from $\boldsymbol{x}_{(k)}$. We define j as the index of such a point.

If x is constrained by the trust region bound $\|x - \boldsymbol{x}_{(k)}\| < \rho$, then the contribution to the error of the model from the position $\boldsymbol{x}_{(j)}$ is approximately the quantity (using Equation 3.30):

$$\frac{1}{6} M \max_x \{ |P_j(x)| \|x - \boldsymbol{x}_{(k)}\|^3 : \|x - \boldsymbol{x}_{(k)}\| \leq \rho \} \tag{3.35}$$

$$\approx \frac{1}{6} M \|\boldsymbol{x}_{(j)} - \boldsymbol{x}_{(k)}\|^3 \max_d \{ |P_j(\boldsymbol{x}_{(k)} + d)| : \|d\| \leq \rho \} \tag{3.36}$$

Therefore the model is considered valid if it satisfies the condition :

$$\frac{1}{6} M \|\boldsymbol{x}_{(j)} - \boldsymbol{x}_{(k)}\|^3 \max_d \{ |P_j(\boldsymbol{x}_{(k)} + d)| : \|d\| \leq \rho \} \leq \epsilon \tag{3.37}$$

ϵ is a bound on the error which must be given to the procedure which checks the validity of the interpolation. See section 6.1 to know how to compute the bound ϵ.

The algorithm which searches for the value of d for which we have

$$\max_d \{ |P_j(\boldsymbol{x}_{(k)} + d)| : \|d\| \leq \rho \} \tag{3.38}$$

is described in Chapter 5.

We are ignoring the dependence of the other Newton polynomials in the hope of finding a useful technique which can be implemented cheaply.

If Equation 3.37 is verified, we now remove the point $\boldsymbol{x}_{(j)}$ from the dataset $\mathcal{J}$ and we iterate: we search among all the points left in $\mathcal{J}$, for the point the further away from $\boldsymbol{x}_{(k)}$. We test this point using 3.37 and continue until the dataset $\mathcal{J}$ is empty.

If the test 3.37 fails for a point $\boldsymbol{x}_{(j)}$, then we change the polynomial: we remove the point $\boldsymbol{x}_{(j)}$ from the interpolating set and replace it with the "better" point: $\boldsymbol{x}_{(k)} + d$ (were d is the solution of 3.38): see section 3.4.4, to know how to do.

3.4.3 Find a good point to replace in the interpolation.

If we are forced to include a new point X in the interpolation set even if the polynomial is valid, we must choose carefully which point $\boldsymbol{x}_{(t)}$ we will drop.

Let us define $\boldsymbol{x}_{(k)}$, the best (lowest) point of the interpolating set.

We want to replace the point $\boldsymbol{x}_{(t)}$ by the point X. Following the remark of Equation 3.24, we must have:

$$|P_t(X)| \text{ as great as possible} \tag{3.39}$$

We also wish to remove a point which seems to be making a relatively large contribution to the bound 3.30 on the error on the quadratic model. Both of these objectives are observed by setting t to the value of i that maximizes the expression:

$$\begin{cases} |P_i(X)| \max\left[1, \frac{\|\boldsymbol{x}_{(i)}-X\|^3}{\rho^3}\right], \quad i=1,\ldots,N & \text{if } f(X) < f(\boldsymbol{x}_{(k)}) \\ |P_i(X)| \max\left[1, \frac{\|\boldsymbol{x}_{(i)}-\boldsymbol{x}_{(k)}\|^3}{\rho^3}\right], \quad i=1,\ldots,k-1,k+1,\ldots,N & \text{if } f(X) > f(\boldsymbol{x}_{(k)}) \end{cases} \tag{3.40}$$

3.4.4 Replace the interpolation point $\boldsymbol{x}_{(t)}$ by a new point X.

Let $\tilde{P}_i \quad i=1,\ldots,N$, be the new Lagrange polynomials after the replacement of $\boldsymbol{x}_{(t)}$ by X.

The difference $\tilde{P}_i - P_i$ has to be a multiple of $\tilde{P}_t$, in order that $\tilde{P}_i$ agrees with P_i at all the old interpolation points that are retained. Thus we deduce the formula:

$$\tilde{P}_t(x) = \frac{P_t(x)}{P_t(X)} \tag{3.41}$$

$$\tilde{P}_i(x) = P_i(x) - P_i(X)\tilde{P}_t(x), \quad i \neq t \tag{3.42}$$

$L(x) = \sum_{j=1}^{N} f(\boldsymbol{x}_{(j)}) P_j(x)$ has to be revised too. The difference $L_{new} - L_{old}$ is a multiple of $\tilde{P}_t(x)$ to allow the old interpolation points to be retained. We finally obtain:

$$L_{new}(x) = L_{old}(x) + [f(X) - L_{old}(X)]\tilde{P}_t(x) \tag{3.43}$$

3.4.5 Generation of the first set of point $\{\boldsymbol{x}_{(1)}, \dots, \boldsymbol{x}_{(N)}\}$.

To be able to generate the first set of interpolation point $\{\boldsymbol{x}_{(1)}, \dots, \boldsymbol{x}_{(N)}\}$, we need:

- The base point $x_{(base)}$ around which $\begin{cases} \text{the function will be interpolated.} \\ \text{the set will be constructed} \end{cases}$
- A length ρ which will be used to separate 2 interpolation point.

If we have already at disposal, $N = (n+1)(n+2)/2$ points situated inside a circle of radius $2\ \rho$ around $x_{(base)}$, we try to construct directly a Lagrange polynomial using them. If the construction fails (points are not poised.), or if we don't have enough point we generate the following interpolation set:

- First point: $\boldsymbol{x}_{(1)} = x_{(base)}$
- From $\boldsymbol{x}_{(2)}$ to $\boldsymbol{x}_{(1+n)}$:

$\boldsymbol{x}_{(j+1)} = x_{(base)} + \rho e_j \quad j = 1, \dots n$ (with e_i being a unit vector along the axis j of the space)

Let us define σ_j:

$$\sigma_j := \begin{cases} -1 & \text{if } f(\boldsymbol{x}_{(j+1)}) > f(\boldsymbol{x}_{(base)}) \\ +1 & \text{if } f(\boldsymbol{x}_{(j+1)}) < f(\boldsymbol{x}_{(base)}) \end{cases} \quad j = 1, \dots n$$

- From $\boldsymbol{x}_{(2+n)}$ to $\boldsymbol{x}_{(1+2n)}$:

$$\boldsymbol{x}_{(j+1+n)} = \begin{cases} x_{(base)} - \rho e_j & \text{if } \sigma_j = -1 \\ x_{(base)} + 2\rho e_j & \text{if } \sigma_j = +1 \end{cases} \quad j = 1, \dots n$$

- From $\boldsymbol{x}_{(2+2n)}$ to $\boldsymbol{x}_{(N)}$:

```
Set $k = 2 + 2n$.
For $j = 1, \dots, n$
  1. For $i = 1, \dots, j-1$
     (a) $\boldsymbol{x}_{(k)} = x_{(base)} + \rho(\sigma_i e_i + \sigma_j e_j) \quad 1 \le i < j \le n$
     (b) Increment $k$.
```

3.4.6 Translation of a polynomial.

The precision of a polynomial interpolation is better when all the interpolation points are close to the center of the space ($\|\boldsymbol{x}_{(i)}\|$ are small).

Example: Let all the interpolation points $\boldsymbol{x}_{(i)}$, be near $x_{(base)}$, and $\|x_{(base)}\| >> 0$. We have constructed two polynomials $P_1(x)$ and $P_2(x)$:

- $P_1(x)$ interpolates the function $f(x)$ at all the interpolation sites $\boldsymbol{x}_{(i)}$.
- $P_2(x - x_{(base)})$ interpolates also the function $f(x)$ at all the interpolation sites $\boldsymbol{x}_{(i)}$.

$P_1(x)$ and $P_2(x)$ are both valid interpolator of $f(x)$ around $x_{(base)}$ BUT it's more interesting to work with P_2 rather then P_1 because of the greater accuracy in the interpolation.

How to obtain $P_2(x)$ from $P_1(x)$?

$P_2(x)$ is the polynomial $P_1(x)$ after the translation $x_{(base)}$.

We will only treat the case where $P_1(x)$ and $P_2(x)$ are quadratics. Let's define $P1(x)$ and $P_2(x)$ the following way:

$$\begin{cases} P_1(x) = a_1 + g_1^T x + \frac{1}{2} x^T H_1 x \\ P_2(x) = a_2 + g_2^T x + \frac{1}{2} x^T H_2 x \end{cases}$$

Using the secant Equation 13.27, we can write:

$$\begin{cases} a_2 := P_1(x_{(base)}) \\ g_2 := g_1 + H_1 x_{(base)} \\ H_2 := H_1 \end{cases} \tag{3.44}$$

Chapter 4

The Trust-Region subproblem

We seek the solution s^* of the minimization problem:

$$\min_{s \in \Re^n} q(x_k + s) \equiv f(x_k) + \langle g_k, s \rangle + \frac{1}{2} \langle s, H_k s \rangle$$
$$\text{subject to } \|s\|_2 < \Delta$$

The following minimization problem is equivalent to the previous one after a translation of the polynomial q in the direction x_k (see Section 3.4.6 about polynomial translation). Thus, this problem will be discussed in this chapter:

$$\min_{s \in \Re^n} q(s) \equiv \langle g, s \rangle + \frac{1}{2} \langle s, Hs \rangle$$
$$\text{subject to } \|s\|_2 < \Delta$$

We will indifferently use the terms, polynomial, quadratic or model. The "trust region" is defined by the set of all the points which respects the constraint $\|s\|_2 \leq \Delta$.

Definition: The trust region $\mathcal{B}_k$ is the set of all points such that $\mathcal{B}_k = \{x \in \Re^n | \|x - x_k\|_k \leq \Delta_k\}$.

Note that we must always have at the end of the algorithm:

$$q(s^*) \leq 0 \tag{4.1}$$

The material of this chapter is based on the following references: [CGT00c, MS83].

4.1 $H(\lambda^*)$ must be positive definite.

The solution we are seeking lies either interior to the trust region ($\|s\|_2 < \Delta$) or on the boundary.

If it lies on the interior, the trust region may as well not have been there and therefore s^* is the unconstrained minimizer of $q(s)$. We have seen in Equation 2.4 ($Hs = -g$) how to find it. We have seen in Equation 2.6 that

$$H \text{ must be definite positive} \quad (s^T H s > 0 \quad \forall s) \tag{4.2}$$

in order to be able to apply 2.4.

If we found a value of s^* using 2.4 which lies outside the trust region, it means that s^* lies on the trust region boundary. Let's take a closer look to this case:

Theorem 1

Any global minimizer of $q(s)$ subject to $\|s\|_2 = \Delta$ satisfies the Equation

$$H(\lambda^*)s^* = -g, \tag{4.3}$$

where $H(\lambda^*) \equiv H + \lambda^* I$ is positive semidefinite. If $H(\lambda^*)$ is positive definite, s^* is unique.

First, we rewrite the constraints $\|s\|_2 = \Delta$ as $c(s) = \frac{1}{2}\Delta - \frac{1}{2}\|s\|_2 = 0$. Now, we introduce a Lagrange multiplier λ for the constraint and use first-order optimality conditions (see Annexe, section 13.3). This gives:

$$\mathcal{L}(s, \lambda) = q(s) - \lambda c(s) \tag{4.4}$$

Using first part of Equation 13.22,we have

$$\nabla_s \mathcal{L}(s^*, \lambda^*) = \nabla q(s^*) - \lambda^* \nabla_s c(s^*) = Hs^* + g + \lambda^* s^* = 0 \tag{4.5}$$

which is 4.3.

We will now proof that $H(\lambda^*)$ must be positive (semi)definite.

Suppose s^F is a feasible point ($\|s^F\| = \Delta$), we obtain:

$$q(s^F) = q(s^*) + \langle s^F - s^*, g(s^*) \rangle + \frac{1}{2}\langle s^F - s^*, H(s^F - s^*) \rangle \tag{4.6}$$

Using the secant equation (see Annexes, Section 13.4), $g'' - g' = H(x'' - x') = Hs \quad (s = x'' - x')$, we can rewrite 4.5 into $g(s^*) = -\lambda^* s^*$. This and the restriction that ($\|s^F\| = \|s^*\| = \Delta$) implies that:

$$\begin{aligned}
\langle s^F - s^*, g(s^*) \rangle =& \langle s^* - s^F, s^* \rangle \lambda^* \\
=& (\Delta^2 - \langle s^F, s^* \rangle)\lambda^* \\
=& [\frac{1}{2}(\langle s^F, s^F \rangle + \langle s^*, s^* \rangle) - \langle s^F, s^* \rangle]\lambda^* \\
=& [\frac{1}{2}(\langle s^F, s^F \rangle + \langle s^*, s^* \rangle) - \frac{1}{2}\langle s^F, s^* \rangle - \frac{1}{2}\langle s^F, s^* \rangle]\lambda^* \\
=& \frac{1}{2}[\langle s^F, s^F \rangle - \langle s^F, s^* \rangle + \langle s^*, s^* \rangle - \langle s^F, s^* \rangle]\lambda^* \\
=& \frac{1}{2}[\langle s^F, s^F - s^* \rangle + \langle s^* - s^F, s^* \rangle]\lambda^* \\
=& \frac{1}{2}[\langle s^F, s^F - s^* \rangle - \langle s^*, s^F - s^* \rangle]\lambda^* \\
\langle s^F - s^*, g(s^*) \rangle =& \frac{1}{2}\langle s^F - s^*, s^F - s^* \rangle \lambda^*
\end{aligned} \tag{4.7}$$

Combining 4.6 and 4.7

$$\begin{aligned} q(s^F) =& q(s^*) + \frac{1}{2}\langle s^F - s^*, s^F - s^*\rangle\lambda^* + \frac{1}{2}\langle s^F - s^*, H(s^F - s^*)\rangle \\ =& q(s^*) + \frac{1}{2}\langle s^F - s^*, (H + \lambda^* I)(s^F - s^*)\rangle \\ =& q(s^*) + \frac{1}{2}\langle s^F - s^*, H(\lambda^*)(s^F - s^*)\rangle \end{aligned} \tag{4.8}$$

Let's define a line $s^* + \alpha v$ as a function of the scalar α. This line intersect the constraints $\|s\| = \Delta$ for two values of α: $\alpha = 0$ and $\alpha = \alpha^F \neq 0$ at which $s = s^F$. So $s^F - s^* = \alpha^F v$, and therefore, using 4.8, we have that

$$q(s^F) = q(s^*) + \frac{1}{2}(\alpha^F)^2\langle v, H(\lambda^*)v\rangle$$

Finally, as we are assuming that s^* is a global minimizer, we must have that $s^F \geq s^*$, and thus that $\langle v, H(\lambda^*)v\rangle \geq 0 \quad \forall v$, which is the same as saying that $H(\lambda)$ is positive semidefinite.

If $H(\lambda^*)$ is positive definite, then $\langle s^F - s^*, H(\lambda^*)(s^F - s^*)\rangle > 0$ for any $s^F \neq s^*$, and therefore 4.8 shows that $q(s^F) > q(s^*)$ whenever s^F is feasible. Thus s^* is the unique global minimizer.

Using 4.2 (which is concerned about an interior minimizer) and the previous paragraph (which is concerned about a minimizer on the boundary of the trust region), we can state:

Theorem 2:

Any global minimizer of $q(s)$ subject to $\|s\|_2 \leq \Delta$ satisfies the Equation

$$H(\lambda^*)s^* = -g, \tag{4.9}$$

where $H(\lambda^*) \equiv H + \lambda^* I$ is positive semidefinite, $\lambda^* \geq 0$, and $\lambda^*(\|s^*\| - \Delta) = 0$. If $H(\lambda^*)$ is positive definite, s^* is unique.

The justification of $\lambda^*(\|s^*\| - \Delta) = 0$ is simply the *complementarity condition* (see Section 13.3 for explanation, Equation 13.22).

The parameter λ is said to "regularized" or "modify" the model such that the modified model is convex and so that its minimizer lies on or within the trust region boundary.

4.2 Explanation of the Hard case.

Theorem 2 tells us that we should be looking for solutions to 4.9 and implicitly tells us what value of λ we need. Suppose that H has an eigendecomposition:

$$H = U^T \Lambda U \tag{4.10}$$

where Λ is a diagonal matrix of eigenvalues $\lambda_1 < \lambda_2 < \ldots < \lambda_n$ and U is an orthonormal matrix of associated eigenvectors. Then

$$H(\lambda) = U^T(\Lambda + \lambda I)U \tag{4.11}$$

We deduce immediately from Theorem 2 that the value of λ we seek must satisfy $\lambda^* > \min[0, -\lambda_1]$ (as only then is $H(\lambda)$ positive semidefinite) (λ_1 is the least eigenvalue of H). We can compute a solution $s(\lambda)$ for a given value of λ using:

$$s(\lambda) = -H(\lambda)^{-1} g = -U^T (\Lambda + \lambda I)^{-1} U g \tag{4.12}$$

The solution we are looking for depends on the non-linear inequality $\|s(\lambda)\|_2 < \Delta$. To say more we need to examine $\|s(\lambda)\|_2$ in detail. For convenience we define $\psi(\lambda) \equiv \|s(\lambda)\|_2^2$. We have that:

$$\psi(\lambda) = \|U^T (\Lambda + \lambda I)^{-1} U g\|_2^2 = \|(\Lambda + \lambda I)^{-1} U g\|_2^2 = \sum_{i=1}^{n} \frac{\gamma_i^2}{\lambda_i + \lambda} \tag{4.13}$$

where γ_i is $[Ug]_i$, the i^{th} component of Ug.

4.2.1 Convex example.

Suppose the problem is defined by:

$$g = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \end{pmatrix}, H = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 4 \end{pmatrix}$$

We plot the function $\psi(\lambda)$ in Figure 4.1. Note the pole of $\psi(\lambda)$ at the negatives of each eigenvalues of H. In view of theorem 2, we are only interested in $\lambda > 0$. If $\lambda = 0$, the optimum lies inside the trust region boundary. Looking at the figure, we obtain $\lambda = \lambda^* = 0$, for $\psi(\lambda) = \Delta^2 > 1.5$. So, it means that if $\Delta^2 > 1.5$, we have an internal optimum which can be computed using 4.9. If $\Delta^2 < 1.5$, there is a unique value of $\lambda = \lambda^*$ (given in the figure and by

$$\|s(\lambda)\|_2 - \Delta = 0 \tag{4.14}$$

which, used inside 4.9, give the optimal s^*.

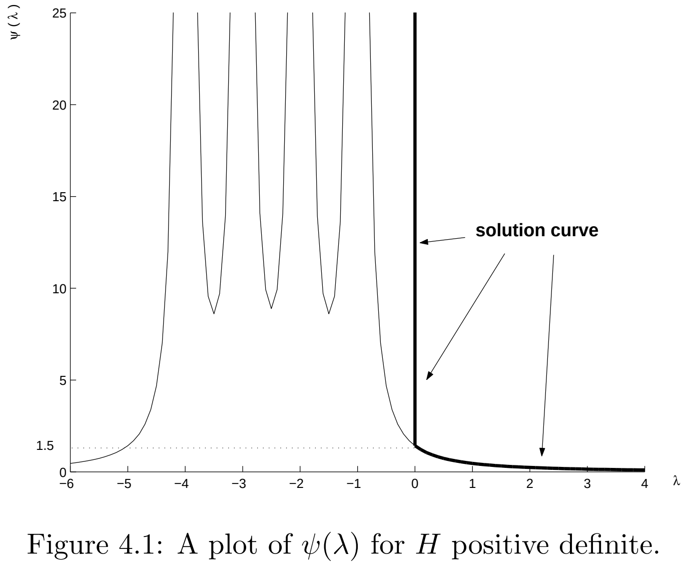

Figure 4.1: A plot of $\psi(\lambda)$ for H positive definite.

4.2.2 Non-Convex example.

Suppose the problem is defined by:

$$g = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \end{pmatrix}, H = \begin{pmatrix} -2 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

We plot the function $\psi(\lambda)$ in Figure 4.2. Recall that λ_1 is defined as the least eigenvalue of H. We are only interested in values of $\lambda > -\lambda_1$, that is $\lambda > 2$. For value of $\lambda < \lambda_1$, we have $H(\lambda)$ NOT positive definite. This is forbidden due to theorem 2. We can see that for any value of Δ, there is a corresponding value of $\lambda > 2$. Geometrically, H is indefinite, so the model function is unbounded from below. Thus the solution lies on the trust-region boundary. For a given λ^*, found using 4.14, we obtain the optimal s^* using 4.9.

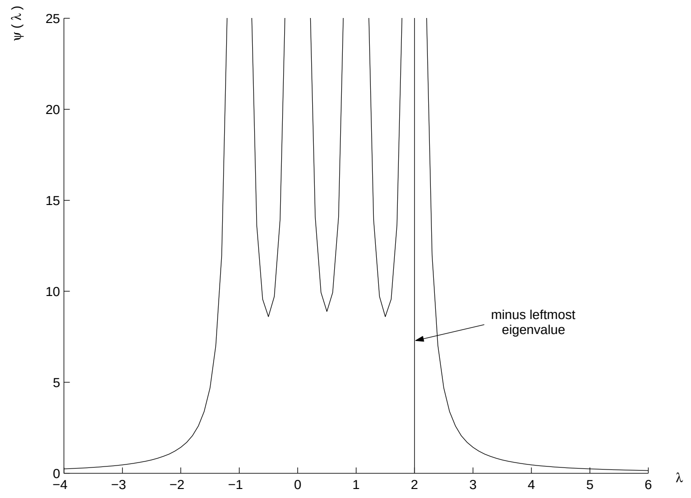

Figure 4.2: A plot of $\psi(\lambda)$ for H indefinite.

4.2.3 The hard case.

Suppose the problem is defined by:

$$g = \begin{pmatrix} 0 \\ 1 \\ 1 \\ 1 \end{pmatrix}, H = \begin{pmatrix} -2 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

We plot the function $\psi(\lambda)$ in Figure 4.3. Again, $\lambda < 2$, is forbidden due to theorem 2. If, $\Delta > \Delta_{critical} \approx 1.2$, there is no acceptable value of λ. This difficulty can only arise when g is orthogonal to the space $\mathcal{E}_\infty$, of eigenvectors corresponding to the most negative eigenvalue of H. When $\Delta = \Delta_{cri}$, then equation 4.9 has a limiting solution s_{cri}, where $s_{cri} = \lim_{\lambda \to \lambda_1} s(\lambda)$.

$H(\lambda_1)$ is positive semi-definite and singular and therefore 4.9 has several solutions. In particular, if u_1 is an eigenvector corresponding to λ_1, we have $H(-\lambda_1)u_1 = 0$, and thus:

$$H(-\lambda_1)(s_{cri} + \alpha u_1) = -g \tag{4.15}$$

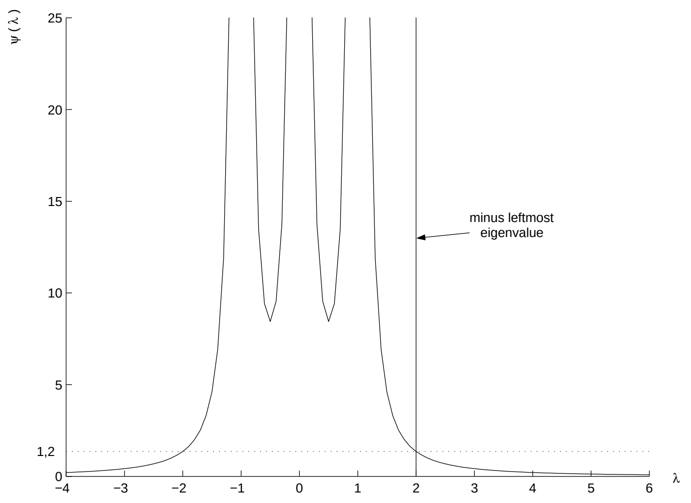

Figure 4.3: A plot of $\psi(\lambda)$ for H semi-definite and singular(hard case).

holds for any value of the scalar α. The value of α can be chosen so that $\|s_{cri} + \alpha u_1\|_2 = \Delta$. There are two roots to this equation: α_1 and α_2. We evaluate the model at these two points and choose as solution $s^* = s_{cri} + \alpha^* u_1$, the lowest one.

4.3 Finding the root of $\|s(\lambda)\|_2 - \Delta = 0$.

We will apply the 1D-optimization algorithm called "1D Newton's search" (see Annexes, Section 13.5) to the *secular equation*:

$$\phi(\lambda) = \frac{1}{\|s(\lambda)\|_2} - \frac{1}{\Delta} \tag{4.16}$$

We use the secular equation instead of $\psi(\lambda) - \Delta^2 = \|s(\lambda)\|_2^2 - \Delta^2 = 0$ inside the "1D Newton's search" because this last function is better behaved than $\psi(\lambda)$. In particular $\phi(\lambda)$ is strictly increasing when $\lambda > \lambda_1$, and concave. It's first derivative is:

$$\phi'(\lambda) = -\frac{\langle s(\lambda), \nabla_\lambda s(\lambda)\rangle}{\|s(\lambda)\|_2^3} \tag{4.17}$$

where

$$\nabla_\lambda s(\lambda) = -H(\lambda)^{-1} s(\lambda) \tag{4.18}$$

The proof of these properties will be skipped.

In order to apply the "1D Newton's search":

$$\lambda_{k+1} = \lambda_k - \frac{\phi(\lambda)}{\phi'(\lambda)} \tag{4.19}$$

we need to evaluate the function $\phi(\lambda)$ and $\phi'(\lambda)$. The value of $\phi(\lambda)$ can be obtained by solving the Equation 4.9 to obtain $s(\lambda)$. The value of $\phi'(\lambda)$ is available from 4.17 once $\nabla_\lambda s(\lambda)$ has been found using 4.18. Thus both values may be found by solving linear systems involving $H(\lambda)$. Fortunately, in the range of interest, $H(\lambda)$ is definite positive, and thus, we may use its Cholesky factors $H(\lambda) = L(\lambda)L(\lambda)^T$ (see Annexes, Section 13.7 for notes about Cholesky decomposition). Notice that we do not actually need tho find $\nabla_\lambda s(\lambda)$, but merely the numerator $\langle s(\lambda), \nabla_\lambda s(\lambda)\rangle = -\langle s(\lambda), H(\lambda)^{-1} s(\lambda)\rangle$ of 4.17. The simple relationship

$$\langle s(\lambda), H(\lambda)^{-1} s(\lambda)\rangle = \langle s(\lambda), L^{-T} L^{-1} s(\lambda)\rangle = \langle L^{-1} s(\lambda), L^{-1} s(\lambda)\rangle = \|\omega\|^2 \tag{4.20}$$

explains why we compute ω in step 3 of the following algorithm. Step 4 of the algorithm follows directly from 4.17 and 4.19. Newton's method to solve $\phi(\lambda) = 0$:

1. find a value of λ such that $\lambda > \lambda_1$ and $\lambda < \lambda^*$.
2. factorize $H(\lambda) = LL^T$
3. solve $LL^T s = -g$
4. Solve $L\omega = s$
5. $$\text{Replace } \lambda \text{ by } \lambda + \left(\frac{\|s(\lambda)\|_2 - \Delta}{\Delta}\right)\left(\frac{\|s(\lambda)\|_2^2}{\|\omega\|_2^2}\right) \tag{4.21}$$
6. If stopping criteria are not met, go to step 2.

Once the algorithm has reached point 2. It will always generate values of $\lambda > \lambda_1$. Therefore, the Cholesky decomposition will never fails and the algorithm will finally find λ^*. We skip the proof of this property.

4.4 Starting and safe-guarding Newton's method

In step 1 of Newton's method, we need to find a value of λ such that $\lambda > \lambda_1$ and $\lambda < \lambda^*$. What happens if $\lambda > \lambda^*$ (or equivalently $\|s(\lambda)\| < \|\Delta\|$)? The Cholesky factorization succeeds and so we can apply 4.21. We get a new value for λ but we must be careful because this new value can be in the forbidden region $\lambda < \lambda_1$.

If we are in the hard case, it's never possible to get $\lambda < \lambda_*$ (or equivalently $\|s(\lambda)\| > \|\Delta\|$), therefore we will never reach point 2 of the Newton's method.

In the two cases described in the two previous paragraphs, the Newton's algorithm fails. We will now describe a modified Newton's algorithm which prevents these failures:

1. Compute λ_L and λ_U respectively a lower and upper bound on the lowest eigenvalue λ_1 of H.
2. Choose $\lambda \in [\lambda_L \lambda_U]$. We will choose: $\lambda = \dfrac{\|g\|}{\Delta}$
3. Try to factorize $H(\lambda) = LL^T$ (if not already done).
   - **Success:**
     (a) solve $LL^T s = -g$

(b) Compute $\|s\|$:
   - $\|s\| < \Delta$: $\lambda_U := min(\lambda_U, \lambda)$

     We must be careful: the next value of λ can be in the forbidden $\lambda < \lambda_1$ region. We may also have interior convergence: Check λ:
     * $\lambda = 0$: The algorithm is finished. We have found the solution s^* (which is inside the trust region).
     * $\lambda \neq 0$: We are maybe in the hard case. Use the methods described in the paragraph containing the Equation 4.15 to find $\|s + \alpha u_1\|_2 = \|\delta\|_2$. Check for termination for the hard case.
   - $\|s\| > \Delta$: $\lambda_L := max(\lambda_L, \lambda)$

(c) Check for termination for the normal case: s^* is on the boundary of the trust region.

(d) Solve $L\omega = s$

(e) Compute $\lambda_{new} = \lambda + (\frac{\|s(\lambda)\|_2 - \Delta}{\Delta})(\frac{\|s(\lambda)\|_2^2}{\|\omega\|_2^2})$

(f) Check λ_{new}: Try to factorize $H(\lambda_{new}) = LL^T$
   - **Success:** replace λ by λ_{new}
   - **Failure:** $\lambda_L = max(\lambda_L, \lambda_{new})$

     The Newton's method, just failed to choose a correct λ. Use the "alternative" algorithm: pick λ inside $[\lambda_L \lambda_U]$ (see Section 4.5).

- **Failure:** Improve λ_L using Rayleigh's quotient trick (see Section 4.8). Use the "alternative" algorithm: pick λ inside $[\lambda_L \lambda_U]$ (see section 4.5).

4. return to step 3.

4.5 How to pick λ inside $[\lambda_L \lambda_U]$?

The simplest choice is to pick the midpoint:

$$\lambda = \frac{1}{2}(\lambda_L + \lambda_U)$$

A better solution (from experimental research)is to use ($\theta = 0.01$):

$$\lambda = max(\sqrt{\lambda_L \lambda_U}, \lambda_L + \theta(\lambda_U - \lambda_L)) \tag{4.22}$$

4.6 Initial values of λ_L and λ_U

Using the well-known *Gershgorin* bound:

$$\min_i \left([H]_{i,i} - \sum_{i \neq j} |[H]_{i,j}| \right) \leq \lambda_{min}[H] \leq \lambda_{max}[H] \leq \max_i \left([H]_{i,i} + \sum_{i \neq j} |[H]_{i,j}| \right)$$

The *frobenius* or Euclidean norm:

$$\|H\|_F = \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n}[H]_{i,j}^2}$$

The infinitum norm:

$$\|H\|_\infty = \max_{1\leq i\leq \text{nLine}} \|H^T e_i\|_1$$

We finally obtain:

$$\lambda_L := \max\left[0, -\min_i [H]_{i,i}, \frac{\|g\|_2}{\Delta} - \min\left[\max_i\left[[H]_{i,i} + \sum_{i\neq j}|[H]_{i,j}|\right], \|H\|_F, \|H\|_\infty\right]\right]$$

$$\lambda_U := \max\left[0, \frac{\|g\|_2}{\Delta} + \min\left[\max_i\left[-[H]_{i,i} + \sum_{i\neq j}|[H]_{i,j}|\right], \|H\|_F, \|H\|_\infty\right]\right]$$

4.7 How to find a good approximation of u_1 : LINPACK METHOD

u_1 is the unit eigenvector corresponding to λ_1. We need this vector in the hard case (see the paragraph containing equation 4.15). Since u_1 is the eigenvector corresponding to λ_1, we can write:

$$(H - \lambda_1 I)u_1 = 0 \Rightarrow H(\lambda_1)u_1 = 0$$

We will try to find a vector u which minimizes $\langle u, H(\lambda)u\rangle$. This is equivalent to find a vector v which maximize $\omega := H(\lambda)^{-1}v = L^{-T}L^{-1}v$. We will choose the component of v between $+1$ and -1 in order to make $L^{-1}v$ large. This is achieved by ensuring that at each stage of the forward substitution $L\omega = v$, the sign of v is chosen to make ω as large as possible. In particular, suppose we have determined the first $k-1$ components of ω during the forward substitution, then the k^{th} component satisfies:

$$l_{kk}\omega_k = v_k - \sum_{i=1}^{k-1} l_{ki}\omega_i,$$

and we pick v_k to be ± 1 depending on which of

$$\frac{1 - \sum_{i=1}^{k-1} l_{ki}\omega_i}{l_{kk}} \quad \text{or} \quad \frac{-1 - \sum_{i=1}^{k-1} l_{ki}\omega_i}{l_{kk}}$$

is larger. Having found ω, u is simply $L^{-T}\omega/\|L^{-T}\omega\|_2$. The vector u found this way has the useful property that

$$\langle u, H(\lambda)u\rangle \longrightarrow 0 \text{ as } \lambda \longrightarrow -\lambda_1$$

4.8 The Rayleigh quotient trick

If H is symmetric and the vector $p \neq 0$, then the scalar

$$\frac{\langle p, Hp\rangle}{\langle p, p\rangle}$$

is known as the *Rayleigh quotient* of p. The Rayleigh quotient is important because it has the following property:

$$\lambda_{min}[H] \leq \frac{\langle p, Hp\rangle}{\langle p, p\rangle} \leq \lambda_{max}[H] \tag{4.23}$$

During the Cholesky factorization of $H(\lambda)$, we have encountered a negative pivot at the k^{th} stage of the decomposition for some $k \leq n$. The factorization has thus failed (H is indefinite). It is then possible to add $\delta = \sum_{j=1}^{k-1} l_{kj}^2 - h_{kk}(\lambda) \geq 0$ to the k^{th} diagonal of $H(\lambda)$ so that the leading k by k submatrix of

$$H(\lambda) + \delta e_k e_k^T$$

is singular. It's also easy to find a vector v for which

$$H(\lambda + \delta e_k e_k^T)v = 0 \tag{4.24}$$

using the Cholesky factors accumulated up to step k. Setting $v_j = 0$ for $j > k, v_k = 1$, and back-solving:

$$v_j = -\frac{\sum_{i=j+1}^{k} l_{ij} v_i}{l_{jj}} \text{ for } j = k-1, \ldots, 1$$

gives the required vector. We then obtain a lower bound on $-\lambda_1$ by forming the inner product of 4.24 with v, using the identity $\langle e_k, v\rangle = v_k = 1$ and recalling that the Rayleigh quotient is greater then $\lambda_{min} = \lambda_1$, we can write:

$$0 = \frac{\langle v, (H + \lambda I)v\rangle}{\langle v, v\rangle} + \delta\frac{\langle e_k, v\rangle^2}{\langle v, v\rangle} \geq \lambda + \lambda_1 + \frac{\delta}{\|v\|_2^2}$$

This implies the bound on λ_1:

$$\lambda + \frac{\delta}{\|v\|_2^2} \leq -\lambda_1$$

In the algorithm, we set $\lambda_L = \max[\lambda_L, \lambda + \frac{\delta}{\|v\|_2^2}]$

4.9 Termination Test.

If v is any vector such that $\|s(\lambda)+v\| = \Delta$, and if we have:

$$\langle v, H(\lambda)v\rangle \leq \ \kappa\ (\langle s(\lambda), H(\lambda)s(\lambda)\rangle + \lambda\Delta^2) \tag{4.25}$$

from some $\kappa \in (0,1)$ then $\hat{s} = s(\lambda)+v$ achieves the condition (see Equation 4.1):

$$q(\hat{s}) \leq (1-\kappa) q(s^*) \tag{4.26}$$

In other words, if κ is small, then the reduction in q that occurs at the point $\hat{s}$ is close to the greatest reduction that is allowed by the trust region constraint.

Proof

for any v, we have the identity:

$$q(s(\lambda)+v) = \langle g, s(\lambda)+v\rangle + \frac{1}{2}\langle s(\lambda)+v, H(s(\lambda)+v)\rangle$$

(using $H(\lambda) = H + \lambda I$:)

$$= \langle g, s(\lambda)+v\rangle + \frac{1}{2}\langle s(\lambda)+v, H(\lambda)(s(\lambda)+v)\rangle - \frac{1}{2}\lambda\|s(\lambda)+v\|_2^2$$

(using $H(\lambda)s(\lambda) = -g$:)

$$\begin{aligned}
&= -\langle H(\lambda)s(\lambda), (s(\lambda)+v)\rangle + \frac{1}{2}\langle s(\lambda)+v, H(\lambda)(s(\lambda)+v)\rangle - \frac{1}{2}\lambda\|s(\lambda)+v\|_2^2 \\
&= \frac{1}{2}\langle v, H(\lambda)v\rangle - \frac{1}{2}\langle s(\lambda), H(\lambda)s(\lambda)\rangle - \frac{1}{2}\lambda\|s(\lambda)+v\|_2^2
\end{aligned} \tag{4.27}$$

If we choose v such that $s(\lambda)+v = s^*$, we have:

$$q(s^*) \geq -\frac{1}{2}(\langle s(\lambda), H(\lambda)s(\lambda)\rangle + \lambda\|s(\lambda)+v\|_2^2) \geq -\frac{1}{2}(\langle s(\lambda), H(\lambda)s(\lambda)\rangle + \lambda\Delta^2)$$

$$\Rightarrow \quad -\frac{1}{2}(\langle s(\lambda), H(\lambda)s(\lambda)\rangle + \lambda\Delta^2) \leq q(s^*) \tag{4.28}$$

From 4.27, using the 2 hypothesis:

$$\begin{aligned}
q(s(\lambda)+v) &= \frac{1}{2}\langle v, H(\lambda)v\rangle - \frac{1}{2}\langle s(\lambda), H(\lambda)s(\lambda)\rangle - \frac{1}{2}\lambda\|s(\lambda)+v\|_2^2 \\
&= \frac{1}{2}\langle v, H(\lambda)v\rangle - \frac{1}{2}\langle s(\lambda), H(\lambda)s(\lambda)\rangle - \frac{1}{2}\lambda\Delta^2
\end{aligned}$$

(Using Equation 4.25:)

$$\begin{aligned}
&\leq \frac{1}{2}\ \kappa\ (\langle s(\lambda), H(\lambda)s(\lambda)\rangle + \lambda\Delta^2) - \frac{1}{2}\langle s(\lambda), H(\lambda)s(\lambda)\rangle - \frac{1}{2}\lambda\Delta^2 \\
&\leq -\frac{1}{2}\ (1-\kappa)\ (\langle s(\lambda), H(\lambda)s(\lambda)\rangle + \lambda\Delta^2)
\end{aligned} \tag{4.29}$$

Combining 4.28 and 4.29, we obtain finally 4.26.

4.9.1 $s(\lambda)$ is near the boundary of the trust region: normal case

Lemma

Suppose $|\|s(\lambda)\|_2 - \Delta| \leq \kappa_{easy}\Delta$, *then we have:*

$$q(s(\lambda)) < (1 - \kappa_{easy}^2) q(s^*) \tag{4.30}$$

From the hypothesis:

$$\begin{aligned} |\|s(\lambda)\|_2 - \Delta| &\leq \kappa_{easy}\Delta \\ \|s(\lambda)\|_2 &\geq (1 - \kappa_{easy})\Delta \end{aligned} \tag{4.31}$$

Combining 4.31 and 4.27 when $v = 0$ reveals that:

$$\begin{aligned} q(s(\lambda)) &= -\frac{1}{2}(\langle s(\lambda), H(\lambda)s(\lambda)\rangle + \lambda\|s(\lambda)\|_2^2) \\ &\leq -\frac{1}{2}(\langle s(\lambda), H(\lambda)s(\lambda)\rangle + \lambda(1-\kappa_{easy})^2\Delta^2) \\ &\leq -\frac{1}{2}(1-\kappa_{easy})^2(\langle s(\lambda), H(\lambda)s(\lambda)\rangle + \lambda\Delta^2) \end{aligned} \tag{4.32}$$

The required inequality 4.30 is immediate from 4.28 and 4.32.

We will use this lemma with $\kappa_{easy} = 0.1$.

4.9.2 $s(\lambda)$ is inside the trust region: hard case

We will choose $\hat{s}$ as (see paragraph containing Equation 4.15 for the meaning of α^* and u_1):

$$\hat{s} = s(\lambda) + \alpha^* u_1 \tag{4.33}$$

Thus, the condition for ending the trust region calculation simplifies to the inequality:

$$\alpha^2\langle u, H(\lambda)u\rangle < \kappa_{hard}(s(\lambda)^T H(\lambda)s(\lambda) + \lambda\Delta^2) \tag{4.34}$$

We will choose $\kappa_{hard} = 0.02$.

4.10 An estimation of the slope of $q(x)$ at the origin.

An estimation of the slope of $q(x)$ at the origin is given by λ_1. In the optimization program, we will only compute λ_1 when we have interior convergence. The algorithm to find λ_1 is the following:

1. Set $\lambda_L := 0$.
2. Set $\lambda_U := \min\left[\max_i \left[[H]_{i,i} + \sum_{i\neq j} |[H]_{i,j}|\right], \|H\|_F, \|H\|_\infty\right]$
3. Set $\lambda := \dfrac{\lambda_L + \lambda_U}{2}$

4. Try to factorize $H(-\lambda) = LL^T$.
   - **Success:** Set $\lambda_L := \lambda$
   - **Failure:** Set $\lambda_U := \lambda$
5. If $\lambda_L < 0.99\ \lambda_U$ go back to step 3.
6. The required value of λ_1 (=the approximation of the slope at the origin) is inside λ_L

Chapter 5

The secondary Trust-Region subproblem

The material of this chapter is based on the following reference: [Pow00].

We seek an approximation to the solution s^*, of the maximization problem:

$$\max_{s\in\Re^n} |q(x_k+s)| \equiv |f(x_k) + \langle g_k, s\rangle + \frac{1}{2}\langle s, H_k s\rangle|$$
$$\text{subject to } \|s\|_2 < \Delta$$

The following maximization problem is equivalent (after a translation) and will be discussed in the chapter:

$$\max_{s\in\Re^n} |q(s)| \equiv |\langle g, s\rangle + \frac{1}{2}\langle s, Hs\rangle| \tag{5.1}$$
$$\text{subject to } \|s\|_2 \le \Delta$$

We will indifferently use the term, polynomial, quadratic or model. The “trust region” is defined by the set of all points which respect the constraint $\|s\|_2 \le \Delta$.

Further, the shape of the trust region allows s to be replaced by $-s$, it's thus equivalent to consider the computation

$$\max_{s\in\Re^n} |\langle g, s\rangle| + \frac{1}{2}|\langle s, Hs\rangle| \tag{5.2}$$
$$\text{subject to } \|s\|_2 < \Delta$$

Now, if $\hat{s}$ and $\tilde{s}$ are the values that maximize $|\langle g, s\rangle|$ and $|\langle s, Hs\rangle|$, respectively, subject to $\|s\|_2 < \Delta$, then s may be an adequate solution of the problem 5.1, if it is the choice between $\pm\hat{s}$ and $\pm\tilde{s}$ that gives the largest value of the objective function of the problem. Indeed, for every feasible s, including the exact solution of the present computation, we find the elementary bound

$$\begin{aligned}|\langle g, s\rangle| + \frac{1}{2}|\langle s, Hs\rangle| &\le \left(|\langle g, \hat{s}\rangle| + \frac{1}{2}|\langle \hat{s}, H\hat{s}\rangle|\right) + \left(|\langle g, \tilde{s}\rangle| + \frac{1}{2}|\langle \tilde{s}, H\tilde{s}\rangle|\right) \\ &\le 2\max\left[|\langle g, \hat{s}\rangle| + \frac{1}{2}|\langle \hat{s}, H\hat{s}\rangle|,\ \ |\langle g, \tilde{s}\rangle| + \frac{1}{2}|\langle \tilde{s}, H\tilde{s}\rangle|\right] \qquad (5.3) \\ & \qquad (5.4)\end{aligned}$$

It follows that the proposed choice of s gives a value of $|q(s)|$ that is, at least, half of the optimal value. Now, $\hat{s}$ is the vector $\pm\rho g/\|g\|$, while $\tilde{s}$ is an eigenvector of an eigenvalue of H of largest modulus, which would be too expensive to compute. We will now discuss how to generate $\tilde{s}$. We will use a method inspired by the *power method* for obtaining large eigenvalues.

Because $|\langle\tilde{s}, H\tilde{s}\rangle|$ is large only if $\|H\tilde{s}\|$ is substantial, the technique begins by finding a column of H, ω say, that has the greatest Euclidean norm. Hence letting $v_1, v_2, \ldots, v_n$ be the columns of the symmetric matrix H, we deduce the bound

$$\begin{aligned}\|H\omega\| &\geq \|\omega\|^2 = \max_k\{\|v_k\| : k = 1, \ldots, n\}\|\omega\| \\ &\geq \|\omega\|\sqrt{\frac{1}{n}\sum_{k=1}^{n}\|v_k\|^2} \qquad (5.5)\\ &\geq \frac{\|\omega\|}{\sqrt{n}}\,\sigma(H) \qquad (5.6)\end{aligned}$$

Where $\sigma(H)$ is the spectral radius of H. It may be disastrous, however to set $\tilde{s}$ to a multiple of ω, because $\langle\omega, H\omega\rangle$ is zero in the case:

$$H = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & -2/3 & -2/3 \\ 1 & -2/3 & -1 & -2/3 \\ 1 & -2/3 & -2/3 & -1 \end{pmatrix} \qquad (5.7)$$

Therefore, the algorithm picks $\tilde{s}$ from the two dimensional linear subspace of $\Re^n$ that is spanned by ω and $H\omega$. Specifically, $\tilde{s}$ has the form $\alpha\omega + \beta H\omega$, where the ratio α/β is computed to maximize the expression

$$\frac{|\langle\alpha\omega + \beta H\omega, H(\alpha\omega + \beta H\omega)\rangle|}{\|\alpha\omega + \beta H\omega\|^2} \qquad (5.8)$$

which determines the direction of $\tilde{s}$. Then the length of $\tilde{s}$ is defined by $\|\tilde{s}\| = \rho$, the sign of $\tilde{s}$, being unimportant.

5.1 Generating $\tilde{s}$.

Let us define $V := \omega$, $D := H\omega$, $r := \beta/\alpha$, equation 5.8, can now be rewritten:

$$\begin{aligned}\frac{(V + rD)^T H(V + rD)}{(V + rD)^2} &= \frac{V^T HV + rV^T HD + rD^T HV + r^2 D^T HD}{V^2 + 2rV^T D + r^2 D^2} \\ &= \frac{r^2 D^T HD + 2rV^T HD + V^T HV}{V^2 + 2rV^T D + r^2 D^2} = f(r)\end{aligned}$$

We will now search for r^*, root of the Equation

$$\begin{aligned}&\frac{\partial f(r^*)}{\partial r} = 0 \\ \Leftrightarrow\ &(2rD^T HD + 2V^T HD)(V^2 + 2rV^T D + r^2 D^2) \\ &\quad - (r^2 D^T HD + 2rV^T HD + V^T HV)(2rD^2 + 2V^T D) = 0 \\ \Leftrightarrow\ &\Big[(D^T HD)(V^T D) - D^2 D^2\Big] r^2 + \Big[(D^T HD)V^2 - D^2(V^T D)\Big] r + \Big[D^2 V^2 - (V^T D)^2\Big] = 0\end{aligned}$$

(Using the fact that $D = HV \Leftrightarrow D^T = V^T H^T = V^T H$.)

We thus obtain a simple equation $ax^2 + bx + c = 0$. We find the two roots of this equation and choose the one r^* which maximize 5.8. $\tilde{s}$ is thus $V + r^* D$.

5.2 Generating $\hat{u}$ and $\tilde{u}$ from $\hat{s}$ and $\tilde{s}$

Having generated $\tilde{s}$ and $\hat{s}$ in the ways that have been described, the algorithm sets s to a linear combination of these vectors, but the choice is not restricted to $\pm\hat{s}$ or $\pm\tilde{s}$ as suggested in the introduction of this chapter(unless $\tilde{s}$ and $\hat{s}$ are nearly or exactly parallel). Instead, the vectors $\hat{u}$ and $\tilde{u}$ of unit length are found in the span of $\tilde{s}$ and $\hat{s}$ that satisfy the condition $\hat{u}^T\tilde{u} = 0$ and $\hat{u}^T H\tilde{u} = 0$. The final s will be a combination of $\hat{u}$ and $\tilde{u}$.

If we set:

$$\begin{cases} G = \tilde{s} \\ V = \hat{s} \end{cases}$$

We have

$$\begin{cases} \tilde{u} = \cos(\theta)G + \sin(\theta)V \\ \hat{u} = -\sin(\theta)G + \cos(\theta)V \end{cases} \qquad \text{we have directly } \hat{u}^T\tilde{u} = 0 \tag{5.9}$$

We will now find θ such that $\hat{u}^T H\tilde{u} = 0$:

$$\begin{aligned} &\hat{u}^T H\tilde{u} = 0 \\ \Leftrightarrow &(-\sin(\theta)G + \cos(\theta)V)^T H(\cos(\theta)G + \sin(\theta)V) = 0 \\ \Leftrightarrow &(\cos^2(\theta) - \sin^2(\theta))V^T HG + (G^T HG - V^T HV)\sin(\theta)\cos(\theta) = 0 \end{aligned}$$

Using

$$\begin{cases} \sin(2\theta) = 2\sin(\theta)\cos(\theta) \\ \cos(2\theta) = \cos^2(\theta) - \sin^2(\theta) \\ tg(\theta) = \frac{\sin(\theta)}{\cos(\theta)} \end{cases}$$

We obtain

$$\begin{aligned} &(V^T HG)\cos(2\theta) + \frac{G^T HG - V^T HV}{2}\sin(2\theta) = 0 \\ \Leftrightarrow &\theta = \frac{1}{2} arctg\left(\frac{2V^T HG}{V^T HV - G^T HG}\right) \end{aligned} \tag{5.10}$$

Using the value of θ from Equation 5.10 in Equation 5.9 give the required $\hat{u}$ and $\tilde{u}$.

5.3 Generating the final s from $\hat{u}$ and $\tilde{u}$.

The final s has the form $s = \rho\Big(cos(\phi)\ \hat{u} + sin(\phi)\ \tilde{u}\Big)$ and ϕ has one of the following values: $\{0, \frac{\pi}{4}, \frac{\pi}{2}, \frac{3\pi}{4}, \pi, \frac{-\pi}{4}, \frac{-\pi}{2}, \frac{-3\pi}{4}\}$. We will choose the value of ϕ which maximize 5.2.

5.4 About the choice of $\tilde{s}$

The choice of $\tilde{s}$ is never bad because it achieves the property

$$|\tilde{s}^T H \tilde{s}| \geq \frac{1}{2} \frac{1}{\sqrt{n}} \sigma(H) \rho^2 \tag{5.11}$$

The proof will be skipped.

Chapter 6

The CONDOR unconstrained algorithm.

I strongly suggest the reader to read first the Section 2.4 which presents a global, simplified view of the algorithm. Thereafter, I suggest to read this section, disregarding the parallel extensions which are not useful to understand the algorithm.

Let n be the dimension of the search space.
Let $f(x)$ be the objective function to minimize.
Let x_{start} be the starting point of the algorithm.
Let ρ_{start} and ρ_{end} be the initial and final value of the global trust region radius.
Let $noise_a$ and $noise_r$, be the absolute and relative errors on the evaluation of the objective function.

1. Set $\Delta = \rho$, $\rho = \rho_{start}$ and generate a first interpolation set $\{\boldsymbol{x}_{(1)}, \ldots, \boldsymbol{x}_{(N)}\}$ around x_{start} (with $N = (n+1)(n+2)/2$), using technique described in section 3.4.5 and evaluate the objective function at these points.

   Parallel extension: do the N evaluations in parallel in a cluster of computers

2. Choose the index k of the best (lowest) point of the set $\mathcal{J} = \{\boldsymbol{x}_{(1)}, \ldots, \boldsymbol{x}_{(N)}\}$. Let $x_{(base)} := \boldsymbol{x}_{(k)}$. Set $F_{old} := f(x_{(base)})$. Apply a translation of $-x_{(base)}$ to all the dataset $\{\boldsymbol{x}_{(1)}, \ldots, \boldsymbol{x}_{(N)}\}$ and generate the polynomial $q(x)$ of degree 2, which intercepts all the points in the dataset (using the technique described in Section 3.3.2).

3. *Parallel extension:* Start the parallel process: make a local copy $q_{copy}(x)$ of $q(x)$ and use it to choose good sampling site using Equation 3.38 on $q_{copy}(x)$.

4. *Parallel extension:* Check the results of the computation made by the parallel process. Update $q(x)$ using all these evaluations. We will possibly have to update the index k of the best point in the dataset and F_{old}. Replace q_{copy} with a fresh copy of $q(x)$.

5. Find the trust region step s^*, the solution of $\min_{s\in\Re^n} q(\boldsymbol{x}_{(k)}+s)$ subject to $\|s\|_2 < \Delta$, using the technique described in Chapter 4.

   In the constrained case, the trust region step s^* is the solution of:

$$\min_{s\in\Re^n} q(\boldsymbol{x}_{(k)}+s) \equiv f(x_k) + \langle g_k, s\rangle + \frac{1}{2}\langle s, H_k s\rangle$$
$$\text{subject to} \begin{cases} b_l \leq x \leq b_u, & b_l, b_u \in \Re^n \\ Ax \geq b, & A \in \Re^{m\times n}, b \in \Re^m \\ c_i(x) \geq 0, & i = 1,\dots,l \\ \|s\|_2 < \Delta \end{cases} \tag{6.1}$$

   where b_l, b_u are the box constraints, $Ax \geq b$ are the linear constraints and $c_i(x) \geq$ are the non-linear constraints.

6. If $\|s\| < \dfrac{\rho}{2}$, then break and go to step 16: we need to be sure that the model is valid before doing a step so small.

7. Let $R := q(\boldsymbol{x}_{(k)}) - q(\boldsymbol{x}_{(k)} + s^*) \geq 0$, the predicted reduction of the objective function.

8. Let $noise := \frac{1}{2}\max[noise_a * (1 + noise_r), noise_r |f(\boldsymbol{x}_{(k)})|]$. If $(R < noise)$, break and go to step 16.

9. Evaluate the objective function $f(x)$ at point $x_{(base)} + \boldsymbol{x}_{(k)} + s^*$. The result of this evaluation is stored in the variable F_{new}.

10. Compute the agreement r between $f(x)$ and the model $q(x)$:

$$r = \frac{F_{old} - F_{new}}{R} \tag{6.2}$$

11. Update the local trust region radius: change Δ to:

$$\begin{cases} \max[\Delta, \frac{5}{4}\|s\|, \rho + \|s\|] & \text{if } 0.7 \leq r, \\ \max[\frac{1}{2}\Delta, \|s\|] & \text{if } 0.1 \leq r < 0.7, \\ \frac{1}{2}\|s\| & \text{if } r < 0.1 \end{cases} \tag{6.3}$$

    If $(\Delta < \dfrac{1}{2}\rho)$, set $\Delta := \rho$.

12. Store $\boldsymbol{x}_{(k)} + s^*$ inside the interpolation dataset: choose the point $\boldsymbol{x}_{(t)}$ to remove using technique of Section 3.4.3 and replace it by $\boldsymbol{x}_{(k)} + s^*$ using the technique of Section 3.4.4. Let us define the $ModelStep := \|\boldsymbol{x}_{(t)} - (\boldsymbol{x}_{(k)} + s^*)\|$

13. Update the index k of the best point in the dataset. Set $F_{new} := \min[F_{old}, F_{new}]$.

14. Update the value of M which is used during the check of the validity of the polynomial around $\boldsymbol{x}_{(k)}$ (see Section 3.4.1 and more precisely Equation 3.34).

15. If there was an improvement in the quality of the solution OR if $(\|s^*\| > 2\rho)$ OR if $ModelStep > 2\rho$ then go back to point 4.

16. *Parallel extension:* same as point 4.

17. We must now check the validity of our model using the technique of Section 3.4.2. We will need, to check this validity, a parameter ϵ: see Section 6.1 to know how to compute it.

    - **Model is invalid:** We will improve the quality of our model $q(x)$. We will remove the worst point $\boldsymbol{x}_{(j)}$ of the dataset and replace it by a better point (we must also update the value of M if a new function evaluation has been made). This algorithm is described in Section 3.4.2. We will possibly have to update the index k of the best point in the dataset and F_{old}. Once this is finished, return to step 4.
    - **Model is valid** If $\|s^*\| > \rho$ return to step 4, otherwise continue.

18. If $\rho = \rho_{end}$, we have nearly finished the algorithm: go to step 21, otherwise continue to the next step.

19. Update of the global trust region radius.

$$\rho_{new} = \begin{cases} \rho_{end} & \text{if } \rho_{end} < \rho \leq 16\rho_{end} \\ \sqrt{\rho_{end}\,\rho} & \text{if } 16\rho_{end} < \rho \leq 250\rho_{end} \\ 0.1\rho & \text{if } 250\rho_{end} < \rho \end{cases} \tag{6.4}$$

    Set $\Delta := \max[\frac{\rho}{2}, \rho_{new}]$. Set $\rho := \rho_{new}$.

20. Set $x_{(base)} := x_{(base)} + \boldsymbol{x}_{(k)}$. Apply a translation of $-\boldsymbol{x}_{(k)}$ to $q(x)$, to the set of Newton polynomials P_i which defines $q(x)$ (see Equation 3.26) and to the whole dataset $\{\boldsymbol{x}_{(1)}, \dots, \boldsymbol{x}_{(N)}\}$. Go back to step 4.

21. The iteration are now complete but one more value of $f(x)$ may be required before termination. Indeed, we recall from step 6 and step 8 of the algorithm that the value of $f(x_{(base)} + \boldsymbol{x}_{(k)} + s^*)$ has maybe not been computed. Compute $F_{new} := f(x_{(base)} + \boldsymbol{x}_{(k)} + s^*)$.

    - if $F_{new} < F_{old}$, the solution of the optimization problem is $x_{(base)} + \boldsymbol{x}_{(k)} + s^*$ and the value of f at this point is F_{new}.
    - if $F_{new} > F_{old}$, the solution of the optimization problem is $x_{(base)} + \boldsymbol{x}_{(k)}$ and the value of f at this point is F_{old}.

Notice the simplified nature of the trust-region update mechanism of ρ (step 16). This is the formal consequence of the observation that the trust-region radius should not be reduced if the model has not been guaranteed to be valid in the trust region $\delta_k \leq \Delta_k$.

6.1 The bound ϵ.

See Section 3.4.2 to know about ϵ.

If we have updated the value of M less than 10 times, we will set $\epsilon := 0$ (see Section 3.4.1 to know about M). This is because we are not sure of the value of M if it has been updated less than 10 times.

If the step size $\|s^*\|$ we have computed at step 3 of the CONDOR algorithm is $\|s^*\| \geq \frac{\rho}{2}$, then $\epsilon = 0$.

Otherwise, $\epsilon = \frac{1}{2}\rho^2\lambda_1$, where λ_1 is an estimate of the slope of $q(x)$ around $\boldsymbol{x}_{(k)}$ (see Section 4.10 to know how to compute λ_1).

We see that, when the slope is high, we permit a more approximative (= big value of ϵ) model of the function.

6.2 Note about the validity check.

When the computation for the current ρ is complete, we check the model (see step 14 of the algorithm) around $\boldsymbol{x}_{(k)}$, then one or both of the conditions:

$$\|\boldsymbol{x}_{(j)} - \boldsymbol{x}_{(k)}\| \leq 2\rho \tag{6.5}$$

$$\frac{1}{6}M\|\boldsymbol{x}_{(j)} - \boldsymbol{x}_{(k)}\|^3 \max_d\{|P_j(\boldsymbol{x}_{(k)} + d)| : \|d\| \leq \rho\} \leq \epsilon \tag{6.6}$$

must hold for every points in the dataset. When ρ is reduced by formula 6.4, the equation 6.5 is very often NOT verified. Only Equation 6.6, prevents the algorithm from sampling the model at $N = (n+1)(n+2)/2$ new points. Numerical experiments indicate that the algorithm is highly successful in that it computes less then $\frac{1}{2}n^2$ new points in most cases.

6.3 The parallel extension of CONDOR

We will use a client-server approach. The main node, the server will have two concurrent process:

- The **main process** on the main computer is the classical non-parallelized version of the algorithm, described at the beginning of Chapter 6. There is an exchange of information with the second/parallel process on steps 4 and 16 of the original algorithm.
- The goal of the **second/parallel process** on the main computer is to increase the quality of the model $q_k(s)$ by using client computers to sample $f(x)$ at specific interpolation sites.

The client nodes are performing the following:

1. Wait to receive from the second/parallel process on the server a sampling site (a point).
2. Evaluate the objective function at this site and return immediately the result to the server.
3. Go to step 1.

Several strategies have been tried to select good sampling sites. We describe here the most promising one. The second/parallel task is the following:

A. Make a local copy $q_{(copy)}(s)$ of $q_k(s)$ (and of the associated Lagrange Polynomials $P_j(x)$)

B. Make a local copy $\mathcal{J}_{(copy)}$ of the dataset $\mathcal{J} = \{\boldsymbol{x}_{(1)}, \ldots, \boldsymbol{x}_{(N)}\}$.

C. Find the index j of the point inside $\mathcal{J}_{(copy)}$ the further away from $\boldsymbol{x}_{(k)}$.

D. Replace $\boldsymbol{x}_{(j)}$ by a better point which will increase the quality of the approximation of $f(x)$. The computation of this point is done using Equation 3.38: $\boldsymbol{x}_{(j)}$ is replaced in $\mathcal{J}_{(copy)}$ by $\boldsymbol{x}_{(k)} + d$ where d is the solution of the following problem:

$$\max_d \{ |P_{j,(copy)}(\boldsymbol{x}_{(k)} + d)| : \|d\| \leq \rho \} \tag{6.7}$$

E. Ask for an evaluation the objective function at point $\boldsymbol{x}_{(k)} + d$ using a free client computer to performs the evaluation. If there is still a client doing nothing, go back to step **C**.

F. Wait for a node to complete its evaluation of the objective function $f(x)$.

G. Update $q_{(copy)}(x)$ using this new evaluation. Remove j from $\mathcal{J}_{(copy)}$. go to step **C**.

In the parallel/second process we are always working on a copy of $q_k(x)$, $\mathcal{J}$, $P_{j,(copy)}(x)$ to avoid any side effect with the main process which is guiding the search. The communication and exchange of information between these two processes are done only at steps 4 and 16 of the main algorithm described in the previous section. Each time the main loop (main process) checks the results of the parallel computations the following is done:

i. Wait for the parallel/second task to enter the step **F** described above and block the parallel task inside this step **F** for the time needed to perform the points **ii** and **iii** below.

ii. Update of $q_k(s)$ using all the points calculated in parallel, discarding the points that are too far away from x_k (at a distance greater than ρ). This update is performed using technique described in Section 3.4.3 and Section 3.4.4. We will possibly have to update the index k of the best point in the dataset $\mathcal{J}$ and F_{old}.

iii. Perform operations described in point **A** & **B** of the parallel/second task algorithm above: " Copy $q_{(copy)}(x)$ from $q(x)$. Copy $\mathcal{J}_{(copy)}$ from $\mathcal{J} = \{\boldsymbol{x}_{(1)}, \ldots, \boldsymbol{x}_{(N)}\}$ ".

Chapter 7

Numerical Results of CONDOR.

7.1 Random objective functions

We will use for the tests, the following objective function:

$$f(x) = \sum_{i=1}^{n} \left[a_i - \sum_{j=1}^{n} (S_{ij} \sin x_j + C_{ij} \cos x_j) \right]^2, x \in \Re^n \tag{7.1}$$

The way of generating the parameters of $f(x)$ is taken from [RP63], and is as follows. The elements of the $n \times n$ matrices S and C are random integers from the interval $[-100,\ 100]$, and a vector x^* is chosen whose components are random numbers from $[-\pi,\ \pi]$. Then, the parameters $a_i, i = 1, \dots, n$ are defined by the equation $f(x^*) = 0$, and the starting vector x_{start} is formed by adding random perturbations of $[-0.1\pi,\ 0.1\pi]$ to the components of x^*. All distributions of random numbers are uniform. There are two remarks to do on this objective function:

- Because the number of terms in the sum of squares is equals to the number of variables, it happens often that the Hessian matrix H is ill-conditioned around x^*.
- Because $f(x)$ is periodic, it has many saddle points and maxima.

Using this test function, it is possible to cover every kind of problems, (from the easiest one to the most difficult one).

We will compare the CONDOR algorithm with an older algorithm: "CFSQP". CFSQP uses line-search techniques. In CFSQP, the Hessian matrix of the function is reconstructed using a $BFGS$ update, the gradient is obtained by finite-differences.

Parameters of CONDOR: $\rho_{start} = 0.1 \quad \rho_{end} = 10^{-8}$.
Parameters of $CFSQP$: $\epsilon = 10^{-10}$. The algorithm stops when the step size is smaller than ϵ.

Recalling that $f(x^*) = 0$, we will say that we have a success when the value of the objective function at the final point of the optimization algorithm is lower then 10^{-9}.

We obtain the following results, after 100 runs of both algorithms:

Dimension n of the space	Mean number of function evaluations		Number of success		Mean best value of the objective function	
	CONDOR	CFSQP	CONDOR	CFSQP	CONDOR	CFSQP
3	44.96	246.19	100	46	3.060873e-017	5.787425e-011
5	99.17	443.66	99	27	5.193561e-016	8.383238e-011
10	411.17	991.43	100	14	1.686634e-015	1.299753e-010
20	1486.100000	—	100	—	3.379322e-016	—-

We can now give an example of execution of the algorithm to illustrate the discussion of Section 6.2:

Rosenbrock's function ($n = 2$)		
function evaluations	Best Value So Far	ρ_{old}
33	5.354072×10^{-1}	10^{-1}
88	7.300849×10^{-8}	10^{-2}
91	1.653480×10^{-8}	10^{-3}
94	4.480416×10^{-11}	10^{-4}
97	4.906224×10^{-17}	10^{-5}
100	7.647780×10^{-21}	10^{-6}
101	7.647780×10^{-21}	10^{-7}
103	2.415887×10^{-30}	10^{-8}

With the *Rosenbrock's function*$= 100 * (x_1 - x_0^2)^2 + (1 - x_0)^2$

We will use the same choice of parameters (for ρ_{end} and ρ_{start}) as before. The starting point is $(-1.2\ ;\ 1.0)$.

As you can see, the number of evaluations performed when ρ is reduced is far inferior to $(n+1)(n+2)/2 = 6$.

7.2 Hock and Schittkowski set

The tests problems are arbitrary and have been chosen by A.R.Conn, K. Scheinberg and Ph.L. Toint. to test their DFO algorithm. The performances of DFO are thus expected to be, at least, good. We list the number of function evaluations that each algorithm took to solve the problem. We also list the final function values that each algorithm achieved. We do not list the CPU time, since it is not relevant in our context. The "*" indicates that an algorithm terminated early because the limit on the number of iterations was reached. The default values for all the parameters of each algorithm is used. The stopping tolerance of DFO was set to 10^{-4}, for the other algorithms the tolerance was set to appropriate comparable default values. The comparison between the algorithms is based on the number of function evaluations needed to reach the SAME precision. For the most fair comparison with DFO, the stopping criteria (ρ_{end}) of CONDOR has been chosen so that CONDOR is always stopping with a little more precision on the result than DFO. This precision is sometime insufficient to reach the true optima of the objective function.

In particular, in the case of the problems GROWTHLS and HEART6LS, the CONDOR algorithm can find a better optimum after some more evaluations (for a smaller ρ_{end}). All algorithms were implemented in Fortran 77 in double precision except COBYLA which is implemented in Fortran 77 in single precision and CONDOR which is written in C++ (in double precision). The trust region minimization subproblem of the DFO algorithm is solved by NPSOL [GMSM86], a fortran 77 non-linear optimization package that uses an SQP approach. For CONDOR, the number in parenthesis indicates the number of function evaluation needed to reach the optimum without being assured that the value found is the real optimum of the function. For example, for the WATSON problem, we find the optimum after (580) evaluations. CONDOR still continues to sample the objective function, searching for a better point. It's loosing 87 evaluations in this search. The total number of evaluation (reported in the first column) is thus 580+87=667.

CONDOR and UOBYQA are both based on the same ideas and have nearly the same behavior. Small differences can be due to the small difference between algorithms of Chapter 4&5 and the algorithms used inside UOBYQA.

PDS stands for "*Parallel Direct Search*" [DT91]. The number of function evaluations is high and so the method doesn't seem to be very attractive. On the other hand, these evaluations can be performed on several CPU's reducing considerably the computation time.

Lancelot [CGT92] is a code for large scale optimization when the number of variable is $n > 10000$ and the objective function is easy to evaluate (less than $1ms.$). Its model is build using finite differences and BFGS update. This algorithm has not been design for the kind of application we are interested in and is thus performing accordingly.

COBYLA [Pow94] stands for "*Constrained Optimization by Linear Approximation*" by Powell. It is, once again, a code designed for large scale optimization. It is a derivative free method, which uses linear polynomial interpolation of the objective function.

DFO [CST97, CGT98] is an algorithm by A.R.Conn, K. Scheinberg and Ph.L. Toint. It's very similar to UOBYQA and CONDOR. It has been specially designed for small dimensional problems and high-computing-load objective functions. In other words, it has been designed for the same kind of problems that CONDOR. DFO also uses a model build by interpolation. It is using a Newton polynomial instead of a Lagrange polynomial. When the DFO algorithm starts, it builds a linear model (using only $n + 1$ evaluations of the objective function; n is the dimension of the search space) and then directly uses this simple model to guide the research into the space. In DFO, when a point is "too far" from the current position, the model could be *invalid* and could not represent correctly the local shape of the objective function. This "far point" is rejected and replaced by a closer point. This operation unfortunately requires an evaluation of the objective function. Thus, in some situation, it is preferable to lower the degree of the polynomial which is used as local model (and drop the "far" point), to avoid this evaluation. Therefore, DFO is using a polynomial of degree oscillating between 1 and a "full" 2. In UOBYQA and CONDOR, we use the Moré and Sorenson algorithm [MS83, CGT00c] for the computation of the trust region step. It is very stable numerically and give *very high* precision results. On the other hand, DFO uses a general purpose tool (NPSOL [GMSM86]) which gives high quality results *but* that cannot be compared to the Moré and Sorenson algorithm when precision is critical. An other critical difference between DFO and CONDOR/UOBYQA is the formula used to update the local model. In DFO, the quadratical model built at each iteration is not defined uniquely.

Name	Dim	Number of Function Evaluation						final function value					
		CONDOR	UOB.	DFO	PDS	LAN.	COB.	CONDOR	UOBYQA	DFO	PDS	LANCELOT	COBYLA
ROSENBR	2	82 (80)	87	81	2307	94	8000	2.0833e-08	4.8316e-08	1.9716e-07	1.2265e-07	5.3797e-13	4.6102e+04*
SNAIL	2	316 (313)	306	246	2563	715	8000	9.3109e-11	1.8656e-10	1.2661e-08	2.6057e-10	4.8608e+00	7.2914e+00*
SISSER	2	40 (40)	31	27	1795	33	46	8.7810e-07	2.5398e-07	1.2473e-06	9.3625e-20	1.3077e-08	1.1516e-20
CLIFF	2	145 (81)	127	75	3075	84	36	1.9978e-01	1.9978e-01	1.9979e-01	1.9979e-01	1.9979e-01	2.0099e-01
HAIRY	2	47 (47)	305	51	2563	357	3226	2.0000e+01	2.0000e+01	2.0000e+01	2.0000e+01	2.0000e+01	2.0000e+01
PFIT1LS	3	153 (144)	158	180	5124	216	8000	2.9262e-04	1.5208e-04	4.2637e-04	3.9727e-06	1.1969e+00	2.8891e-02*
HATFLDE	3	96 (89)	69	95	35844	66	8000	5.6338e-07	6.3861e-07	3.8660e-06	1.7398e-05	5.1207e-07	3.5668e-04*
SCHMVETT	3	32 (31)	39	53	2564	32	213	-3.0000e+00	3.0000e+00	-3.0000e+00	-3.0000e+00	-3.0000e+00	-3.0000e+00
GROWTHLS	3	104 (103)	114	243	2308	652	6529	1.2437e+01	1.2446e+01	1.2396e+01	1.2412e+01	1.0040e+00	1.2504e+01
GULF	3	170 (160)	207	411	75780	148	8000	2.6689e-09	3.8563e-08	1.4075e-03	3.9483e-02	7.0987e-17	6.1563e+00*
BROWNDEN	4	91 (87)	107	110	5381	281	540	8.5822e+04	8.5822e+04	8.5822e+04	8.5822e+04	8.5822e+04	8.5822e+04
EIGENALS	6	123 (118)	119	211	5895	35	1031	3.8746e-09	2.4623e-07	9.9164e-07	1.1905e-05	2.0612e-16	7.5428e-08
HEART6LS	6	346 (333)	441	1350	37383	6652	8000	4.3601e-01	4.0665e-01	4.3167e-01	1.6566e+00	4.1859e-01	4.1839e+00*
BIGGS6	6	284 (275)	370	1364	31239	802	8000	1.1913e-05	7.7292e-09	1.7195e-05	7.5488e-05	8.4384e-12	8.3687e-04*
HART6	6	64 (64)	64	119	6151	57	124	-3.3142e+00	-3.2605e+00	-3.3229e+00	-3.3229e+00	-3.3229e+00	-3.3229e+00
CRAGGLVY	10	545 (540)	710	1026	13323	77	1663	1.8871e+00	1.8865e+00	1.8866e+00	1.8866e+00	1.8866e+00	1.8866e+00
VARDIM	10	686 (446)	880	2061	33035	165	4115	8.7610e-13	1.1750e-11	2.6730e-07	8.5690e-05	1.8092e-26	4.2233e-06
MANCINO	10	184 (150)	143	276	11275	88	249	3.7528e-09	6.1401e-08	1.5268e-07	2.9906e-04	2.2874e-16	2.4312e-06
POWER	10	550 (494)	587	206	13067	187	368	9.5433e-07	2.0582e-07	2.6064e-06	1.6596e-13	8.0462e-09	6.8388e-18
MOREBV	10	110 (109)	113	476	75787	8000	8000	1.0100e-07	1.6821e-05	6.0560e-07	1.0465e-05	1.9367e-13	2.2882e-06*
BRYBND	10	505 (430)	418	528	128011	8000	8000	4.4280e-08	1.2695e-05	9.9818e-08	1.9679e-02	7.5942e-15	8.2470e-03*
BROWNAL	10	331 (243)	258	837	14603	66	103	4.6269e-09	4.1225e-08	9.2867e-07	1.3415e-03	1.1916e-11	9.3470e-09
DQDRTIC	10	201 (79)	80	403	74507	33	7223	2.0929e-18	1.1197e-20	1.6263e-20	1.1022e-04	1.6602e-23	3.8218e-06
WATSON	12	667 (580)	590	1919	76813	200	8000	7.9451e-07	2.1357e-05	4.3239e-05	2.5354e-05	2.0575e-07	7.3476e-04*
DIXMAANK	15	964 (961)	1384	1118	63504	2006	2006	1.0000e+00	1.0000e+00	1.0000e+00	1.0000e+00	1.0000e+00	1.0001e+00
FMINSURF	16	695 (615)	713	1210	21265	224	654	1.0000e+00	1.0000e+00	1.0000e+00	1.0000e+00	1.0000e+00	1.0000e+00
Total Number of Function Evaluation		7531 (6612)	8420	14676	> 20000								

Figure 7.1: Comparative results between CONDOR, UOBYQA, DFO, PDS, LANCELOT and COBYLA on one CPU.

For a unique quadratical model in n variables one needs at least $\frac{1}{2}(n+1)(n+2) = N$ points and their function values. "In DFO, models are often build using many fewer points and such models are not uniquely defined" (citation from [CGT98]). The strategy used inside DFO is to select the model with the smallest Frobenius norm of the Hessian matrix. This update is highly numerically instable [Pow04]. Some recent research at this subject have maybe found a solution [Pow04] but this is still "work in progress". The model DFO is using can thus be very inaccurate.

In CONDOR and in UOBYQA the *validity* of the model is checked using the two Equations 6.5 and 6.6, which are restated here for clarity:

$$\begin{array}{l l}
\textit{All the interpolation points must} & \\
\textit{be close to the current point } \boldsymbol{x}_{(k)} & : \|\boldsymbol{x}_{(j)} - \boldsymbol{x}_{(k)}\| \leq 2\rho \qquad j = 1, \ldots, N \\
\textit{Powell's} & \\
\textit{heuristic} & : \frac{M}{6}\|\boldsymbol{x}_{(j)} - \boldsymbol{x}_{(k)}\|^3 \max_d \{|P_j(\boldsymbol{x}_{(k)} + d)| : \|d\| \leq \rho\} \leq \epsilon \quad j = 1, \ldots, N
\end{array}$$

The first equation (6.5) is also used in DFO. The second equation (6.6) is NOT used in DFO. This last equation allows us to "keep far points" inside the model, still being assured that it is valid. It allows us to have a "full" polynomial of second degree for a "cheap price". The DFO algorithm cannot use equation 6.6 to check the validity of its model because the variable ϵ (which is computed in UOBYQA and in CONDOR as a by-product of the computation of the "Moré and Sorenson Trust Region Step") is not cheaply available. In DFO, the trust region step is calculated using an external tool: NPSOL [GMSM86]. ϵ is difficult to obtain and is not used.

UOBYQA and CONDOR are always using a full quadratical model. This enables us to compute Newton's steps. The Newton's steps have a proven quadratical convergence speed [DS96]. Unfortunately, some evaluations of the objective function are lost to build the quadratical model. So, we only obtain *near* quadratic speed of convergence. We have Q-superlinear convergence (see original paper of Powell [Pow00]). (In fact the convergence speed is often directly proportional to the quality of the approximation H_k of the real Hessian matrix of $f(x)$). Usually, the price (in terms of number of function evaluations) to construct a good quadratical model is very high but using equation (6.6), UOBYQA and CONDOR are able to use very few function evaluations to update the local quadratical model.

When the dimension of the search space is greater than 25, the time needed to start, building the first quadratic, is so important (N evaluations) that DFO may becomes attractive again. Especially, if you don't want the optimum of the function but only a small improvement in a small time. If several CPU's are available, then CONDOR once again imposes itself. The function evaluations needed to build the first quadratic are parallelized on all the CPU's without any loss of efficiency when the number of CPU increases (the maximum number of CPU is $N+1$). This first construction phase has a great parallel efficiency, as opposed to the rest of the optimization algorithm where the efficiency becomes soon very low (with the number of CPU increasing). In contrast to CONDOR, the DFO algorithm has a very short initialization phase and a long research phase. This last phase can't be parallelized very well. Thus, when the number of CPU's is high, the most promising algorithm for parallelization is CONDOR. A parallel version of CONDOR has been implemented. Very encouraging experimental results on the parallel code are given in the next section.

When the local model is not convex, no second order convergence proof (see [CGT00d]) is available. It means that, when using a linear model, the optimization process can prematurely stop.

This phenomenon *can* occur with DFO which uses from time to time a simple linear model. CONDOR is very robust and always converges to a local optimum (extensive numerical tests have been made [VB04]).

From the numerical results, the CONDOR algorithm (on 1 CPU) outperforms the DFO algorithm when the dimension of the search space is greater than two. This result can be explained by the fact that, most of the time, DFO uses a simple linear approximation (with few or no second degree terms) of the objective function to guide its search. This poor model gives "sufficiently" good search directions when $n = 2$. But when $n > 2$, the probability to choose a bad search direction is higher. The high instability of the least-Frobenius-norm update of the model used in DFO can also give poor model, degrading the speed of the algorithm.

7.3 Parallel results on the Hock and Schittkowski set

We are using the same test conditions as in the previous section (standard objective functions with standard starting points).

Since the objective function is assumed to be time-expensive to evaluate, we can neglect the time spent inside the optimizer and inside the network transmissions. To be able to make this last assumption (negligible network transmissions times), a wait loop of 1 second is embedded inside the code used to evaluate the objective function (only 1 second: to be in the worst case possible).

Table 7.2 indicates the number of function evaluations performed on the master CPU (to obtain approximatively the total number of function evaluations cumulated over the master and all the slaves, multiply the given number on the list by the number of CPU's). The CPU time is thus directly proportional to the numbers listed in columns 3 to 5 of the Table 7.2.

Suppose a function evaluation takes 1 hour. The parallel/second process on the main computer has asked 59 minutes ago to a client to perform one such evaluation. We are at step 4(a)i of the main algorithm. We see that there are no new evaluation available from the client computers. Should we go directly to step 4(a)ii and use later this new information, or wait 1 minute? The response is clear: wait a little. This bad situation occurs very often in our test examples since every function evaluation takes exactly the same time (1 second). But what's the best strategy when the objective function is computing, randomly, from 40 to 80 minutes at each evaluation (this is for instance the case for objective functions which are calculated using CFD techniques)? The response is still to investigate. Currently, the implemented strategy is: never wait. Despite, this simple strategy, the current algorithm gives already some non-negligible improvements.

7.4 Noisy optimization

We will assume that objective functions derived from CFD codes have usually a simple shape but are subject to high-frequency, low amplitude noise. This noise prevents us to use simple finite-differences gradient-based algorithms. Finite-difference is highly sensitive to the noise. Simple Finite-difference quasi-Newton algorithms behave so badly because of the noise, that most researchers choose to use optimization techniques based on GA,NN,... [CAVDB01, PVdB98, Pol00]. The poor performances of finite-differences gradient-based algorithms are either due to

Name	Dim	Number of Function Evaluations on the main node			final function value		
		1CPU	2CPU	3CPU	1 CPU	2 CPU	3 CPU
ROSENBR	2	82	81 (1.2%)	70 (14.6%)	2.0833e-08	5.5373e-09	3.0369e-07
SNAIL	2	314	284 (9.6%)	272 (13.4%)	9.3109e-11	4.4405e-13	6.4938e-09
SISSER	2	40	35 (12.5%)	40 (0.0%)	8.7810e-07	6.7290e-10	2.3222e-12
CLIFF	2	145	87 (40.0%)	69 (52.4%)	1.9978e-01	1.9978e-01	1.9978e-01
HAIRY	2	47	35 (25.5%)	36 (23.4%)	2.0000e+01	2.0000e+01	2.0000e+01
PFIT1LS	3	153	91 (40.5%)	91 (40.5%)	2.9262e-04	1.7976e-04	2.1033e-04
HATFLDE	3	96	83 (13.5%)	70 (27.1%)	5.6338e-07	1.0541e-06	3.2045e-06
SCHMVETT	3	32	17 (46.9%)	17 (46.9%)	-3.0000e+00	-3.0000e+00	-3.0000e+00
GROWTHLS	3	104	85 (18.3%)	87 (16.3%)	1.2437e+01	1.2456e+01	1.2430e+01
GULF	3	170	170 (0.0%)	122 (28.2%)	2.6689e-09	5.7432e-04	1.1712e-02
BROWNDEN	4	91	60 (34.1%)	63 (30.8%)	8.5822e+04	8.5826e+04	8.5822e+04
EIGENALS	6	123	77 (37.4%)	71 (42.3%)	3.8746e-09	1.1597e-07	1.5417e-07
HEART6LS	6	346	362 (4.4%)	300 (13.3%)	4.3601e-01	4.1667e-01	4.1806e-01
BIGGS6	6	284	232 (18.3%)	245 (13.7%)	1.1913e-05	1.7741e-06	4.0690e-07
HART6	6	64	31 (51.6%)	17 (73.4%)	-3.3142e+00	-3.3184e+00	-2.8911e+00
CRAGGLVY	10	545	408 (25.1%)	339 (37.8%)	1.8871e+00	1.8865e+00	1.8865e+00
VARDIM	10	686	417 (39.2%)	374 (45.5%)	8.7610e-13	3.2050e-12	1.9051e-11
MANCINO	10	184	79 (57.1%)	69 (62.5%)	3.7528e-09	9.7042e-09	3.4434e-08
POWER	10	550	294 (46.6%)	223 (59.4%)	9.5433e-07	3.9203e-07	4.7188e-07
MOREBV	10	110	52 (52.7%)	43 (60.9%)	1.0100e-07	8.0839e-08	9.8492e-08
BRYBND	10	505	298 (41.0%)	198 (60.8%)	4.4280e-08	3.0784e-08	1.7790e-08
BROWNAL	10	331	187 (43.5%)	132 (60.1%)	4.6269e-09	1.2322e-08	6.1906e-09
DQDRTIC	10	201	59 (70.6%)	43 (78.6%)	2.0929e-18	2.0728e-31	3.6499e-29
WATSON	12	667	339 (49.2%)	213 (68.1%)	7.9451e-07	1.1484e-05	1.4885e-04
DIXMAANK	15	964	414 (57.0%)	410 (57.5%)	1.0000e+00	1.0000e+00	1.0000e+00
FMINSURF	16	695	455 (34.5%)	333 (52.1%)	1.0000e+00	1.0000e+00	1.0000e+00
Total Number of Function Evaluation		7531	4732	3947			

Figure 7.2: Improvement due to parallelism

the difficulty in choosing finite-difference step sizes for such a rough function, or the often cited tendency of derivative-based methods to converge to a local optimum [BDF+98]. Gradient-based algorithms can still be applied but a clever way to retrieve the derivative information must be used. One such algorithm is DIRECT [GK95, Kel99, BK97] which is using a technique called implicit filtering. This algorithm makes the same assumption about the noise (low amplitude, high frequency) and has been successful in many cases [BK97, CGP+01, SBT+92]. For example, this optimizer has been used to optimize the cost of fuel and/or electric power for the compressor stations in a gas pipeline network [CGP+01]. This is a two-design-variables optimization problem. You can see in the right of Figure 7.5 a plot of the objective function. Notice the simple shape of the objective function and the small amplitude, high frequency noise. Another family of optimizers is based on interpolation techniques. DFO, UOBYQA and CONDOR belongs to this last family. DFO has been used to optimize (minimize) a measure of the vibration of a helicopter rotor blade [BDF+98]. This problem is part of the Boeing problems set [BCD+95]. The blade are characterized by 31 design variables. CONDOR will soon be used in industry on a daily basis to optimize the shape of the blade of a centrifugal impeller [PMM+03]. All these problems (gas pipeline, rotor blade and impeller blade) have an objective function based on CFD code and are both solved using gradient-based techniques. In particular, on the rotor blade design, a comparative study between DFO and other approaches like GA, NN,... has demonstrated the clear superiority of gradient-based techniques approach combined with interpolation techniques [BDF+98].

We will now illustrate the performances of CONDOR in two simple cases which have sensibly the same characteristics as the objective functions encountered in optimization based on CFD codes. The functions, the amplitude of the artificial noise applied to the objective functions (uniform noise distribution) and all the parameters of the tests are summarized in Table 7.4. In this table "NFE" stands for *Number of Function Evaluations.* Each columns represents 50 runs of the optimizer.

Objective function	Rosenbrock	A simple quadratic: $\sum_{i=1}^{4}(x_i-2)^2$				
starting point	$(-1.2\ \ 1)^t$	$(0\ 0\ 0\ 0)^t$				
ρ_{start}	1					
ρ_{end}	1e-4					
average NFE	96.28 (88.02)	82.04 (53.6)	89.1 (62.20)	90.7 (64.56)	99.4 (66.84)	105.36 (68.46)
max NFE	105	117	116	113	129	124
min NFE	86	58	74	77	80	91
average best val	2.21e-5	6.5369e-7	3.8567e-6	8.42271e-5	8.3758e-4	1.2699e-2
noise	1e-4	1e-5	1e-4	1e-3	1e-2	1e-1

Figure 7.3: Noisy optimization.

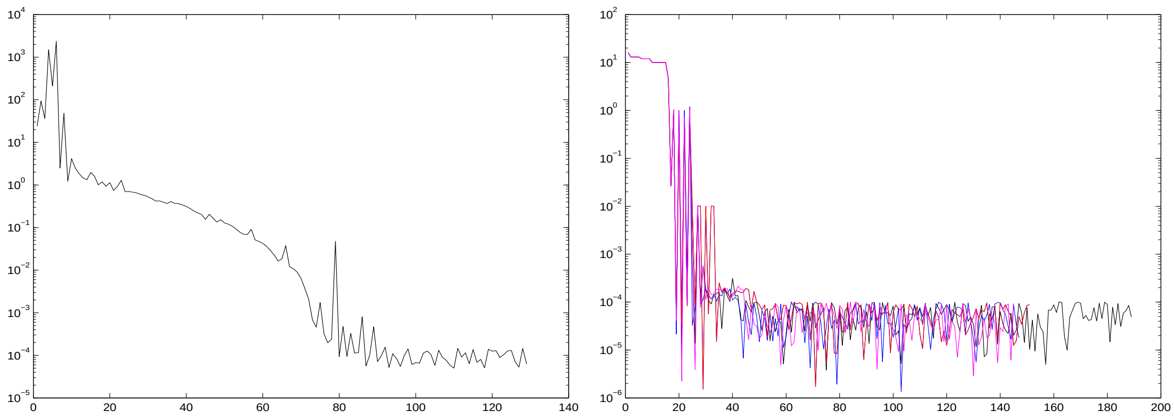

Figure 7.4: On the left: A typical run for the optimization of the noisy Rosenbrock function. On the right:Four typical runs for the optimization of the simple noisy quadratic (noise=1e-4).

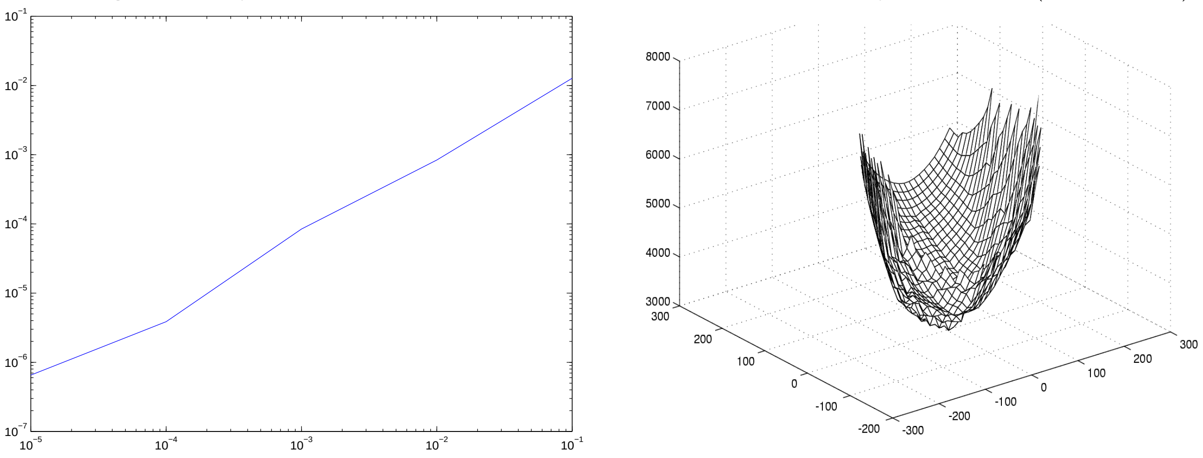

Figure 7.5: On the left: The relation between the noise (X axis) and the average best value found by the optimizer (Y axis). On the right: Typical shape of objective function derived from CFD analysis.

A typical run for the optimization of the noisy Rosenbrock function is given in the left of Figure 7.4. Four typical runs for the optimization of the simple noisy quadratic in four dimension are given in the right of figure 7.4. The noise on these four runs has an amplitude of 1e-4. In these conditions, CONDOR stops in average after 100 evaluations of the objective function but we can see in figure 7.4 that we usually already have found a quasi-optimum solution after only 45 evaluations.

As expected, there is a clear relationship between the noise applied on the objective function and the average best value found by the optimizer. This relationship is illustrated in the left of figure 7.4. From this figure and from the Table 7.4 we can see the following: When you have a noise of 10^{n+2}, the difference between the best value of the objective function found by the optimizer AND the real value of the objective function at the optimum is around 10^n. In other words, in our case, if you apply a noise of 10^{-2}, you will get a final value of the objective function around 10^{-4}. Obviously, this strange result only holds for this simple objective function (the simple quadratic) and these particular testing conditions. Nevertheless, the robustness against noise is impressive.

If this result can be generalized, it will have a great impact in the field of CFD shape optimization. This simply means that if you want a gain of magnitude 10^n in the value of the objective function, you have to compute your objective function with a precision of at least 10^{n+2}. This gives you an estimate of the precision at which you have to calculate your objective function. Usually, the more precision, the longer the evaluations are running. We are always tempted to lower the precision to gain in time. If this strange result can be generalized, we will be able to adjust tightly the precision and we will thus gain a precious time.

Part II

Constrained Optimization

Chapter 8

A short review of the available techniques.

In the industry, the objective function is very often a simulator of a complex process. The constraints usually represents bounds on the validity of the simulator. Sometimes the simulator can simply crash when evaluating an infeasible point (a point which do not respects the constraints). For this reason, the optimization algorithm generates only feasible points (due to rounding errors, some points may be infeasible especially when there are non-linear constraints. Anyway, the generated infeasible points are always very close to feasibility).

There are two possible approaches. The first approach is now described.

The steps s_k of the unconstrained algorithm are the solution of:

$$\min_{s\in\Re^n} q(x_k+s) = q_k(s) \equiv f(x_k) + \langle g_k, s\rangle + \frac{1}{2}\langle s, H_k s\rangle \tag{8.1}$$
$$\text{subject to } \|s\|_2 < \Delta$$

In the first approach (="*approach 1*"), the step s_k of the constrained algorithm are the solution of:

$$\min_{s\in\Re^n} q(x_k+s) \equiv f(x_k) + \langle g_k, s\rangle + \tfrac{1}{2}\langle s, H_k s\rangle$$
$$\text{subject to } \begin{cases} b_l \le x \le b_u, & b_l, b_u \in \Re^n \\ Ax \ge b, & A \in \Re^{m\times n}, b \in \Re^m \\ c_i(x) \ge 0, & i = 1, \dots, l \\ \|s\|_2 < \Delta \end{cases} \tag{8.2}$$

The problem of solving 8.2 is not trivial at all. It's in fact as difficult as the original problem. The only advantage in solving this subproblem at each step is that the objective function (which is $q_k(s)$) evaluations are cheap and thus we can have very precise steps leading (hopefully) to a fast convergence to the solution of the original problem.

The same methods used for solving the subproblem 8.2 can be directly applied to the original non-linear objective function. This is our second approach (="*approach 2*").

There are specific methods for box or linear constraints and for non-linear constraints. We will describe them in two separate chapters.

8.1 Linear constraints

We want to find $x^* \in \mathcal{R}^n$ which satisfies:

$$\mathcal{F}(x^*) = \min_x \mathcal{F}(x) \quad \text{Subject to: } Ax \geq b, \; A \in \Re^{n \times m}, b \in \Re^m \tag{8.3}$$

where n is the dimension of the search space and m is the number of linear constraints.

8.1.1 Active set - Null space method

The material of this section is based on the following reference: [Fle87].

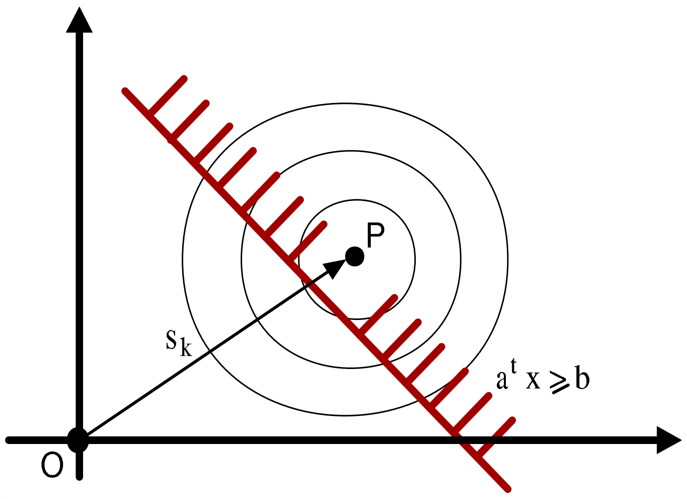

Figure 8.1: Violation of a constraint.

At each step s_k computed from 8.1, we check if one of the m linear constraints has been violated. On the figure 8.1, the linear constraint $a^t x \geq b$ has been violated and it's thus "activated". Without loosing generality, let's define $A_a \in \Re^{n \times m_a}$ and $b_a \in \Re^{m_a}$ the set of the m_a active constraints.

Let Y and Z be $n \times m_a$ and $n \times (n - m_a)$ matrices respectively such that $[Y : Z]$ is non-singular, and in addition let $A_a Y = I$ and $A_a Z = 0$. The step must be feasible, so we must have to $A_a s \geq b_a$. A solution to this equation is given by: $s = Y b_a + Z y$ where y can be any vector. Indeed, we have the following: $A_a s = A_a (Y b_a + Z y) = A_a Y b_a + A_a Z y = b_a$. The situation is illustrated in figure 8.2. We will choose y as the solution of

$$\min_y q(x_k + Y b_a + Z y) \quad \text{subject to } \|Zy\| \leq \Delta_r \tag{8.4}$$

with $\Delta_r = \sqrt{\Delta^2 - \|Y b_a\|^2}$. In other words, y is the minimum of the quadratical approximation of the objective function limited to the reduced space of the active linear constraints and limited to the trust region boundaries. We have already developed an algorithm able to compute y in chapter 4.

When using this method, there is no difference between "*approach 1*" and "*approach 2*".

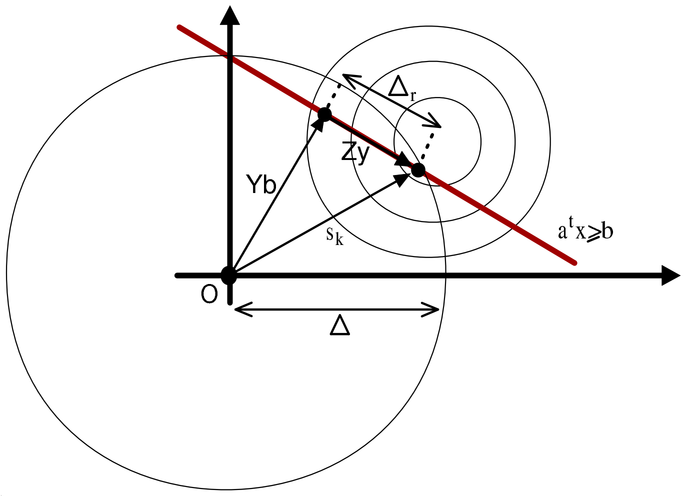

Figure 8.2: A search in the reduced space of the active constraints gives as result y

This algorithm is very stable in regards to rounding error. It's very fast because we can make Newton step (quadratical convergence speed) in the reduced space. Beside, we can use software developed in chapter 4. For all these reasons, it has been chosen and implemented. It will be fully described in chapter 9.

8.1.2 Gradient Projection Methods

The material of this section is based on the following reference: [CGT00e].

In these methods, we will follow the "steepest descent steps": we will follow the gradient. When we enter the infeasible space, we will simply project the gradient into the feasible space. A straightforward (unfortunately false) extension to this technique is the "Newton step projection algorithm" which is illustrated in figure 8.3. In this figure the current point is $x_k = O$. The Newton step (s_k) lead us to point P which is infeasible. We project P into the feasible space: we obtain B. Finally, we will thus follow the trajectory OAB, which *seems* good.

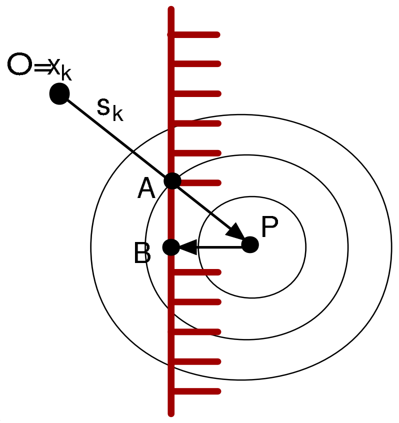

Figure 8.3: "newton's step projection algorithm" seems good.

In figure 8.4, we can see that the "Newton step projection algorithm" can lead to a false minimum. As before, we will follow the trajectory OAB. Unfortunately, the real minimum of the problem is C.

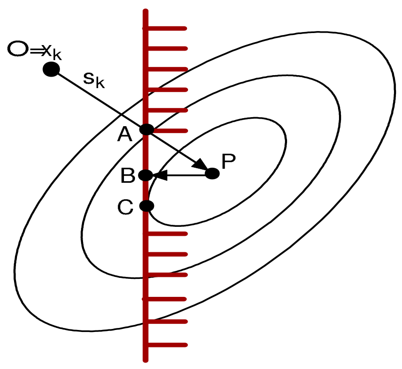

Figure 8.4: "newton's step projection algorithm" is false.

We can therefore only follow the gradient,not the Newton step. The speed of this algorithm is thus, at most, linear, requiring many evaluation of the objective function. This has little consequences for "*approach 1*" but for "*approach 2*" it's intolerable. For these reasons, the Null-space method seems more promising and has been chosen.

8.2 Non-Linear constraints: Penalty Methods

The material of this section is based on the following reference: [Fle87].

Consider the following optimization problem:

$$f(x^*) = \min_x f(x) \quad \text{Subject to: } c_i(x) \geq 0, \; i = 1, \ldots, m \tag{8.5}$$

A *penalty function* is some combination of f and c which enables f to be minimized whilst controlling constraints violations (or near constraints violations) by penalizing them. A primitive penalty function for the inequality constraint problem 8.5 is

$$\phi(x, \sigma) = f(x) + \frac{1}{2}\sigma \sum_{i=1}^{m} [\min(c_i(x), 0)]^2 \tag{8.6}$$

The penalty parameter σ increases from iteration to iteration to ensure that the final solution is feasible. The penalty function is thus more and more ill-conditioned (it's more and more difficult to approximate it with a quadratic polynomial). For these reason, penalty function methods are slow. Furthermore, they produce infeasible iterates. Using them for "*approach 2*" is not possible. However, for "*approach 1*" they can be a good alternative, especially if the constraints are very non-linear.

8.3 Non-Linear constraints: Barrier Methods

The material of this section is based on the following references: [NW99, BV04, CGT00f].

Consider the following optimization problem:

$$f(x^*) = \min_x f(x) \quad \text{Subject to: } c_i(x) \geq 0,\ i = 1, \ldots, m \tag{8.7}$$

We aggregate the constraints and the objective function in one function which is:

$$\varphi_t(x) := f(x) - t \sum_{i=1}^{m} ln(c_i(x)) \tag{8.8}$$

t is called the barrier parameter. We will refer to $\varphi_t(x)$ as the *barrier function*. The degree of influence of the barrier term "$-t \sum_{i=1}^{m} ln(c_i(x))$" is determined by the size of t. Under certain conditions x_t^* converges to a local solution x^* of the original problem when $t \to 0$. Consequently, a strategy for solving the original NLP (Non-Linear Problem) is to solve a sequence of barrier problems for decreasing barrier parameter t_l, where l is the counter for the sequence of subproblems. Since the exact solution $x_{t_l}^*$ is not of interest for large t_l, the corresponding barrier problem is solved only to a relaxed accuracy ϵ_l, and the approximate solution is then used as a starting point for the solution of the next barrier problem. The radius of convergence of Newton's method applied to 8.8 converges to zero as $t \to 0$. When $t \to 0$, the barrier function becomes more and more difficult to approximate with a quadratical function. This lead to poor convergence speed for the newton's method. The conjugate gradient (CG) method can still be very effective, especially if good preconditionners are given. The barrier methods have evolved into primal-dual interior point which are faster. The relevance of these methods in the case of the optimization of high computing load objective functions will thus be discussed at the end of the section relative to the primal-dual interior point methods.

8.4 Non-Linear constraints: Primal-dual interior point

The material of this section is based on the following references: [NW99, BV04, CGT00f].

8.4.1 Duality

Consider an optimization problem in this form:

$$\min f(x) \quad \text{Subject to:} \quad c_i(x) \geq 0,\ i = 1, \ldots, m \tag{8.9}$$

We assume its domain is $\mathcal{D} = \left(\bigcap_{i=1}^{m} \mathbf{dom} c_i(x) \right) \bigcap (\mathbf{dom} f(x))$. We define the *Lagrangian* $\mathcal{L}$: $\Re^n \times \mathcal{R}^m \to \Re$ associated with the problem 8.9 (see section 13.3 about the *Lagrangian* function):

$$\mathcal{L}(x, \lambda) = f(x) - \sum_{i=1}^{m} \lambda_i c_i(x) \tag{8.10}$$

We refer to λ_i as the *Lagrange multiplier* or *dual variables* associated with the i^{th} inequality constraint $c_i(x) \geq 0$. We define the *Lagrange dual function* (or just *dual function*) $g : \Re^m \to \Re$ as the minimum value of the Lagrangian over x:

$$g(\lambda) = \inf_{x \in \mathcal{D}} \mathcal{L}(x, \lambda) \tag{8.11}$$

Lower Bounds on optimal value

The dual function yields lower bounds on the optimal value $f(x^*)$ of the problem 8.9: for any $\lambda > 0$, we have

$$g(\lambda) \leq f(x^*) \tag{8.12}$$

The proof follow. Suppose $\bar{x}$ is a feasible point of the problem 8.9. Then we have:

$$\sum_{i=1}^{m} \lambda_i c_i(\bar{x}) \geq 0 \tag{8.13}$$

since each term is non-negative. And therefore:

$$\mathcal{L}(\bar{x}, \lambda) = f(\bar{x}) - \sum_{i=1}^{m} \lambda_i c_i(\bar{x}) \leq f(\bar{x}) \tag{8.14}$$

Hence

$$g(\lambda) = \inf_{x \in \mathcal{D}} \mathcal{L}(x, \lambda) \leq \mathcal{L}(\bar{x}, \lambda) \leq f(\bar{x}) \tag{8.15}$$

Since $g(\lambda) \leq f(\bar{x})$ holds for every feasible point $\bar{x}$, the inequality 8.12 follows.

When the feasible domain is convex and when the objective function is convex, the problem is said to be convex. In this case, we have $\max_\lambda g(\lambda) = \min_x f(x)$. The proof will be skipped. When the problem is not convex, we will define the *Duality gap* $= \min_x f(x) - \max_\lambda g(\lambda) \geq 0$.

The Lagrange dual problem of a linear problem in inequality form

Lets calculate the Lagrange dual of an inequality form LP:

$$\min c^t x \quad \text{Subject to:} \quad Ax \leq b \Leftrightarrow b - Ax \geq 0 \tag{8.16}$$

The Lagrangian is

$$\mathcal{L}(x, \lambda) = c^t x - \lambda^t (b - Ax) \tag{8.17}$$

So the dual function is

$$g(\lambda) = \inf_x \mathcal{L}(x, \lambda) = -b^t \lambda + \inf_x (A^t \lambda + c)^t x \tag{8.18}$$

The infinitum of a linear function is $-\infty$, except in the special case when its identically zero, so the dual function is:

$$g(\lambda) = \begin{cases} -b^t \lambda & A^t \lambda + c = 0 \\ -\infty & otherwise. \end{cases} \tag{8.19}$$

The dual variable λ is dual feasible if $\lambda > 0$ and $A^t \lambda + c = 0$. The Lagrange dual of the LP 8.16 is to maximize g over all $\lambda > 0$. We can reformulate this by explicitly including the dual feasibility conditions as constraints, as in:

$$\max b^t \lambda \quad \text{Subject to:} \quad \begin{cases} A^t \lambda + c = 0 \\ \lambda \geq 0 \end{cases} \tag{8.20}$$

which is an LP in standard form.

Since the feasible domain is convex and the objective function is also convex, we have a convex problem. The solution of this dual problem is thus equal to the solution of the primal problem.

8.4.2 A primal-dual Algorithm

Consider an optimization problem in this form:

$$\min c^t x \quad \text{Subject to:} \quad Ax = b, x \geq 0,$$

where $c, x \in \Re^n, b \in \Re^m, A \in \Re^{m \times n}$. We will now compute the dual problem:

$$\begin{aligned}\mathcal{L}(x) =& c^t x - \lambda^t(Ax - b) - s^t x \\ g(\lambda, s) =& \inf_{x \in \mathcal{D}} \mathcal{L}(x, \lambda) \\ =& \inf_{x \in \mathcal{D}} x^t(c - A^t\lambda - s) + b^t\lambda\end{aligned}$$

Since the problem is convex, we have ($\lambda \in \Re^m$ are the Lagrange multipliers associated to the constraints $Ax = b$ and $s \in \Re^n$ are the Lagrange multipliers associated to the constraints $x \geq 0$):

$$\begin{aligned}\min_x c^t \quad \text{Subject to:} \quad \begin{matrix} Ax = b \\ x \geq 0 \end{matrix} &= \max_{\lambda, s} g(\lambda, s) \\ &= \max_{\lambda} b^t\lambda \quad \text{Subject to:} \quad \begin{matrix} A^t\lambda + s = c \\ s \geq 0, \end{matrix}\end{aligned}$$

The associated KKT conditions are:

$$A^t\lambda + s = c, \tag{8.21}$$

$$Ax = b, \tag{8.22}$$

$$x_i s_i = 0, \quad i = 1, \dots, n \tag{8.23}$$

(the *complementarity condition* for the constraint $x \geq 0$)

$$(x, s) \geq 0 \tag{8.24}$$

Primal dual methods find solutions (x^*, λ^*, s^*) of this system by applying variants of Newton's method (see section 13.6 for Newton's method for non-linear equations) to the three equalities 8.21-8.23 and modifying the search direction and step length so that inequalities $(x, s) \geq 0$ are satisfied strictly at every iteration. The nonnegativity condition is the source of all the complications in the design and analysis of interior-point methods. Let's rewrite equations 8.21-8.24 in a slightly different form:

$$F(x, \lambda, s) = \begin{bmatrix} A^t\lambda + s - c \\ Ax - b \\ XSe \end{bmatrix} = 0 \tag{8.25}$$

$$(x, s) \geq 0 \tag{8.26}$$

where $X = diag(x_1, \dots, x_n), S = diag(s_1, \dots, s_n)$ and $e = (1, \dots, 1)^t$. Primal dual methods generate iterates (x^k, λ^k, s^k) that satisfy the bounds 8.26 strictly. This property is the origin of the term *interior-point*.

As mentioned above, the search direction procedure has its origin in Newton's method for the set of nonlinear equations 8.25 (see section 13.6 for Newton's method for non-linear equations). We obtain the search direction $(\Delta x, \Delta\lambda, \Delta s)$ by solving the following system of linear equations:

$$J(x, \lambda, s) \begin{bmatrix} \Delta x \\ \Delta\lambda \\ \Delta s \end{bmatrix} = -F(x, \lambda, s) \tag{8.27}$$

where J is the Jacobian of F. If the current point is strictly feasible, we have:

$$\begin{bmatrix} 0 & A^t & I \\ A & 0 & 0 \\ S & 0 & X \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta\lambda \\ \Delta s \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ -XSe \end{bmatrix} \tag{8.28}$$

A full step along this direction is not permissible, since it would violate the bound $(x, s) > 0$. To avoid this difficulty, we perform a line search along the Newton direction so that the new iterate is

$$(x, \lambda, s) + \alpha(\Delta x, \Delta\lambda, \Delta s) \tag{8.29}$$

for some line search parameter $\alpha \in (0, 1]$. Unfortunately, we often can only take a small step along this direction (= the *affine scaling direction*). To overcome this difficult, primal-dual methods modify the basic Newton procedure in two important ways:

1. They bias the search direction toward the interior of the nonnegative orthant $(x, s) \geq 0$, so that we can move further along this direction before one components of (x, s) becomes negative.

2. They keep the components of (x, s) from moving "too close" to the boundary of the nonnegative orthant.

We will consider these two modifications in turn in the next subsections.

8.4.3 Central path

The central path $\mathcal{C}$ is an arc of strictly feasible points that is parameterized by a scalar $\tau > 0$. We can define the central path as

$$\mathcal{C} = \{(x_\tau, \lambda_\tau, s_\tau) | \tau > 0\} \tag{8.30}$$

Where each point $(x_\tau, \lambda_\tau, s_\tau) \in \mathcal{C}$ solves the following system, which is a perturbation of the original system 8.25

$$F(x, \lambda, s) = \begin{bmatrix} A^t\lambda + s - c \\ Ax - b \\ XSe - \tau \end{bmatrix} = 0 \tag{8.31}$$

$$(x, s) \geq 0 \tag{8.32}$$

A plot of $\mathcal{C}$ for a typical problem, projected into the space of primal variables x, is shown in figure 8.5.

The equation 8.31 approximate 8.25 more and more closely as τ goes to zero. Primal-Dual algorithms take Newton's steps toward points on $\mathcal{C}$ for which $\tau > 0$. Since these steps are biased toward the interior of the nonnegative orthant $(x, s) \geq 0$, it is usually possible to take longer steps along them than along pure Newton's steps for F, before violating the positivity condition. To describe the biased search direction, we introduce a *duality measure* μ defined by

$$\mu = \frac{1}{n} \sum_{i=1}^{n} x_i s_i = \frac{x^t s}{n} \tag{8.33}$$

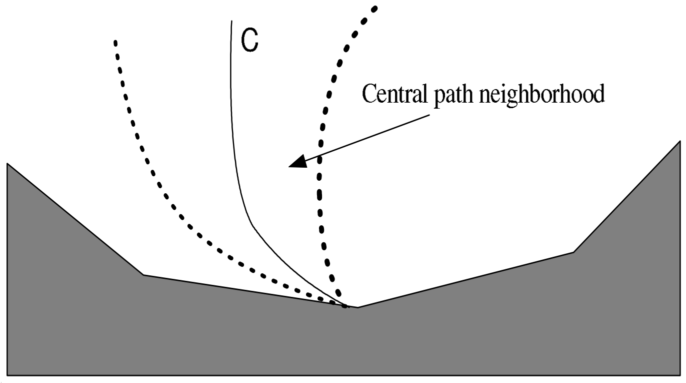

Figure 8.5: Central Path

which measure the average value of the pairwise products $x_i s_i$. We also define a *centering parameter* $\sigma \in [0,1] = \frac{\tau}{\mu}$. Applying Newton's method to the system 8.31, we obtain:

$$\begin{bmatrix} 0 & A^t & I \\ A & 0 & 0 \\ S & 0 & X \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta \lambda \\ \Delta s \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ -XSe + \sigma\mu e \end{bmatrix} \tag{8.34}$$

If $\sigma = 1$, the equations 8.34 define a *centering direction*, a Newton step toward the point $(x_\mu, \lambda_\mu, s_\mu) \in \mathcal{C}$. Centering direction are usually biased strongly toward the interior of the nonnegative orthant and make little progress in reducing the duality measure μ. However, by moving closer to $\mathcal{C}$, they set the scene for substantial progress on the next iterations. At the other extreme, the value $\sigma = 0$ gives the standard Newton's step.

The choice of centering parameter σ_k and step length α_k are crucial to the performance of the method. Techniques for controlling these parameters, directly and indirectly, give rise to a wide variety of methods with varying theoretical properties.

8.4.4 Link between Barrier method and Interior point method

In this section we want to find the solution of:

$$f(x^*) = \min_x f(x) \quad \text{Subject to: } c_i(x) \geq 0,\ i = 1, \dots, m \tag{8.35}$$

There exist a value $\lambda^* \in \Re^m$ of the dual variable and a value $x^* \in \Re^n$ of the primal variable which satisfy the KKT conditions:

$$\nabla f(x^*) - \sum_{i=1}^{m} \lambda_i^* \nabla c_i(x^*) = 0 \tag{8.36}$$

$$c_i(x^*) \geq 0 \tag{8.37}$$

$$\lambda_i^* c_i(x^*) = 0 \tag{8.38}$$

$$\lambda^* \geq 0 \tag{8.39}$$

Equations 8.36-8.39 are comparable to equations 8.21-8.24. There are the base equations for the primal-dual iterations. In the previous section, we motivated the following perturbation of these equations:

$$\nabla f(x^*) + \sum_{i=1}^{m} \lambda_i^* \nabla c_i(x^*) = 0 \tag{8.40}$$

$$c_i(x) \geq 0 \tag{8.41}$$

$$\lambda_i^* c_i(x^*) = \tau \tag{8.42}$$

$$\lambda^* \geq 0 \tag{8.43}$$

Let's now consider the following equivalent optimization problem (barrier problem):

$$\varphi_t(x) := f(x) - t \sum_{i=1}^{m} ln(c_i(x)) \text{ and } t \rightarrow 0 \tag{8.44}$$

For a given t, at the optimum x^* of equation 8.44, we have:

$$0 = \nabla \varphi_t(x^*) = \nabla f(x^*) - \sum_{i=1}^{m} \frac{t}{c_i(x^*)} \nabla c_i(x^*) \tag{8.45}$$

If we define

$$\lambda_i^* = \frac{t}{c_i(x^*)} \tag{8.46}$$

We will now proof that λ^* is dual feasible. First it's clear that $\lambda^* \geq 0$ because $c_i(x^*) \geq 0$ because x^* is inside the feasible domain. We now have to proof that $g(\lambda^*) = f(x^*)$. Let's compute the value of the dual function of 8.35 at (λ^*):

$$\mathcal{L}(x, \lambda) = f(x) - \sum_{i=1}^{m} \lambda_i c_i(x)$$

$$g(\lambda^*) = f(x^*) - \sum_{i=1}^{m} \lambda_i^* c_i(x^*)$$

Using equation 8.46:

$$= f(x^*) - mt \tag{8.47}$$

When $t \rightarrow 0$, we will have $g(\lambda^*) = f(x^*) = p^*$ which concludes the proof. mt is the *duality gap*.

Let's define p^* the minimum value of the original problem 8.35, and (x^*, λ^*) the solution of $min_x \varphi_t(x)$ for a given value of t. From equation 8.47, we have the interesting following result: $f(x^*) - p^* \leq mt$. That is: x^* is no more than mt suboptimal.

What's the link between equations 8.40-8.43, which are used inside the primal-dual algorithm and the barrier method? We see that if we combine equations 8.45 and 8.46, we obtain 8.40. The equations 8.42 and 8.46 are the same, except that $t = \tau$. The "perturbation parameter" τ in primal-dual algorithm is simply the barrier parameter t in barrier algorithms. In fact, barrier method and primal-dual methods are very close. The main difference is: In primal-dual methods, we update the primal variables x AND the dual variable λ at each iteration. In barrier methods, we only update the primal variables.

8.4.5 A final note on primal-dual algorithms.

The primal-dual algorithms are usually following this scheme:

1. Set $l := 0$
2. Solve *approximatively* the barrier problem 8.44 for a given value of the centering parameter σ_l (see equation 8.34 about σ) using as starting point the solution of the previous iteration.
3. Update the barrier parameter σ. For example: $\sigma_{l+1} := min\{\ 0.2\ \sigma_l,\ \sigma_l^{1.5}\ \}$
   Increase the iteration counter $l := l + 1$
   If not finished, go to step 2.

The principal difficulties in primal-dual methods are:

- The choice of centering parameter σ which is crucial to the performance of the method. If the decrease is too slow, we will loose our time evaluating the objective function far from the real optimum. For "*approach 2*", where these evaluations are very expensive (in time), it's a major drawback.
- In step 2 above, we have to solve *approximatively* an optimization problem. What are the tolerances? What's the meaning of *approximatively*?
- The feasible space should be convex. Extension to any feasible space is possible but not straight forward.
- The starting point should be feasible. Extension to any starting point is possible but not straight forward.
- The "switching mechanism" from the unconstrained steps when no constrained are violated to the constrained case is difficult to implement (if we want to keep the actual mechanism for step computation when no constraints are active).

These kinds of algorithm are useful when the number of constraints are huge. In this case, identifying the set of active constraints can be very time consuming (it's a combinatorial problem which can really explode). Barrier methods and primal-dual methods completely avoids this problem. In fact, most recent algorithms for linear programming and for quadratic programming are based on primal-dual methods for this reason. The main field of application of primal-dual methods is thus optimization in very large dimensions (can be more than 10000) and with many constraints (can be more than 6000000). Since we have only a reduced amount of constraints, we can use an active set method without any problem.

8.5 Non-Linear constraints: SQP Methods

The material of this section is based on the following references: [NW99, Fle87].

SQP stands for "*Sequential quadratic Programming*". We want to find the solution of:

$$f(x^*) = \min_x f(x) \quad \text{Subject to: } c_i(x) \geq 0,\ i = 1, \dots, m \tag{8.48}$$

As in the case of the Newton's method in unconstrained optimization, we will do a quadratical approximation of the function to optimize and search for the minimum of this quadratic. The function to be approximated will be the Lagrangian function $\mathcal{L}$.

The quadratical approximation of $\mathcal{L}$ is:

$$\begin{aligned}\mathcal{L}(x_k+\delta_x,\lambda_k+\delta_\lambda)\approx\mathcal{Q}(\delta_x,\delta_\lambda) &= \mathcal{L}(x_k,\lambda_k)+\blacktriangledown\mathcal{L}(x_k,\lambda_k)\begin{pmatrix}\delta_x\\ \delta_\lambda\end{pmatrix}+\\ &\quad \frac{1}{2}\begin{pmatrix}\delta_x & \delta_\lambda\end{pmatrix}[\blacktriangledown^2\mathcal{L}(x_k,\lambda_k)]\begin{pmatrix}\delta_x\\ \delta_\lambda\end{pmatrix}\end{aligned}$$

$$\begin{aligned}\Longleftrightarrow \mathcal{Q}(\delta_x,\delta_\lambda) &= \mathcal{L}(x_k,\lambda_k)+\begin{pmatrix}g_k+A_k\lambda_k & c_k\end{pmatrix}\begin{pmatrix}\delta_x\\ \delta_\lambda\end{pmatrix}+\\ &\quad \frac{1}{2}\begin{pmatrix}\delta_x & \delta_\lambda\end{pmatrix}\begin{bmatrix}H_k & -A_k\\ -A_k & 0\end{bmatrix}\begin{pmatrix}\delta_x\\ \delta_\lambda\end{pmatrix}\end{aligned}\tag{8.49}$$

With A_k : the Jacobian matrix of the constraints evaluated at x_k:
$(A_k)_{i,j}=\dfrac{\partial c_i(x)}{\partial j}$ or $(A_k)_i=\nabla c_i(x)$
H_k : the Hessian Matrix $\nabla^2_x\mathcal{L}(x_k,\lambda_k)$
$H_k=\nabla^2_x f(x_k)-\sum_i(\lambda_k)_i\nabla^2 c_i(x_k)$

The full Hessian W_k of $\mathcal{L}$ is thus :

$$W_k=\begin{bmatrix}H_k & -A_k\\ -A_k & 0\end{bmatrix}(=[\blacktriangledown^2\mathcal{L}(x_k,\lambda_k)])\tag{8.50}$$

If we are on the boundary of the i^{th} constraint, we will have $(\delta_x)^t\nabla c_i(x)=0$, thus we can write:

on the constraints boundaries: $A_k\delta_x\approx 0$ (8.51)

We want to find δ_x which minimize $\mathcal{Q}(\delta_x,\delta_\lambda)$, subject to the constraints $c_i(x)\geq 0,\ i=1,...,m$.

From equation 8.49 and using equation 8.51, we obtain:

$$\begin{matrix}\min\limits_{\delta_x}\mathcal{Q} & & \min\limits_{\delta_x} g_k^t\delta_x++\frac{1}{2}\delta_x^t H_k\delta_x\\ & \overset{\text{approx.}}{\Longleftrightarrow} & \\ \textit{subject to } c_j(x_k+\delta_x)\geq 0,\ j=1,...,r & & \textit{subject to } c_j(x_k+\delta_x)\geq 0,\ j=1,...,r\end{matrix}\tag{8.52}$$

Using a first order approximation of the constraints around x_k, we have the

$$\begin{aligned}&\textit{Quadratic Program QP}:\\ &\min_{\delta_x} g_k^t\delta_x+\frac{1}{2}\delta_x^t H_k\delta_x\\ &\textit{subject to } c_j(x_k)+(\delta_x)^t\nabla c_j(x_k)\geq 0,\ \ j=1,...,r\end{aligned}\tag{8.53}$$

DEFINITION: a *Quadratic Program (QP)* is a function which finds the minimum of a quadratic subject to linear constraints.

Note that $H_k = \nabla_x^2 \mathcal{L}(x_k, \lambda_k) \neq \nabla^2 f(x_k)$.

We can now define the SQP algorithm:

1. solve the QP subproblem described on equation 8.53 to determine δ_x and let λ_{k+1} be the vector of the Lagrange multiplier of the linear constraints obtained from the QP.
2. set $x_{k+1} = x_k + \delta_x$
3. Increment k. Stop if $\nabla \mathcal{L}(x_k, \lambda_k) \approx 0$. Otherwise, go to step 1.

This algorithm is the generalization of the "Newton's method" for the constrained case. It has the same properties. It has, near the optimum (and under some special conditions), quadratical convergence.

A more robust implementation of the SQP algorithm adjust the length of the steps and performs "second order correction steps" (or "*SOC* steps").

8.5.1 A note about the H matrix in the SQP algorithm.

As already mentioned, we have $H_k = \nabla_x^2 \mathcal{L}(x_k, \lambda_k) = \nabla_x^2 f(x_k) - \sum_i (\lambda_k)_i \nabla^2 c_i(x_k) \neq \nabla^2 f(x_k)$

The inclusion of the second order constraint terms in the subproblem is important in that otherwise second order convergence for nonlinear constraints would not be obtained. This is well illustrated by the following problem:

$$\begin{array}{l} \text{minimize} - x_1 - x_2 \\ \text{subject to } 1 - x_1^2 - x_2^2 \geq 0 \end{array} \tag{8.54}$$

in which the objective function is linear so that it is only the curvature of the constraint which causes a solution to exist. In this case the sequence followed by the SQP algorithm is only well-defined if the constraint curvature terms are included.

We can obtain a good approximation of this matrix using an extension of the BFGS update to the constrained case.

8.5.2 Numerical results of the SQP algorithm.

The following table compares the number of function evaluations and gradient evaluations required to solve Colville's (1969) [Col73] first three problems and gives some idea of the relative performances of the algorithms.

Problem	Extrapolated barrier function	Multiplier penalty function	SQP method (Powell, 1978a) [Pow77]
TP1	177	47	6
TP2	245	172	17
TP3	123	73	3

The excellent results of the SQP method are very attractive.

8.6 The final choice of a constrained algorithm

We will choose "*approach 1*". To cope with box and linear constraints, we will use an active set/null-space strategy. Non-linear constraints are handled using an SQP method in the reduced-space of the active constraints. The SQP method require a QP (quadratical program) in order to work.

In the next chapter, we will see a detailed description of this algorithm.

Chapter 9

Detailed description of the constrained step

The steps s_k of the constrained algorithm are the solution of:

$$\min_{s\in\Re^n} q(x_k+s) \equiv f(x_k) + \langle g_k, s\rangle + \tfrac{1}{2}\langle s, H_k s\rangle$$
$$\text{subject to } \begin{cases} b_l \leq x \leq b_u, & b_l, b_u \in \Re^n \\ Ax \geq b, & A \in \Re^{m\times n}, b \in \Re^m \\ c_i(x) \geq 0, & i = 1, \dots, l \\ \|s\|_2 < \Delta \end{cases}$$

The calculation of these steps will involves two main algorithm:

- a QP algorithm
- a SQP algorithm

We will give a detailed discussion of these two algorithms in the next sections. The last section of this chapter will summarize the algorithm which computes the constrained step.

9.1 The QP algorithm

The material of this section is based on the following reference: [Fle87].

We want to find the solution of:

$$\min_x g^t x + \frac{1}{2} x^t H x \quad \text{subject to} \quad Ax \geq b \tag{9.1}$$

where $A \in \Re^{m\times n}$ and $b \in \Re^m$.

We will assume in this chapter that H is positive definite. There is thus only one solution. We will first see how to handle the following simpler problem:

$$\min_x g^t x + \frac{1}{2} x^t H x \quad \text{subject to} \quad Ax = b \tag{9.2}$$

9.1.1 Equality constraints

Let Y and Z be $n \times m$ and $n \times (n-m)$ matrices respectively such that $[Y : Z]$ is non-singular, and in addition let $AY = I$ and $AZ = 0$. The solution of the equation $Ax = b$ is given by:

$$x = Yb + Zy \tag{9.3}$$

where y can be any vector. Indeed, we have the following: $Ax = A(Yb + Zy) = AYb + AZy = b$. The matrix Z has linearly independent columns $z_1, \dots, z_{n-m}$ which are inside the null-space of A and therefore act as bases vectors (or reduced coordinate directions) for the null space. At any point x any **feasible** correction δ can be written as:

$$\delta = Zy = \sum_{i=1}^{n-m} z_i y_i \tag{9.4}$$

where $y_1, \dots, y_{n-m}$ are the components (or reduced variables) in each reduced coordinate direction (see Figure 9.1).

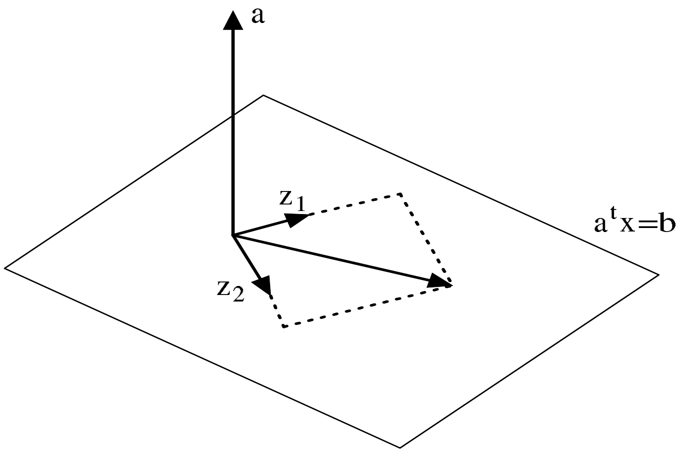

Figure 9.1: The null-space of A

Combining equation 9.2 and 9.3, we obtain the reduced quadratic function:

$$\psi(y) = \frac{1}{2} y^t Z^t H Z y + (g + HYb)^t Z y + \frac{1}{2}(g + HYb)^t Y b \tag{9.5}$$

If $Z^t H Z$ is positive definite then a minimizer y^* exists which solves the linear system:

$$(Z^t H Z) y = -Z^t (g + GYb) \tag{9.6}$$

The solution y^* is computed using Cholesky factorization. Once y^* is known we can compute x^* using equation 9.3 and g^* using the secant equation 13.29: $g^* = Hx^* + g$. Let's recall equation 13.19:

$$g(x) = \sum_{j \in E} \lambda_j \nabla c_j(x) \tag{9.7}$$

Let's now compute λ^*: From equation 9.7 we have:

$$g^* = A^t \lambda^* \tag{9.8}$$

Pre-Multiply 9.8 by Y^t and using $Y^t A^t = I$, we have:

$$Y^t g^* = \lambda^* \tag{9.9}$$

Depending on the choice of Y and Z, a number of methods exist. We will obtain Y and Z from a QR factorization of the matrix A^t (see annexe for more information about the QR factorization of a matrix). This can be written:

$$A^t = Q \begin{bmatrix} R \\ 0 \end{bmatrix} = [Q_1 \; Q_2] \begin{bmatrix} R \\ 0 \end{bmatrix} = Q_1 R \tag{9.10}$$

where $Q \in \Re^{n\times n}$ is an orthogonal matrix, $R \in \Re^{m\times m}$ is an upper triangular matrix, $Q_1 \in \Re^{n\times m}$ and $Q_2 \in \Re^{n\times(n-m)}$. The choices

$$Y = Q_1 (R^t)^{-1}, \quad Z = Q_2 \tag{9.11}$$

have the correct properties. Moreover the vector Yb in equation 9.3 and figure 9.2 is orthogonal to the constraints. The reduced coordinate directions z_i are also mutually orthogonal. Yb is calculated by forward substitution in $R^t u = b$ followed by forming $Yb = Q_1 u$. The multipliers λ^* are calculated by backward substitution in $R\lambda^* = Q_1^t g^*$.

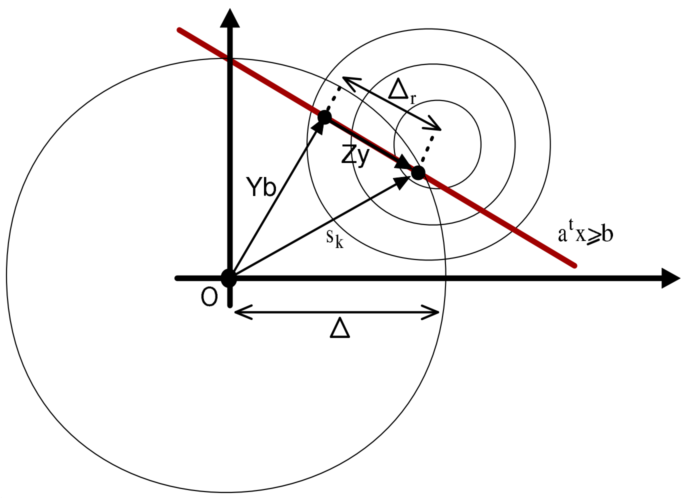

Figure 9.2: A search in the reduced space of the active constraints give as result y

9.1.2 Active Set methods

Certain constraints, indexed by the *Active set* $\mathcal{A}$, are regarded as equalities whilst the rest are temporarily disregarded, and the method adjusts this set in order to identify the correct active constraints at the solution to 9.1. Each iteration attempts to locate the solution to an *Equality Problem* (EP) in which only the active constraints occur. This is done by solving:

$$\min_x g^t x + \frac{1}{2} x^t H x \quad \text{subject to} \quad a_i x = b_i, \; i \in \mathcal{A} \tag{9.12}$$

Let's define, as usual, $x_{k+1} = x_k + \alpha_k s_k$. If x_{k+1} is infeasible then the length α_k of the step is chosen to solve:

$$\alpha_k = \min \left(1, \min_{i:i \notin \mathcal{A}} \frac{b_i - a_i x_k}{a_i s_k} \right) \tag{9.13}$$

If $\alpha_k < 1$ then a new constraint becomes active, defined by the index which achieves the min in 9.13, and this index is added to the active set $\mathcal{A}$. Suppose we have the solution x^* of the EP. We will now make a test to determine if a constraint should become inactive. All the constraints which have a negative associated Lagrange multiplier λ can become inactive. To summarize the algorithm, we have:

1. Set $k = 1, \ \ \mathcal{A} = \emptyset, \ \ x_1 =$ *a feasible point.*
2. Solve the EP.
3. Compute the Lagrange multipliers λ_k. If $\lambda_k \geq 0$ for all constraints then terminate. Remove the constraints which have negative λ_k.
4. Solve 9.13 and activate a new constraint if necessary.
5. set $k := k + 1$ and go to (2).

An illustration of the method for a simple QP problem is shown in figure 9.3. In this QP, the constraints are the bounds $x \geq 0$ and a general linear constraint $a^t x \geq b$.

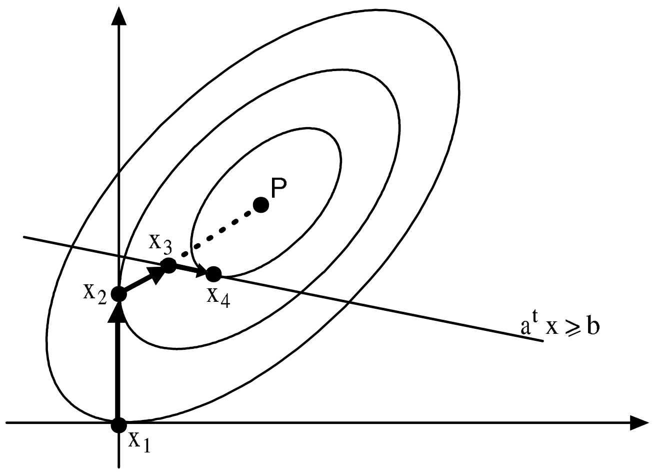

Figure 9.3: Illustration of a simple QP

9.1.3 Duality in QP programming

If H is positive definite, the dual of ($x \in \Re^n, b \in \Re^m$)

$$\min_x \frac{1}{2} x^t H x + x^t g \ \text{ subject to } Ax \geq b \tag{9.14}$$

is given by

$$\max_{x,\lambda} \frac{1}{2} x^t H x + x^t g - \lambda^t (Ax - b) \ \text{ subject to } \begin{cases} Hx + g - A^t \lambda = 0 \\ \lambda \geq 0 \end{cases} \tag{9.15}$$

By eliminating x from the first constraint equation, we obtain the bound constrained problem:

$$\max_{\lambda} \frac{1}{2}\lambda^t(AH^{-1}A^t)\lambda + \lambda^t(b + AH^{-1}g) - \frac{1}{2}g^tH^{-1}g \quad \text{subject to} \quad \lambda \geq 0 \tag{9.16}$$

This problem can be solved by means of the gradient projection method, which normally allows us to identify the active set more rapidly than with classical active set methods. The matrix $(AH^{-1}A^t)$ is semi-definite positive when H is positive definite. This is good. Unfortunately, if we have linearly dependent constraints then the matrix $(AH^{-1}A^t)$ becomes singular (this is always the case when $m > n$). This lead to difficulties when solving 9.16. There is, for example, no Cholesky factorization possible.

My first algorithm used gradient projection to identify the active set. It then project the matrix $(AH^{-1}A^t)$ into the space of the active constraints (the projection is straight forward and very easy) and attempt a Cholesky factorization of the reduced matrix. This fails very often. When it fails, it uses "steepest descent algorithm" which is sloooooooow and useless. The final implemented QP works in the primal space and use QR factorization to do the projection.

9.1.4 A note on the implemented QP

The QP which has been implemented uses normalization techniques on the constraints to increase robustness. It's able to start with an infeasible point. When performing the QR factorization of A^t, it is using pivoting techniques to improve numerical stability. It has also some very primitive technique to avoid cycling. Cycling occurs when the algorithm returns to a previous active set in the sequence (because of rounding errors). The QP is also able to "warm start". "Warm start" means that you can give to the QP an approximation of the optimal active set $\mathcal{A}$. If the given active set and the optimal active set $\mathcal{A}^*$ are close, the solution will be found very rapidly. This feature is particularly useful when doing SQP. The code can also compute very efficiently "Second Order Correction steps" (SOC step) which are needed for the SQP algorithm. The SOC step is illustrated in figure 9.4. The SOC step is perpendicular to the active constraints. The length of the step is based on Δb which is calculated by the SQP algorithm. The SOC step is simply computed using:

$$SOC = -Y\Delta b \tag{9.17}$$

The solution to the EP (equation 9.12) is computed in one step using a Cholesky factorization of H. This is very fast but, for badly scaled problem, this can lead to big rounding errors in the solution. The technique to choose which constraint enters the active set is very primitive (it's based on equation 9.13) and can also lead to big rounding errors. The algorithm which finds an initial feasible point when the given starting point is infeasible could be improved. All the linear algebra operations are performed with dense matrix.

This QP algorithm is very far from the "state-of-the-art". Some numerical results shows that the QP algorithm is really the weak point of all the optimization code. Nevertheless, for most problems, this implementation gives sufficient results (see numerical results).

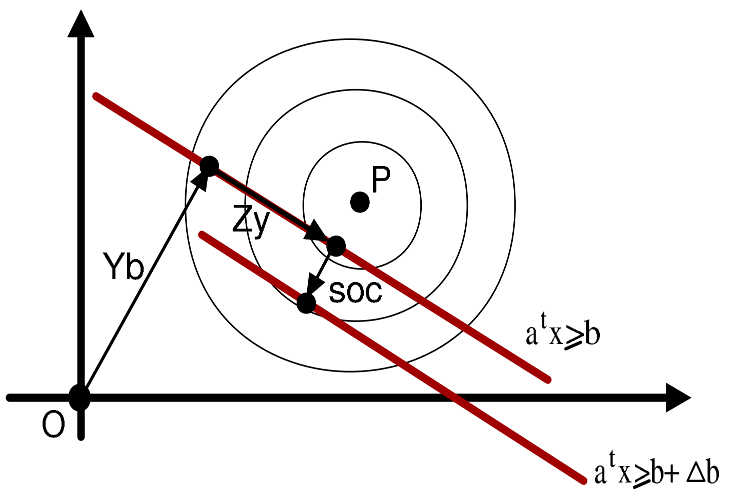

Figure 9.4: The SOC step

9.2 The SQP algorithm

The material of this section is based on the following references: [NW99, Fle87, PT93].

The SQP algorithm is:

1. set $k := 1$
2. Solve the QP subproblem described on equation 8.53 to determine δ_k and let λ_{k+1} be the vector of the Lagrange multiplier of the linear constraints obtained from the QP.
3. Compute the length α_k of the step and set $x_{k+1} = x_k + \alpha_k \delta_k$
4. Compute H_{k+1} from H_k using a quasi-Newton formula
5. Increment k. Stop if $\nabla \mathcal{L}(x_k, \lambda_k) \approx 0$. Otherwise, go to step 2.

We will now give a detailed discussion of each step. The QP subproblem has been described in the previous section.

9.2.1 Length of the step.

From the QP, we got a direction δ_k of research. To have a more robust code, we will apply the first Wolf condition (see equation 13.4) which is recalled here:

$$f(\alpha) \leq f(0) + \rho \alpha f'(0) \qquad \rho \in (0, \frac{1}{2}) \tag{9.18}$$

where $f(\alpha) = f(x_k + \alpha \delta_k)$. This condition ensure a "sufficient enough" reduction of the objective function at each step. Unfortunately, in the constrained case, sufficient reduction of the objective function is not enough. We must also ensure reduction of the infeasibility. Therefore, we will use a modified form of the first Wolf condition where $f(\alpha) = \phi(x_k + \alpha \delta_k)$ and $\phi(x)$ is a merit function. We will use in the optimizer the l_1 *merit function*:

$$\phi(x) = f(x) + \frac{1}{\mu} \|c(x)\|_1 \tag{9.19}$$

where $\mu > 0$ is called the penalty parameter. A suitable value for the penalty parameter is obtained by choosing a constant $K > 0$ and defining μ at every iteration too be

$$\mu^{-1} = \|\lambda_{k+1}\|_\infty + K \tag{9.20}$$

In equation 9.18, we must calculate $f'(0)$: the directional derivative of ϕ in the direction δ_k at x_k:

$$f'(0) = D_{\delta_k}\phi(x_k) = \delta_k^t g_k - \frac{1}{\mu}\left(\sum_{i\in\mathcal{F}_k}\delta_k^t a_i + \sum_{i\in\mathcal{Z}_k}\max(-\delta_k^t a_i, 0)\right) \tag{9.21}$$

where $\mathcal{F}_k = \{i : c_i(x_k) < 0\}$, $\mathcal{Z}_k = \{i : c_i(x_k) = 0\}$, $g_k = \nabla f(x_k)$, $a_i = \nabla c_i(x_k)$. The algorithm which computes the length of the step is thus:

1. Set $\alpha_k := 1$, $x_k :=$ *current point*, $\delta_k :=$ *direction of research*
2. Test the Wolf condition equation 9.18: $\phi(x_k + \alpha_k\delta_k) \lneqq \phi(x_k) + \rho\alpha_k D_{\delta_k}\phi(x_k)$
   - **True:** Set $x_{k+1} := x_k + \alpha_k\delta_k$ and go to step 3.
   - **False:** Set $\alpha_k = 0.7 * \alpha_k$ and return to the beginning of step 2.
3. We have successfully computed the length $\|\alpha_k\delta_k\|$ of the step.

9.2.2 Maratos effect: The SOC step.

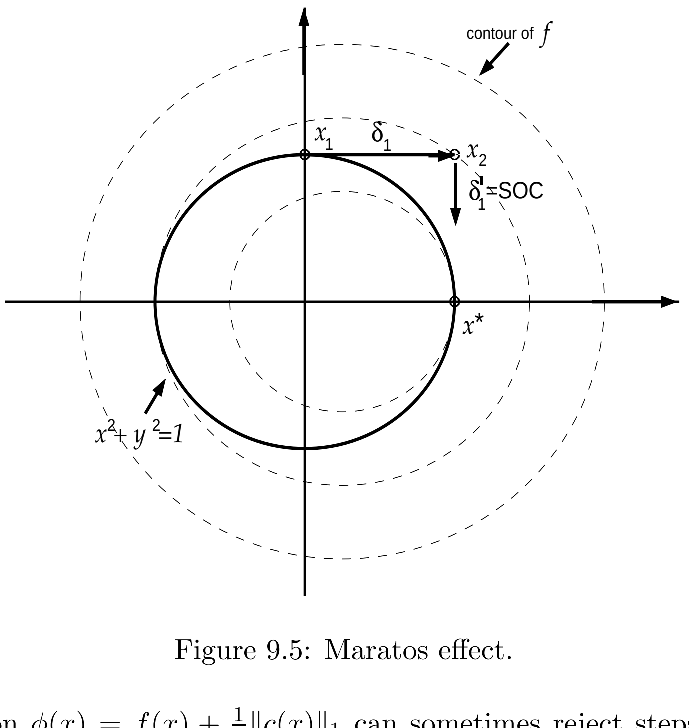

Figure 9.5: Maratos effect.

The l_1 merit function $\phi(x) = f(x) + \frac{1}{\mu}\|c(x)\|_1$ can sometimes reject steps (=to give $\alpha_k = 0$) that are making good progress towards the solution. This phenomenon is known as the Maratos effect. It is illustrated by the following example:

$$\min_{x,y} f(x,y) = 2(x^2 + y^2 - 1) - x, \quad \text{Subject to} \quad x^2 + y^2 - 1 = 0 \tag{9.22}$$

The optimal solution is $x^* = (1,0)^t$. The situation is illustrated in figure 9.5. The SQP method moves ($\delta_1 = (1,0)$) from $x_1 = (0,1)^t$ to $x_2 = (1,1)^t$.

In this example, a step $\delta_1 = \delta_k$ along the constraint will always be rejected ($\alpha_k = 0$) by $\phi = l_1$ *merit function*. If no measure are taken, the Maratos effect can dramatically slow down SQP methods. To avoid the Maratos effect, we can use a *second-order correction step* (SOC) in which we add to δ_k a step ${\delta_k}'$ which is computed at $c(x_k + \delta_k)$ and which provides sufficient decrease in the constraints. Another solution is to allow the merit function ϕ to increase on certain iterations (watchdog, non-monotone strategy).

Suppose we have found the SQP step δ_k. δ_k is the solution of the following QP problem:

$$\begin{aligned} &\min_\delta g_k^t \delta + \frac{1}{2}\delta^t H_k \delta \\ &\text{subject to: } \; c_i(x_k) + \nabla c_i(x_k)^t \delta \geq 0 \quad i = 1,\ldots,m \end{aligned} \tag{9.23}$$

Where we have used a linear approximation of the constraints at point x_k to find δ_k. Suppose this first order approximation of the constraint is poor. It's better to replace δ_k with s_k, the solution of the following problem, where we have used a quadratical approximation of the constraints:

$$\min_s g_k^t s + \frac{1}{2} s^t H_k s \tag{9.24}$$

$$\text{subject to: } \; c_i(x_k) + \nabla c_i(x_k)^t s + \frac{1}{2} s^t \nabla^2 c_i(x_k) s \geq 0 \quad i = 1,\ldots,m \tag{9.25}$$

but it's not practical, even if the hessian of the constraints are individually available, because the subproblem becomes very hard to solve. Instead, we evaluate the constraints at the new point $x_k + \delta_k$ and make use of the following approximation. Ignoring third-order terms, we have:

$$c_i(x_k + \delta_k) = c_i(x_k) + \nabla c_i(x_k)^t \delta_k + \frac{1}{2}\delta_k^t \nabla^2 c_i(x_k)\delta_k \tag{9.26}$$

We will assume that, near x_k, we have:

$$\delta_k^t \nabla^2 c_i(x_k)\delta_k \approx s\nabla^2 c_i(x_k) s \qquad \forall s \;\; small \tag{9.27}$$

Using 9.27 inside 9.26, we obtain:

$$\frac{1}{2} s\nabla^2 c_i(x_k) s \approx c_i(x_k + \delta_k) - c_i(x_k) - \nabla c_i(x_k)^t \delta_k \tag{9.28}$$

Using 9.28 inside 9.25, we obtain:

$$\begin{aligned} c_i(x_k) + &\nabla c_i(x_k)^t s + \frac{1}{2} s^t \nabla^2 c_i(x_k) s \\ &\approx c_i(x_k) + \nabla c_i(x_k)^t s + \Big(c_i(x_k + \delta_k) - c_i(x_k) - \nabla c_i(x_k)^t \delta_k \Big) \\ &= \Big(c_i(x_k + \delta_k) - \nabla c_i(x_k)^t \delta_k \Big) + \nabla c_i(x_k)^t s \end{aligned} \tag{9.29}$$

Combining 9.24 and 9.29, we have:

$$\begin{aligned} &\min_s g_k^t s + \frac{1}{2} s^t H_k s \\ &\text{subject to : } \; \Big(c_i(x_k + \delta_k) - \nabla c_i(x_k)^t \delta_k \Big) + \nabla c_i(x_k)^t s \geq 0 \quad i = 1,\ldots,m \end{aligned} \tag{9.30}$$

Let's define s_k the solution to problem 9.30. What we really want is ${\delta_k}' = s_k - \delta_k \Leftrightarrow s_k = \delta_k + {\delta_k}'$. Using this last equation inside 9.30, we obtain: ${\delta_k}'$ is the solution of

$$\begin{aligned}&\min_{\delta'} g_k^t(\delta_k + \delta') + \frac{1}{2}(\delta_k + \delta')^t H_k (\delta_k + \delta') \\ &\text{subject to: } c_i(x_k + \delta_k) + \nabla c_i(x_k)^t \delta' \geq 0 \quad i = 1, \dots, m\end{aligned}$$

If we assume that $\nabla f(x_k + \delta_k) \approx \nabla f(x_k)$ ($\Rightarrow H_k \delta_k \approx 0$)(equivalent to assumption 9.27), we obtain:

$$\begin{aligned}&\min_{\delta'} g_k^t \delta' + \frac{1}{2}\delta'^t H_k \delta' \\ &\text{subject to: } c_i(x_k + \delta_k) + \nabla c_i(x_k)^t \delta' \geq 0 \quad i = 1, \dots, m\end{aligned} \tag{9.31}$$

which is similar to the original equation 9.23 where the constant term of the constraints are evaluated at $x_k + \delta_k$ instead of x_k. In other words, there has been a small shift on the constraints (see illustration 9.4). We will assume that the active set of the QP described by 9.23 and the QP described by 9.31 are the same. Using the notation of section 9.1.1, we have:

$${\delta_k}' = -Yb' \quad \text{where } {b_i}' = c_i(x_k + \delta_k) \quad i = 1, \dots, m \tag{9.32}$$

Where Y is the matrix calculated during the first QP 9.23 which is used to compute δ_k. The SOC step is thus not computationally intensive: all what we need is an new evaluation of the active constraints at $x_k + \delta_k$. The SOC step is illustrated in figure 9.4 and figure 9.5. It's a shift perpendicular to the active constraints with length proportional to the amplitude of the violation of the constraints. Using a classical notation, the SOC step is:

$${\delta_k}' = -A_k^t (A_k A_k^t)^{-1} c(x_k + \delta_k) \tag{9.33}$$

where A_k is the jacobian matrix of the active constraints at x_k.

9.2.3 Update of H_k

H_k is an approximation of the hessian matrix of the Lagrangian of the optimization problem.

$$H_k \approx \nabla_x^2 \mathcal{L}(x_k, \lambda_k) = \nabla_x^2 f(x_k) - \sum_i (\lambda_k)_i \nabla^2 c_i(x_k) \tag{9.34}$$

The QP problem makes the hypothesis that H_k is definite positive. To obtain a definite positive approximation of $\nabla_x^2 \mathcal{L}$ we will use the damped-BFGS updating for SQP (with $H_1 = I$):

$$\begin{aligned}s_k &:= x_{k+1} - x_k \\ y_k &:= \nabla_x \mathcal{L}(x_{k+1}, \lambda_{k+1}) - \nabla_x \mathcal{L}(x_k, \lambda_{k+1}) \\ \theta_k &:= \begin{cases} 1 & \text{if } \; s_k^t y_k \geq 0.2 s_k^t H_k s_k, \\ \dfrac{0.8 s_k^t H_k s_k}{s_k^t H_k s_k - s_k^t y_k} & \text{otherwise} \end{cases} \\ r_k &:= \theta_k y_k + (1 - \theta_k) H_k s_k \\ H_{k+1} &:= H_k - \frac{H_k s_k s_k^t H_k}{s_k^t H_k s_k} + \frac{r_k r_k^t}{s_k^t r_k}\end{aligned} \tag{9.35}$$

The formula 9.35 is simply the standard BFGS update formula, with y_k replaced by r_k. It guarantees that H_{k+1} is positive definite.

9.2.4 Stopping condition

All the tests are in the form:

$$\|r\| \leq (1 + \|V\|)\tau \tag{9.36}$$

where r is a residual and V is a related reference value.

We will stop in the following conditions:

1. The length of the step is very small.
2. The maximum number of iterations is reached.
3. The current point is inside the feasible space, all the values of λ are positive or null, The step's length is small.

9.2.5 The SQP in detail

The SQP algorithm is:

1. set $k := 1$
2. If termination test is satisfied then stop.
3. Solve the QP subproblem described on equation 8.53 to determine δ_k and let λ_{k+1} be the vector of the Lagrange multiplier of the linear constraints obtained from the QP.
4. Choose μ_k such that δ_k is a descent direction for ϕ at x_k, using equation 9.20.
5. Set $\alpha_k := 1$
6. Test the Wolf condition equation 9.18: $\sigma(x_k + \alpha_k \delta_k) \overset{?}{\leq} \sigma(x_k) + \rho \alpha_k D_{\delta_k} \phi(x_k)$
   - **True:** Set $x_{k+1} := x_k + \alpha_k \delta_k$ and go to step 7.
   - **False:** Compute $\delta_k{}'$ using equation 9.32
     and test: $\sigma(x_k + \alpha_k \delta_k + \delta_k{}') \overset{?}{\leq} \sigma(x_k) + \rho \alpha_k D_{\delta_k} \phi(x_k)$
     - **True:** Set $x_{k+1} := x_k + \alpha_k \delta_k + \delta_k{}'$ and go to step 7.
     - **False:** Set $\alpha_k = 0.7 * \alpha_k$ and go back to the beginning of step 6.
7. Compute H_{k+1} from H_k using a quasi-Newton formula
8. Increment k. Stop if $\nabla \mathcal{L}(x_k, \lambda_k) \approx 0$ otherwise, go to step 1.

9.3 The constrained step in detail

The step s_k of the constrained algorithm are the solution of:

$$\min_{s \in \Re^n} q(x_k + s) \equiv f(x_k) + \langle g_k, s \rangle + \frac{1}{2} \langle s, H_k s \rangle$$
$$\text{subject to} \begin{cases} b_l \leq x \leq b_u, & b_l, b_u \in \Re^n \\ Ax \geq b, & A \in \Re^{m \times n}, b \in \Re^m \\ c_i(x) \geq 0, & i = 1, \ldots, l \\ \|s\|_2 < \Delta \end{cases}$$

We will use a null-space, active set approach. We will follow the notations of section 9.1.1.

1. Let λ be a vector of Lagrange Multiplier associated with all the linear constraints. This vector is recovered from the previous calculation of the constrained step. Set $k = 1$, $\mathcal{A} =$ *The constraints which are active are determined by a non-null* λ , $s_1 = 0$. If a λ associated with a non-linear constraint is not null, set *NLActive*= *true*, otherwise set *NLActive*= $false$.

2. Compute the matrix Y and Z associated with the reduced space of the active box and linear constraints. The active set is determined by $\mathcal{A}$.

3. We will now compute the step **in the reduced-space of the active box and linear constraints**. Check *NLActive*:

   - **True:** We will use the SQP algorithm described in the previous section to compute the step s_k. Once the step is calculated, check the Lagrange Multipliers associated with the non-linear constraints. If they are all null or negative, set *NLActive*= $false$.
   - **False:** We will use the Dennis-Moré algorithm of chapter 4 to compute the step. The step s_k is computed using:

     $$\begin{aligned} y_k &\text{ is the solution of} \\ &\min_y \frac{1}{2} y^t Z^t H_k Z y + (g_k + H_k Y b)^t Z y; \text{ subject to } \|y\|_2 < \Delta_r \\ s_k &= Yb + Zy_k \end{aligned}$$

     Where the trust region radius Δ_r used in the reduced spaced is $\Delta_r = \sqrt{\Delta^2 - \|Yb\|^2}$ as illustrated in figure 9.6.

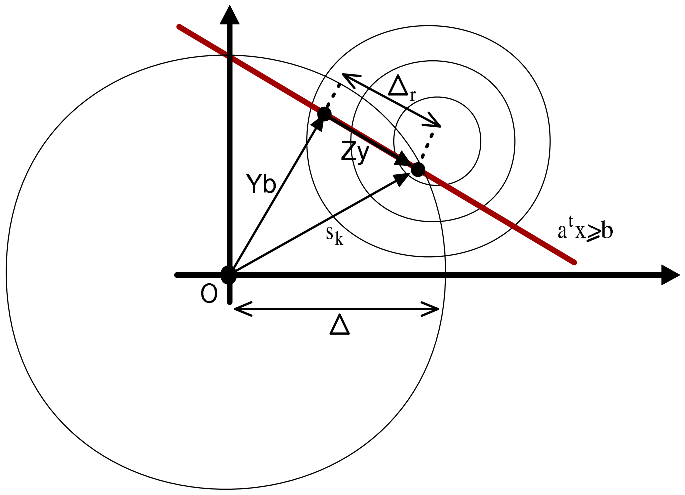

Figure 9.6: A search in the reduced space of the active constraints give as result y

4. Compute the Lagrange multipliers λ_k. If $\lambda_k \geq 0$ for all constraints then terminate. Remove from $\mathcal{A}$ the constraints which have negative λ_k.

5. Check if a non-linear constraint has been violated. If the test is true, set *NLActive*= *true*, set $k := k + 1$ and go to (2).

6. Solve 9.13 and add if necessary a new box or linear constraint inside $\mathcal{A}$. Set $k := k + 1$ and go to (2).

This is really a small, simple sketch of the implemented algorithm. The real algorithm has some primitive techniques to avoid cycling. As you can see, the algorithm is also able to "warm start", using the previous λ computed at the previous step.

9.4 Remarks about the constrained step.

This algorithm combines the advantages of both *trust region* and *line-search* worlds. We are using a *trust region* for his robustness and speed when confronted to highly non-linear objective function. We are using *line-search* techniques because of their superiority when confronted to non-linear constraints. When no non-linear constraints are active, we are using the Moré and Sorensen algorithm [CGT00c, MS83] which gives us high accuracy in step calculation and which leads to very fast convergence.

Chapter 10

Numerical Results for the constrained algorithm

We will compare CONDOR with other optimizers on constrained test problems from the set of Hock and Schittkowski [HS81]. All the constraints of the problems from the Hock and Schittkowski set were qualified as “easy” constraints. We list the number of function evaluations that each algorithm took to solve the problem. We also list the final function values that each algorithm achieved. We do not list the CPU time, since it is not relevant in our context. The “*” symbol next to the function values in the DFO column indicates that the algorithm terminated early because the trust region minimization algorithm failed to produce a feasible solution. Even when the DFO algorithm terminates early, it often does so after finding a local optimal solution, as in the case of HS54 and HS106. In other columns, the “*” indicates that an algorithm terminated early because the limit on the number of iterations was reached.

The descriptions in AMPL (”*Advanced Mathematical Programming Language*”) (see [FGK02] about AMPL) of the optimization problems are given in the code section 14.3. All the starting points are the default ones used in the literature and are found in the AMPL file.

The stopping tolerances of CONDOR and DFO was set to 10^{-4}, for the other algorithms the tolerance was set to appropriate comparable default values. In the constrained case, the stopping criteria algorithm for CONDOR is not very performant. Indeed, very often, the optimal value of the function is found since a long time and the optimizer is still running. The number in parenthesis for the CONDOR algorithm gives the number of function evaluations needed to reach the optimum (or, at least, the final value given in the table). Unfortunately, some extra, un-useful evaluations are performed. The total is given in the column 1 of the CONDOR results. In particular, for problem hs268, the optimum is found after only 25 function evaluations but CONDOR still performs 152 more useless evaluations (leading to the total of 177 evaluations reported in the table).

All algorithms were implemented in Fortran 77 in double precision except COBYLA which is implemented in Fortran 77 in single precision and CONDOR which is written in C++ (in double precision). The trust region minimization subproblem of the DFO algorithm is solved by NPSOL [GMSM86], a fortran 77 non-linear optimization package that uses an SQP approach.

Name	Dim	Number of Function Evaluation				final function value			
		CONDOR	DFO	LAN.	COB.	CONDOR	DFO	LAN.	COB.
hs022	2	13 (1)	15	24	24	1.0000e+00	1.0000e+00	1.0000e+00	1.0000e+00
hs023	2	11 (9)	16	1456	28	2.0000e+00	2.0000e+00	2.0000e+00	2.0000e+00
hs026	3	115 (49)	49	1677	1001	2.7532e-13	1.9355e-09	6.1688e-09	1.0250e-07
hs034	3	21 (18)	22	405	41	-8.3403e-01	-8.3403e-01	-8.3403e-01	-8.3403e-01
hs038	4	311 (245)	536	128	382	7.8251e-13	1.6583e-07	4.3607e-13	7.8770e+00
hs044	4	23 (18)	26	35	45	-1.5000e+01	-1.5000e+01	-1.5000e+01	-1.5000e+01
hs065	3	20 (17)	35	1984	102	9.5352e-01	9.5352e-01	9.5352e-01	9.5352e-01
hs076	4	21 (17)	29	130	76	-4.6818e+00	-4.6818e+00	-4.6818e+00	-4.6818e+00
hs100	7	48 (38)	127	6979	175	6.8199e+02	6.8063e+02	6.9138e+02*	6.8063e+02
hs106	8	201 (103)	63	196	93	8.9882e+03	7.0492e+03*	1.4995e+04	1.4994e+04
hs108	9	90 (77)	62	2416	177	-7.8167e-01	-8.6603e-01	-8.6602e-01	-8.6603e-01
hs116	13	141 (118)	87	12168	119	8.0852e+01	9.7485e+01*	5.0000e+01	3.5911e+02
hs268	5	177 (25)	84	5286	676	-1.5917e-11	-1.8190e-11	2.6576e+01	5.9238e+00

From the experimental results, we can see that the CONDOR algorithm performs very well compared to the other algorithms on problems with linear constraints only. On these problems (hs038, hs044, hs076, hs268), the step is always calculated using a reduced-space version of the Moré and Sorensen algorithm of Chapter 4 and is thus very accurate. When using the Moré and Sorensen algorithm, the variable ϵ needed to use the bound of Equation 6.6 is computed. We have seen in Section 7.2 that:

- Equation 6.6 is not used by DFO.
- Equation 6.6 allows very fast convergence speed.

These facts explain the superior performance of CONDOR on box and linear constraints.

The starting point of problem hs022 is infeasible. When this situation occurs, CONDOR searches for the closest feasible point from the given starting point. Once this "feasibility restoration phase" is finished, the real optimization process begins: the optimizer asks for evaluations of the objective function. In problem hs022 the point which is found after the "feasibility restoration phase" is also the optimum of the objective function. This explains the value "1" in parenthesis for this problem for the CONDOR algorithm.

For the problem hs116, the value of the objective function at the optimum is 97.588409. The problem is so badly scaled that LANCELOT and CONDOR are both finding an infeasible point as result. Among all the problems considered, this difficulty arises only for problem hs116. During the development of the optimizer, I had to laugh a little because, on this problem, on some iterations, the Hessian matrix is filled with number of magnitude around 10^{40} and the gradient vector is filled with number of magnitude around 10^{-10}. That's no surprise that the algorithm that compute the step is a little "lost" in these conditions!

The problems where chosen by A.R.Conn, K.Schneinberg and Ph.L.Toint to test their DFO algorithm, during its development. Thus, the performance of DFO on these problems is expected to be very good or, at least, good. The problems hs034, hs106 and hs116 have linear objective functions. The problems hs022, hs023, hs038, hs044, hs065, hs076, hs100 and hs268 have quadratical objective functions. The problems hs026, hs065 and hs100 have few simple non-linear terms in the objective function. The problems are not chosen so that CONDOR outperforms the other algorithms. The test set is arbitrary and not tuned for CONDOR. However, this test-set has been used during CONDOR's development for debug purposes. In fact, this test-set is particularly well-suited for DFO because the objective function is very "gentle", mostly linear with very few quadratic/non-linear terms. In these conditions, using a near-linear model for the objective function is a good idea. That's exactly what DFO does (and not CONDOR). The main difficulty here is really the constraints.

On the problem hs268 which has only simple linear constraints, Lancelot and COBYLA are both failing. The constraints on these problems are qualified as "gentle" but is it really the case? CONDOR was tested on many other more gentle, constrained objective functions during its development and everything worked fine.

The less good results are obtained for problems with badly scaled non-linear constraints (hs108, h116). On these problems, the quality of the QP algorithm is very important. The home-made QP shows unfortunately its limits. A good idea for future extension of CONDOR is to implement a better QP.

These results are very encouraging. Despite its simple implementation, CONDOR is competitive compared to high-end, commercial optimizers. In the field of interest, where we encounter mostly box or linear constraints, CONDOR seems to have the best performances (at least, on this reduced test set).

Chapter 11

The METHOD project

My work on optimization techniques for high computing load, continuous function without derivatives available is financed by the LTR European project METHOD (METHOD stands for Achievement Of Maximum Efficiency For Process Centrifugal Compressors THrough New Techniques Of Design). The goal of this project is to optimize the shape of the blades inside a Centrifugal Compressor (see the figure 11.1 for an illustration of the blades of the compressor).

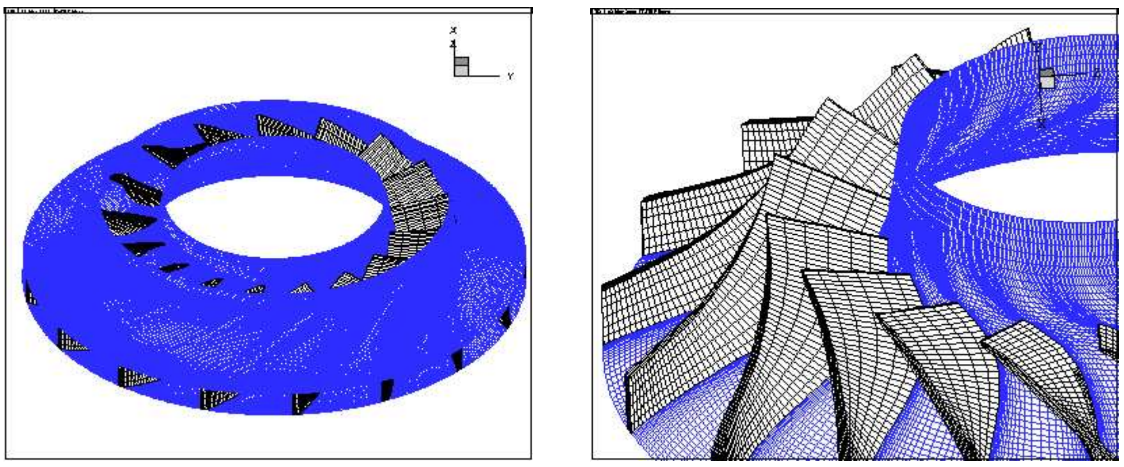

Figure 11.1: Illustration of the blades of the compressor

The shape of the blades is described by 31 parameters. The objective function is computed in the following way:

- Based on the 31 parameters, generate a 3D grid.
- Use this grid to simulate the flow of the gas inside the turbine.
- Wait for stationary conditions (often compute during 1 hour).
- Extract from the simulation the outlet pressure, the outlet velocy, the energy transmit to the gas at stationary conditions.
- Aggregate all these indices in one general overall number representing the quality of the turbine. (That's this number we are optimizing.)

We simply have an objective function which takes as input a vector of 31 parameters and gives as output the associated quality. We want to maximize this function. This objective function has been implemented by the Energetics Department “Sergio Stecco” of the ”Università degli Studi di Firenze” (DEF). For more information, see [PMM+03].

11.1 Parametrization of the shape of the blades

A shape can be parameterized using different tools:

- Discrete approach (fictious load)
- Bezier & B-Spline curves
- Uniform B-Spline (NURBS)
- Feature-based solid modeling (in CAD)

In collaboration with DEF, we have decided to parameterize the shape of the blade using “Bezier curves”. An illustration of the parametrization of an airfoil shape using Bezier curves is given in figures 11.2 and 11.3. The parametrization of the shape of the blades has been designed by DEF.

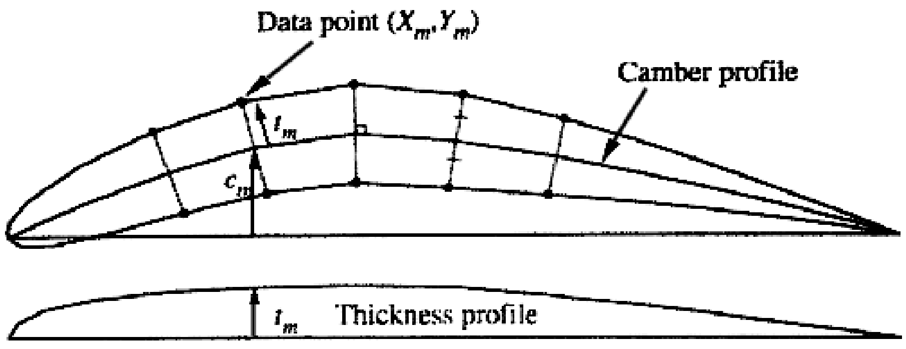

Figure 11.2: Superposition of thickness normal to camber to generate an airfoil shape

Some set of shape parameters generates infeasible geometries. The ”feasible space” of the constrained optimization algorithm is defined by the set of parameters which generates feasible geometries. A good parametrization of the shape to optimize should only involve box or linear constraints. Non-linear constraints should be avoided.

In the airfoil example, if we want to express that the thickness of the airfoil must be non-null, we can simply write $b_8 > 0, b_{10} > 0, b_{14} > 0$ (3 box constraints) (see Figure 11.3 about b_8, b_{10} and b_{14}). Expressing the same constraint (non-null thickness) in an other, simpler, parametrization of the airfoil shape (direct description of the upper and lower part of the airfoil using 2 bezier curves) can lead to non-linear constraints. The parametrization of the airfoil proposed here is thus very good and can easily be optimized.

11.2 Preliminary research

Before implementing CONDOR, several kinds of optimizer have been tested. The following table describe them:

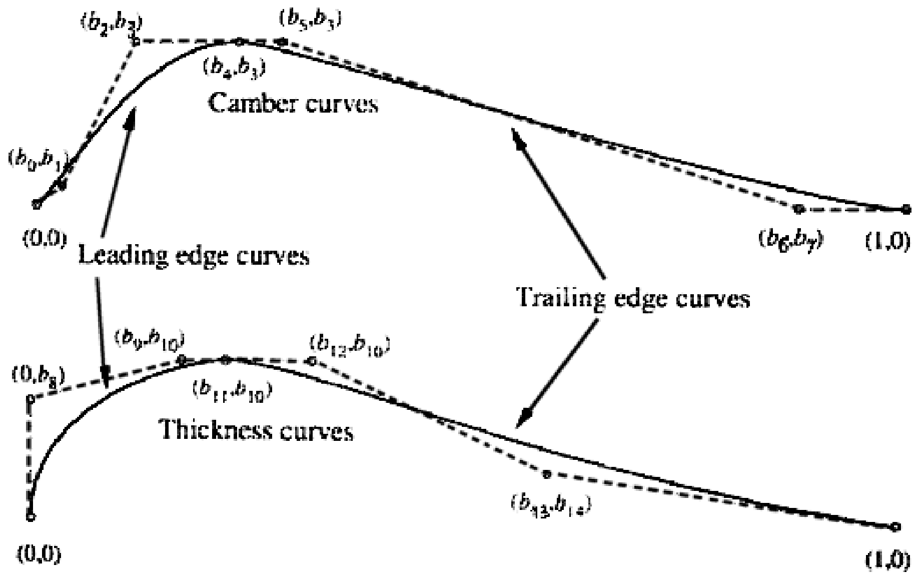

Figure 11.3: Bezier control variable required to form an airfoil shape

Available Optimization algorithms	Derivatives required	Type of optimum	constraints	Number of design variables	Type of design variables	Noise
Pattern Search (Discrete Rosenbrock's method, simplex, PDS,...)	no	local	**box**	Large	continuous	Small
Finite-differences Gradient-Based Approach (FSQP, Lancelot,NPSOL,...)	**yes**	local	Non-linear	medium	continuous	**Nearly no noise**
Genetic Algorithm	no	global	**box**	**small**	mixed	Small
Gradient-Based Approach using Interpolation techniques (CONDOR, UOBYQA, DFO)	no	local	Non-linear	medium	continous	Small

The pattern search and the genetic algorithm were rejected because numerical results on simple test functions demonstrated that they were too slow (They require many function evaluations). Furthermore, we can only have box constraints with these kinds of algorithms. This is insufficient for the method project where we have box, linear and non-linear constraints. The finite-difference gradient-based approach (more precisely: the FSQP algorithm) was used. The gradient was computed in parallel in a cluster of computers. Unfortunately, this approach is very sensitive to the noise, as already discussed in Section 7.4. The final, most appropriate solution, is CONDOR.

11.3 Preliminary numerical results

The goal of the objective function is to maximize the polytropic efficiency of the turbine keeping the flow coefficient and the polytropic head constant:

$$f = -(\eta_p)^2 + 10((\tau\eta_p)_{req} - \tau\eta_p)^2$$

where $(\tau\eta_p)_{req}$ represents the required polytropic head.

The optimization of 9 out of 31 parameters (corresponding to a 2D compressor) using CFSQP gives as result an increase of 3.5 % in efficiency (from 85.3 % to 88.8 %) in about 100 functions evaluations (4 days of computation). The CFSQP stopped prematurely because of the noise on the objective function.

11.4 Interface between CONDOR and XFLOS / Pre-Solve phase

We want to optimize optimizes a non-linear function

$$y = \mathcal{F}(x), \; x \in \Re^n \; y \in \Re \qquad \text{Subject to: } \begin{cases} b_l \leq x \leq b_u, & b_l, b_u \in \Re^n \\ Ax \geq b, & A \in \Re^{m \times n}, b \in \Re^m \\ c_i(x) \geq 0, & i = 1, \ldots, l \end{cases} \tag{11.1}$$

For the Method project, the objective function $\mathcal{F}(x)$ is an external code (an executable program): XFLOS. I developed a simple interface between CONDOR and XFLOS. This interface is configured via a configuration text file: "optim.cfg". I will now describe the content of "optim.cfg".

- **Parameter 1:**
  The filename of the executable that must be launched to run XFLOS.
- **Parameters 2 & 3:**
  The information exchange between CONDOR and XFLOS is based on files written on the hard drive. There are two files: The first one is a file written by CONDOR to tell XFLOS what's the current point we want to evaluate. The second one is written by XFLOS and is the result of the evaluation. Parameters 2 & 3 are the name of these files.
- **Parameters 4:**
  Dimension of the search space (31)
- **Parameters 5,6 & 7:**
  The result of a run of XFLOS is a vector of 20 values which must be aggregated into one single value which is the value of the objective function at the current point. The aggregation is based on Parameters 5,6 & 7.
- **Parameter 8:**
  In the industry, there are two kinds of impellers:
  - 2D impellers
  - 3D impellers

The 2D impellers are simpler to manufacture and are thus cheaper. The set of parameters describing a 2D impeller is a subset (9 parameters) of the 31 parameters needed to describe a 3D impeller. When optimizing a 2D impeller we must fix 22 parameters and only optimize 9 parameters. Config-file-Parameter 8 sets the variables to optimize (=active variable) and those which must be fixed. Let's define $\mathcal{J}$, the set of active variables.

Warning ! If you want, for example, to optimize $n+m$ variables, never do the following:

- Activate the first n variables, let the other m variables fixed, and run CONDOR (Choose as starting point the best point known so far)
- Activate the second set of m variables, let the first set of n variables fixed, and run CONDOR (Choose as starting point the best point known so far).
- If the stopping criteria is met then stop, otherwise go back to step 1.

This algorithm is really bad. It will results in a very slow linear speed of convergence as illustrated in Figure 11.4. The config-file-parameter 8 allows you to activate/deactivate some variables, it's sometime a useful tool but don't abuse from it! Use with care!

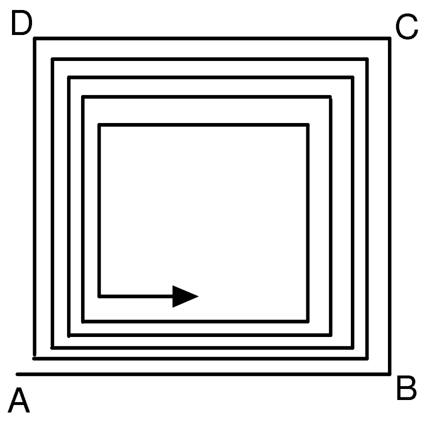

Figure 11.4: Illustration of the slow linear convergence when performing consecutive optimization runs with some variables deactivated.

- **Parameter 9:**
  Starting point x_{start}.
- **Parameter 10:**
  If some runs of XFLOS have already been computed and saved on the hard drive, it's possible to tell CONDOR to use the old evaluations (*warm start*). If a *warm start* is performed, the evaluations needed to build the first quadratic will be lowered. Beside, it may be interesting to use as starting point the best point known so far, instead of value specified at parameter 9. Parameter 10 tells to CONDOR which alternative it must use for the choice of the starting point.
- **Parameter 11:**
  Lower bound on x: b_l
- **Parameter 12:**
  Upper bound on x: b_u

- **Parameter 13:**
  Number m of linear inequalities $Ax \geq b \;\; A \in \Re^{m \times n}, b \in \Re^m$ for constrained optimization.

- **Parameter 14:**
  The linear inequalities are described here. Each line represents a constraint. On each line, you will find: $A_{ij} \quad (j = 1, \ldots, n)$ and b_i. Using parameter 7, we can let some variables fixed. In this case, some linear constraints may:

  - be simply removed:
    $A_{ij} = 0 \qquad j \in \mathcal{J} \qquad (A_{ij}$ is zero for all the active variables)
    ($\mathcal{J}$ is defined using config-file-parameter 8).
  - be removed and replaced by a tighter bound on the variable x_k $(k \in \mathcal{J})$
    $A_{ij} = 0 \qquad j = \mathcal{J} \backslash \{k\} \quad$ and $\quad A_{ik} \neq 0$.
    The k^{th} component of b_l or b_u will maybe be updated.

  This simple elimination of some linear constraints is implemented inside CONDOR. It is called in the literature the "pre-solve phase".

- **Parameter 15:**
  The normalization factor for each variables (see Section 12.3 about normalization).

- **Parameter 16:**
  The stopping criteria: ρ_{end}. This criteria is tested inside the normalized space.

- **Parameter 17:**
  We consider that the evaluation of the objective function inside XFLOS has failed when the result of this evaluation is greater than parameter 17. It means that a "virtual constraint" has been encountered. See Section 12.2.2 for more information.

- **Parameter 18:**
  In the method project, some constraints are non-linear and have been hard-coded inside the class "METHODObjectiveFunction". Parameter 18 activates/deactivates these non-linear constraints.

- **Parameter 19:**
  This parameter is the name of the file which contains all the evaluations of the objective function already performed. This file can be used to *warm start* (see parameter 10).

- **Parameter 20:**
  When XFLOS is running, CONDOR is waiting. CONDOR regularly checks the hard drive to see if the result file of XFLOS has appeared. Parameter 20 defines the time interval between two successive checks.

Here is a simple example of a "server configuration file":

```
; number of CPU's (not used currently)
;1

; IP's (not used currently)
;127.0.0.1

; blackbox objective function
;/home/andromeda_2/fld_user/METHOD/TD21/splitter
/home/fvandenb/L6/splitter
;testoptim

; objective function: input file:
/home/fvandenb/L6/000/optim.out
```

```
; objective function: output file:
/home/fvandenb/L6/000/xflos.out

; number of input variables (x vector) for the objective function
31

; The objective function is ofunction= sum_over_i ( w_i * ( O_i - C_i)^(e_i) )
; of many variables. The weights (w_i) are:
; 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19     20
  0 0 0 0 0 0 0 0 0  0  0  0  0  0  0  0 20  0 -1     20

; C_i are:
; 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17       18 19     20
  0 0 0 0 0 0 0 0 0  0  0  0  0  0  0  0  0.0956  0  0  0.521

; e_i are:
; 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19     20
  2 2 2 2 2 2 2 2 2  2  2  2  2  2  2  2  2  2  2      2

; optimization of only a part of the variables:
;dz zh1 rh2 zs1 rs2 sleh sles sh1 sh2 th1 th3 ss1 ss2 ts0 ts1 ts3 tkh tks b0 b2 r2  r0 delt bet2 str coel dle el11 $
$el12 el21 el22
; 1  2   3   4   5   6    7   8   9   10  11  12  13  14  15  16  17  18  19 20 21  22   23  24   25  26   27   28   $
$29   30   31
  0  1   1   1   1   0    0   1   1   1   1   1   1   1   1   1   1   1   0  0   0   0    1   1   1    1   1    1    1$
$    1    1

; a priori estimated x (starting point)-
;     1     2     3     4     5     6   7   8      9       10  11  12   13     14       15    16    17      18     19 $
$   20
;   21   22       23       24    25   26    27    28    29    30   31
;    dz   zh1   rh2   zs1   rs2   sleh  sles   sh1 sh2    th1      th3 ss1 ss2  ts0    ts1      ts3   tkh   tks      $
$b0     b2    r2
;   r0  delt     bet2      str coel  dle   el11   el12   el21   el22
0.1068 0.0010 0.055555 0.0801 0.2245 0.0096 0.000 0.3 0.37 -0.974 -1.117010721 0.297361 0.693842 -0.301069296  -0.8 $
$-1.117010721  0.004 0.0051 0.07647 0.035 0.225 0.07223 0.0 -0.99398 0.00 4.998956  0.000  0.000  0.000  0.000  0.0000

; use previous line as starting point:
; - 1: yes
; - 0: no, use best point found in database.
1

; lower bounds for x
; 1   2   3   4   5   6   7   8   9   10   11  12   13   14   15   16   17   18  19  20  21   22    23   24   25   26 $
$  27   28   29   30   31
;dz zh1 rh2 zs1 rs2 sleh sles sh1 sh2  th1  th3 ss1 ss2  ts0  ts1  ts3  tkh   tks  b0  b2  r2  r0 delt  bet2  str coel$
$  dle el11 el12 el21 el22
  0   0   0   0   0   0   0   0   0 -1.7 -1.7   0  0  -1.7 -1.7 -1.7 .002  .002   0   0   0 .05    0  -1.0  -.5  0.1 $
$-0.05    0    0    0    0

; upper bounds for x
; 1   2   3   4   5   6   7   8   9   10   11  12   13   14   15   16   17   18  19  20  21   22    23   24   25   26 $
$  27   28   29   30   31
;dz zh1 rh2 zs1 rs2 sleh sles sh1 sh2  th1  th3 ss1 ss2  ts0  ts1  ts3  tkh   tks  b0  b2  r2  r0 delt  bet2  str coel$
$  dle el11 el12 el21 el22
  1   1   2   1   2   1   1   1   1  1.7  1.7   1  1   1.7  1.7  1.7  .05   .05   2   1   2   1    2  -0.4   .5  10   $
$0.05  0.02  0.02  0.02  0.02

; number of inequalities
15
;0

; here would be the matrix for inequalities definition if they were needed
; 1   2   3   4   5   6   7    8   9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24   25    26   27   28 $
$  29   30   31
;dz zh1 rh2 zs1 rs2 sleh sles sh1 sh2 th1 th3 ss1 ss2 ts0 ts1 ts3 tkh tks  b0  b2  r2  r0 delt bet2 str coel  dle el11$
$ el12 el21 el22  RHS
 -1   0   0   0   0   0   0    0   0   0   0   0   0   0   0   0   0   0   0   1   0   0   0   0   1    0    0    0 $
$   0    0    0    0
  0   0   1   0   0   0   0    0   0   0   0   0   0   0   0   0   0   0   0   0  -1   0   1   0   0    0    0    0 $
$   0    0    0    0
  0   0   0   0  -1   0   0    0   0   0   0   0   0   0   0   0   0   0   1   0   0   1   0   0   0    0    0    0 $
$   0    0    0    0
  0   0   0   0   1   0   0    0   0   0   0   0   0   0   0   0   0   0   0   0  -1   0   1   0   0    0    0    0 $
$   0    0    0    0
  0   0   0   0   0   1   0    0   0   0   0   0   0   0   0   0   0   0   0   0  -1   0   0   0   0    0    0    0 $
$   0    0    0    0
  0   0   0   0   0   0   1    0   0   0   0   0   0   0   0   0   0   0   0   0  -1   0   0   0   0    0    0    0 $
$   0    0    0    0
  0   0   0   0   0   0   0    1  -1   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0    0    0    0 $
$   0    0    0    0
  0   0   0   0   0   0   0    0   0   0   0   1  -1   0   0   0   0   0   0   0   0   0   0   0   0    0    0    0 $
$   0    0    0    0
  0   0   0   0   0   0   0    0   0   0   1   0   0   0   0  -1   0   0   0   0   0   0   0   0   0    0    0    0 $
$   0    0    0  .35
  0   0   0   0   0   0   0    0   0   0  -1   0   0   0   0   1   0   0   0   0   0   0   0   0   0    0    0    0 $
$   0    0    0  .35
  0   0   0   0   0   0   0    0   0   0   0   0   0   0   0   0   0   0   1   0  -1   1   1   0   0    0    0    0 $
$   0    0    0    0
  0   0   0   0   0   0   0    0   0   0   0   0   0   0   0   0   0   0   0  -1   0   0   0   0   0    0    0    1 $
$   0    0    0    0
  0   0   0   0   0   0   0    0   0   0   0   0   0   0   0   0   0   0   0  -1   0   0   0   0   0    0    0    0 $
$   1    0    0    0
```

```
  0   0   0   0   0   0   0    0   0   0   0   0   0   0   0   0   0   0   0  -1   0   0   0   0   0    0     0    0 $
$   0    1    0    0
  0   0   0   0   0   0   0    0   0   0   0   0   0   0   0   0   0   0   0  -1   0   0   0   0   0    0     0    0 $
$   0    0    1    0

; scaling factor for the normalization of the variables.
; 1     2    3    4    5    6    7    8    9   10   11   12   13   14   15   16   17   18   19   20   21   22   23   $
$24   25   26   27   28   29   30   31
; dz  zh1  rh2  zs1  rs2 sleh sles  sh1  sh2  th1  th3  ss1  ss2  ts0  ts1  ts3  tkh  tks   b0   b2   r2   r0 delt $
$bet2  str coel  dle el11 el12 el21 el22
1e-3 1e-3 1e-3 1e-3 1e-3 1e-3 1e-3 1e-2 1e-2 1e-2 1e-2 1e-2 1e-2 1e-2 1e-2 1e-2 5e-4 5e-4 1e-3 1e-3 1e-3 1e-3 1e-3 1e$
$-2 1e-3 1e-2 1e-3 1e-3 1e-3 1e-3 1e-3


; stopping criteria \rho_end=
1e-4
;1e-8

; bad value of the objective function
3

; Nuovo Pignone non-linear constaints hard coded into code must be
; - activated : 1
; - desactivated: 0
;1
1

; the data are inside a file called:
;/home/andromeda_2/fld_user/METHOD/TD21/data.ll
/home/fvandenb/L6/dataNEW.ll

; when waiting for the result of the evaluation of the objective
; function, we check every xxx seconds for an arrival of the file
; containing the results
3
```

11.4.1 Config file on client node

When running CONDOR inside a cluster of computers (to do parallel optimization), the user must start on each client-node the CONDOR-client-software. This software is performing the following:

1. Wait to receive from the server a sampling site (a point) (using nearly no CPU time).
2. Evaluate the objective function at this site and return immediately the result to the server. Go to step 1.

Each client node will use a different “client configuration file”. These files contain simply the parameters 1 to 7 of the ”server configuration file” (described in Section 11.4). The IP addresses and the port numbers of the client nodes are currently still hard coded inside the code (at the beginning of file “parallel.cpp”).

11.5 Lazy Learning, Artificial Neural Networks and other functions approximators used inside a optimizer.

At the beginning of my research on continuous optimization, I was using the ”Lazy Learning” as local model to guide the optimization process. Here is a brief resume of my unsuccessful experiences with the Lazy Learning.

Lazy learning is a local identification tool. Lazy learning postpones all the computation until an explicit request for a prediction is received. The request is fulfilled by modelling locally the relevant data stored in the database and using the newly constructed model to answer the request (see figure 11.5). Each prediction (or evaluation of the LL model) requires therefore a

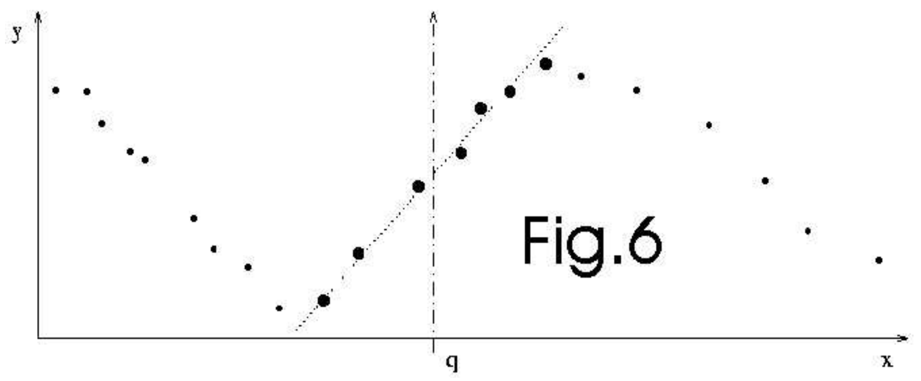

Figure 11.5: Lazy Learning identification

local modelling procedure. We will use, as regressor, a simple polynomial. Lazy Learning is a regression technique meaning that the newly constructed polynomial will NOT go exactly through all the points which have been selected to build the local model. The number of points used for the construction of this polynomial (in other words, the kernel size or the bandwidth) and the degree of the polynomial (0,1, or 2) are chosen using leave-one out cross-validation technique. The identification of the polynomial is performed using a recursive least square algorithm. The cross-validation is performed at high speed using PRESS statistics.

What is leave-one out cross-validation? This is a technique used to evaluate to quality of a regressor build on a dataset of n points. First, take $n-1$ points from the dataset (These points are the training set). The last point P will be used for validation purposes (validation set). Build the regressor using only the points in the training set. Use this new regressor to predict the value of the only point P which has not been used during the building process. You obtain a fitting error between the true value of P and the predicted one. Continue, in a similar way, to build regressors, each time on a different dataset of $n-1$ points. You obtain n fitting errors for the n regressors you have build. The global cross-validation error is the mean of all these n fitting errors.

We will choose the polynomial which has the lowest cross-validation error and use it to answer the query.

We can also choose the k polynomials which have the lowest cross-validation error and use, as final model, a new model which is the weighted average of these k polynomials. The weight is the inverse of the leave-one out cross-validation error. In the Lazy learning toolbox, other possibilities of combination are possible.

When using artificial neural networks as local model, the number of neurons on the hidden layer is usually chosen using a *loocv* technique.
A simple idea is to use the leave-one out cross-validation error (*loocv*) to obtain an estimate of the quality of the prediction. An other simple idea is to use the *loocv* to assess the *validity* of the model used to guide the optimization process. The *validity* of the local model is a very important notion. In CONDOR, the validity of the model is checked using equation 3.37 of chapter 3.4.2. No guarantee of convergence (even to a local optima) can be obtained without a local model which is *valid*.

When using Lazy Learning or Artificial Neural Networks as local models, we obtain the *loocv*, as a by-product of the computation of the approximator. Most of the time this *loocv* is used to assess the *quality* of the approximator. I will now give a small example which demonstrate that the *loocv* cannot be used to check the *validity* of the model. This will therefore demonstrate that Lazy Learning and Artificial Neural Networks cannot be used as local model to correctly guide the optimization process. Lazy Learning or Neural Networks can still be used but an external algorithm (not based on *loocv*) assessing their *validity* must be added (this algorithm is, most of the time, missing).

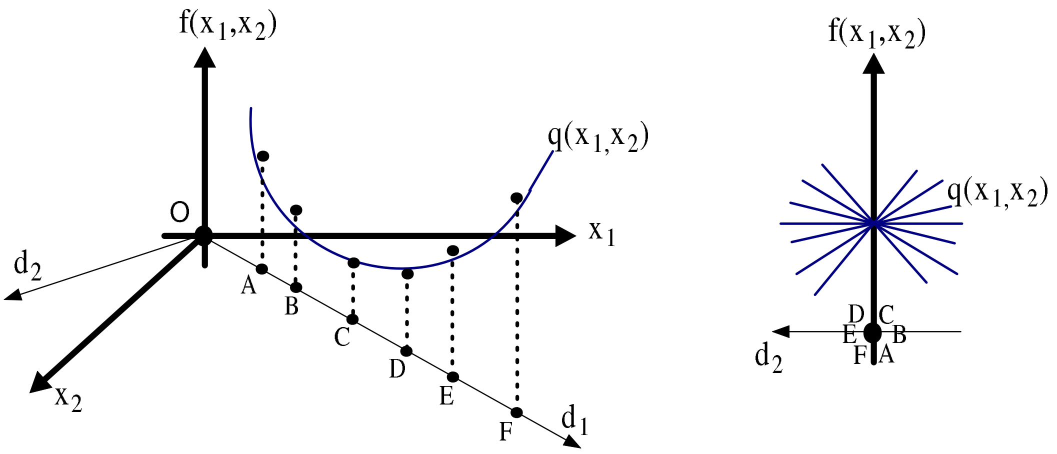

Figure 11.6: *loocv* is useless to assess the *quality* of the model

Let's consider the identification problem illustrated in figure 11.6. The local model $q(x_1, x2)$ of the function $f(x_1, x_2)$ to identify is build using the set of (points,values): $\{(A, f(A)); (B, f(B)); (C, f(C)); (D, f(D)); (E, f(E)); (F, f(F))\}$. All these points are on the line d_1. You can see on the left of figure 11.6 that the leave-one out cross-validation error is very low: the approximator intercepts nearly perfectly the dataset. However there lacks some information about the function $f(x_1, x_2)$ to identify: In the direction d_2 (which is perpendicular to d_1) we have no information about $f(x_1, x_2)$. This leads to a simple infinity of approximators $q(x_1, x_2)$ which are all of equivalent (poor) quality. This infinity of approximators is illustrated on the right of figure 11.6. Clearly the model $q(x_1, x_2)$ cannot represents accurately $f(x_1, x_2)$. $q(x_1, x_2)$ is thus *invalid*. This is in contradiction with the low value of the *loocv*. Thus, the *loocv* cannot be used to assess the *quality* of the model.

The identification dataset is composed of evaluation of $f(x)$ which are performed during the optimization process. Let's consider an optimizer based on line-search technique (CONDOR is based on trust region technique and is thus more robust). The algorithm is the following:

1. Search for a descent direction s_k around x_k (x_k is the current position).
2. In the direction s_k, search the length α_k of the step.
3. Make a step in this direction: $x_{k+1} = x_k + \delta_k$ (with $\delta_k = \alpha_k s_k$)
4. Increment k. Stop if $\alpha_k \approx 0$ otherwise, go to step 1.

In the step 2 of this algorithm, many evaluation of $f(x)$ are performed aligned on the same line s_k. This means that the local,reduced dataset which will be used inside the lazy learning algorithm will very often be composed by aligned points. This situation is bad, as described at the beginning of this section. This explains why the Lazy Learning performs so poorly when used as local model inside a line-search optimizer. The situation is even worse than expected: the algorithm used inside the lazy learning to select the kernel size prevents to use points which are not aligned with d_1, leading to a poor, unstable model (complete explanation of this algorithm goes beyond the scope of this section). This phenomenon is well-known by statisticians and is referenced in the literature as "degeneration due to multi-collinearity" [Mye90]. The Lazy learning has actually no algorithm to prevent this degeneration and is thus of no-use in most cases.

To summarize: The *validity* of the local model must be checked to guarantee convergence to a local optimum. The leave-one out cross-validation error cannot be used to check the *validity* of the model. The lazy learning algorithm makes an extend use of the *loocv* and is thus to proscribe. Optimizer which are using as local model Neural Networks, Fuzzy sets, or other approximators can still be used if they follow closely the surrogate approach [KLT97, BDF$^+$99] which is an exact and good method. In particular approach based on kriging models seems to give good results [BDF$^+$98].

Chapter 12

Conclusions

When the search space dimension is comprised between 2 and 20 and when the noise on the objective function evaluation is limited, among the best optimizers available are CONDOR and UOBYQA. When several CPU's are used, the experimental results tend to show that CONDOR becomes the fastest optimizer in its category (fastest in terms of number of function evaluations).

From my knowledge, CONDOR is the *ONLY* optimizer which is completely:

- free of copyrights (GNU).
- stand-alone (no costly, copyrighted code needed).
- written in C++, using OO approach.
- cross-platform: it's able to compile on windows,unix,... Everywhere!

The fully stand-alone code is currently available at my homepage:

http://iridia.ulb.ac.be/∼fvandenb/

All what you need is a C++ compiler (like GCC) and you can go!

12.1 About the code

The code of the optimizer is a complete C/C++ **stand-alone** package written in pure structural programmation style. There is no call to fortran, external, unavailable, expensive libraries. You can compile it under unix or windows. The only library needed is the standard TCP/IP network transmission library based on sockets (only in the case of the parallel version of the code) which is available on almost every platform. You don't have to install any special libraries such as MPI or PVM to build the executables. The clients on different platforms/OS'es can be mixed together to deliver a huge computing power.

The code has been highly optimized (with extended use of memcpy function, special fast matrix manipulation, fast pointer arithmetics, and so on...). However, BLAS libraries have not been used to allow a full Object-Oriented approach. Anyways, the dimension of the problems is rather low so BLAS is un-useful. OO style programming allows a better comprehension of the code for the possible reader. The linear-algebra package is NOT LaPack. It's a home-made code inspired by LaPack, but optimized for C++ notations.

A small C++ SIF-file reader has also been implemented (to be able to use the problems coded in SIF from the CUTEr database, [GOT01]).

An AMPL interface has also been implemented. This is currently the most-versatile and most-used language for mathematical programming (It is used to describe objective functions).

If you are able to describe your problem in AMPL or in SIF, it means that the time needed for an evaluation of the objective function is *light* (it's NOT a high-computing-load objective function). You should thus use an other optimizer, like the one described in annexe in Section 13.9. The AMPL and SIF interface are mainly useful to test the code.

The fully stand-alone code is currently available at my homepage:

http://iridia.ulb.ac.be/~fvandenb/

12.2 Improvements

12.2.1 Unconstrained case

The algorithm is still limited to search space of dimension lower than 50 ($n < 50$). This limitation has two origin:

- The number of evaluation of the function needed to construct the first interpolation polynomial is very high: $n = (n + 1)(n + 2)/2$. It is possible to build a quadratic using less point (using a "Least frobenius norm updating of quadratic models that satisfy interpolation conditions"), as in the DFO algorithm, but the technique seems currently numerically very instable. Some recent works of Powell about this subject suggest that maybe a solution will soon be found (see [Pow02]).

- The algorithm to update the Lagrange polynomial (when we replace one interpolation point by another) is very slow. Its complexity is $\mathcal{O}(n^4)$. Since the calculation involved are very simple, it should be possible to parallelize this process. Another solution would be to use "Multivariate Newton polynomials" instead of "Multivariate Lagrange Polynomials".

Other improvements are possible:

- Improve the "warm-start" capabilities of the algorithm.
- Use a better strategy for the parallel case (see end of section 7.3)
- Currently the trust region is a simple ball: this is linked to the L2-norm $\|s\|_2$ used in the trust region step computation of Chapter 4. It would be interesting to have a trust region which reflects the underlying geometry of the model and not give undeserved weight to certain directions (like by using a H-norm: see Section 12.4 about this norm). The Dennis-Moré algorithm can be easily modified to use the H-norm. This improvement will have a small effect provided the variables have already been correctly normalized (see section 12.3 about normalization).

- The Dennis-Moré algorithm of of Chapter 4 requires many successive Cholesky factorization of the matrix $H + \lambda I$, with different value of λ. It's possible to partially transform H in tri-diagonal form to speed-up the successive Cholesky factorizations, as explained in [Pow97].

- Another possible improvement would be to use a more clever algorithm for the update of the two trust regions radius ρ and Δ. In particular, the update of ρ is actually not linked at all to the success of the polynomial interpolation. It can be improved.

- Some researches can also be made in the field of kriging models (see [BDF$^+$98]). These models need very few "model improvement steps" to obtain a good *validity*. The *validity* of the approximation can also be easily checked. On the contrary, in optimization algorithms based on other models (or surrogate: see [KLT97, BDF$^+$99]) like Neural Networks, Fuzzy Set, Lazy learning, ... the *validity* of the model is hard to assess (there is often no mathematical tool to allow this). The surrogate approach is a serious, correct and strong theory. Unfortunately, most optimizers based on NN, Fuzzy set,... do not implement completely the surrogate approach. In particular, most of the time, these kind of optimizers doesn't care for the *validity* of their model. They should thus be proscribed because they can easily fail to converge, even to a simple local optimum. Furthermore, they usually need many "model improvement step" to ensure *validity* and turn to be very slow.

12.2.2 Constrained case

The home-made QP is not very performant and could be upgraded.

Currently, we are using an SQP approach to handle non-linear constraints. It could be interesting to use a penalty-function approach.

When the model is *invalid*, we have to sample the objective function at a point of the space which will increase substantially the quality of the model. This point is calculated using Equation 3.38:

$$\max_d \{ |P_j(\boldsymbol{x}_{(k)} + d)| : \|d\| \leq \rho \} \tag{12.1}$$

The method to solve this equation is described in Chapter 5. This method does not take into account the constraints. As a result, CONDOR may ask for some evaluations of the objective function in the infeasible space. The infeasibility is never excessive (it's limited by ρ: see equation 12.1) but can sometime be a problem. A major improvement is to include some appropriate techniques to have a fully feasible-only algorithm.

Sometimes the evaluation of the objective function fails. This phenomenon is usual in the field of shape design optimization by CFD code. It simply means that the CFD code has not converged. This is referred in the literature as "virtual constraints" [CGT98]. In this case, a simple strategy is to reduce the Trust Region radius Δ and continue normally the optimization process. This strategy has been implemented and tested on some small examples and shows good results. However, It is still in development and tuning phase. It is the subject of current, ongoing research.

12.3 Some advice on how to use the code.

The steps s_k of the unconstrained algorithm are the solution to the following minimization problem (see Chapter 4):

$$\min_{s \in \Re^n} q_k(s) = g_k^t f_k + s^t H_k s \text{ subject to } \|s\|_2 < \Delta \tag{12.2}$$

where $q_k(s)$ is the local model of the objective function around x_k, $g_k \approx \nabla f(x_k)$ and $H_k \approx \nabla^2 f(x_k)$. The size of the steps are limited by the trust region radius Δ which represents the domain of validity of the local model $q_k(s)$. It is assumed that the validity of the local model at the point $x_k + s$ is only related to the distance $\|s\|_2$ and not to the direction of s. This assumption can be false: In some directions, the model can be completely false for small $\|s\|_2$ and in other directions the model can still be valid for large $\|s\|_2$.

Currently the trust region is a simple ball (because we are using the L2-norm $\|s\|_2$). If we were using the H-norm, the trust region would be an ellipsoid (see next Section 12.4 about H-norm). The H-norm allows us to link the *validity* of the model to the norm AND the direction of s.

Since we are using a L2-norm, it's very important to scale correctly the variables. An example of bad-scaling is given in Table 12.1.

Name	Normal Rosenbrock	Bad scale Rosenbrock	Bad Scale Rosenbrock corrected using **CorrectScaleOF**
Objective function	$100 * (x_2 - x_1^2)^2 + (1 - x_1)^2$	$100 * (\frac{x_2}{1000} - x_1^2)^2 + (1 - x_1)^2$	
starting point	$(-1.2 \;\; 1)^t$	$(-1200 \;\; 1)^t$	
ρ_{start}	.1		
ρ_{end}	1e-5	1e-3	1e-5
Number of function evaluations	100 (89)	376 (360)	100 (89)
Best value found	1.048530e-13	5.543738e-13	1.048569e-13

Table 12.1: Illustration of bad scaling

When all the variables are of the same order of magnitude, the optimizer is the fastest. For example, don't mix together variables which are degrees expressed in radians (values around 1) and variables which are height of a house expressed in millimeters (values around 10000). You have to *scale* or *normalize* the variables. There is inside the code a C++ class which can do automatically the scaling for you: "**CorrectScaleOF**". The scaling factors used in CorrectScaleOF are based on the values of the components of the starting point or are given by the user.

The same advice (scaling) can be given for the constraints: The evaluation of a constraint should give results of the same order of magnitude as the evaluation of the objective function.

12.4 The H-norm

The shape of an ideal trust region should reflect the geometry of the model and not give undeserved weight to certain directions.

Perhaps the ideal trust region would be in the H-norm, for which

$$\|s\|^2_{|H|} = \langle s, |H|s\rangle \tag{12.3}$$

and where the absolute value $|H|$ is defined by $|H| = U|\Lambda|U^T$, where Λ is a diagonal matrix constituted by the eigenvalues of H and where U is an orthonormal matrix of the associated eigenvectors and where the absolute value $|\Lambda|$ of the diagonal matrix Λ is simply the matrix formed by taking absolute values of its entries.

This norm reflects the proper scaling of the underlying problem - directions for which the model is changing fastest, and thus directions for which the model may differ most from the true function are restricted more than those for which the curvature is small.

The eigenvalue decomposition is extremely expensive to compute. A solution, is to consider the less expensive symmetric, indefinite factorization $H = PLBL^TP^T$ (P is a permutation matrix, L is unit lower triangular, B is block diagonal with blocks of size at most 2). We will use $|H| \approx PL|B|L^TP^T$ with $|B|$ computed by taking the absolute values of the 1 by 1 pivots and by forming an independent spectral decomposition of each of the 2 by 2 pivots and reversing the signs of any resulting negative eigenvalues.

For more information see [CGT00g].

Chapter 13

Annexes

The material of this chapter is based on the following references: [Fle87, DS96, CGT00a, PTVF99, GVL96].

13.1 Line-Search addenda.

13.1.1 Speed of convergence of Newton's method.

We have:

$$B_k \delta_k = -g_k \tag{13.1}$$

The Taylor series of g around x_k is:

$$\begin{aligned} g(x_k + h) &= g(x_k) + B_k h + o(\|h\|^2) \\ \Longleftrightarrow g(x_k - h) &= g(x_k) - B_k h + o(\|h\|^2) \end{aligned} \tag{13.2}$$

If we set in Equation 13.2 $h = h_k = x_k - x^*$, we obtain :

$$g(x_k + h_k) = g(x^*) = \underline{0 = g(x_k) - B_k h_k + o(\|h_k\|^2)}$$

If we multiply the left and right side of the previous equation by B_k^{-1}, we obtain, using 13.1:

$$\begin{array}{lrcl} & 0 &=& -\delta_k - h_k + o(\|h_k\|^2) \\ \Longleftrightarrow & \delta_k + h_k &=& o(\|h_k\|^2) \\ \Longleftrightarrow & (x_{k+1} - x_k) + (x_k - x^*) &=& o(\|h_k\|^2) \\ \Longleftrightarrow & x_{k+1} - x^* &=& o(\|h_k\|^2) \\ \Longleftrightarrow & h_{k+1} &=& o(\|h_k\|^2) \end{array}$$

By using the definition of o:

$$\|h_{k+1}\| < c\|h_k\|^2$$

with $c > 0$.

This is the definition of quadratic convergence. Newton's method has quadratic convergence speed.

Note

If the objective function is locally quadratic and if we have exactly $B_k = H$, then we will find the optimum in one iteration of the algorithm.

Unfortunately, we usually don't have H, but only an approximation B_k of it. For example, if this approximation is constructed using several "BFGS update", it becomes close to the real value of the Hessian H only after n updates. It means we will have to wait at least n iterations before going in "one step" to the optimum. In fact, the algorithm becomes "only" super-linearly convergent.

13.1.2 How to improve Newton's method : Zoutendijk Theorem.

We have seen that Newton's method is very fast (quadratic convergence) but has no global convergence property (B_k can be negative definite). We must search for an alternative method which has global convergence property. One way to prove global convergence of a method is to use Zoutendijk theorem.

Let us define a general method:

1. find a search direction s_k
2. search for the minimum in this direction using 1D-search techniques and find α_k with
$$x_{k+1} = x_k + \alpha_k s_k \tag{13.3}$$
3. Increment k. Stop if $g_k \approx 0$ otherwise, go to step 1.

The 1D-search must respect the Wolf conditions. If we define $f(\alpha) = f(x + \alpha s)$, we have:

$$f(\alpha) \leq f(0) + \rho\alpha f'(0) \qquad \rho \in (0, \frac{1}{2}) \tag{13.4}$$
$$f'(\alpha) > \sigma f'(0) \qquad \sigma \in (\rho, 1) \tag{13.5}$$

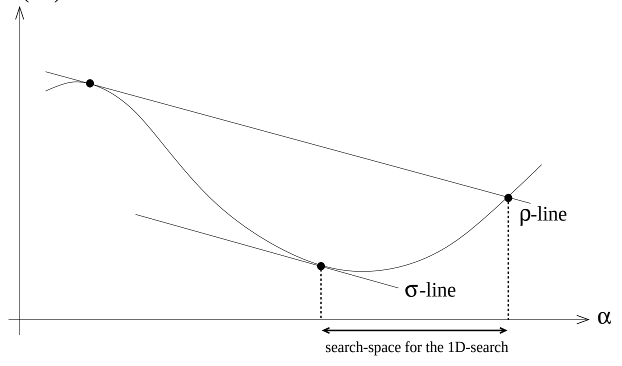

Figure 13.1: bounds on α: Wolf conditions

The objective of the Wolf conditions is to give a lower bound (equation 13.5) and upper bound (equation 13.4) on the value of α, such that the 1D-search algorithm is easier: see Figure 13.1. Equation 13.4 expresses that the objective function $\mathcal{F}$ must be reduced sufficiently. Equation 13.5 prevents too small steps. The parameter σ defines the precision of the 1D-line search:

- exact line search : $\sigma \approx 0.1$
- inexact line search : $\sigma \approx 0.9$

We must also define θ_k which is the angle between the steepest descent direction $(= -g_k)$ and the current search direction $(= s_k)$:

$$\cos(\theta_k) = \frac{-g_k^T s_k}{\|g_k\| \|s_k\|} \tag{13.6}$$

Under the following assumptions:

- $\mathcal{F} : \Re^n \to \Re$ bounded below
- $\mathcal{F}$ is continuously differentiable in a neighborhood $\mathcal{N}$ of the level set $\{x | f(x) < f(x_1)\}$
- $g = \nabla f$ is lipschitz continuous:

$$\exists L > 0 : \ \|g(x) - g(\tilde{x})\| < L\|x - \tilde{x}\| \quad \forall x, \tilde{x} \in \mathcal{N} \tag{13.7}$$

We have:

$$\textbf{Zoutendijk Theorem:} \ \sum_{k>1} \cos^2(\theta_k) \|g_k\|^2 < \infty \tag{13.8}$$

From Equation 13.5, we have:

$$\begin{aligned} f'(\alpha) &> \sigma f'(0) \\ \Longleftrightarrow g_{k+1}^T s_k &> \sigma g_k^T s_k \\ \text{we add } -g_k^T s_k \text{ on both side:} & \\ \Longleftrightarrow (g_{k+1}^T - g_k^T) s_k &> (\sigma - 1) g_k^T s_k \end{aligned} \tag{13.9}$$

From Equation 13.7, we have:

$$\begin{aligned} \|g_{k+1} - g_k\| &< L\|x_{k+1} - x_k\| \\ \text{using Equation 13.3 :} & \\ \Longleftrightarrow \|g_{k+1} - g_k\| &< \alpha_k L \|s_k\| \\ \text{we multiply by } \|s_k\| \text{ both sides:} & \\ \Longleftrightarrow (g_{k+1} - gk^T) s_k < \|g_{k+1} - g_k\| \|s_k\| &< \alpha_k L \|s_k\|^2 \end{aligned} \tag{13.10}$$

Combining Equation 13.9 and 13.10 we obtain:

$$\begin{aligned} (\sigma - 1) g_k^T s_k \quad &< (g_{k+1} - gk^T) s_k < \alpha_k L \|s_k\|^2 \\ \Longleftrightarrow \frac{(\sigma - 1) g_k^T s_k}{L \|s_k\|^2} \quad &< \quad \alpha_k \end{aligned} \tag{13.11}$$

We can replace in Equation 13.4:

$$f(\alpha) \le f(0) + \underbrace{\underbrace{\rho}_{>0} \quad \underbrace{f'(0)}_{=g_k^T s_k<0}}_{<0} \quad \alpha_k$$

the α_k by its lower bound from Equation 13.11. We obtain:

$$f_{k+1} \le fk + \frac{\rho(\sigma-1)(g_k^T s_k)^2}{L\|s_k\|^2} \frac{\|g_k\|^2}{\|g_k\|^2}$$

If we define $c = \frac{\rho(\sigma-1)}{L} < 0$ and if we use the definition of θ_k (see eq. 13.6), we have:

$$f_{k+1} \le f_k + \underbrace{\underbrace{c}_{<0} \underbrace{\cos^2(\theta_k)\|g_k\|^2}_{>0}}_{<0} \tag{13.12}$$

$$\frac{f_{k+1}-f_k}{c} \ge \cos^2(\theta_k)\|g_k\|^2 \tag{13.13}$$

Summing Equation 13.13 on k, we have

$$\frac{1}{c}\sum_k (f_{k+1}-f_k) \ge \sum_k \cos^2(\theta_k)\|g_k\|^2 \tag{13.14}$$

We know that $\mathcal{F}$ is bounded below, we also know from Equation 13.12, that $f_{k+1} \le f_k$. So, for a given large value of k (and for all the values above), we will have $f_{k+1} = f_k$. The sum on the left side of Equation 13.14 converges and is finite: $\sum_k (f_{k+1}-f_k) < \infty$. Thus,we have:

$$\sum_{k>1} \cos^2(\theta_k)\|g_k\|^2 < \infty$$

which concludes the proof.

Angle test.

To make an algorithm globally convergent, we can make what is called an "angle test". It consists of always choosing a search direction such that $cos(\theta_k) > \epsilon > 0$. This means that the search direction do not tends to be perpendicular to the gradient. Using the Zoutendijk theorem (see Equation 13.8), we obtain:

$$\lim_{k\to\infty} \|g_k\| = 0$$

which means that the algorithm is globally convergent.

The "angle test" on Newton's method prevents quadratical convergence. We must not use it.

The "steepest descent" trick.

If, regularly (lets say every $n+1$ iterations), we make a "steepest descent" step, we will have for this step, $cos(\theta_k) = 1 > 0$. It will be impossible to have $cos(\theta_k) \to 0$. So, using Zoutendijk, the only possibility left is that $\lim_{k\to\infty} \|g_k\| = 0$. The algorithm is now globally convergent.

13.2 Gram-Schmidt orthogonalization procedure.

We have a set of independent vectors $\{a_1, a_2, \dots, a_n\}$. We want to convert it into a set of orthonormal vectors $\{b_1, b_2, \dots, b_n\}$ by the Gram-Schmidt process.

The scalar product between vectors x and y will be noted $< x, y >$

Algorithm 1.

1. **Initialization** $b_1 = a_1$, $k = 2$
2. **Orthogonalisation**

$$\tilde{b_k} = a_k - \sum_{j=1}^{k} < a_k, b_j > b_j \tag{13.15}$$

   We will take a_k and transform it into $\tilde{b_k}$ by removing from a_k the component of a_k parallel to all the previously determined b_j.

3. **Normalisation**

$$b_k = \frac{\tilde{b_k}}{\|\tilde{b_k}\|} \tag{13.16}$$

4. **Loop** increment k. If $k < n$ go to step 2.

Algorithm 2.

1. **Initialization** k=1;
2. **Normalisation**

$$b_k = \frac{a_k}{\|a_k\|} \tag{13.17}$$

3. **Orthogonalisation** for $j = k + 1$ to n do:

$$a_j = a_j - < a_j, b_k > b_k \quad j = k + 1, \dots, n \tag{13.18}$$

   We will take the a_j which are left and remove from all of them the component parallel to the current vector b_k.

4. **Loop** increment k. If $k < n$ go to step 2.

13.3 Notions of constrained optimization

Let us define the problem:

Find the minimum of $f(x)$ subject to m constraints $c_j(x) \geq 0 (j = 1, ..., m)$.

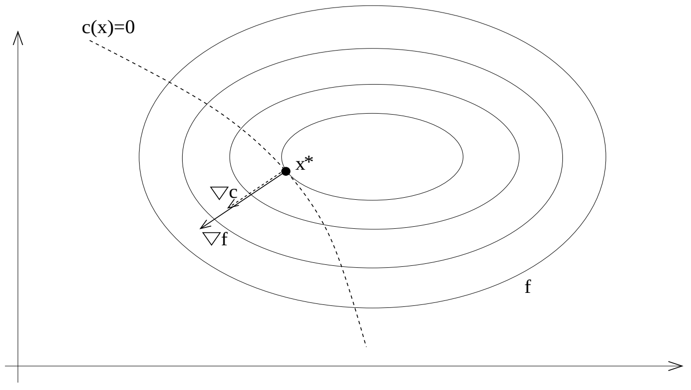

Figure 13.2: Existence of Lagrange Multiplier λ

To be at an optimum point x^* we must have the equi-value line (the contour) of $f(x)$ tangent to the constraint border $c(x) = 0$. In other words, when we have $r = 1$ constraints, we must have (see illustration in Figure 13.2) (the gradient of f and the gradient of c must aligned):

$$\nabla f = \lambda \nabla c$$

In the more general case when $r > 1$, we have:

$$\nabla f(x) = g(x) = \sum_{j \in E} \lambda_j \nabla c_j(x) \qquad\qquad c_j(x) = 0, \; j \in E \tag{13.19}$$

Where E is the set of active constraints, that is, the constraints which have $c_j(x) = 0$
We define Lagrangian function $\mathcal{L}$ as:

$$\mathcal{L}(x, \lambda) = f(x) - \sum_i \lambda_i c_i(x). \tag{13.20}$$

The Equation 13.19 is then equivalent to:

$$\blacktriangledown \mathcal{L}(x^*, \lambda^*) = 0 \text{ where } \blacktriangledown = \begin{pmatrix} \nabla_x \\ \nabla_\lambda \end{pmatrix} \tag{13.21}$$

In unconstrained optimization, we found an optimum x^* when $g(x^*) = 0$. In constrained optimization, we find an optimum point (x^*, λ^*), called a KKT point (Karush-Kuhn-Tucker point) when:

$$(x^*, \lambda^*) \text{ is a KKT point} \iff \begin{array}{rcl} \nabla_x \mathcal{L}(x^*, \lambda^*) & = & 0 \\ \lambda_j^* \, c_j(x^*) & = & 0, \;\; i = 1, ..., r \end{array} \tag{13.22}$$

The second equation of 13.22 is called the *complementarity condition.* It states that both λ^* and c_i^* cannot be non-zero, or equivalently that inactive constraints have a zero multiplier. An illustration is given on figure 13.3.

To get an other insight into the meaning of Lagrange Multipliers λ, consider what happens if the right-hand sides of the constraints are perturbated, so that

$$c_i(x) = \epsilon_i, \;\; i \in E \tag{13.23}$$

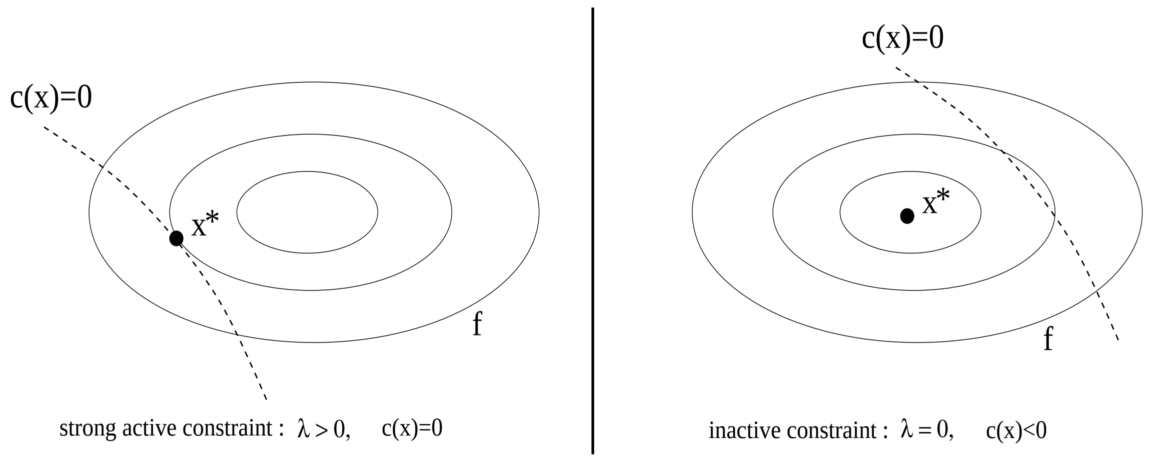
c(x)=0
x*
f
strong active constraint : λ > 0, c(x)=0
c(x)=0
x*
f
inactive constraint : λ = 0, c(x)<0

Figure 13.3: complementarity condition

Let $x(\epsilon)$, $\lambda(\epsilon)$ denote how the solution and multipliers change as ϵ changes. The Lagrangian for this problem is:

$$\mathcal{L}(x,\lambda,\epsilon) = f(x) - \sum_{i\in E} \lambda_i (c_i(x) - \epsilon_i) \tag{13.24}$$

From 13.23, $f(x(\epsilon)) = \mathcal{L}(x(\epsilon),\lambda(\epsilon),\epsilon)$, so using the chain rule, we have

$$\frac{df}{d\epsilon_i} = \frac{d\mathcal{L}}{d\epsilon_i} = \frac{dx^t}{d\epsilon_i}\nabla_x \mathcal{L} + \frac{d\lambda^t}{d\epsilon_i}\nabla_\lambda \mathcal{L} + \frac{d\mathcal{L}}{d\epsilon_i} \tag{13.25}$$

Using Equation 13.21, we see that the terms $\nabla_x\mathcal{L}$ and $\nabla_\lambda\mathcal{L}$ are null in the previous equation. It follows that:

$$\frac{df}{d\epsilon_i} = \frac{d\mathcal{L}}{d\epsilon_i} = \lambda_i \tag{13.26}$$

Thus the Lagrange multiplier of any constraint measure the rate of change in the objective function, consequent upon changes in that constraint function. This information can be valuable in that it indicates how sensitive the objective function is to changes in the different constraints.

13.4 The secant equation

Let us define a general polynomial of degree 2:

$$q(x) = q(0) + < g(0), x > + \frac{1}{2} < x, H(0)x > \tag{13.27}$$

where $H(0), g(0), q(0)$ are constant. From the rule for differentiating a product, it can be verified that:

$$\nabla(< u, b >) =< \nabla u, v > + < \nabla v, u >$$

if u and v depend on x. It therefore follows from 13.27 (using $u = x, v = H(0)x$) that

$$\nabla q(x) = g(x) = H(0)x + g(0) \tag{13.28}$$

$$\nabla^2 q(x) = H(0)$$

A consequence of 13.28 is that if $x_{(1)}$ and $x_{(2)}$ are two given points and if $g_{(1)} = \nabla q(x_{(1)})$ and $g_{(2)} = \nabla q(x_{(2)})$ (we simplify the notation $H := H(0)$), then

$$g_{(2)} - g_{(1)} = H(x_{(2)} - x_{(1)}) \tag{13.29}$$

This is called the "*Secant Equation*". That is the Hessian matrix maps the differences in position into differences in gradient.

13.5 1D Newton's search

Suppose we want to find the root of $f(x) = x^2 - 3$ (see Figure 13.4). If our current estimate of the answer is $x_k = 2$, we can get a better estimate x_{k+1} by drawing the line that is tangent to $f(x)$ at $(2, f(2)) = (2, 1)$, and find the point x_{k+1} where this line crosses the x axis. Since,

$$x_{k+1} = x_k - \Delta x,$$

and

$$f'(x_k) = \frac{\Delta y}{\Delta x} = \frac{f(x_k)}{\Delta x},$$

we have that

$$f'(x_k)\Delta x = F(x_k)$$

or

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)} \tag{13.30}$$

which gives $x_{k+1} = 2 - \frac{1}{4} = 1.75$. We apply the same process and iterate on k.

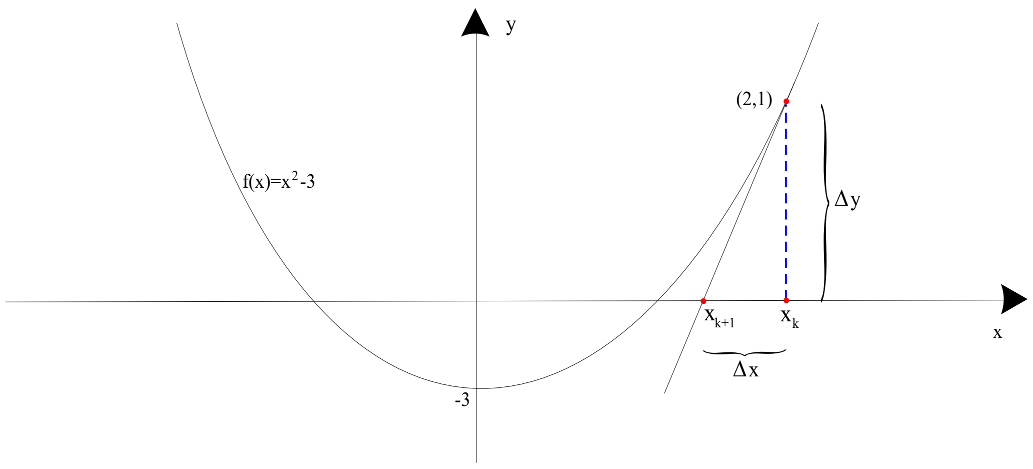

Figure 13.4: A plot of $\psi(\lambda)$ for H indefinite.

13.6 Newton's method for non-linear equations

We want to find the solution x of the set of non-linear equations:

$$r(x) = \begin{bmatrix} r_1(x) \\ \vdots \\ r_n(x) \end{bmatrix} = 0 \tag{13.31}$$

The algorithm is the following:

1. Choose x_0
2. Calculate a solution δ_k to the Newton equation:
$$J(x_k)\delta_k = -r(x_k) \tag{13.32}$$
3. $x_{k+1} = x_k + \delta_k$

We use a linear model to derive the Newton step (rather than a quadratical model as in unconstrained optimization) because the linear model normally as a solution and yields an algorithm with fast convergence properties (Newton's method has superlinear convergence when the Jacobian J is a continuous function and local quadratical convergence when J is Liptschitz continous). Newton's method for unconstrained optimization can be derived by applying Equation 13.32 to the set of nonlinear equations $\nabla f(x) = 0$).

13.7 Cholesky decomposition.

The Cholesky decomposition can be applied on any square matrix A which is symmetric and positive definite. The Cholesky decomposition is one of the fastest decomposition available. It constructs a lower triangular matrix L which has the following property:

$$L \cdot L^T = A \tag{13.33}$$

This factorization is sometimes referred to, as "taking the square root" of the matrix A.

The Cholesky decomposition is a particular case of the LU decomposition. The LU decomposition is the following:

$$L \cdot U = A \tag{13.34}$$

where L is a lower triangular matrix and U is a upper triangular matrix. For example, in the case of a 4×4 matrix A, we have:

$$\begin{pmatrix} \alpha_{11} & 0 & 0 & 0 \\ \alpha_{21} & \alpha_{22} & 0 & 0 \\ \alpha_{31} & \alpha_{32} & \alpha_{33} & 0 \\ \alpha_{41} & \alpha_{42} & \alpha_{43} & \alpha_{44} \end{pmatrix} \cdot \begin{pmatrix} \beta_{11} & \beta_{12} & \beta_{13} & \beta_{14} \\ 0 & \beta_{22} & \beta_{23} & \beta_{24} \\ 0 & 0 & \beta_{33} & \beta_{34} \\ 0 & 0 & 0 & \beta_{44} \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix} \tag{13.35}$$

We can use the LU decomposition to solve the linear set: $Ax = B \Leftrightarrow (LU)x = B$ by first solving for the vector y such that $Ly = B$ and then solving $Ux = y$. These two systems are trivial to solve because they are triangular.

13.7.1 Performing *LU* decomposition.

First let us rewrite the component a_{ij} of A from the equation 13.34 or 13.35. That component is always a sum beginning with

$$a_{i,j} = \alpha_{i1}\beta_{1j} + \cdots$$

The number of terms in the sum depends, however on wether i or j is the smallest number. We have, in fact three cases:

$$i < j: \quad a_{ij} = \alpha_{i1}\beta_{1j} + \alpha_{i2}\beta_{2j} + \cdots + \alpha_{ii}\beta_{ij} \tag{13.36}$$

$$i = j: \quad a_{ii} = \alpha_{i1}\beta_{1i} + \alpha_{i2}\beta_{2i} + \cdots + \alpha_{ii}\beta_{ii} \tag{13.37}$$

$$i > j: \quad a_{ij} = \alpha_{i1}\beta_{1j} + \alpha_{i2}\beta_{2j} + \cdots + \alpha_{ij}\beta_{jj} \tag{13.38}$$

Equations 13.36 - 13.38, totalize n^2 equations for the $n^2 + n$ unknown α's and β's (the diagonal being represented twice). Since the number of unknowns is greater than the number of equations, we have to specify n of the unknowns arbitrarily and then solve for the others. In fact, as we shall see, it is always possible to take:

$$\alpha_{ii} \equiv 1 \quad i = 1, \ldots, n \tag{13.39}$$

A surprising procedure, now, is *Crout's algorithm*, which, quite trivially, solves the set of $n^2 + n$ Equations 13.36 - 13.38 for all the α's and β's by just arranging the equation in a certain order! That order is as follows:

- Set $\alpha_{ii} = 1, \quad i = 1, \ldots, n$
- For each $j = 1, \ldots, n$ do these two procedures:

  First, for $i = 1, \ldots, j$ use 13.36, 13.37 and 13.39 to solve for β_{ij}, namely

$$\beta_{ij} = a_{ij} - \sum_{k=1}^{i-1} \alpha_{ik}\beta_{kj} \tag{13.40}$$

  Second, for $i = j + 1, \ldots, n$, use 13.38 to solve for α_{ij}, namely

$$\alpha_{ij} = \frac{1}{\beta_{jj}} \left(a_{ij} - \sum_{k=1}^{j-1} \alpha_{ik}\beta_{kj} \right) \tag{13.41}$$

  Be sure to do both procedures before going on to the next j.

If you work through a few iterations of the above procedure, you will see that the α's and β's that occur on the right-hand side of the Equation 13.40 and 13.41 are already determined by the time they are needed.

13.7.2 Performing Cholesky decomposition.

We can obtain the analogs of Equations 13.40 and 13.41 for the Cholesky decomposition:

$$L_{ii} = \sqrt{a_{ii} - \sum_{k=1}^{i-1} L_{ik}^2} \tag{13.42}$$

and

$$L_{ji} = \frac{1}{L_{ii}}\left(a_{ij} - \sum_{k=1}^{i-1} L_{ik}L_{jk}\right) \quad j = i+1, \ldots, n \tag{13.43}$$

If you apply Equation 13.42 and 13.43 in the order $i = 1, \ldots, n$, you will see the the L's that occur on the right-hand side are exactly determined by the time they are needed. Also, only components a_{ij} with $j > i$ are referenced.

If the matrix A is not positive definite, the algorithm will stop, trying to take the square root of a negative number in equation 13.42.

What about pivoting? Pivoting (i.e., the selection of a salubrious pivot element for the division in Equation 13.43) is not really required for the stability of the algorithm. In fact, the only cause of failure is if the matrix A (or, with roundoff error, another very nearby matrix) is not positive definite.

13.8 QR factorization

There is another matrix factorization that is sometimes very useful, the so-called QR decomposition,

$$A = Q\begin{bmatrix} R \\ 0 \end{bmatrix} \quad A \in \Re^{m\times n}(m \geq n), Q \in \Re^{m\times m}, R \in \Re^{n\times n} \tag{13.44}$$

Here R is upper triangular, while Q is orthogonal, that is,

$$Q^tQ = I \tag{13.45}$$

where Q^t is the transpose matrix of Q. The standard algorithm for the QR decomposition involves successive Householder transformations. The Householder algorithm reduces a matrix A to the triangular form R by $n-1$ orthogonal transformations. An appropriate Householder matrix applied to a given matrix can zero all elements in a column of the matrix situated below a chosen element. Thus we arrange for the first Householder matrix P_1 to zero all elements in the first column of A below the first element. Similarly P_2 zeroes all elements in the second column below the second element, and so on up to P_{n-1}. The Householder matrix P has the form:

$$P = 1 - 2ww^t \tag{13.46}$$

where w is a real vector with $\|w\|^2 = 1$. The matrix P is orthogonal, because $P^2 = (1 - 2ww^t)(1-2ww^t) = 1 - 4ww^t + 4w(w^tw)w^t = 1$. Therefore $P = P^{-1}$. But $P^t = P$, and so $P^t = P^{-1}$, proving orthogonality. Let's rewrite P as

$$P = 1 - \frac{uu^t}{H} \quad \text{with } H = \frac{1}{2}\|u\|_2 \tag{13.47}$$

and u can now be any vector. Suppose x is the vector composed of the first column of A. Choose $u = x \mp \|x\|e_1$ where e_1 is the unit vector $[1, 0, \ldots, 0]^t$, and the choice of signs will be made later. Then

$$\begin{aligned} Px &= x - \frac{u}{H}(x \mp \|x\|e_1)^t x \\ &= x - \frac{2u(\|x\|^2 \mp \|x\|x_1)}{2\|x\|^2 \mp 2\|x\|x_1} \\ &= x - u \\ &= \mp\|x\|e_1 \end{aligned}$$

This shows that the Householder matrix P acts on a given vector x to zero all its elements except the first one. To reduce a symmetric matrix A to triangular form, we choose the vector x for the first Householder matrix to be the first column. Then the lower $n-1$ elements will be zeroed:

$$P_1 A = A' = \left[\begin{array}{c|c} a'_{11} & \\ 0 & \\ & \textit{irrelevant} \\ \vdots & \\ 0 & \end{array}\right] \qquad (13.48)$$

If the vector x for the second Householder matrix is the lower $n-1$ elements of the second column, then the lower $n-2$ elements will be zeroed:

$$\left[\begin{array}{c|c} 1 & 0\ldots 0 \\ \hline 0 & \\ \vdots & P_2 \\ 0 & \end{array}\right] A' = A'' = \left[\begin{array}{cc|c} a'_{11} & a'_{12} & a'_{13}\ldots a'_{1m} \\ \hline 0 & a''_{22} & \\ 0 & 0 & \\ & & \textit{irrelevant} \\ \vdots & \vdots & \\ 0 & 0 & \end{array}\right] \qquad (13.49)$$

Where $P_2 \in \Re^{(n-1)\times(n-1)}$ and the quantity a''_{22} is simply plus or minus the magnitude of the vector $[\ a'_{22}\ \cdots\ a'_{n2}\ ]^t$. Clearly a sequence of $n-1$ such transformation will reduce the matrix A to triangular form R. Instead of actually carrying out the matrix multiplications in PA, we compute a vector $p := \frac{Au}{H}$. Then $PA = (1 - \frac{uu^t}{H})A = A - up^t$. This is a computationally useful formula. We have the following: $A = P_1 \ldots P_{n-1}R$. We will thus form $Q = P_1 \ldots P_{n-1}$ by recursion after all the P's have been determined:

$$\begin{aligned} Q_{n-1} &= P_{n-1} \\ Q_j &= P_j Q_{j+1} \quad j = n-2, \ldots, 1 \\ Q &= Q_1 \end{aligned}$$

No extra storage is needed for intermediate results but the original matrix is destroyed.

13.9 A simple direct optimizer: the Rosenbrock optimizer

The Rosenbrock method is a 0^{th} order search algorithm (i.e, it does not require any derivatives of the target function. Only simple evaluations of the objective function are used). Yet, it approximates a gradient search thus combining advantages of 0^{th} order and 1^{st} order strategies. It was published by Rosenbrock [Ros60] in the 70^{th}.

This method is particularly well suited when the objective function does not require a great deal of computing power. In such a case, it's useless to use very complicated optimization algorithms. We will spend much time in the optimization calculations instead of making a little bit more evaluations of the objective function which will finally lead to a shorter calculation time.

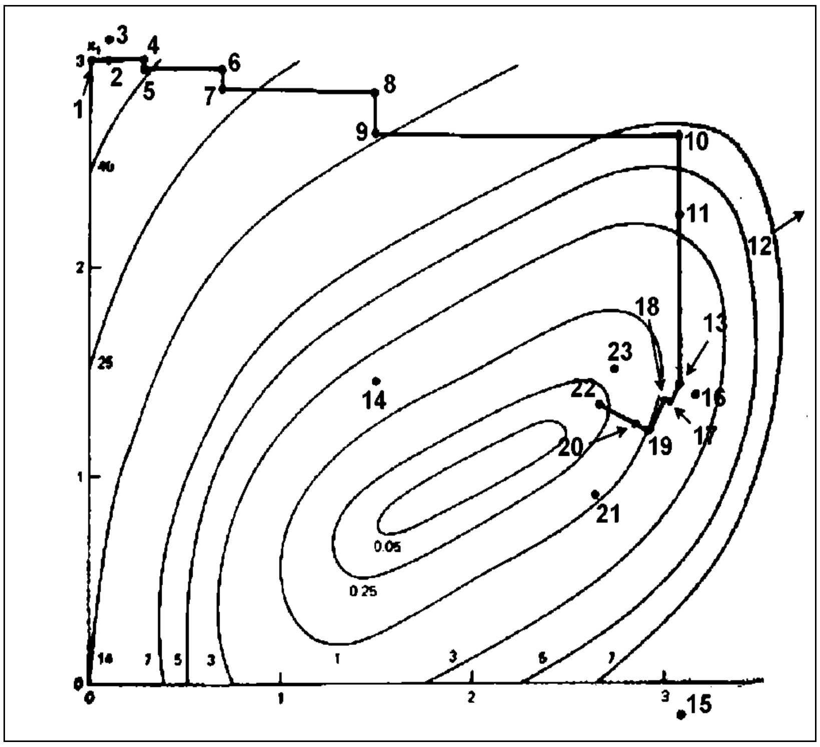

Figure 13.5: Illustration of the Rosenbrock procedure using discrete steps (the number denotes the order in which the points are generated))

In the first iteration, it is a simple 0^{th} order search in the directions of the base vectors of an n-dimensional coordinate system. In the case of a success, which is an attempt yielding a new minimum value of the target function, the step width is increased, while in the case of a failure it is decreased and the opposite direction will be tried (see points 1 to 15 in the Figure 13.5). Once a success has been found and exploited in each base direction, the coordinate system is rotated in order to make the first base vector point into the direction of the gradient (in Figure 13.5, the points 13, 16 & 17 are defining the new base). Now all step widths are initialized and the process is repeated using the rotated coordinate system (points 16 to 23).

The creation of a new rotated coordinate system is usually done using a Gram-Shmidt orthogonalization procedure. This algorithm is numerically instable. This instability can lead to a premature ending of the optimization algorithm. J.R.Palmer [Pal69] has proposed a beautiful solution to this problem.

Initializing the step widths to rather big values enables the strategy to leave local optima and to go on with search for more global minima. It has turned out that this simple approach is more stable than many sophisticated algorithms and it requires much less calculations of the target function than higher order strategies [Sch77]. This method has also been proved to always converge (global convergence to a local optima assured) [BSM93].

Finally a user who is not an optimization expert has a real chance to understand it and to set and tune its parameters properly.

The code of my implementation of the Rosenbrock algorithm is available in the code section. The code of the optimizer is standard C and doesn't use any special libraries. It can be compiled under windows or unix. The code has been highly optimized to be as fast as possible (with extend use of memcpy function, special fast matrix manipulation and so on...). The improvement of J.R. Palmer is used. This improvement allows still faster speed. The whole algorithm is only 107 lines long (with correct indentations). It's written in pure structural programmation (i.e., there is no "goto instruction"). It is thus very easy to understand/customize. A small example of use (testR1.cpp) is available. In this example, the standard Rosenbrock banana function is minimized.

Chapter 14

Code

14.1 Rosenbrock's optimizer

14.1.1 rosenbrock.cpp

This is the core of the Rosenbrock optimizer. The variables b_l and b_u are currently ignored.

```
#include<stdio.h>
#include<math.h>
#include<stdlib.h>
#include<memory.h>

char rosenbrock_version[] = "rosenbrock 0.99";

#define MIN(a,b) ((a)<(b)?(a):(b))
#define MAX(a,b) ((a)>(b)?(a):(b))

void rosenbrock(int n, double *x, double *bl, double *bu,
                double bigbnd, int maxiter, double eps, $
$int verbose,
                void obj(int,double *,double *,void *), $
$void *extraparams)
{
    double **xi=(double**)calloc(n,sizeof(double*)),
           *temp1=(double*)calloc(n*n,sizeof(double)),
           **A=(double**)calloc(n,sizeof(double*)),
           *temp2=(double*)calloc(n*n,sizeof(double)),
           *d=(double*)calloc(n,sizeof(double)),
           *lambda=(double*)calloc(n,sizeof(double)),
           *xk=(double*)calloc(n,sizeof(double)),
           *xcurrent=(double*)calloc(n,sizeof(double)),
           *t=(double*)calloc(n,sizeof(double)),
           alpha=2,
           beta=0.5,
           yfirst,yfirstfirst,ybest,ycurrent,mini,div;
    int i,k,j,restart,numfeval=0;

    memset(temp1,0,n*n*sizeof(double));
    for(i=0; i<n; i++)
    {
        temp1[i]=1; xi[i]=temp1; temp1+=n;
                     A[i]=temp2; temp2+=n;
    };
    // memcpy(destination,source,nbre_of_byte)
    memcpy(xk,x,n*sizeof(double));
    for (i=0; i<n; i++) d[i]=.1;
    memset(lambda,0,n*sizeof(double));
    (*obj)(n,x,&yfirstfirst,extraparams); numfeval++;

    do
    {
        ybest=yfirstfirst;
        do
        {
            yfirst=ybest;
            for (i=0; i<n; i++)
            {
                for (j=0; j<n; j++) xcurrent[j]=xk[j]+d[i$
$]*xi[i][j];
                (*obj)(n,xcurrent,&ycurrent,extraparams); $
$numfeval++;
                if (ycurrent<ybest)
                {
                    lambda[i]+=d[i];        // success
                    d[i]*=alpha;
                    ybest=ycurrent;
                    memcpy(xk,xcurrent,n*sizeof(double));
                } else
                {
                    d[i]*=-beta;             // failure
                }
            }
        } while (ybest<yfirst);

        mini=bigbnd;
        for (i=0; i<n; i++) mini=MIN(mini,fabs(d[i]));
        restart=mini>eps;

        if (ybest<yfirstfirst)
        {
            mini=bigbnd;
            for (i=0; i<n; i++) mini=MIN(mini,fabs(xk[i]-x$
$[i]));
            restart=restart||(mini>eps);

            if (restart)
            {
                // nous avons:
                // xk[j]-x[j]=(somme sur i de) lambda[i]*$
$xi[i][j];

                for (i=0; i<n; i++) A[n-1][i]=lambda[n-1]*$
$xi[n-1][i];
                for (k=n-2; k>=0; k--)
                    for (i=0; i<n; i++) A[k][i]=A[k+1][i]+$
$lambda[k]*xi[k][i];

                t[n-1]=lambda[n-1]*lambda[n-1];
                for (i=n-2; i>=0; i--) t[i]=t[i+1]+lambda[$
$i]*lambda[i];
                for (i=n-1; i>0; i--)
                {
                    div=sqrt(t[i-1]*t[i]);
                    if (div!=0)
                        for (j=0; j<n; j++)
                            xi[i][j]=(lambda[i-1]*A[i][j]-$
$xi[i-1][j]*t[i])/div;
                }
                div=sqrt(t[0]);
                for (i=0; i<n; i++) xi[0][i]=A[0][i]/div;

                memcpy(x,xk,n*sizeof(double));
                memset(lambda,0,n*sizeof(double));
                for (i=0; i<n; i++) d[i]=.1;
                yfirstfirst=ybest;
            }
        }
    } while ((restart)&&(numfeval<maxiter));
    // the maximum number of evaluation is approximative
    // because in 1 iteration there is n function $
$evaluations.
```

```
    if (verbose)
    {
        printf("ROSENBROCK method for local optimization ($
$minimization)\n"
                "number of evaluation of the objective $
$function= %i\n\n",numfeval);
    }

    free(xi[0]);
    free(A[0]);
    free(d);
    free(lambda);
    free(xk);
    free(xcurrent);
    free(t);
}
```

14.1.2 rosenbrock.h

```
#ifndef __INCLUDE__ROSEN_H___ #define __INCLUDE__ROSEN_H___

void rosenbrock(int n, double *x, double *bl, double *bu,
                double bigbnd, int maxiter, double eps, int verbose,
                void obj(int,double *,double *,void *), void *extraparams);

#endif
```

14.1.3 testR1.cpp

This is an example code where we use the Rosenbrock optimizer to optimize the Rosenbrock banana function.

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <memory.h>
#include "rosenbrock.h"

#define SQR(a) ((a)*(a))

void obj32(int nparam,double *x,double *fj,void *$
$extraparams) {
//    *fj=pow((x[0]-2.0),4.0)+pow((x[0]-2.0*x[1]),2.e0);
    *fj=100*SQR(x[1]-SQR(x[0]))+SQR(1-x[0]);
   return;
}

void message(int n,double *x) {
    double y;
    int i;
    printf("optimum found at:\n");
    for (i=0; i<n; i++)
        printf(" x[%i]=%f\n",i+1,x[i]);
    obj32(n,x,&y,NULL);
    printf("objective function value= %f\n", y);
};

int main() {
    int nparam,maxIter,verbosity;
    double *x,*bl,*bu,bigbnd,eps;

    nparam=2;
    x=(double*)calloc(nparam,sizeof(double));
    bl=(double *)calloc(nparam,sizeof(double));
    bu=(double *)calloc(nparam,sizeof(double));
    bigbnd=1.e10;
    maxIter=5000;
    eps=1.e-5;
    verbosity=1;

    bl[0]=bl[1]=-10;
    bu[0]=bu[1]=10;

    x[0]=5;
    x[1]=5;

    rosenbrock(nparam,x,bl,bu,bigbnd,maxIter,eps,verbosity$
$,obj32,NULL);

    message(nparam,x);

    free(x);
    free(bl);
    free(bu);
    return 0;
}
```

14.2 CONDOR

14.2.1 Matrix.cpp

```
#include <memory.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include "Matrix.h"
#include "tools.h"

Matrix Matrix::emptyMatrix;

void Matrix::init(int _nLine, int _nColumn, int _extLine, $
$int _extColumn,MatrixData* md)
{
    if (md==NULL)
    {
        d=(MatrixData*)malloc(sizeof(MatrixData));
        d->ref_count=1;
    } else d=md;
    d->nLine=_nLine; d->nColumn=_nColumn;
    d->extLine=_extLine; d->extColumn=_extColumn;

    if ((_extLine>0)&&(_extColumn>0))
        {
            double **t,*t2;
        t=d->p=(double**)malloc(_extLine*sizeof(double*));
        t2=(double*)malloc(_extColumn*_extLine*sizeof($
$double));
            while(_extLine--)
        {
            *(t++)=t2; t2+=_extColumn;
        }
    } else d->p=NULL;
}

Matrix::Matrix(int _nLine,int _nColumn)
{
    init(_nLine,_nColumn,_nLine, _nColumn);
};

void Matrix::diagonal(double dd)
{
    zero();
    double *p=*d->p;
    int n=nLine(), i=d->extColumn+1;
    while (n--) { *p=dd; p+=i; }

}

Matrix::Matrix(Vector a, Vector b)  // a * b^T
{
    int nl=a.sz(), nc=b.sz(), i,j;
    double *pa=a, *pb=b;
    init(nl,nc,nl,nc);
    double **pp=d->p;
    for (i=0; i<nl; i++)
        for (j=0; j<nc; j++)
            pp[i][j]=pa[i]*pb[j];
}

Matrix::Matrix(int _nLine,int _nColumn,int _extLine,int $
$_extColumn)
{
    init(_nLine,_nColumn,_extLine,_extColumn);
};

Matrix::Matrix(char *filename)
{
    unsigned _nLine,_nColumn;
    FILE *f=fopen(filename,"rb");
    if (f==NULL)
    {
        printf("file not found.\n"); exit(255);
    }
    fread(&_nLine, sizeof(unsigned), 1, f);
    fread(&_nColumn, sizeof(unsigned), 1, f);
    init(_nLine,_nColumn,_nLine,_nColumn);
    fread(*d->p,sizeof(double)*d->nColumn*d->nLine,1,f);
    fclose(f);
}

void Matrix::extendLine()
{
    d->nLine++;
    if (d->nLine>d->extLine) setExtSize(d->nLine+9,d->$
$extColumn);
}

void Matrix::setNLine(int _nLine)
{
    d->nLine=_nLine;
    if (_nLine>d->extLine) setExtSize(_nLine,d->extColumn)$
$;
}

void Matrix::extendColumn()
{
    d->nColumn++;
    if (d->nColumn>d->extColumn) setExtSize(d->extLine,d->$
$nColumn+9);
}

void Matrix::setNColumn(int _nColumn)
{
    d->nColumn=_nColumn;
    if (_nColumn>d->extColumn) setExtSize(d->extLine,$
$_nColumn);
}

void Matrix::setSize(int _nLine,int _nColumn)
{
    d->nLine=_nLine;
    d->nColumn=_nColumn;
    if ((_nLine>d->extLine)||(_nColumn>d->extColumn)) $
$setExtSize(_nLine,_nColumn);
}

void Matrix::setExtSize(int _extLine, int _extColumn)
{
    int ec=d->extColumn;
    if ((ec==0)||(d->extLine==0))
    {
        init(d->nLine,d->nColumn,_extLine,_extColumn,d);
        return;
    }
    if (_extColumn>ec)
    {
        int nc=d->nColumn,i;
            double *tmp,*tmp2,**tmp3=d->p,*oldBuffer=*tmp3$
$;

        if (d->extLine<_extLine)
            tmp3=d->p=(double**)realloc(tmp3,_extLine*$
$sizeof(double*));
        else _extLine=d->extLine;
        tmp2=tmp=(double *)malloc(_extLine*_extColumn*$
$sizeof(double));
                if (tmp==NULL)
        {
            printf("memory allocation error");
            getchar(); exit(255);
        }

        i=_extLine;
                while (i--)
                {
                    *(tmp3++)=tmp2;
                    tmp2+=_extColumn;
                };

        if ((nc)&&(d->nLine))
        {
                tmp2=oldBuffer;
            i=d->nLine;
            nc*=sizeof(double);
            while(i--)
                    {
                        memmove(tmp,tmp2,nc);
                        tmp+=_extColumn;
                        tmp2+=ec;
                    };
            free(oldBuffer);
        };
        d->extLine=_extLine;
        d->extColumn=_extColumn;
        return;
    }
    if (_extLine>d->extLine)
    {
        int i;
        double *tmp,**tmp3;
        tmp=(double *)realloc(*d->p,_extLine*ec*sizeof($
$double));
                if (tmp==NULL)
        {
            printf("memory allocation error");
            getchar(); exit(255);
        }
        free(d->p);
        tmp3=d->p=(double **)malloc(_extLine*sizeof(double$
$*));
        i=_extLine;
                while (i--)
                {
                    *(tmp3++)=tmp;
                    tmp+=ec;
                };
        d->extLine=_extLine;
    }
}

void Matrix::save(char *filename,char ascii)
{
    double **p=(d->p);
    int i,j;
    FILE *f;
    if (ascii)
    {
        f=fopen(filename,"w");
        for (i=0; i<d->nLine; i++)
        {
            for (j=0; j<d->nColumn-1; j++)
                fprintf(f,"%1.10f ",p[i][j]);
            fprintf(f,"%1.10f\n",p[i][d->nColumn-1]);
        }
    } else
    {
        f=fopen(filename,"wb");
        fwrite(&d->nLine, sizeof(unsigned), 1, f);
        fwrite(&d->nColumn, sizeof(unsigned), 1, f);
        for (i=0; i<d->nLine; i++)
            fwrite(p[i],sizeof(double)*d->nColumn,1,f);
    };
    fclose(f);
}

void Matrix::updateSave(char *saveFileName)
{
    FILE *f=fopen(saveFileName,"r+b");
    if (f==NULL)
    {
        save(saveFileName,0);
        return;
    }
    fseek(f,0,SEEK_END);
    long l=ftell(f);
    int nc=d->nColumn, nlfile=(l-sizeof(unsigned)*2)/(nc*$
$sizeof(double)), nl=d->nLine, i;
```

```
    double **p=d->p;
    for (i=nlfile; i<nl; i++)
        fwrite(p[i],sizeof(double)*nc,1,f);
    fseek(f,0,SEEK_SET);
    fwrite(&d->nLine, sizeof(unsigned), 1, f);
    fflush(f);
    fclose(f);
}

void Matrix::exactshape()
{
    int i, nc=d->nColumn, ec=d->extColumn, nl=d->nLine, el$
$=d->extLine;
    double *tmp,*tmp2,**tmp3;

    if ((nc==ec)&&(nl==el)) return;

    if (nc!=ec)
    {
        i=nl;
        tmp=tmp2=*d->p;
        while(i--)
                {
                    memmove(tmp,tmp2,nc*sizeof(double));
                    tmp+=nc;
                    tmp2+=ec;
                };
    }

    tmp=(double *)realloc(*d->p,nl*nc*sizeof(double));
        if (tmp==NULL)
    {
        printf("memory allocation error");
        getchar(); exit(255);
    }
    if (tmp!=*d->p)
    {
        tmp3=d->p=(double **)realloc(d->p,nl*sizeof(double$
$*));
        i=nl;
        while (i--)
            {
                    *(tmp3++)=tmp;
                tmp+=nc;
            };
    } else d->p=(double **)realloc(d->p,nl*sizeof(double*)$
$);

    d->extLine=nl; d->extColumn=nc;
};


void Matrix::print()
{
    double **p=d->p;
    int i,j;

    printf("[");
    for (i=0; i<d->nLine; i++)
    {
        for (j=0; j<d->nColumn; j++)
            if (p[i][j]>=0.0) printf(" %2.3f ",p[i][j]);
            else printf("%2.3f ",p[i][j]);
            if (i==d->nLine-1) printf("]\n"); else printf($
$";\n");
    }
    fflush(0);
}

Matrix::~Matrix()
{
    destroyCurrentBuffer();
};

void Matrix::destroyCurrentBuffer()
{
    if (!d) return;
    (d->ref_count) --;
        if (d->ref_count==0)
    {
        if (d->p) { free(*d->p); free(d->p); }
        free(d);
    }
}

Matrix& Matrix::operator=( const Matrix& A )
{
    // shallow copy
    if (this != &A)
        {
        destroyCurrentBuffer();
        d=A.d;
                (d->ref_count) ++ ;
```

```
        }
        return *this;
}

Matrix::Matrix(const Matrix &A)
{
    // shallow copy
    d=A.d;
        (d->ref_count)++ ;
}

Matrix Matrix::clone()
{
    // a deep copy
    Matrix m(nLine(),nColumn());
    m.copyFrom(*this);
    return m;
}

void Matrix::copyFrom(Matrix m)
{
    int nl=m.nLine(),nc=m.nColumn(), ec=m.d->extColumn;
    if ((nl!=nLine())||(nc!=nColumn()))
    {
        printf("Matrix: copyFrom: size do not agree");
        getchar(); exit(254);
    }
    if (ec==nc)
    {
        memcpy(*d->p,*m.d->p,nc*nl*sizeof(double));
        return;
    }
    double *pD=*d->p,*pS=*m.d->p;
    while(nl--)
    {
        memcpy(pD,pS,nc*sizeof(double));
        pD+=nc;
        pS+=ec;
    };
}

void Matrix::transposeInPlace()
{
    int nl=nLine(),nc=nColumn(),i,j;
    if (nl==nc)
    {
        double **p=(*this),t;
        for (i=0; i<nl; i++)
            for (j=0; j<i; j++)
            {
                t=p[i][j];
                p[i][j]=p[j][i];
                p[j][i]=t;
            }
        return;
    }
    Matrix temp=clone();
    setSize(nc,nl);
    double **sp=temp, **dp=(*this);
    i=nl;
    while (i--)
    {
        j=nc;
        while (j--) dp[j][i]=sp[i][j];
    }
}

void Matrix::transpose(Matrix temp)
{
    int nl=nLine(),nc=nColumn(),i,j;
    temp.setSize(nc,nl);
    double **sp=temp, **dp=(*this);
    i=nl;
    while (i--)
    {
        j=nc;
        while (j--) sp[j][i]=dp[i][j];
    }
}

Matrix Matrix::transpose()
{
    Matrix temp(nColumn(),nLine());
    transpose(temp);
    return temp;
}

//Matrix Matrix::deepCopy()
//{
//    Matrix cop(this); // contructor of class matrix
//    return cop;    // copy of class Matrix in return $
$Variable
//                     // destruction of instance cop.
```

```
//};

void Matrix::zero()
{
    memset(*d->p,0,nLine()*d->extColumn*sizeof(double));
};

void Matrix::multiply(Matrix R, Matrix Bplus)
{
    if (Bplus.nLine()!=nColumn())
    {
        printf("(matrix * matrix) error");
        getchar(); exit(249);
    }
    int i,j,k, nl=nLine(), nc=Bplus.nColumn(), n=nColumn()$
$;
    R.setSize(nl,nc);
    double sum,**p1=(*this),**p2=Bplus,**pr=R;

    for (i=0; i<nl; i++)
        for (j=0; j<nc; j++)
        {
            sum=0;
            for (k=0; k<n; k++) sum+=p1[i][k]*p2[k][j];
            pr[i][j]=sum;
        }
}

void Matrix::transposeAndMultiply(Matrix R, Matrix Bplus)
{
    if (Bplus.nLine()!=nLine())
    {
        printf("(matrix^t * matrix) error");
        getchar(); exit(249);
    }
    int i,j,k, nl=nColumn(), nc=Bplus.nColumn(), n=nLine()$
$;
    R.setSize(nl,nc);
    double sum,**p1=(*this),**p2=Bplus,**pr=R;

    for (i=0; i<nl; i++)
        for (j=0; j<nc; j++)
        {
            sum=0;
            for (k=0; k<n; k++) sum+=p1[k][i]*p2[k][j];
            pr[i][j]=sum;
        }
}

void Matrix::multiplyByTranspose(Matrix R, Matrix Bplus)
{
    if (Bplus.nColumn()!=nColumn())
    {
        printf("(matrix * matrix^t) error");
        getchar(); exit(249);
    }
    int i,j,k, nl=nLine(), nc=Bplus.nLine(), n=nColumn();
    R.setSize(nl,nc);
    double sum,**p1=(*this),**p2=Bplus,**pr=R;

    for (i=0; i<nl; i++)
        for (j=0; j<nc; j++)
        {
            sum=0;
            for (k=0; k<n; k++) sum+=p1[i][k]*p2[j][k];
            pr[i][j]=sum;
        }
}

Matrix Matrix ::multiply(Matrix Bplus)
{
    Matrix R(nLine(),Bplus.nColumn());
    multiply(R,Bplus);
    return R;
}

void Matrix::multiplyInPlace(double dd)
{
    int i,j, nl=nLine(), nc=nColumn();
    double **p1=(*this);

    for (i=0; i<nl; i++)
        for (j=0; j<nc; j++)
            p1[i][j]*=dd;
}

void Matrix::multiply(Vector rv, Vector v)
{
    int i,j, nl=nLine(), nc=nColumn();
    rv.setSize(nl);
    if (nc!=(int)v.sz())
    {
        printf("matrix multiply error");
        getchar(); exit(250);
    };
    double **p=(*this), *x=v, *r=rv, sum;

    for (i=0; i<nl; i++)
    {
        sum=0; j=nc;
        while (j--) sum+=p[i][j]*x[j];
        r[i]=sum;
    }
}

void Matrix::transposeAndMultiply(Vector rv, Vector v)
{
    int i,j, nc=nLine(), nl=nColumn();
    rv.setSize(nl);
    if (nc!=(int)v.sz())
    {
        printf("matrix multiply error");
        getchar(); exit(250);
    };
    double **p=(*this), *x=v, *r=rv, sum;

    for (i=0; i<nl; i++)
    {
        sum=0; j=nc;
        while (j--) sum+=p[j][i]*x[j];
        r[i]=sum;
    }
}

Vector Matrix::multiply(Vector v)
{
    Vector r(nLine());
    multiply(r,v);
    return r;
}

double Matrix::scalarProduct(int nl, Vector v)
{
    double *x1=v, *x2=d->p[nl], sum=0;
    int n=v.sz();
    while (n--) { sum+=*(x1++) * *(x2++); };
    return sum;
}

void Matrix::addInPlace(Matrix B)
{
    if ((B.nLine()!=nLine())||
        (B.nColumn()!=nColumn()))
    {
        printf("matrix addition error");
        getchar(); exit(250);
    };

    int i,j, nl=nLine(), nc=nColumn();
    double **p1=(*this),**p2=B;

    for (i=0; i<nl; i++)
        for (j=0; j<nc; j++)
            p1[i][j]+=p2[i][j];
}

void Matrix::addMultiplyInPlace(double d, Matrix B)
{
    if ((B.nLine()!=nLine())||
        (B.nColumn()!=nColumn()))
    {
        printf("matrix addition error");
        getchar(); exit(250);
    };

    int i,j, nl=nLine(), nc=nColumn();
    double **p1=(*this),**p2=B;

    for (i=0; i<nl; i++)
        for (j=0; j<nc; j++)
            p1[i][j]+=d*p2[i][j];
}

//inline double sqr(double a){return a*a;};

#ifndef NOMATRIXTRIANGLE
MatrixTriangle MatrixTriangle::emptyMatrixTriangle(0);

Matrix::Matrix(MatrixTriangle A, char bTranspose)
{
    int n=A.nLine(),i,j;
    init(n,n,n,n);
    double **pD=(*this), **pS=A;

    if (bTranspose)
    {
        for (i=0; i<n; i++)
            for (j=0; j<n; j++)
                if (j>=i) pD[i][j]=pS[j][i];
                else pD[i][j]=0;
    } else
    {
        for (i=0; i<n; i++)
            for (j=0; j<n; j++)
                if (j<=i) pD[i][j]=pS[i][j];
                else pD[i][j]=0;
    }
}

bool Matrix::cholesky(MatrixTriangle matL, double lambda, $
$double *lambdaCorrection)
// factorize (*this)+lambda.I into L.L^t
{
    double s,s2;
    int i,j,k,n=nLine();
    matL.setSize(n);

    double **A=(*this), **L_=matL;
    if (lambdaCorrection) *lambdaCorrection=0;

    for (i=0; i<n; i++)
        {
       s2=A[i][i]+lambda; k=i;
       while ( k-- ) s2-=sqr(L_[i][k]);
       if (s2<=0)
       {
           if (lambdaCorrection)
           {
                // lambdaCorrection
               n=i+1;
               Vector X(n); // zero everywhere
               double *x=X, sum;
               x[i]=1.0;
               while(i--)
               {
                   sum=0.0;
                   for (k=i+1; k<n; k++) sum-=L_[k][i]*x[k$
$];
                   x[i]=sum/L_[i][i];
               }
               *lambdaCorrection=-s2/X.euclidianNorm();
           }
           return false;
       }
       L_[i][i] = s2 = sqrt(s2);

       for (j=i+1; j<n; j++)
       {
           s=A[i][j]; k=i;
           while (k--) s-=L_[j][k]*L_[i][k];
           L_[j][i]=s/s2;
       }
        }
    return true;
}

void Matrix::choleskySolveInPlace(Vector b)
{
    MatrixTriangle M(nLine());
    if (!cholesky(M))
    {
        b.setSize(0); // no cholesky decomposition => $
$return emptyVector
        return;
    }
    M.solveInPlace(b);
    M.solveTransposInPlace(b);
}

void Matrix::QR(Matrix Q, MatrixTriangle Rt, VectorInt $
$vPermutation)
{
//      QR factorization of the transpose of (*this)
//      beware!! :
//          1. (*this) is destroyed during the process.
//          2. Rt contains the tranpose of R (get an easy $
$manipulation matrix using:
//                   Matrix R(Rt,1); ).
//          3. use of permutation IS tested
//
//
//      subroutine qrfac(m,n,a,lda,pivot,ipvt,lipvt,rdiag,$
$acnorm,wa)
//      integer m,n,lda,lipvt
//      integer ipvt(lipvt)
//      logical pivot
//
//
//      double precision a(lda,n),rdiag(n),acnorm(n),wa(n)
```

```
//c     **********
//c
//c     subroutine qrfac
//c
//c     this subroutine uses householder transformations $
$with column
//c     pivoting (optional) to compute a qr factorization $
$of the
//c     m by n matrix a. that is, qrfac determines an $
$orthogonal
//c     matrix q, a permutation matrix p, and an upper $
$trapezoidal
//c     matrix r with diagonal elements of nonincreasing $
$magnitude,
//c     such that a*p = q*r. the householder $
$transformation for
//c     column k, k = 1,2,...,min(m,n), is of the form
//c
//c                            t
//c           i - (1/u(k))*u*u
//c
//c     where u has zeros in the first k-1 positions. the $
$form of
//c     this transformation and the method of pivoting $
$first
//c     appeared in the corresponding linpack subroutine.
//c
//c     the subroutine statement is
//c
//c       subroutine qrfac(m,n,a,lda,pivot,ipvt,lipvt,$
$rdiag,acnorm,wa)
//c
//c     where
//c
//c       m is a positive integer input variable set to $
$the number
//c         of rows of a.
//c
//c       n is a positive integer input variable set to $
$the number
//c         of columns of a.
//c
//c       a is an m by n array. on input a contains the $
$matrix for

//c         which the qr factorization is to be computed. $
$on output
//c         the strict upper trapezoidal part of a $
$contains the strict
//c         upper trapezoidal part of r, and the lower $
$trapezoidal
//c         part of a contains a factored form of q (the $
$non-trivial
//c         elements of the u vectors described above).
//c
//c       lda is a positive integer input variable not $
$less than m
//c         which specifies the leading dimension of the $
$array a.
//c
//c       pivot is a logical input variable. if pivot is $
$set true,
//c         then column pivoting is enforced. if pivot is $
$set false,
//c         then no column pivoting is done.
//c
//c       ipvt is an integer output array of length lipvt.$
$ ipvt
//c         defines the permutation matrix p such that a*p$
$ = q*r.
//c         column j of p is column ipvt(j) of the $
$identity matrix.
//c         if pivot is false, ipvt is not referenced.
//c
//c       lipvt is a positive integer input variable. if $
$pivot is false,
//c         then lipvt may be as small as 1. if pivot is $
$true, then
//c         lipvt must be at least n.
//c
//c       rdiag is an output array of length n which $
$contains the
//c         diagonal elements of r.
//c
//c       wa is a work array of length n. if pivot is $
$false, then wa
//c         can coincide with rdiag.
    char pivot=!(vPermutation==VectorInt::emptyVectorInt);
    int i,j,k,kmax,minmn;
    double ajnorm,sum,temp;
     // data one,p05,zero /1.0d0,5.0d-2,0.0d0/

    const double epsmch = 1e-20; // machine precision
    int nc=nColumn(), nl=nLine();

    if (nl>nc)
    {
        printf("QR factorisation of A^t is currently not $
$possible when number of lines is greater than number of $
$columns.\n");
        getchar(); exit(255);
    }

    Vector vWA(nl), vRDiag;
    int *ipvt;
    double *wa=vWA, *rdiag, **a=*this;

    if (pivot)
    {
        vPermutation.setSize(nl);
        ipvt=vPermutation;
        vRDiag.setSize(nl);
        rdiag=vRDiag;
    } else rdiag=wa;

//c
//c     compute the initial line norms and initialize $
$several arrays.
//c
    for (j=0; j<nl; j++)
    {
        rdiag[j]=wa[j]=euclidianNorm(j);
        if (pivot) ipvt[j]=j;
    }
//c
//c     reduce a to r with householder transformations.
//c
    minmn=mmin(nl,nc);
    for (j=0; j<minmn; j++)
    {
        if (pivot)
        {
//c
//c        bring the line of largest norm into the pivot $
$position.
//c
            kmax=j;
            for (k=j+1; k<nl; k++)
                if (rdiag[k]>rdiag[kmax]) kmax=k;

            if (kmax!=j)
            {
                for (i=0; i<nc; i++)
                {
                    temp = a[j][i];
                    a[j][i] = a[kmax][i];
                    a[kmax][i] = temp;
                }
                rdiag[kmax] = rdiag[j];
                wa[kmax] = wa[j];
                k = ipvt[j];
                ipvt[j] = ipvt[kmax];
                ipvt[kmax] = k;
            }
        }
//c
//c        compute the householder transformation to $
$reduce the
//c        j-th line of a to a multiple of the j-th unit $
$vector.
//c

//         ajnorm = enorm(nl-j+1,a(j,j))
        ajnorm=::euclidianNorm(nc-j, &a[j][j]);

        if (ajnorm==0.0) { rdiag[j]=0.0; continue; }
        if (a[j][j]<0.0) ajnorm = -ajnorm;
        for (i=j; i<nc; i++) a[j][i]=a[j][i]/ajnorm;
        a[j][j]+=1.0;

//c
//c        apply the transformation to the remaining lines
//c        and update the norms.
//c
        if (j>=nc) { rdiag[j] = -ajnorm; continue; }

        for (k = j+1; k<nl; k++)
        {
            sum=0.0;
            for (i=j; i<nc; i++) sum=sum+a[j][i]*a[k][i];

            temp = sum/a[j][j];
            for (i=j; i<nc; i++) a[k][i]=a[k][i]-temp*a[j$
$][i];

            if ((!pivot)||(rdiag[k]==0.0)) continue;
            temp = a[k][j]/rdiag[k];
            rdiag[k] *= sqrt(mmax(0.0,1.0-temp*temp));

            if (0.05*sqr(rdiag[k]/wa[k])> epsmch) continue$
$;

            //rdiag(k) = enorm(nl-j,a(jp1,k))
            rdiag[k]=::euclidianNorm(nc-j, &a[k][j+1]);
            wa[k] = rdiag[k];
        }
        rdiag[j] = -ajnorm;
    }
//c
//c     last card of subroutine qrfac.
//c
    if (!(Rt==MatrixTriangle::emptyMatrixTriangle))
    {
        Rt.setSize(minmn);
        double **r=Rt;
        for (i=0; i<minmn; i++)
        {
            r[i][i]=rdiag[i];
            for (j=i+1; j<minmn; j++)
                r[j][i]=a[j][i];
        }
    }

    if (!(Q==Matrix::emptyMatrix))
    {
        Q.setSize(nc,nc);
        double **q=Q;
        Q.diagonal(1.0);
        for (j=nl-1; j>=0; j--)
        {
            if (a[j][j]==0.0) continue;
            for (k=j; k<nc; k++)
            {
                sum=0.0;
                for (i=j; i<nc; i++) sum=sum+a[j][i]*q[i][$
$k];

                temp = sum/a[j][j];
                for (i=j; i<nc; i++) q[i][k]=q[i][k]-temp*$
$a[j][i];
            }
        }
    }
}

#endif

void Matrix::addUnityInPlace(double dd)
{
    int nn=d->extColumn+1, i=nLine();
    double *a=*d->p;
    while (i--) { (*a)+=dd; a+=nn; };
}

double Matrix::frobeniusNorm()
{
// no tested
// same code for the Vector eucilidian norm
/*
    double sum=0, *a=*p;
    int i=nLine()*nColumn();
    while (i--) sum+=sqr(*(a++));
    return sqrt(sum);
*/
    return ::euclidianNorm(nLine()*nColumn(),*d->p);
}

double Matrix::LnftyNorm()
{
// not tested
    double m=0, sum;
    int j,nl=nLine(), nc=nColumn();
    double **a=(*this), *xp;
    while (nl--)
    {
        sum=0; j=nc; xp=*(a++);
        while (j--) sum+=abs(*(xp++));
        m=::mmax(m,sum);
    }
    return m;
}

Vector Matrix::getMaxColumn()
{
    double **a=(*this), sum, maxSum=0;
    int i=nColumn(),j,k=0, nl=nLine();
    while (i--)
    {
        sum=0; j=nl;
        while(j--) sum+=sqr(a[j][i]);
        if (sum>maxSum)
        {
            maxSum=sum; k=i;
        }
    }
    Vector rr(nl);
    double *r=rr;
    j=nl;
//    while (j--) *(r++)=a[j][k];
    while (j--) r[j]=a[j][k];
    return rr;
}

Vector Matrix::getLine(int i, int n)
{
    if (n==0) n=nColumn();
    Vector r(n,d->p[i]);
    return r;
}

void Matrix::getLine(int i, Vector r, int n)
{
    if (n==0) n=nColumn();
    r.setSize(n);
    memcpy((double*)r, d->p[i], n*sizeof(double));
}

Vector Matrix::getColumn(int i, int n)
{
    if (n==0) n=nLine();
    Vector r(n);
    double **d=*this, *rr=r;
    while (n--) rr[n]=d[n][i];
    return r;
}

void Matrix::getColumn(int i, Vector r, int n)
{
    if (n==0) n=nLine();
    r.setSize(n);
    double **d=*this, *rr=r;
    while (n--) rr[n]=d[n][i];
}

void Matrix::setLine(int i, Vector v, int n)
{
    if (n==0) n=nColumn();
    memcpy(d->p[i], (double*)v, n*sizeof(double));
}

void Matrix::setLines(int indexDest, Matrix Source, int $
$indexSource, int number)
{
        if (!Source.nLine()) return;
        double **dest=(*this), **sour=Source;
        int snl=d->nColumn*sizeof(double);
    if (number==0) number=Source.nLine()-indexSource;
        while (number--) memcpy(dest[indexDest+number], $
$sour[indexSource+number], snl);
}

double Matrix::euclidianNorm(int i)
{
    return ::euclidianNorm(nColumn(), d->p[i]);
}

void Matrix::getSubMatrix(Matrix R, int startL, int startC$
$, int nl, int nc)
{
    if (nl==0) nl=  nLine()-startL; else nl=mmin(nl,  $
$nLine()-startL);
    if (nc==0) nc=nColumn()-startC; else nc=mmin(nc,$
$nColumn()-startC);
    R.setSize(nl,nc);
    double **sd=(*this), **dd=R;
    while (nl--)
        memcpy(dd[nl], sd[nl+startL]+startC, nc*sizeof($
$double));
}

void Matrix::swapLines(int i, int j)
{
    if (i==j) return;
    int n=nColumn();
    double *p1=d->p[i], *p2=d->p[j], t;
    while (n--)
    {
        t=p1[n];
        p1[n]=p2[n];
        p2[n]=t;
    }
}

/*
int Matrix::solve(Vector vB)
{
        double t;
        int i, j, k, l, info=0;
    int nl=nLine(), nc=nColumns();

        // gaussian elimination with partial pivoting
        if ( nl>1 )
        {
                for ( k=0; k<nl-1 ; k++ )
                {
                        // find l = pivot index
            l=k; maxp=abs(x[k][k]);
            for (i=k+1; i<nl; i++)
                if (abs(x[i][k])>maxp) { maxp=abs(x[i][k])$
$; l=i; }

            jpvt[k] = l;
                        // zero pivot implies this column $
$already triangularized
                        if ( maxp==0.0 ) info = k;
                        else
                        {
                                // interchange if $
$necessary
                                if ( l!=k )
                                {
                    for (i=k; i<nc; i++)
                    {
                                            t=x[l][i];
                                            x[l][i]=x[k][k$
$];
                                            x[k][k]=t;
                    }
                    t=b[l]; b[l]=b[k]; b[k]=t;
                                }
                                // compute multipliers
                                maxp=-1.0/maxp;
                                for (i=k+1; i<nc; j++ ) x[$
$k][i]*=maxp;

                                // row elimination
                                for (j=k+1; j<nl; j++ )
                                {
                                        t=x[k][j];
                                        for (i=k+1; i<nc; $
$i++) x[j][i] += t*x[k][i];
                                }
                }
                }
        }
        if ( x[nl-1][nl-1]==0.0 ) info=nl-1;
        return;
}
*/
```

14.2.2 Matrix.h

```
#ifndef _MPI_MATRIX_H
#define _MPI_MATRIX_H

#include "Vector.h"
#include "VectorInt.h"

#ifndef NOMATRIXTRIANGLE
#include "MatrixTriangle.h"
#endif

class Matrix
{
  protected:
    typedef struct MatrixDataTag
    {
       int nLine ,nColumn, extColumn, extLine;
       int ref_count;
       double **p;
    } MatrixData;
    MatrixData *d;
    void init(int _nLine, int _nColumn, int _extLine, int $
$_extColumn, MatrixData* d=NULL);
    void setExtSize(int _extLine, int _extColumn);
    void destroyCurrentBuffer();

  public:

// creation & management of Matrix:
    Matrix(int _ligne=0,int _nColumn=0);
    Matrix(int _ligne,int _nColumn, int _extLine,int $
$_extColumn);
    Matrix(char *filename);
    Matrix(Vector a, Vector b);  // a * b^T
    void save(char *filename,char ascii);
    void updateSave(char *saveFileName); // only binary
    void extendLine();
    void setNLine(int _nLine);
    void extendColumn();
    void setNColumn(int _nColumn);
    void setSize(int _nLine,int _nColumn);
    void exactshape();
    void print();

// allow shallow copy:
    ~Matrix();
    Matrix(const Matrix &A);
    Matrix& operator=( const Matrix& A );
    Matrix clone();
    void copyFrom(Matrix a);

// accessor method
    inline bool operator==( const Matrix& A ) { return (A.$
$d==d);}
    inline int nLine()   { return d->nLine; };
    inline int nColumn() { return d->nColumn; };
    inline double *operator [](int i) { return d->p[i]; };
    inline operator double**() const { return d->p; };
    Vector getLine(int i, int n=0);
    void getLine(int i, Vector r, int n=0);
    Vector getColumn(int i, int n=0);
    void getColumn(int i, Vector r, int n=0);
    void getSubMatrix(Matrix R, int startL, int StartC, $
$int nl=0, int nc=0);
    void setLine(int i, Vector v, int n=0);
    void setLines(int indexDest, Matrix Source, int $
$indexSource=0, int number=0);
    void swapLines(int i, int j);

// simple math tools:
    void zero();
    void diagonal(double d);
    Matrix multiply(Matrix B);
    void multiplyInPlace(double d);
    void multiply(Vector R, Vector v);  // result in R
    void transposeAndMultiply(Vector R, Vector a);// $
$result in R
    void multiply(Matrix R, Matrix a); // result in R
    void transposeAndMultiply(Matrix R, Matrix a);// $
$result in R
    void multiplyByTranspose(Matrix R, Matrix a); // $
$result in R
    Vector multiply(Vector v);
    void addInPlace(Matrix B);
    void addMultiplyInPlace(double d, Matrix B);
    void addUnityInPlace(double d);
    void transposeInPlace();
    Matrix transpose();
    void transpose(Matrix trans);
    double scalarProduct(int nl, Vector v);

#ifndef NOMATRIXTRIANGLE
    Matrix(MatrixTriangle A, char bTranspose=0);
    bool cholesky(MatrixTriangle matL, double lambda=0, $
$double *lambdaCorrection=NULL);
    void choleskySolveInPlace(Vector b);
    void QR(Matrix Q=Matrix::emptyMatrix, MatrixTriangle R$
$=MatrixTriangle::emptyMatrixTriangle,
                        VectorInt permutation=VectorInt::$
$emptyVectorInt);

#endif
    double frobeniusNorm();
    double LnftyNorm();
    double euclidianNorm(int i);
    Vector getMaxColumn();

// default return matrix in case of problem in a function
    static Matrix emptyMatrix;
};

#endif
```

14.2.3 MatrixTriangle.h

```
#ifndef _MPI_MATRIXTRIANGLE_H
#define _MPI_MATRIXTRIANGLE_H

#include "Vector.h"

class Matrix;

class MatrixTriangle // lower triangular
{
  friend class Matrix;
  protected:
    void destroyCurrentBuffer();
    typedef struct MatrixTriangleDataTag
    {
       int n;
       int ext;
       int ref_count;
       double **p;
    } MatrixTriangleData;
    MatrixTriangleData *d;

  public:

// creation & management of Matrix:
    MatrixTriangle(int _n=0);
    void setSize(int _n);

// allow shallow copy:
    ~MatrixTriangle();
    MatrixTriangle(const MatrixTriangle &A);
    MatrixTriangle& operator=( const MatrixTriangle& A );
    MatrixTriangle clone();
    void copyFrom(MatrixTriangle r);

// accessor method
    inline bool operator==( const MatrixTriangle& A ) { $
$return (A.d==d);}
    inline int nLine() { return d->n; };
    inline double *operator [](int i) { return d->p[i]; };
    inline operator double**() const { return d->p; };

// simple math tools:
    void solveInPlace(Vector b);
    void solveTransposInPlace(Vector y);
    //void invert();
    void LINPACK(Vector &u);

// default return matrix in case of problem in a function
    static MatrixTriangle emptyMatrixTriangle;
};

#endif
```

14.2.4 MatrixTiangle.cpp

```
#include <stdio.h>
#include <memory.h>
#include "MatrixTriangle.h"

MatrixTriangle::MatrixTriangle(int _n)
{
    d=(MatrixTriangleData*)malloc(sizeof($
$MatrixTriangleData));
    d->n=_n; d->ext=_n;
    d->ref_count=1;
    if (_n>0)
```

```
        {
            double **t,*t2;
        int i=1;
        t=d->p=(double**)malloc(_n*sizeof(double*));
        t2=(double*)malloc((_n+1)*_n/2*sizeof(double));
            while(_n--)
        {
            *(t++)=t2; t2+=i; i++;
        }
    } else d->p=NULL;
}

void MatrixTriangle::setSize(int _n)
{
    d->n=_n;
    if (_n>d->ext)
    {
        d->ext=_n;
            double **t,*t2;
        if (!d->p)
        {
            t2=(double*)malloc((_n+1)*_n/2*sizeof(double))$
$;
            t=d->p=(double**)malloc(_n*sizeof(double));
        } else
        {
            t2=(double*)realloc(*d->p,(_n+1)*_n/2*sizeof($
$double));
            t=d->p=(double**)realloc(d->p,_n*sizeof(double$
$));
        }
        int i=1;
            while(_n--)
        {
            *(t++)=t2; t2+=i; i++;
        }
    }
}

void MatrixTriangle::solveInPlace(Vector b)
{
    int i,k,n=nLine();
    double **a=(*this), *x=b, sum;

    if ((int)b.sz()!=n)
    {
        printf("error in matrixtriangle solve.\n"); getchar$
$(); exit(254);
    }
    for (i=0; i<n; i++)
    {
        sum=x[i]; k=i;
        while (k--) sum-=a[i][k]*x[k];
        x[i]=sum/a[i][i];
    }
}

void MatrixTriangle::solveTransposInPlace(Vector y)
{
    int n=nLine(),i=n,k;
    double **a=(*this), *x=y, sum;

    while(i--)
    {
        sum=x[i];
        for (k=i+1; k<n; k++) sum-=a[k][i]*x[k];
        x[i]=sum/a[i][i];
    }
}
/*
void MatrixTriangle::invert()
{
    int i,j,k,n=nLine();
    double **a=(*this), sum;
    for (i=0; i<n; i++)
    {
        a[i][i]=1/a[i][i];
        for (j=i+1; j<n; j++)
        {
            sum=0;
            for (k=i; k<j; k++) sum-=a[j][k]*a[k][i];
            a[j][i]=sum/a[j][j];
        }
    }
}
*/
void MatrixTriangle::LINPACK(Vector &R)
{
    int i,j,n=nLine();
    R.setSize(n);
    double **L=(*this), *w=R, sum;

    for (i=0; i<n; i++)
    {
        if (L[i][i]==0) w[i]=1.0;

        sum=0; j=i-1;
        if (i) while (j--) sum+=L[i][j]*w[j];
        if (((1.0-sum)/L[i][i])>((-1.0-sum)/L[i][i])) w[i$
$]=1.0; else w[i]=-1.0;
    }
    solveTransposInPlace(R);
    R.multiply(1/R.euclidianNorm());
};


MatrixTriangle::~MatrixTriangle()
{
    destroyCurrentBuffer();
};

void MatrixTriangle::destroyCurrentBuffer()
{
    if (!d) return;
    (d->ref_count) --;
        if (d->ref_count==0)
    {
        if (d->p) { free(*d->p); free(d->p); }
        free(d);
    };
}

MatrixTriangle::MatrixTriangle(const MatrixTriangle &A)
{
    // shallow copy
    d=A.d;
        (d->ref_count)++ ;
}

MatrixTriangle& MatrixTriangle::operator=( const $
$MatrixTriangle& A )
{
    // shallow copy
    if (this != &A)
        {
        destroyCurrentBuffer();
        d=A.d;
                (d->ref_count) ++ ;
        }
        return *this;
}

MatrixTriangle MatrixTriangle::clone()
{
    // a deep copy
    MatrixTriangle r(nLine());
    r.copyFrom(*this);
    return r;
}

void MatrixTriangle::copyFrom(MatrixTriangle r)
{
    int n=r.nLine();
    setSize(n);
    if (n==0) return;
    memcpy(*d->p,*(r.d->p),(n+1)*n/2*sizeof(double));
}
```

14.2.5 Vector.h

```
#ifndef _MPI_VECTOR_H
#define _MPI_VECTOR_H

#include <stdlib.h> // for the declaration of NULL
#include "VectorInt.h"

class Matrix;

class Vector
{
  public:
    // only use the following method at your own risks!
    void prepareExtend(int new_extention);
    void alloc(int n, int ext);
    typedef struct VectorDataTag
    {
        int n,extention;
        int ref_count;
        double *p;
        char externalData;
    } VectorData;
    VectorData *d;

// creation & management of Vector:
    Vector(int _n=0);
    Vector(int _n, int _ext);
    Vector(int _n, double *dd, char externalData=0);
    Vector(char *filename);
    Vector(char *line, int guess_on_size);

    void getFromLine(char *line);
    void extend();
    void setSize(int _n);
    void exactshape();
    void print();
    void save(char *filename);
    void setExternalData(int _n, double *dd);

// allow shallow copy:
    Vector clone();
    void copyFrom(Vector r, int _n=0);
    Vector( const Vector& P );
        Vector& operator=( const Vector& P );
    void destroyCurrentBuffer();
    ~Vector();

// accessor method
    inline unsigned sz() {return d->n;};
//  inline double &operator [](int i) { return d->p[i]; };
    inline int operator==( const Vector Q) { return d==Q.d$
$; };
    int equals( const Vector Q );
    operator double*() const { if (d) return d->p; else $
$return NULL; };
    //double &operator[]( unsigned i) {return p[i];};
```

```
        void setPart(int i, Vector v, int n=0, int ii=0);

// simple math tools:
    double euclidianNorm();
    double L1Norm();
    double LnftyNorm();
    double euclidianDistance(Vector v);
    double L1Distance(Vector v);
    double LnftyDistance(Vector v);

    double square();
    void multiply(double a);
    void multiply(Vector R, double a);
    void zero(int _i=0, int _n=0);
    void set(double dd);
    void shift(int s);
    double scalarProduct(Vector v);
    double mmin();
    double mmax();
    bool isNull();
    Vector operator-( Vector v);
    Vector operator+( Vector v);
    Vector operator-=( Vector v);
    Vector operator+=( Vector v);
    void addInPlace(double a, Vector v); // this+=a*v
    void addInPlace(double a, int i, Matrix m); // this+=a$
$ * M(i,:)
    void transposeAndMultiply(Vector vR, Matrix M);
    void permutIn(Vector vR, VectorInt viP);
    void permutOut(Vector vR, VectorInt viP);

// default return Vector in case of problem in a function
    static Vector emptyVector;

};

#endif
```

14.2.6 Vector.cpp

```
#include <stdio.h>
#include <memory.h>
#include <string.h> // for memmove: microsoft bug
#include "Vector.h"
#include "Matrix.h"
#include "tools.h"

Vector Vector::emptyVector;

void Vector::alloc(int _n, int _ext)
{
    d=(VectorData*)malloc(sizeof(VectorData));
    d->n=_n;
    d->extention=_ext;
    d->ref_count=1;

    if (_ext==0) { d->p=NULL; return; };

    d->p=(double*)malloc(_ext*sizeof(double));
    if (d->p==NULL) { printf("memory allocation error\n");$
$ getchar(); exit(253); }
}

Vector::Vector(int n)
{
    alloc(n,n);
    zero();
};

Vector::Vector(int n, int ext)
{
    alloc(n,ext);
    zero();
};

Vector::Vector(int n, double *dd, char _exte)
{
    alloc(n,n);
    if (dd)
    {
        if (_exte) { d->externalData=1; d->p=dd; }
        else memcpy(d->p,dd,n*sizeof(double));
    }
    else zero();
```

```
}

void Vector::setExternalData(int _n, double *dd)
{
    if ((d->extention==_n)||(!d->extention))
    {
        d->n=_n; d->extention=_n; d->externalData=1; d->p=$
$dd;
    } else
    {
        printf("do not use this function ('setExternalData$
$'): it's too dangerous.\n");
        getchar(); exit(255);
    }
}


void Vector::zero(int i, int _n)
{
    if (_n==0) _n=d->n-i;
    if (d->p) memset(d->p+i,0,_n*sizeof(double));
}

void Vector::prepareExtend(int new_extention)
{
        if (d->extention<new_extention)
        {
                d->p=(double*)realloc(d->p,new_extention*$
$sizeof(double));
        if (d->p==NULL) { printf("memory allocation error\$
$n"); getchar(); exit(253); }

        // not really necessary:
        memset(d->p+d->extention,0,(new_extention-d->$
$extention)*sizeof(double));
        d->extention=new_extention;
        };
};

void Vector::setSize(int _n)
{
    d->n=_n;
    if (_n==0) { if (d->p) free(d->p); d->p=NULL; d->$
$extention=0; return; }
    prepareExtend(_n);
}

void Vector::extend()
{
    d->n++;
    if (d->n>d->extention) prepareExtend(d->extention+100)$
$;
}

void Vector::exactshape()
{
        if (d->extention!=d->n)
        {
                d->p=(double*)realloc(d->p,d->n*sizeof($
$double));
        if (d->p==NULL) { printf("memory allocation error\$
$n"); getchar(); exit(253); }
                d->extention=d->n;
        };
};

int Vector::equals( Vector Q )
{
  if (Q.d==d) return 1;
  if (Q.d==emptyVector.d)
  {
      double *cP=d->p;
      int i=sz();
      while (i--) if (*(cP++)) return 0;
      return 1;
  }

  if (sz() != Q.sz()) return 0;

  double *cP = d->p, *cQ = Q.d->p;
  int i = sz();

  while( i-- )
  {
    if (*cP!=*cQ) return 0;
    cP++; cQ++;
  }

  return 1;
}

//ostream& Vector::PrintToStream( ostream& out ) const
void Vector::print()
{
    int N=sz();
        printf("[");
        if (!N || !d->p) { printf("]\n"); return; }

    double *up=d->p;
        while (--N) printf("%f,",*(up++));
        printf("%f]\n",*up);
}

Vector::~Vector()
{
    destroyCurrentBuffer();
};

void Vector::destroyCurrentBuffer()
{
    if (!d) return;
    (d->ref_count) --;
        if (d->ref_count==0)
    {
        if ((d->p)&&(!d->externalData)) free(d->p);
        free(d);
    };
}

Vector::Vector(const Vector &A)
{
    // shallow copy
    d=A.d;
        (d->ref_count)++ ;
}

Vector& Vector::operator=( const Vector& A )
{
    // shallow copy
    if (this != &A)
        {
        destroyCurrentBuffer();
        d=A.d;
                (d->ref_count) ++ ;
        }
        return *this;
}

Vector Vector::clone()
{
    // a deep copy
    Vector r(sz());
    r.copyFrom(*this);
    return r;
}

void Vector::copyFrom(Vector r, int _n)
{
    if (_n==0) _n=r.sz();
    setSize(_n);
    if (_n) memcpy(d->p,r.d->p,_n*sizeof(double));
}

double Vector::euclidianNorm()
{
    return ::euclidianNorm(sz(), d->p);
}

double Vector::L1Norm()
{
    if (sz()==0) return 0;
    double *x=d->p, sum=0;
    int ni=sz();
    while (ni--) sum+=abs(*(x++));
    return sum;
}

double Vector::square()
{
    double *xp=d->p, sum=0;
    int ni=sz();
    while (ni--) sum+=sqr(*(xp++));
    return sum;
}

double Vector::euclidianDistance(Vector v)
{
    Vector t=(*this)-v;
    return ::euclidianNorm(sz(), t.d->p);
/*
    double *xp1=d->p, *xp2=v.d->p, sum=0;
    int ni=sz();
    while (ni--) sum+=sqr(*(xp1++)-*(xp2++));
    return sqrt(sum);
*/
}

double Vector::LnftyDistance(Vector v)
{
    double *xp1=d->p, *xp2=v.d->p, sum=-1.0;
    int ni=sz();
    while (ni--) sum=::mmax(sum, abs(*(xp1++)-*(xp2++)));
    return sum;
}

double Vector::LnftyNorm()
{
    double *xp1=d->p, sum=-1.0;
    int ni=sz();
    while (ni--) sum=::mmax(sum, abs(*(xp1++)));
    return sum;
}

double Vector::L1Distance(Vector v)
{
    if (sz()==0) return 0;
    double *xp1=d->p, *xp2=v.d->p, sum=0;
    int ni=sz();
    while (ni--) sum+=abs(*(xp1++)-*(xp2++));
    return sum;
}

void Vector::multiply(double a)
{
    double *xp=d->p;
    int ni=sz();
    while (ni--) *(xp++)*=a;
}

void Vector::multiply(Vector R, double a)
{
    int ni=sz();
    R.setSize(ni);
    double *xs=d->p, *xd=R;
    while (ni--) *(xd++)=a * (*(xs++));
}
```

```
void Vector::transposeAndMultiply(Vector vR, Matrix M)
{
    if ((int)sz()!=M.nLine())
    {
        printf("error in V^t * M.\n"); getchar(); exit$
$(254);
    }
    int n=sz(), szr=M.nColumn(), i;
    vR.setSize(szr);
    double sum, *dv=(*this), **dm=M, *dd=vR;

    while (szr--)
    {
        sum=0.0;
        i=n;
        while (i--) sum+=dv[i]*dm[i][szr];
        dd[szr]=sum;
    }
}

double Vector::scalarProduct(Vector v)
{
    double *xp1=d->p, *xp2=v.d->p, sum=0;
    int ni=sz();
    while (ni--) { sum+=*(xp1++) * *(xp2++); };
    return sum;
}

double Vector::mmin()
{
    if (sz()==0) return 0;
    double *xp=d->p, m=INF;
    int ni=sz();
    while (ni--) m=::mmin(m,*(xp++));
    return m;
}

double Vector::mmax()
{
    if (sz()==0) return 0;
    double *xp=d->p, m=-INF;
    int ni=sz();
    while (ni--) m=::mmax(m,*(xp++));
    return m;
}

bool Vector::isNull()
{
    double *xp=d->p;
    int ni=sz();
    while (ni--) if (*(xp++)!=0) return false;
    return true;
}

Vector Vector::operator-( Vector v)
{
    int ni=sz();
    Vector r(sz());
    double *xp1=r.d->p, *xp2=v.d->p, *xp3=d->p;
    while (ni--)
        *(xp1++)+=*(xp3++)-*(xp2++);
    return r;
}

Vector Vector::operator+( Vector v)
{
    int ni=sz();
    Vector r(sz());
    double *xp1=r.d->p, *xp2=v.d->p, *xp3=d->p;
    while (ni--)
        *(xp1++)+=*(xp3++)+*(xp2++);
    return r;
}

Vector Vector::operator-=( Vector v)
{
    int ni=sz();
    double *xp1=d->p, *xp2=v.d->p;
    while (ni--) *(xp1++)-=*(xp2++);
    return *this;
}

Vector Vector::operator+=( Vector v)
{
    int ni=sz();
    double *xp1=d->p, *xp2=v.d->p;
    while (ni--) *(xp1++)+=*(xp2++);
    return *this;
}

void Vector ::addInPlace(double a, Vector v)
{
    int ni=sz();
    double *xp1=d->p, *xp2=v.d->p;
    if (a==1.0)  while (ni--) *(xp1++)+=     *(xp2++);
    else         while (ni--) *(xp1++)+=a * (*(xp2++));
}

void Vector::addInPlace(double a, int i, Matrix m)
{
    int ni=sz();
    double *xp1=d->p, *xp2=((double**)m)[i];
    while (ni--) *(xp1++)+=a * (*(xp2++));
}

Vector::Vector(char *filename)
{
    unsigned _n;
    FILE *f=fopen(filename,"rb");
    fread(&_n, sizeof(int),1, f);
    alloc(_n,_n);
    fread(d->p, d->n*sizeof(double),1, f);
    fclose(f);
}

void Vector::save(char *filename)
{
    FILE *f=fopen(filename,"wb");
    fwrite(&d->n, sizeof(int),1, f);
    fwrite(d->p, d->n*sizeof(double),1, f);
    fclose(f);
}

void Vector::setPart(int i, Vector v, int n, int ii)
{
        if (n==0) n=v.sz()-ii;
    memcpy(d->p+i, ((double*)v)+ii, n*sizeof(double));
}

void Vector::set(double dd)
{
    double *p=(*this);
    if (!p) return;
    int n=sz();
    while (n--) *(p++)=dd;
}

void Vector::shift(int s)
{
    int n=sz();
    if (!n) return;
    double *ps=(*this), *pd=ps; // pointer source / $
$destination
    if (s==0) return;
    if (s>0) { n-=s; pd+=s; }
    else { n+=s; ps+=s; }
    memmove(pd,ps,n*sizeof(double));
}

void Vector::permutIn(Vector vR, VectorInt viP)
{
    int i,n=sz(), *ii=viP;
    if (!n) return;
    if (n!=viP.sz())
    {
        printf("error in permutation IN: sizes don't agree$
$.\n"); getchar(); exit(255);
    }
    vR.setSize(n);
    double *ps=(*this), *pd=vR; // pointer source / $
$destination
    for (i=0; i<n; i++)
//        *(pd++)=ps[ii[i]];
        pd[ii[i]]=*(ps++);
}

void Vector::permutOut(Vector vR, VectorInt viP)
{
    int i,n=sz(), *ii=viP;
    if (!n) return;
    if (n!=viP.sz())
    {
        printf("error in permutation IN: sizes don't agree$
$.\n"); getchar(); exit(255);
    }
    vR.setSize(n);
    double *ps=(*this), *pd=vR; // pointer source / $
$destination
    for (i=0; i<n; i++)
//        pd[ii[i]]=*(ps++);
        *(pd++)=ps[ii[i]];
}

#define EOL1 13
```

```
#define EOL2 10
Vector::Vector(char *line, int gn)
{
        char *tline=line;

    if (gn==0)
    {
            while ((*tline!=EOL1)&&(*tline!=EOL2))
            {
                    while ((*tline==' ')||
                            (*tline=='\t'))tline++;
                    if ((*tline==EOL1)||(*tline==EOL2)) $
$break;
                    while(((*tline>='0')&&(*tline<='9'))||
                            (*tline=='.')||
                            (*tline=='e')||
                            (*tline=='E')||
                            (*tline=='+')||
                            (*tline=='-')) tline++;
                    gn++;
            };
    };

        if (gn==0) { alloc(0,0); return; };
    alloc(gn,gn);
    getFromLine(line);
};

void Vector::getFromLine(char *line)
{
    double *dp=d->p;
    int n=sz(),k;
        char *tline=line, *oldtline;
        for (k=0; k<n; k++)
    {
                while ((*tline==' ')||
                            (*tline=='\t'))tline++;
                if ((*tline==EOL1)||(*tline==EOL2))
        {
            setSize(k);
                return;
        }
        oldtline=tline;
        while(((*tline>='0')&&(*tline<='9'))||
                        (*tline=='.')||
                        (*tline=='e')||
                        (*tline=='E')||
                        (*tline=='+')||
                        (*tline=='-')) tline++;
        if (oldtline==tline)
        {
            setSize(k);
                return;
        };
        *tline='\0'; tline++;
        dp[k]=atof(oldtline);
    }
}
```

14.2.7 Poly.h

```
//
//      Multivariate Polynomials
//      Public header
//      ...
//      V 0.0


#ifndef _MPI_POLY_H_
#define _MPI_POLY_H_

#include "MultInd.h"
#include "Vector.h"
//#include "tools.h"
#include "Vector.h"
#include "Matrix.h"

// ORDER BY DEGREE !
class Polynomial {
protected:

    typedef struct PolynomialDataTag
    {
            double *coeff;      // Coefficients
            unsigned n,      // size of vector of $
$Coefficients
                    dim,    // Dimensions
                        deg;        // Degree
        int ref_count;
    } PolynomialData;
    PolynomialData *d;
    void init(int _dim, int _deg, double *data=NULL);
    void destroyCurrentBuffer();

public:
    Polynomial(){ init(0,0); };
    Polynomial( unsigned Dim, unsigned deg=0, double *data$
$=0 );
    Polynomial( unsigned Dim, double val ); // Constant $
$polynomial
    Polynomial( MultInd& );  // Monomials
    Polynomial(char *name);


        // Accessor
    inline unsigned dim() { return d->dim; };
    inline unsigned deg() { return d->deg; };
    inline unsigned sz() { return d->n; };
    inline operator double*() const { return d->coeff; };

    // allow shallow copy:
    ~Polynomial();
    Polynomial(const Polynomial &A);
    Polynomial& operator=( const Polynomial& A );
    Polynomial clone();
    void copyFrom(Polynomial a);

        // Arithmetic operations

  // friend Polynomial operator*( const double&, const $
$Polynomial& );
        Polynomial operator*( const double );
        Polynomial operator/( const double );
        Polynomial operator+( Polynomial );
        Polynomial operator-( Polynomial );

        // Unary
        Polynomial operator-( void ); // the opposite ($
$negative of)
        Polynomial operator+( void )
                { return *this; }

        // Assignment+Arithmetics

        Polynomial operator+=( Polynomial );
        Polynomial operator-=( Polynomial );
        Polynomial operator*=( const double );
        Polynomial operator/=( const double );

        // simple math tools

//    double simpleEval( Vector P);
    double shiftedEval( Vector Point, double minusVal);
        double operator()( Vector );
    Polynomial derivate(int i);
    void gradient(Vector P, Vector G);
    void gradientHessian(Vector P, Vector G, Matrix H);
    void translate(Vector translation);

    // Comparison

    inline int operator==( const Polynomial q) { return d$
$==q.d; };
    int equals( Polynomial q );

        // Output

    void print();
    void save(char *name);

        //ostream& PrintToStream( ostream& ) const;

    //behaviour
    static const unsigned int NicePrint;
    static const unsigned int Warning;
    static const unsigned int Normalized;  // Use $
$normalized monomials
```

```
    static unsigned int flags;
    void       setFlag( unsigned int val ) { flags |= val;$
$ }
    void     unsetFlag( unsigned int val ) { flags &= ~val$
$; }
    unsigned queryFlag( unsigned int val ) { return flags $
$& val; }

    static Polynomial emptyPolynomial;
};

unsigned long choose( unsigned n, unsigned k );
```

```
// operator * defined on double:
inline Polynomial operator*( const double& dou, Polynomial$
$& p  )
{
    // we can use operator * defined on Polynomial because$
$ of commutativity
        return p * dou;
}

#endif  /* _MPI_POLY_H_ */
```

14.2.8 Poly.cpp

```
//
//      Multivariate Polynomials
//      Private header
//      ...
//      V 0.0


#ifndef _MPI_POLYP_H_
#define _MPI_POLYP_H_

#include <stdio.h>
#include <memory.h>
//#include <crtdbg.h>
#include "Vector.h"
#include "MultInd.h"
#include "tools.h"
#include "Poly.h"
#include "IntPoly.h"

const unsigned int Polynomial::NicePrint = 1;
const unsigned int Polynomial::Warning   = 2;
const unsigned int Polynomial::Normalized= 4;  // Use $
$normalized monomials
      unsigned int Polynomial::flags = Polynomial::Warning$
$||Polynomial::NicePrint;

Polynomial Polynomial::emptyPolynomial;

void Polynomial::init(int _dim, int _deg, double *data)
{
    int n;
    d=(PolynomialData*)malloc(sizeof(PolynomialData));
    if (_dim) n=d->n=choose( _dim+_deg, _dim );
    else n=d->n=0;

    d->dim=_dim;
    d->deg=_deg;
    d->ref_count=1;

    if (n==0) { d->coeff=NULL; return; };

    d->coeff=(double*)malloc(n*sizeof(double));
    if (d->coeff==NULL) { printf("memory allocation error\$
$n"); getchar(); exit(253); }

    if (data) memcpy(d->coeff, data, d->n*sizeof(double));
    else memset(d->coeff, 0, d->n*sizeof(double));
}
/*
Polynomial::PolyInit( const Polynomial& p )
{ dim = p.dim; deg = p.deg; coeff=((Vector)p.coeff).clone$
$(); }
*/
Polynomial::Polynomial( unsigned Dim, unsigned Deg, double$
$ *data )
{
    init(Dim,Deg,data);
}

Polynomial::Polynomial( unsigned Dim, double val ) // $
$Constant polynomial
{
    init(Dim,0,&val);
}

Polynomial::Polynomial( MultInd& I )
{
    init(I.dim,I.len());
        d->coeff[I.index()] = 1;
}

Polynomial::~Polynomial()
{
    destroyCurrentBuffer();
};

void Polynomial::destroyCurrentBuffer()
{
    if (!d) return;
    (d->ref_count) --;
        if (d->ref_count==0)
    {
        if (d->coeff) free(d->coeff);
        free(d);
    }
}

Polynomial& Polynomial::operator=( const Polynomial& A )
{
    // shallow copy
    if (this != &A)
        {
        destroyCurrentBuffer();
        d=A.d;
                (d->ref_count) ++ ;
        }
        return *this;
}

Polynomial::Polynomial(const Polynomial &A)
{
    // shallow copy
    d=A.d;
        (d->ref_count)++ ;
}

Polynomial Polynomial::clone()
{
    // a deep copy
    Polynomial m(d->dim,d->deg);
    m.copyFrom(*this);
    return m;
}

void Polynomial::copyFrom(Polynomial m)
{
    if (m.d->dim!=d->dim)
    {
        printf("poly: copyFrom: dim do not agree");
        getchar(); exit(254);
    }

    d->deg = mmax(d->deg,m.d->deg);     // New degree
    unsigned N1=sz(), N2=m.sz();
    if (N1!=N2)
    {
        d->coeff=(double*)realloc(d->coeff,N2*sizeof($
$double));
        d->n=m.d->n;
    }
    memcpy((*this),m,N2*sizeof(double));
}

Polynomial Polynomial::operator*( const double t )
{
    int i=sz();
        Polynomial q( d->dim, d->deg );
        double *tq = q.d->coeff, *tp = d->coeff;

        while (i--) *(tq++) = *(tp++) * t;
        return q;
}


Polynomial Polynomial::operator/( const double t )
{
        if (t == 0)
        {
                printf( "op/(Poly,double): Division by zero$
$\n");
                getchar(); exit(-1);
        }
    int i=sz();
        Polynomial q( d->dim, d->deg );
        double *tq = q.d->coeff, *tp = d->coeff;

        while (i--) *(tq++) = *(tp++) / t;
        return q;
}

Polynomial Polynomial::operator+( Polynomial q )
{
        if (d->dim != q.d->dim)
        {
                printf( "Poly::op+ : Different dimension\n"$
$);
                getchar(); exit(-1);
        }

        Polynomial r(d->dim,mmax(d->deg,q.d->deg));
        unsigned N1=sz(), N2=q.sz(), Ni=mmin(N1,N2);
        double  *tr = r, *tp = (*this), *tq = q;
    while (Ni--) *(tr++) = *(tp++) + *(tq++);
    if (N1<N2)
    {
        memcpy(tr,tq,(N2-N1)*sizeof(double));
//        N2-=N1; while (N2--) *(tr++)=*(tq++);
    }
    return r;
}


Polynomial Polynomial::operator-( Polynomial q )
{
        if (d->dim != q.d->dim)
        {
                printf("Poly::op- : Different dimension\n")$
$;
                getchar(); exit(-1);
        }

        Polynomial r(d->dim,mmax(d->deg,q.d->deg));
        unsigned N1=sz(), N2=q.sz(), Ni=mmin(N1,N2);
        double  *tr = r, *tp = (*this), *tq = q;
    while (Ni--) *(tr++) = *(tp++) - *(tq++);
    if (N1<N2)
    {
        N2-=N1; while (N2--) *(tr++)=-(*(tq++));
    }
        return r;
}

Polynomial Polynomial::operator-( void )
{
        unsigned Ni = sz();
    double *tp = (*this);

        if (!Ni || !tp) return *this;   // Take it like it$
$ is ...

        Polynomial r(d->dim,d->deg);
        double *tq = (r);
        while( Ni-- ) *(tq++) = -(*(tp++));
        return r;
}


Polynomial Polynomial::operator+=( Polynomial p )
{
        if (d->dim != p.d->dim)
        {
                printf("Poly::op+= : Different dimension\n"$
$);
                getchar(); exit(-1);
        }

    d->deg = mmax(d->deg,p.d->deg);     // New degree
    unsigned N1=sz(), N2=p.sz(), Ni=mmin(N1,N2);
    if (N1<N2)
    {
        d->coeff=(double*)realloc(d->coeff,N2*sizeof($
$double));
        d->n=p.d->n;
    }
        double *tt = (*this),*tp = p;

    while (Ni--) *(tt++) += *(tp++);

    if (N1<N2)
    {
        memcpy(tt,tp,(N2-N1)*sizeof(double));
//        N2-=N1; while (N2--) *(tt++)=*(tp++);
    }

        return *this;
}


Polynomial Polynomial::operator-=( Polynomial p )
{
        if (d->dim != p.d->dim)
        {
                printf( "Poly::op-= : Different dimension\n$
$");
                getchar(); exit(-1);
        }

    d->deg = mmax(d->deg,p.d->deg);     // New degree
    unsigned N1=sz(), N2=p.sz(), Ni=mmin(N1,N2);
    if (N1<N2)
    {
        d->coeff=(double*)realloc(d->coeff,N2*sizeof($
$double));
        d->n=p.d->n;
    }
        double *tt = (*this),*tp = p;

    while (Ni--) *(tt++) -= *(tp++);

    if (N1<N2)
    {
        N2-=N1; while (N2--) *(tt++)=-(*(tp++));
    }

        return *this;
}


Polynomial Polynomial::operator*=( const double t )
{
    int i=sz();
        double *tp = (*this);

        while (i--) *(tp++) *=t;
        return *this;
}


Polynomial Polynomial::operator/=( const double t )
{
        if (t == 0)
        {
                printf( "Poly::op/= : Division by zero\n");
                getchar(); exit(-1);
        }

    int i=sz();
        double *tp = (*this);

        while (i--) *(tp++) /=t;
        return *this;
}

int Polynomial::equals( Polynomial q )
{
    if (d==q.d) return 1;
        if ( (d->deg != q.d->deg) || (d->dim != q.d->dim) $
$) return 0;

        unsigned N = sz();
        double  *tp = (*this),*tq = q;

        while (N--)
                if ( *(tp++) != *(tq++) ) return 0;

        return 1;
}

//ostream& Polynomial::PrintToStream( ostream& out ) const
void Polynomial::print()
{
        MultInd I( d->dim );
    double *tt = (*this);
        unsigned N = sz();
        bool IsFirst=true;

        if ( !N || !tt ) { printf("[Void polynomial]\n"); $
$return; }

    if (*tt) { IsFirst=false; printf("%f", *tt); }
    tt++; ++I;

        for (unsigned i = 1; i < N; i++,tt++,++I)
```

```
                                if (queryFlag( NicePrint )$
$)

    {
                if (*tt != 0)
                {
                        if (IsFirst)
                        {



{
                    if (*tt<0) printf("-");
                    printf("%f x^",abs(*tt)); I.print();
                                }
                                else
                                {
                    printf("+%f x^",*tt); I.print();
                                }
                                IsFirst = false;
                                continue;
                        }
                        if (queryFlag( NicePrint ))
                        {
                if (*tt<0) printf("-"); else printf("+");
                printf("%f x^",abs(*tt)); I.print();
                        }
                        else
                        {
                printf("+%f x^",*tt); I.print();
                        }

                }
        }
}

/*
double Polynomial::simpleEval(Vector P)
{
    unsigned i=coeff.sz(),j;
    double *cc=coeff,r=0, r0, *p=(double*)P;
    MultInd I(dim);
    while (i--)
    {
        r0=*(cc++); j=dim;
        while (j--) r0*=pow(p[j],I[j]);
        r+=r0; I++;
    }
    return r;
}
*/

double Polynomial::shiftedEval( Vector Point, double $
$minusVal)
{
    double tmp1=d->coeff[0], tmp2;
    d->coeff[0]-=minusVal;
    tmp2=(*this)(Point);
    d->coeff[0]=tmp1;
    return tmp2;
}

// Evaluation oprator
// According to Pena, Sauer, "On the multivariate Horner $
$scheme",
//   SIAM J. Numer. Anal., to appear

double Polynomial::operator()( Vector Point )
{
// I didn't notice any difference in precision:
//  return simpleEval(P);

  unsigned dim=d->dim, deg=d->deg;
  double r,r0;                              // no static $
$here because of the 2 threads !
  double rbuf[100];     // That should suffice // no $
$static here because of the 2 threads !
  double *rbufp = rbuf;
  unsigned lsize = 100;
  double *rptr;
  int i,j;

  if (Point==Vector::emptyVector) return *d->coeff;

  if ( dim != (unsigned)Point.sz() )
  {
    printf( "Polynomial::operator()( Vector& ) : Improper $
$size\n");
    getchar(); exit(-1);
  }

  if ( !sz() )
```

```
  {
    if ( queryFlag( Warning ) )
    {
      printf( "Polynomial::operator()( Vector& ) : $
$evaluating void polynomial\n");
    }
    return 0;
  }

  if ( dim > lsize )    // Someone must be crazy !!!
  {
    if ( queryFlag( Warning ) )
    {
        printf( "Polynomial::operator()( Vector& ) : $
$Warning -> 100 variables\n");
    }
    if ((rbufp != rbuf) && rbufp) delete rbufp;

    lsize=dim;
    rbufp = (double*)malloc(lsize*sizeof(double));      //$
$ So be it ...

    if ( !rbufp )
    {
      printf( "Polynomial::operator()( Vector& ) : Cannot $
$allocate <rbufp>\n");
      getchar(); exit( -1 );
    }
  }

  if (deg==0) return *d->coeff;

  // Initialize
  MultInd *mic=cacheMultInd.get( dim, deg );
  unsigned *nextI=mic->indexesOfCoefInLexOrder(),
           *lcI=mic->lastChanges();
  double *cc = (*this), *P=Point;
  unsigned nxt, lc;

  // Empty buffer (all registers = 0)
  memset(rbufp,0,dim*sizeof(double));

  r0=cc[*(nextI++)];
  i=sz()-1;
  while (i--)
  {
    nxt= *(nextI++);
    lc = *(lcI++);

    r=r0; rptr=rbufp+lc; j=dim-lc;
    while (j--) { r+=*rptr; *(rptr++)=0; }
    rbufp[lc]=P[lc]*r;
    r0=cc[nxt];
  }
  r=r0; rptr=rbufp; i=(int)dim;
  while (i--) r+=*(rptr++);

  return r;
}

Polynomial Polynomial::derivate(int i)
{
    unsigned dim=d->dim, deg=d->deg;
    if (deg<1) return Polynomial(dim,0.0);

    Polynomial r(dim, deg-1);
        MultInd I( dim );
        MultInd J( dim );
    double *tS=(*this), *tD=r;
        unsigned j=sz(), k, *cc, sum,
             *allExpo=(unsigned*)I, *expo=allExpo+i, *$
$firstOfJ=(unsigned*)J;

    while (j--)
    {
        if (*expo)
        {
            (*expo)--;

            sum=0; cc=allExpo; k=dim;
            while (k--) sum+=*(cc++);
            if (sum) k=choose( sum-1+dim, dim ); else k=0;
            J.resetCounter(); *firstOfJ=sum;
            while (!(J==I)) { k++; J++; }

            (*expo)++;
            tD[k]=(*tS) * (double)*expo;
        }
        tS++;
        I++;
    }
    return r;
}
```

```
{


void Polynomial::gradient(Vector P, Vector G)



unsigned
 i=d->dim;
    G.setSize(i);
    double *r=G;
    if (P.equals(Vector::emptyVector))
    {
        memcpy(r,d->coeff+1,i*sizeof(double));
        return;
    }
    while (i--) r[i]=(derivate(i))(P);
}

void Polynomial::gradientHessian(Vector P, Vector G, $
$Matrix H)
{
    unsigned dim=d->dim;
    G.setSize(dim);
    H.setSize(dim,dim);
    double *r=G, **h=H;
    unsigned i,j;

    if (d->deg==2)
    {
        double *c=d->coeff+1;
        memcpy(r,c,dim*sizeof(double));
        c+=dim;
        for (i=0; i<dim; i++)
        {
            h[i][i]=2* *(c++);
            for (j=i+1; j<dim; j++)
                h[i][j]=h[j][i]=*(c++);
        }
        if (P.equals(Vector::emptyVector)) return;
        G+=H.multiply(P);
        return;
    }

    Polynomial *tmp=new Polynomial[dim], a;
    i=dim;
    while (i--)
    {
        tmp[i]=derivate(i);
        r[i]=(tmp[i])(P);
    }

    i=dim;
    while (i--)
    {
        j=i+1;
        while (j--)
        {
            a=tmp[i].derivate(j);
            h[i][j]=h[j][i]=a(P);
        }
    }

//    _CrtCheckMemory();

    delete []tmp;
}

void Polynomial::translate(Vector translation)
{
    if (d->deg>2)
    {
        printf("Translation only for polynomial of degree $
$lower than 3.\n");
        getchar(); exit(255);
    }
    d->coeff[0]=(*this)(translation);
    if (d->deg==1) return;
    int dim=d->dim;
    Vector G(dim);
    Matrix H(dim,dim);
    gradientHessian(translation, G, H);
    memcpy(((double*)d->coeff)+1, (double*)G, dim*sizeof($
$double));
}

void Polynomial::save(char *name)
{
    FILE *f=fopen(name,"wb");
    fwrite(&d->dim, sizeof(int),1, f);
    fwrite(&d->deg, sizeof(int),1, f);
    fwrite(d->coeff, d->n*sizeof(double),1, f);
    fclose(f);
}

Polynomial::Polynomial(char *name)
{
    unsigned _dim,_deg;
    FILE *f=fopen(name,"rb");
    fread(&_dim, sizeof(int),1, f);
    fread(&_deg, sizeof(int),1, f);
    init(_dim,_deg);
    fread(d->coeff, d->n*sizeof(double),1, f);
    fclose(f);
}

#endif  /* _MPI_POLYP_H_ */
```

14.2.9 MultiInd.h

```
//
//      class Multiindex
//
#ifndef _MPI_MULTIND_H_
#define _MPI_MULTIND_H_

#include "VectorInt.h"

class MultInd;

class MultIndCache {
  public:
    MultIndCache();
    ~MultIndCache();
    MultInd *get(unsigned _dim, unsigned _deg);
  private:
    MultInd *head;
};

#ifndef __INSIDE_MULTIND_CPP__
extern MultIndCache cacheMultInd;
#endif

class MultInd {
friend class MultIndCache;
public:
    unsigned dim, deg;

    unsigned *lastChanges();
    unsigned *indexesOfCoefInLexOrder();

    MultInd(unsigned d=0);
    ~MultInd();

    void resetCounter();
    MultInd& operator++();       // prefix
    MultInd& operator++( int ) { return this->operator++()$
$; } // postfix
//    unsigned &operator[]( unsigned i) {return coeffDeg[i$
$];};
    inline operator unsigned*() const { return coeffDeg; $
$};
    MultInd& operator=( const MultInd &P );
    bool operator==( const MultInd& m );
    unsigned index() {return indexV;};
    unsigned len();

  // Print it
    void print();

private:
    MultInd( unsigned _dim, unsigned _deg );
    void fullInit();
    void standardInit();

    VectorInt lastChangesV, indexesOfCoefInLexOrderV;
    unsigned *coeffDeg, *coeffLex, indexV;

    static unsigned *buffer, maxDim;
    // to do the cache:
    MultInd *next;
};


#endif  /* _MPI_MULTIND_H_ */
```

14.2.10 MultiInd.cpp

```
//
//      Multiindex
//
#include <stdlib.h>
#include <stdio.h>
#include <memory.h>

#define __INSIDE_MULTIND_CPP__
#include "MultInd.h"
#undef __INSIDE_MULTIND_CPP__

#include "tools.h"

unsigned MultInd::maxDim;
unsigned *MultInd::buffer;
MultIndCache cacheMultInd;

MultInd& MultInd::operator=( const MultInd &p )
{
    dim=p.dim; deg=p.deg; next=NULL;
    lastChangesV=p.lastChangesV; indexesOfCoefInLexOrderV=$
$p.indexesOfCoefInLexOrderV;
    indexV=p.indexV;
    standardInit();
    if (deg==0) memcpy(coeffDeg,p.coeffDeg,dim*sizeof($
$unsigned));
    return *this;
}

void MultInd::standardInit()
{
    if (deg==0)
    {
        coeffDeg=(unsigned*)malloc(dim*sizeof(unsigned));
        coeffLex=NULL;
    } else
    {
        coeffDeg=buffer;
        coeffLex=buffer+dim;
    };
}

MultInd::MultInd( unsigned _dim, unsigned _deg):
    dim(_dim), deg(_deg), next(NULL)
{
    standardInit();
    fullInit();
    resetCounter();
}

MultInd::MultInd(unsigned d):  dim(d), deg(0), next(NULL)
{
    standardInit();
    resetCounter();
};

MultInd::~MultInd()
{
    if (deg==0) free(coeffDeg);
}

void MultInd::print()
{
        printf("[");
        if (!dim) { printf("]"); return; }

    unsigned N=dim,*up=coeffDeg;
        while (--N) printf("%i,",*(up++));
        printf("%i]",*up);
}

unsigned MultInd::len()
{
    unsigned l=0, *ccDeg=coeffDeg, j=dim;
    while (j--) l+=*(ccDeg++);
    return l;
}

bool MultInd::operator==( const MultInd& m )
{
    unsigned *p1=(*this), *p2=m, n=dim;
    while (n--)
        if (*(p1++)!=*(p2++)) return false;
    return true;
}

void MultInd::fullInit()
{
    unsigned *ccLex, *ccDeg, degree=deg, n=choose(dim+deg,$
$dim),i,k,sum, d=dim-1;
    int j;

    lastChangesV.setSize(n-1);
    indexesOfCoefInLexOrderV.setSize(n);

    memset(coeffLex+1,0,d*sizeof(int));
        *coeffLex=deg;

    for (i=0; i<n; i++)
    {
        sum=0; ccLex=coeffLex; j=dim;
        while (j--) sum+=*(ccLex++);
        if (sum) k=choose( sum+d, dim ); else k=0;

        resetCounter();
        *coeffDeg=sum;

        while(1)
        {
            ccLex=coeffLex; ccDeg=coeffDeg;
            for ( j=d; j>0 ; j--, ccLex++, ccDeg++ ) if (*$
$ccLex != *ccDeg) break;
            if (*ccLex >= *ccDeg) break;
            ++(*this); k++;
        }

        indexesOfCoefInLexOrderV[i]=k;

        if (i==n-1) break;

        // lexical order ++ :
        if (coeffLex[d])
        {
           lastChangesV[i]=d;
           coeffLex[d]--;
        } else
        for (j=d-1; j>=0; j--)
        {
                if (coeffLex[j])
                {
                  lastChangesV[i]=j;
                  sum=--coeffLex[j];
                  for (k=0; k<(unsigned)j; k++) sum+=$
$coeffLex[k];
                  coeffLex[++j]=degree-sum;
                  for (k=j+1; k<=d; k++) coeffLex[k]=0;
                  break;
                }
        }
    }
}

void MultInd::resetCounter()
{
    indexV=0;
    memset(coeffDeg,0,dim*sizeof(unsigned));
}

MultInd& MultInd::operator++()
{
        unsigned *cc = coeffDeg;
    int n=dim, pos, i;

        if (!n || !cc) return *this;

        for (pos = n-2; pos >= 0; pos--)
        {
                if (cc[pos])    // Gotcha
                {
                  cc[pos]--;
                  cc[++pos]++;
                  for (i = pos+1; i < n;i++)
                  {
                    cc[pos] += cc[i];
                    cc[i] = 0;
                  }
          indexV++;
                  return *this;
                }
        }

        (*cc)++;
        for ( i = 1; i < n; i++)
        {
                *cc += cc[i];
                cc[i] = 0;
        }

    indexV++;
    return *this;
}

unsigned *MultInd::lastChanges()
{
    if (deg==0)
    {
        printf("use MultIndCache to instanciate MultInd");
        getchar(); exit(252);
    }
    return (unsigned*)lastChangesV.d->p;
}

unsigned *MultInd::indexesOfCoefInLexOrder()
{
    if (deg==0)
    {
        printf("use MultIndCache to instanciate MultInd");
        getchar(); exit(252);
    }
    return (unsigned*)indexesOfCoefInLexOrderV.d->p;
}

MultIndCache::MultIndCache(): head(NULL)
{
    MultInd::maxDim=100;
    MultInd::buffer=(unsigned*)malloc(MultInd::maxDim*2*$
$sizeof(unsigned));
};

MultIndCache::~MultIndCache()
{
    MultInd *d=head, *d1;
    while (d)
    {
        d1=d->next;
        delete d;
        d=d1;
    }
    free(MultInd::buffer);
}

MultInd *MultIndCache::get(unsigned _dim, unsigned _deg )
{
    if (_deg==0)
    {
        printf("use normal constructor of MultiInd");
        getchar(); exit(252);
    }
    if (_dim>MultInd::maxDim)
    {
        free(MultInd::buffer);
        MultInd::maxDim=_dim;
        MultInd::buffer=(unsigned*)malloc(_dim*2*sizeof($
$unsigned));
    }
    MultInd *d=head;
    while (d)
    {
        if ((_dim==d->dim)&&(_deg==d->deg)) return d;
        d=d->next;
    }

    d=new MultInd(_dim,_deg);
    d->next=head;
    head=d;
    return d;
}
```

14.2.11 IntPoly.h

```
//
//      Multivariate Interpolating Polynomials
//      Application header

//#include <windows.h>

#include "Poly.h"
#include "Vector.h"

#ifndef _MPI_INTPOLY_H_
#define _MPI_INTPOLY_H_


class InterPolynomial : public Polynomial
{
public:

    double M;
    unsigned nPtsUsed, nUpdateOfM;

    // (*this) = sum_i newBasis[i]*NewtonCoefPoly[i]
        Polynomial *NewtonBasis;
    // double *NewtonCoefPoly;

    // data:
    Vector *NewtonPoints;

    double *NewtonCoefficient(double *);
    void ComputeNewtonBasis(double *, unsigned nPtsTotal);

/*
    InterPolynomial() : Polynomial() {}
    InterPolynomial( const Polynomial& p ) : Polynomial( p$
$ ) {};
    InterPolynomial( const InterPolynomial& p ) : $
$Polynomial( p ) {};
*/

    InterPolynomial( unsigned _deg, unsigned nPtsTotal, $
$Vector *_Pp, double *_Yp );

    int findAGoodPointToReplace(int excludeFromT,double $
$rho,
                                                        Vector$
$ pointToAdd, double *modelStep=NULL);
    void replace(int t, Vector pointToAdd, double valueF);
    int maybeAdd(Vector pointToAdd, unsigned k, double rho$
$, double valueF);

    void updateM(Vector newPoint, double valueF);
    int checkIfValidityIsInBound(Vector dd, unsigned k, $
$double bound, double rho);
    int getGoodInterPolationSites(Matrix d, int k, double $
$rho, Vector *v=NULL);
    double interpError(Vector Point);

    void translate(int k);
    void translate(Vector translation);

//    void test();
//    void check(Vector Base, double (*f)(  Vector ) );

        // allow shallow copy:
    ~InterPolynomial();
    InterPolynomial(const InterPolynomial &A);
    InterPolynomial& operator=( const InterPolynomial& A )$
$;
    InterPolynomial clone();
    void copyFrom(InterPolynomial a);
    void copyFrom(Polynomial a);
    InterPolynomial(unsigned dim, unsigned deg);

protected:
    void destroyCurrentBuffer();

};

#ifndef NOOBJECTIVEFUNCTION
#include "ObjectiveFunction.h"
Vector *GenerateData(double **valuesF, double rho,
                     Vector Base, double vBase, $
$ObjectiveFunction *of);
#endif

#endif  /* _MPI_INTPOLY_H_ */
```

14.2.12 IntPoly.cpp

```
//
//    Multivariate Interpolating Polynomials
//    Private header
//    ...
//    V 0.3
#include <stdio.h>
#include "Poly.h"
#include "Vector.h"
#include "tools.h"

#ifndef _MPI_INTPOLYP_H_
#define _MPI_INTPOLYP_H_

#include "IntPoly.h"

Vector LAGMAXModified(Vector G, Matrix H, double rho,$
$double &VMAX);

// Note:
// Vectors do come from outside. Newton Basis and $
$associated permutation
// vector are generated internally and can be deleted.

double *InterPolynomial::NewtonCoefficient(double *yy)
{
   // Initialize to local variables
   unsigned N=nPtsUsed,i;
   Polynomial *pp=NewtonBasis;
   Vector *xx=NewtonPoints;
   double *ts=(double*)calloc(N,sizeof(double)), *tt=ts;

   if (!ts)
   {
      printf("NewtonCoefficient : No mem\n");
      getchar(); exit(251);
   }

   for (i=0; i<N; i++)    // 0th difference everywhere
    *(tt++) = yy[i];

   unsigned deg=d->deg, Dim=d->dim, Nfrom, Nto, j, curDeg;
   double *ti, *tj;

   for (curDeg=0, Nfrom=0; curDeg<deg; Nfrom=Nto )
   {
      Nto=Nfrom+choose( curDeg+Dim-1,Dim-1 );
      for (ti=ts+Nto, i=Nto; i<N; i++,ti++)
      {
         for (tj=ts+Nfrom, j=Nfrom; j<Nto; j++, tj++)
            *ti -= *tj ? // Evaluation takes time
                         *tj * (pp[j])( xx[i] ) : 0;
      }
      curDeg++;
   }
   return ts;
}

void InterPolynomial::ComputeNewtonBasis(double *yy, $
$unsigned nPtsTotal)
{
   const double eps  =  1e-6;
   const double good =  1 ;
   unsigned dim=d->dim,i;

   Vector *xx=NewtonPoints, xtemp;
   Polynomial *pp= new Polynomial[nPtsUsed], *qq=pp;
   NewtonBasis=pp;

   if (!pp)
   {
      printf("ComputeNewtonBasis( ... ) : Alloc for $
$polynomials failed\n");
      getchar(); exit(251);
   }

   MultInd I(dim);
   for (i=0; i<nPtsUsed; i++)
   {
      *(qq++)=Polynomial(I);
      I++;
   }

   unsigned k, kmax;
   double v, vmax, vabs;
#ifdef VERBOSE
   printf("Constructing first quadratic ... (N=%i)\n",$
$nPtsUsed);
#endif
   for (i=0; i<nPtsUsed; i++)
   {
#ifdef VERBOSE
      printf(".");
#endif
```

```
      vmax = vabs = 0;
      kmax = i;
      if (i==0)
      {
          // to be sure point 0 is always taken:
          vmax=(pp[0])( xx[0] );
      } else
      for (k=i; k<nPtsTotal; k++)    // Pivoting
      {
          v=(pp[i])( xx[k] );
          if (fabs(v) > vabs)
          {
              vmax = v;
              vabs = abs(v);
              kmax = k;
          }
          if (fabs(v) > good ) break;
      }

      // Now, check ...
      if (fabs(vmax) < eps)
      {
          printf("Cannot construct newton basis");
          getchar(); exit(251);
      }

      // exchange component i and k of NewtonPoints
      // fast because of shallow copy
      xtemp   =xx[kmax];
      xx[kmax]=xx[i];
      xx[i]   =xtemp;

      // exchange component i and k of newtonData
      v       =yy[kmax];
      yy[kmax]=yy[i];
      yy[i]   =v;

      pp[i]/=vmax;
      for (k=0;  k<i;        k++) pp[k] -= (pp[k])( xx[i] $
$) * pp[i];
      for (k=i+1; k<nPtsUsed; k++) pp[k] -= (pp[k])( xx[i]$
$ ) * pp[i];

      // Next polynomial, break if necessary
   }
#ifdef VERBOSE
   printf("\n");
#endif
}

InterPolynomial::InterPolynomial( unsigned _deg, unsigned $
$_nPtsTotal, Vector *_Pp,
    double *_Yp ) : Polynomial(_Pp->sz(), _deg), M(0.0), $
$nUpdateOfM(0),
      NewtonPoints(_Pp)
{
    nPtsUsed=choose( _deg+d->dim,d->dim );
    if (!_Pp)
    {
        printf( "InterPolynomial::InterPolynomial( double $
$*) : No Vectors\n");
        getchar(); exit(-1);
    }
    if (!_Yp)
    {
        printf( "InterPolynomial::InterPolynomial( double $
$*) : No data\n");
        getchar(); exit(-1);
    }
    if (_nPtsTotal<nPtsUsed)
    {
        printf( "InterPolynomial::InterPolynomial( double $
$*) : Not enough data\n");
        getchar(); exit(-1);
    }

    // Generate basis
    ComputeNewtonBasis(_Yp, _nPtsTotal);
//    test();

    // Compute Interpolant
//    double *NewtonCoefPoly=NewtonCoefficient(_Yp);
    double *NewtonCoefPoly=_Yp;

    double *coc= NewtonCoefPoly+nPtsUsed-1;
    Polynomial *ppc= NewtonBasis+nPtsUsed-1;
    this->copyFrom((Polynomial)(*coc * *ppc));         //$
$take highest degree

    int i=nPtsUsed-1;
    if (i)
      while(i--)
        (*this) += *(--coc) * *(--ppc);
```

```
                // No reallocation here because of the $
$order of
                // the summation
    //free(NewtonCoefPoly);
}

#ifndef NOOBJECTIVEFUNCTION

void calculateNParallelJob(int n,double *vf,Vector *cp,$
$ObjectiveFunction *of);

Vector *GenerateData(double **valuesF, double rho,
                     Vector Base, double vBase, $
$ObjectiveFunction *of )
// generate points to allow start of fitting a polynomial $
$of second degree
// around point Base
{
    int j,k,dim=Base.sz(), N=(dim+1)*(dim+2)/2;
    double *vf=(double*)malloc(N*sizeof(double)); // value$
$ objective function
    *valuesF=vf;
    Vector *ap=new Vector[ N-1 ], *cp=ap, cur; // ap: $
$allPoints
                                               // cp: $
$current Point
    double *sigma=(double*)malloc(dim*sizeof(double));

    for (j=0; j<dim;j++)
    {
        cur=Base.clone();
        cur[j]+=rho;

        *(cp++)=cur;
    }

        calculateNParallelJob(dim,vf,ap,of);

    for (j=0; j<dim; j++)
        {
        cur=Base.clone();
        if (*(vf++)<vBase) { cur[j]+=2*rho; sigma[j]=rho; $
$}
        else { cur[j]-=rho; sigma[j]=-rho; }
        *(cp++)=cur;
        }

    for (j=0; j<dim; j++)
    {
        for (k=0; k<j; k++)
        {
            cur=Base.clone();
            cur[j]+=sigma[j];
            cur[k]+=sigma[k];
            *(cp++)=cur;
        }
    }

    free(sigma);

        // parallelize here !
        calculateNParallelJob(N-dim-1,vf,ap+dim,of);

        return ap;
}

#endif

void InterPolynomial::updateM(Vector newPoint, double $
$valueF)
{
    //not tested
    unsigned i=nPtsUsed;
    double sum=0,a;
    Polynomial *pp=NewtonBasis;
    Vector *xx=NewtonPoints;

    while (i--)
    {
        a=newPoint.euclidianDistance(xx[i]);
        sum+=abs(pp[i]( newPoint ))*a*a*a;
    }
    M=mmax(M, abs((*this)(newPoint)-valueF)/sum);
    nUpdateOfM++;
}

double InterPolynomial::interpError(Vector Point)
{
    unsigned i=nPtsUsed;
    double sum=0,a;
    Polynomial *pp=NewtonBasis;
    Vector *xx=NewtonPoints;
```

```
    while (i--)
    {
        a=Point.euclidianDistance(xx[i]);
        sum+=abs(pp[i]( Point ))*a*a*a;
    }
    return M*sum;
}

int InterPolynomial::findAGoodPointToReplace(int $
$excludeFromT,
                                          double rho, Vector $
$pointToAdd, double *maxd)
{
    //not tested

    // excludeFromT is set to k if not sucess from optim $
$and we want
    //        to be sure that we keep the best point
    // excludeFromT is set to -1 if we can replace the $
$point x_k by
    //        pointToAdd(=x_k+d) because of the success of$
$ optim.

    // choosen t: the index of the point inside the $
$newtonPoints
    //            which will be replaced.

    Vector *xx=NewtonPoints;
    Vector XkHat;
    if (excludeFromT>=0) XkHat=xx[excludeFromT];
    else XkHat=pointToAdd;

    int t=-1, i, N=nPtsUsed;
    double a, aa, maxa=-1.0, maxdd=0;
    Polynomial *pp=NewtonBasis;

  //  if (excludeFromT>=0) maxa=1.0;

    for (i=0; i<N; i++)
    {
        if (i==excludeFromT) continue;
        aa=XkHat.euclidianDistance(xx[i]);
        if (aa==0.0) return -1;
        a=aa/rho;
        // because of the next line, rho is important:
        a=mmax(a*a*a,1.0);
        a*=abs(pp[i] (pointToAdd));

        if (a>maxa)
        {
            t=i; maxa=a; maxdd=aa;
        }
    }
    if (maxd) *maxd=maxdd;
    return t;
}
/*
void InterPolynomial::check(Vector Base, double (*f)(  $
$Vector ) )
{
    int i,n=sz();
    double r, bound;

    for (i=0; i<n; i++)
    {
        r=(*f)(NewtonPoints[i]+Base);
        bound=(*this)(NewtonPoints[i]);
        if ((abs(bound-r)>1e-15)&&(abs(bound-r)>1e-3*abs($
$bound)))
        {
            printf("error\n");
            test();
        }
//                      for (j=0; j<n; j++)
//                          r=poly.NewtonBasis[j](poly.$
$NewtonPoints[i]);
    }
}

void InterPolynomial::test()
{
    unsigned i,j,n=d->n; Matrix M(n,n); double **m=M;
    for (i=0; i<n; i++)
        for (j=0; j<n; j++)
            m[i][j]=NewtonBasis[i](NewtonPoints[j]);
    M.print();
};
*/
void InterPolynomial::replace(int t, Vector pointToAdd, $
$double valueF)
{
    //not tested
    updateM(pointToAdd, valueF);
    if (t<0) return;

    Vector *xx=NewtonPoints;
    Polynomial *pp=NewtonBasis, t1;
    int i, N=nPtsUsed;
    double t2=(pp[t]( pointToAdd ));

    if (t2==0) return;

    t1=pp[t]/=t2;

    for (i=0; i<t; i++)   pp[i]-= pp[i]( pointToAdd )*t1;
    for (i=t+1; i<N; i++) pp[i]-= pp[i]( pointToAdd )*t1;
    xx[t].copyFrom(pointToAdd);

    // update the coefficents of general poly.

    valueF-=(*this)(pointToAdd);
    if (abs(valueF)>1e-11) (*this)+=valueF*pp[t];

//    test();
}

int InterPolynomial::maybeAdd(Vector pointToAdd, unsigned $
$k, double rho, double valueF)
// input: pointToAdd, k, rho, valueF
// output: updated polynomial
{

    unsigned i,N=nPtsUsed;
    int j;
    Vector *xx=NewtonPoints, xk=xx[k];
    double distMax=-1.0,dd;
 /*
    Polynomial *pp=NewtonBasis;
    Vector *xx=NewtonPoints, xk=xx[k],vd;
    Matrix H(n,n);
    Vector GXk(n); //,D(n);
*/
    // find worst point/newton poly

    for (i=0; i<N; i++)
    {
        dd=xk.euclidianDistance(xx[i]);
        if (dd>distMax) { j=i; distMax=dd; };
    }
    dd=xk.euclidianDistance(pointToAdd);

    // no tested:

    if (abs(NewtonBasis[j](pointToAdd))*distMax*distMax*$
$distMax/(dd*dd*dd)>1.0)
    {
        printf("good point found.\n");
        replace(j, pointToAdd, valueF);
        return 1;
    }
    return 0;
}

int InterPolynomial::checkIfValidityIsInBound(Vector ddv, $
$unsigned k, double bound, double rho)
// input: k,bound,rho
// output: j,ddv
{

    // check validity around x_k
    // bound is epsilon in the paper
    // return index of the worst point of J
    // if (j==-1) then everything OK : next : trust region$
$ step
    // else model step: replace x_j by x_k+d where d
    // is calculated with LAGMAX

    unsigned i,N=nPtsUsed, n=dim();
    int j;
    Polynomial *pp=NewtonBasis;
    Vector *xx=NewtonPoints, xk=xx[k],vd;
    Vector Distance(N);
    double *dist=Distance, *dd=dist, distMax, vmax, tmp;
    Matrix H(n,n);
    Vector GXk(n); //,D(n);

    for (i=0; i<N; i++) *(dd++)=xk.euclidianDistance(xx[i$
$]);

    while (true)
    {
        dd=dist; j=-1; distMax=2*rho;
        for (i=0; i<N; i++)
        {
            if (*dd>distMax) { j=i; distMax=*dd; };
            dd++;
        }
        if (j<0) return -1;

        // to prevent to choose the same point once again:
        dist[j]=0;

        pp[j].gradientHessian(xk,GXk,H);
//        d=H.multiply(xk);
//        d.add(G);

        tmp=M*distMax*distMax*distMax;

        if (tmp*rho*(GXk.euclidianNorm()+0.5*rho*H.$
$frobeniusNorm())>=bound)
        {
/*                      vd=L2NormMinimizer(pp[j], xk, rho)$
$;
            vd+=xk;
            vmax=abs(pp[j](vd));

                        Vector vd2=L2NormMinimizer(pp[j], $
$xk, rho);
            vd2+=xk;
            double vmax2=abs(pp[j](vd));

                        if (vmax<vmax2) { vmax=vmax2; vd=$
$vd2; }
*/
                        vd=LAGMAXModified(GXk,H,rho,vmax);
//            tmp=vd.euclidianNorm();
                vd+=xk;
                        vmax=abs(pp[j](vd));

            if (tmp*vmax>=bound) break;
        }
    }
    if (j>=0) ddv.copyFrom(vd);
    return j;
}

int InterPolynomial::getGoodInterPolationSites(Matrix d, $
$int k, double rho, Vector *v)
// input: k,rho
// output: d,r
{
    //not tested

    unsigned i, N=nPtsUsed, n=dim();
    int ii, j,r=0;
    Polynomial *pp=NewtonBasis;
    Vector *xx=NewtonPoints, xk,vd;
    Vector Distance(N);
    double *dist=Distance, *dd=dist, distMax, vmax;
    Matrix H(n,n);
    Vector GXk(n);
    if (k>=0) xk=xx[k]; else xk=*v;

    for (i=0; i<N; i++) *(dd++)=xk.euclidianDistance(xx[i$
$]);

    for (ii=0; ii<d.nLine(); ii++)
    {
        dd=dist; j=-1;
        distMax=-1.0;
        for (i=0; i<N; i++)
        {
            if (*dd>distMax) { j=i; distMax=*dd; };
            dd++;
        }
        // to prevent to choose the same point once again:
        dist[j]=-1.0;

        if (distMax>2*rho) r++;
        pp[j].gradientHessian(xk,GXk,H);
                vd=LAGMAXModified(GXk,H,rho,vmax);
            vd+=xk;

        d.setLine(ii,vd);
    }
    return r;
}

void InterPolynomial::translate(Vector translation)
{
    Polynomial::translate(translation);
    int i=nPtsUsed;
    while (i--) NewtonBasis[i].translate(translation);
    i=nPtsUsed;
    while (i--) if (NewtonPoints[i]==translation) $
$NewtonPoints[i].zero();
                else NewtonPoints[i]-=translation;
}
```

```
// to allow shallow copy:

void InterPolynomial::destroyCurrentBuffer()
{
    if (!d) return;
        if (d->ref_count==1)
    {
        delete[] NewtonBasis;
        if (NewtonPoints) delete[] NewtonPoints;
//          free(ValuesF);
    }
}

InterPolynomial::~InterPolynomial()
{
    destroyCurrentBuffer();
}

InterPolynomial::InterPolynomial(const InterPolynomial &A)
{
    // shallow copy for inter poly.
    d=A.d;
    NewtonBasis=A.NewtonBasis;
    NewtonPoints=A.NewtonPoints;
//    ValuesF=A.ValuesF;
        M=A.M;
    nPtsUsed=A.nPtsUsed;
    nUpdateOfM=A.nUpdateOfM;
        (d->ref_count)++;

}

InterPolynomial& InterPolynomial::operator=( const $
$InterPolynomial& A )
{
    // shallow copy
    if (this != &A)
        {
        destroyCurrentBuffer();

        d=A.d;
        NewtonBasis=A.NewtonBasis;
        NewtonPoints=A.NewtonPoints;
//          ValuesF=A.ValuesF;
            M=A.M;
        nPtsUsed=A.nPtsUsed;
        nUpdateOfM=A.nUpdateOfM;
                (d->ref_count) ++ ;
        }
        return *this;
}

InterPolynomial::InterPolynomial(unsigned _dim, unsigned $
$_deg): Polynomial(_dim, _deg), M(0.0), nUpdateOfM(0)
{
    nPtsUsed=choose( _deg+_dim,_dim );
    NewtonBasis= new Polynomial[nPtsUsed];
        NewtonPoints= new Vector[nPtsUsed];
}

InterPolynomial InterPolynomial::clone()
{
    // a deep copy
    InterPolynomial m(d->dim, d->deg);
    m.copyFrom(*this);
    return m;
}

void InterPolynomial::copyFrom(InterPolynomial m)
{
    if (m.d->dim!=d->dim)
    {
        printf("poly: copyFrom: dim do not agree\n");
        getchar(); exit(254);
    }
    if (m.d->deg!=d->deg)
    {
        printf("poly: copyFrom: degree do not agree\n");
        getchar(); exit(254);
    }
    Polynomial::copyFrom(m);
        M=m.M;
//    nPtsUsed=m.nPtsUsed; // not usefull because dim and $
$degree already agree.
    nUpdateOfM=m.nUpdateOfM;
// ValuesF

        int i=nPtsUsed;
        while (i--)
        {
//              NewtonBasis[i]=m.NewtonBasis[i];
//              NewtonPoints[i]=m.NewtonPoints[i];
                NewtonBasis[i]=m.NewtonBasis[i].clone();
                NewtonPoints[i]=m.NewtonPoints[i].clone();
        }
}

void InterPolynomial::copyFrom(Polynomial m)
{
    Polynomial::copyFrom(m);
}

#endif      /* _MPI_INTPOLYP_H_ */
```

14.2.13 KeepBests.h

```
#ifndef __INCLUDE_KEEPBEST__
#define __INCLUDE_KEEPBEST__

#define INF 1.7E+308

typedef struct cell_tag
    {
        double K;
        double value;
        double *optValue;
        struct cell_tag *prev;
    } cell;

class KeepBests
{
public:
    KeepBests(int n);
    KeepBests(int n, int optionalN);
    void setOptionalN(int optinalN);
    ~KeepBests();
    void reset();
    void add(double key, double value);
    void add(double key, double value, double $
$optionalValue);
    void add(double key, double value, double *$
$optionalValue);
    void add(double key, double value, double *$
$optionalValue, int nn);
    double getKey(int i);
    double getValue(int i);
    double getOptValue(int i, int n);
    double* getOptValue(int i);
    int sz() {return n;};
private:
    void init();
    cell *ctable,*end,*_local_getOptValueC;
    int n,optionalN,_local_getOptValueI;
};

#endif
```

14.2.14 KeepBests.cpp

```
#include "KeepBests.h"
#include <stdlib.h>
#include <string.h>

KeepBests::KeepBests(int _n): n(_n), optionalN(0)
{
    init();
}

KeepBests::KeepBests(int _n, int _optionalN): n(_n), $
$optionalN(_optionalN)
```

```
{
    init();
}

void KeepBests::init()
{
    int i;
    double *t;
    ctable=(cell*)malloc(n*sizeof(cell));
    if (optionalN) t=(double*)malloc(optionalN*n*sizeof($
$double));
    for (i=0; i<n; i++)
    {
        if (optionalN)
        {
            ctable[i].optValue=t;
            t+=optionalN;
        }
        ctable[i].K=INF;
        ctable[i].prev=ctable+(i-1);
    }
    ctable[0].prev=NULL;
    end=ctable+(n-1);
    _local_getOptValueI=-1;
}

void KeepBests::setOptionalN(int _optionalN)
{
    int i;
    double *t;
    if (optionalN) t=(double*)realloc(ctable[0].optValue,$
$_optionalN*n*sizeof(double));
    else t=(double*)malloc(_optionalN*n*sizeof(double));
    for (i=0; i<n; i++)
    {
        ctable[i].optValue=t;
        t+=_optionalN;
    }
    optionalN=_optionalN;
}

KeepBests::~KeepBests()
{
    if (optionalN) free(ctable[0].optValue);
    free(ctable);
}

void KeepBests::reset()
{
    int i;
    for (i=0; i<n; i++) ctable[i].K=INF;
//    if (optionalN) memset(ctable[0].optValue,0,optionalN$
$*n*sizeof(double));
}

void KeepBests::add(double key, double value)
{
    add(key,value,NULL,0);
}
void KeepBests::add(double key, double value, double $
$optionalValue)
{
    add(key,value,&optionalValue,1);
}

void KeepBests::add(double key, double value, double *$
$optionalValue)
{
    add(key,value,optionalValue,optionalN);
}

void KeepBests::add(double key, double value, double *$
$optionalValue, int nn)
{
    cell *t=end, *prev, *t_next=NULL;
    while ((t)&&(t->K>key)) { t_next=t; t=t->prev; };
    if (t_next)
    {
        if (t_next==end)
        {
            end->K=key;
            end->value=value;
            if ((optionalN)&&(optionalValue))
            {
                memcpy(end->optValue, optionalValue, nn*$
$sizeof(double));
                if (optionalN-nn>0)
                    memset(end->optValue+nn,0,(optionalN-$
$nn)*sizeof(double));
            }
        } else
        {
            prev=end->prev;
            end->prev=t;
            t_next->prev=end;

            end->K=key;
            end->value=value;
            if ((optionalN)&&(optionalValue))
            {
                memcpy(end->optValue, optionalValue, nn*$
$sizeof(double));
                if (optionalN-nn)
                    memset(end->optValue+nn,0,(optionalN-$
$nn)*sizeof(double));
            }
            end=prev;
        };
    };
}

double KeepBests::getValue(int i)
{
    cell *t=end;
    i=n-i-1;
    while (i) { t=t->prev; i--; }
    return t->value;
}

double KeepBests::getKey(int i)
{
    cell *t=end;
    i=n-i-1;
    while (i) { t=t->prev; i--; }
    return t->K;
}

double KeepBests::getOptValue(int i, int no)
{
    if (i==_local_getOptValueI) return _local_getOptValueC$
$->optValue[no];
    _local_getOptValueI=i;
    cell *t=end;
    i=n-i-1;
    while (i) { t=t->prev; i--; }
    _local_getOptValueC=t;
    return t->optValue[no];
}

double *KeepBests::getOptValue(int i)
{
    cell *t=end;
    i=n-i-1;
    while (i) { t=t->prev; i--; }
    return t->optValue;
}
```

14.2.15 ObjectiveFunction.h

```
#ifndef OBJECTIVEFUNCTION_INCLUDE
#define OBJECTIVEFUNCTION_INCLUDE

#include <stdio.h>
#include "Vector.h"
#include "Matrix.h"

#define INF 1.7E+308

class ObjectiveFunction
{
  public:
    char name[9];
    Vector xStart, xBest, xOptimal;
    // xOptimal is the theoretical exact solution of the $
$optimization problem.
    // xBest is the solution given by the optimization $
$algorithm.
    double valueOptimal, valueBest, noiseAbsolute, $
$noiseRelative, objectiveConst;
    // valueOptimal is the value of the obj.funct. at the $
$theoretical exact solution of the optimization problem.
    // valueBest is the value of the obj.funct. at the $
$solution given by the optimization algorithm.
    // objectiveConst is use inside method "printStats" to$
$ give correction evaluation of the obj.funct.
    Matrix data;
    int nfe,nfe2,t,nNLConstraints, isConstrained;
    // nfe: number of function evalution
    // t: type of the OF
    // nNLConstraints: number of non-linear constraints

    // CONSTRAINTS:
    // for lower/upper bounds (box constraints)
    Vector bl, bu;

        // for linear constraints Ax>=b
        Matrix A;
        Vector b;

    // for non-linear constraints
    virtual double evalNLConstraint(int j, Vector v, int *$
$nerror=NULL)=0;
    virtual Vector evalGradNLConstraint(int j, Vector v, $
$int *nerror=NULL);
    virtual void evalGradNLConstraint(int j, Vector v, $
$Vector result, int *nerror=NULL)=0;

    // tolerances for constraints
    double tolRelFeasibilityForNLC, tolNLC;
    double tolRelFeasibilityForLC, tolLC;

    ObjectiveFunction() : valueOptimal(INF), valueBest(INF$
$), noiseAbsolute(0.0),
           noiseRelative(0.0), objectiveConst(0.0), nfe(0)$
$, nfe2(0), nNLConstraints(0),
           isConstrained(1), tolRelFeasibilityForNLC(1e-9)$
$, tolNLC(1e-6),
           tolRelFeasibilityForLC(1e-6), tolLC(1e-8),
           saveFileName(NULL), dfold(INF), maxNormLC(0.0),$
$ maxNormNLC(0.0) { };
    virtual ~ObjectiveFunction(){ if (saveFileName) free($
$saveFileName); };
    virtual double eval(Vector v, int *nerror=NULL)=0;
    int dim();
    void initDataFromXStart();
    virtual void saveValue(Vector tmp,double valueOF);
    virtual void printStats(char cc=1);
    virtual void finalize(){};
    void setName(char *s);
    void setSaveFile(char *b=NULL);
    void updateCounter(double df, Vector vX);
    char isFeasible(Vector vx, double *d=NULL);
    void initBounds();
    void endInit();
    void initTolLC(Vector vX);
    void initTolNLC(Vector c, double delta);
  private:
    char *saveFileName;
    double dfold, dfref, maxNormLC, maxNormNLC;
};

class UnconstrainedObjectiveFunction : public $
$ObjectiveFunction
{
  public:
    UnconstrainedObjectiveFunction(): ObjectiveFunction(){$
$ isConstrained=0; }
    ~UnconstrainedObjectiveFunction() {};

    virtual double evalNLConstraint(int j, Vector v, int *$
$nerror=NULL){ return 0; };
    virtual Vector evalGradNLConstraint(int j, Vector v, $
$int *nerror=NULL){ return Vector::emptyVector; };
    virtual void evalGradNLConstraint(int j, Vector v, $
$Vector result, int *nerror=NULL) { result=Vector::$
$emptyVector; };
};

class FletcherTest: public ObjectiveFunction
{
    // practical method of optimization
    // page 199 equation 9.1.15
    // page 142 figure 7.1.3
  public:
    FletcherTest(int _t);
    ~FletcherTest(){};

    double eval(Vector v, int *nerror=NULL);
    virtual double evalNLConstraint(int j, Vector v, int *$
$nerror=NULL);
    virtual void evalGradNLConstraint(int j, Vector v, $
$Vector result, int *nerror=NULL);
};

class FletcherTest2: public ObjectiveFunction
{
    // practical method of optimization
    // page 199 equation 9.1.15
    // page 142 figure 7.1.3
  public:
    FletcherTest2(int _t);
    ~FletcherTest2(){};
```

```
    double eval(Vector v, int *nerror=NULL);
    virtual double evalNLConstraint(int j, Vector v, int *$
$nerror=NULL){return 0.0;};
    virtual void evalGradNLConstraint(int j, Vector v, $
$Vector result, int *nerror=NULL){};
};

class SuperSimpleConstrainedObjectiveFunction: public $
$ObjectiveFunction
{
  public:
    SuperSimpleConstrainedObjectiveFunction(int _t);
    ~SuperSimpleConstrainedObjectiveFunction(){};

    double eval(Vector v, int *nerror=NULL);
    virtual double evalNLConstraint(int j, Vector v, int *$
$nerror=NULL){ return 0; };
    virtual void evalGradNLConstraint(int j, Vector v, $
$Vector result, int *nerror=NULL) {};
};

class Rosenbrock : public UnconstrainedObjectiveFunction
{
  public:
    Rosenbrock(int _t);
    ~Rosenbrock(){};
    double eval(Vector v, int *nerror=NULL);
};

class NoisyRosenbrock : public $
$UnconstrainedObjectiveFunction
{
  public:
    NoisyRosenbrock(int _t);
    ~NoisyRosenbrock(){};
    double eval(Vector v, int *nerror=NULL);
};

class NoisyQuadratic : public $
$UnconstrainedObjectiveFunction
{
  public:
    NoisyQuadratic(int _t);
    ~NoisyQuadratic(){};
    double eval(Vector v, int *nerror=NULL);
  private:
    int n;
};

class BADScaleRosenbrock : public $
$UnconstrainedObjectiveFunction
{
  public:
    BADScaleRosenbrock(int _t);
    ~BADScaleRosenbrock(){};
    double eval(Vector v, int *nerror=NULL);
};

class CorrectScaleOF: public ObjectiveFunction
{
  public:
    CorrectScaleOF(int _t, ObjectiveFunction *_of, Vector $
$_rescaling);
    CorrectScaleOF(int _t, ObjectiveFunction *_of);
    ~CorrectScaleOF(){};
    double eval(Vector v, int *nerror=NULL);
    virtual double evalNLConstraint(int j, Vector v, int *$
$nerror=NULL);
    virtual void evalGradNLConstraint(int j, Vector v, $
$Vector result, int *nerror=NULL);
    virtual void finalize();
  private:
    void init();
    ObjectiveFunction *of;
    Vector rescaling, xTemp;
};

class RandomOF: public UnconstrainedObjectiveFunction
{
  public:
    RandomOF(int _t,int n);
    RandomOF(int _t,char *);
    ~RandomOF(){};
    double eval(Vector v, int *nerror=NULL);
    double ff(Vector v);
    void save(char *);

  private:
    Vector A;
    Matrix S,C;
    void alloc(int n);
};
```

```
ObjectiveFunction *getObjectiveFunction(int i, double *rho$
$=NULL);
```

```
#endif
```

14.2.16 ObjectiveFunction.cpp

```
#ifdef WIN32
#include <windows.h>
#else
#include <unistd.h>
#endif

#include <string.h>
#include "ObjectiveFunction.h"
#include "tools.h"

char ObjectiveFunction::isFeasible(Vector vx, double *d)
{
    if (!isConstrained) return 1;
    int i, dim=vx.sz();
    char feasible=1;
    double *bbl=bl, *bbu=bu, *x=vx, t;

    if (d) *d=0.0;
    initTolLC(vx);

    for (i=0; i<dim; i++)
        if ((t=bbl[i]-x[i])>tolLC)
            { if (d) *d=mmax(*d,t); else return 0; $
$feasible=0; }

    for (i=0; i<dim; i++)
        if ((t=x[i]-bbu[i])>tolLC)
            { if (d) *d=mmax(*d,t); else return 0; $
$feasible=0; }

    for (i=0; i<A.nLine(); i++)
        if ((t=b[i]-A.scalarProduct(i,vx))>tolLC)
            { if (d) *d=mmax(*d,t); else return 0; $
$feasible=0; }

    for (i=0; i<nNLConstraints; i++)
        if ((t=-evalNLConstraint(i,vx))>tolNLC )
            { if (d) *d=mmax(*d,t); else return 0; $
$feasible=0; }

//    printf("");
    return feasible;
}

void ObjectiveFunction::endInit()
 // init linear tolerances and init variable "$
$isConstrained"
{
    int i,mdim=dim();
    double *bbl=bl, *bbu=bu;

    isConstrained=0;
    for (i=0; i<mdim; i++)
    {
        if (bbl[i]>-INF) {isConstrained=1; maxNormLC=mmax($
$maxNormLC, abs(bbl[i])); }
        if (bbu[i]< INF) {isConstrained=1; maxNormLC=mmax($
$maxNormLC, abs(bbu[i])); }
    }
    if (b.sz()) { isConstrained=1; maxNormLC=mmax($
$maxNormLC,b.LnftyNorm()); }
    tolLC=(1.0+maxNormLC)*tolRelFeasibilityForLC*(mdim*2+A$
$.nLine());

    if (nNLConstraints) isConstrained=1;
}

void ObjectiveFunction::initTolLC(Vector vX)
{
    int i;
    double *ofb=b;

    for (i=0; i<A.nLine(); i++)
        maxNormLC=mmax(maxNormLC, abs(ofb[i]-A.$
$scalarProduct(i,vX)));

    tolLC=(1.0+maxNormLC)*tolRelFeasibilityForLC*(dim()*2+$
$A.nLine());
}

void ObjectiveFunction::initTolNLC(Vector c, double delta)
{
    int i;

    for (i=0; i<nNLConstraints; i++) maxNormNLC=mmax($
$maxNormNLC,abs(c[i]));
    if (delta<INF) maxNormNLC=mmax(maxNormNLC,delta*delta)$
$;

    tolNLC=(1.0+maxNormNLC)*tolRelFeasibilityForNLC*$
$nNLConstraints;
}

void ObjectiveFunction::updateCounter(double df, Vector vX$
$)
{
    nfe++;
    if (dfold==INF) { dfref=(1+abs(df))*1e-8; dfold=df; $
$nfe2=nfe; return; }
    if (dfold-df<dfref) return;
    if (!isFeasible(vX)) return;
    nfe2=nfe;  dfold=df;
}

void ObjectiveFunction::setSaveFile(char *s)
{
    char buffer[300];
    if (saveFileName) free(saveFileName);
    if (s==NULL)
    {
        strcpy(buffer,name); strcat(buffer,".dat"); s=$
$buffer;
    }
    saveFileName=(char*)malloc(strlen(s)+1);
    strcpy(saveFileName,s);
}

void ObjectiveFunction::setName(char *s)
{
    char *p=s+strlen(s)-1;
    while ((*p!='.')&&(p!=s)) p--;
    if (p==s) { strncpy(name,s, 8); name[8]=0; return; }
    *p='\0';
    while ((*p!='\\')&&(*p!='/')&&(p!=s)) p--;
    if (p==s) { strncpy(name,s, 8); name[8]=0; return; }
    p++;
    strncpy(name,p, 8); name[8]=0;
}

void ObjectiveFunction::printStats(char cc)
{
    printf("\n\nProblem Name: %s\n",name);
    printf("Dimension of the search space: %i\n",dim());
    printf("best (lowest) value found: %e\n", valueBest+$
$objectiveConst);
    printf("Number of funtion Evalution: %i (%i)\n",nfe,$
$nfe2);
    if (xOptimal.sz())
    {
        printf("Lnfty distance to the optimum: %e\n", xBest$
$.LnftyDistance(xOptimal));
   //   printf("Euclidian distance to the optimum: %e\n", $
$xBest.euclidianDistance(xOptimal));
    }
    int idim=dim(),j=0;
    if (idim<20)
    {
        double *dd=xBest;
        printf("Solution Vector is : \n[%e",dd[0]);
        for (j=1; j<idim; j++) printf(", %e",dd[j]);
        printf("]\n"); j=0;
    }
    if ((cc==0)||(!isConstrained)) return;
    double *dbl=bl,*dbu=bu;
    while (idim--)
    {
        if (*(dbl++)>-INF) j++;
        if (*(dbu++)< INF) j++;
    }
    printf("number of         box constraints:%i\n"
           "number of      linear constraints:%i\n"
```

```
            "number of non-linear constraints:%i\n",j,A.$
$nLine(),nNLConstraints);

}

void ObjectiveFunction::saveValue(Vector tmp,double $
$valueOF)
{
    int nl=data.nLine(), dim=tmp.sz();
    if (nl==0) data.setSize(1,dim+1); else data.extendLine$
$();
    data.setLine(nl,tmp,dim);
    ((double**)data)[nl][dim]=valueOF;
    if (saveFileName) data.updateSave(saveFileName);
}

int ObjectiveFunction::dim()
{
    int n=xStart.sz();
    if (n>1) return n;
    return data.nColumn()-1;
}

void ObjectiveFunction::initDataFromXStart()
{
    if (data.nLine()>0) return;
    saveValue(xStart, eval(xStart));
}

void RandomOF::alloc(int n)
{
    xOptimal.setSize(n);
    xStart.setSize(n);
    A.setSize(n);
    S.setSize(n,n);
    C.setSize(n,n);
    strcpy(name,"RandomOF");
}

RandomOF::RandomOF(int _t,int n)
{
    t=_t;
    alloc(n);
    double *xo=xOptimal, *xs=xStart, *a=A, **s=S, **c=C, $
$sum;
    int i,j;
    initRandom();
    for (i=0; i<n ; i++)
    {
        xo[i]=(rand1()*2-1)*PI;
        xs[i]=xo[i]+(rand1()*0.2-0.1)*PI;
        for (j=0; j<n; j++)
        {
            s[i][j]=rand1()*200-100;
            c[i][j]=rand1()*200-100;
        }
    }

    for (i=0; i<n; i++)
    {
        sum=0;
        for (j=0; j<n; j++) sum+=s[i][j]*sin(xo[j])+c[i][j$
$]*cos(xo[j]);
        a[i]=sum;
    }

    valueOptimal=0.0;
}

void RandomOF::save(char *filename)
{
    int n=A.sz();

    FILE *f=fopen(filename,"wb");
    fwrite(&n, sizeof(int),1, f);
    fwrite((double*)A, n*sizeof(double),1, f);
    fwrite((double*)xOptimal, n*sizeof(double),1, f);
    fwrite((double*)xStart, n*sizeof(double),1, f);
    fwrite(*S, n*n*sizeof(double),1, f);
    fwrite(*C, n*n*sizeof(double),1, f);
    fclose(f);
}

RandomOF::RandomOF(int _t,char *filename)
{
    t=_t;
    int n;

    FILE *f=fopen(filename,"rb");
    fread(&n, sizeof(int),1, f);

    alloc(n);
    fread((double*)A, n*sizeof(double),1, f);
    fread((double*)xOptimal, n*sizeof(double),1, f);
    fread((double*)xStart, n*sizeof(double),1, f);
    fread(*S, n*n*sizeof(double),1, f);
    fread(*C, n*n*sizeof(double),1, f);
    fclose(f);
}

double RandomOF::eval(Vector X, int *nerror)
{
    double *x=X, *a=A, **s=S, **c=C, sum, r=0;
    int i,j,n=X.sz();

    for (i=0; i<n; i++)
    {
        sum=0;
        for (j=0; j<n; j++) sum+=s[i][j]*sin(x[j])+c[i][j$
$]*cos(x[j]);
        r+=sqr(a[i]-sum);
    }
#ifdef WIN32
 //   Sleep(1000); // 30 seconds sleep
#else
 //   sleep(1);
#endif

    updateCounter(r,X);
    return r;

    return r;
}

double RandomOF::ff(Vector X) // fast eval
{
    double *x=X, *a=A, **s=S, **c=C, sum, r=0;
    int i,j,n=X.sz();

    for (i=0; i<n; i++)
    {
        sum=0;
        for (j=0; j<n; j++) sum+=s[i][j]*sin(x[j])+c[i][j$
$]*cos(x[j]);
        r+=sqr(a[i]-sum);
    }
    updateCounter(r,X);
    return r;
}

double Rosenbrock::eval(Vector X, int *nerror)
{
    double *x=X, r=100*sqr(x[1]-sqr(x[0]))+sqr(1-x[0]);
    updateCounter(r,X);
    return r;
}

Rosenbrock::Rosenbrock(int _t)
{
    t=_t;
    strcpy(name,"ROSEN");

    xOptimal.setSize(2);
    xStart.setSize(2);

    xOptimal[0]=1.0;
    xOptimal[1]=1.0;

    valueOptimal=0.0;

    xStart[0]=-1.2;
    xStart[1]=1.0;
}

double NoisyRosenbrock::eval(Vector X, int *nerror)
{
    double *x=X,r;
    r=100*sqr(x[1]-sqr(x[0]))+sqr(1-x[0])+rand1()*1e-4;
    updateCounter(r,X);
    return r;
}

NoisyRosenbrock::NoisyRosenbrock(int _t)
{
    t=_t;
    strcpy(name,"NOROSEN");

    xOptimal.setSize(2);
    xStart.setSize(2);

    xOptimal[0]=1.0;
    xOptimal[1]=1.0;

    valueOptimal=0.0;
    xStart[0]=-1.2;
    xStart[1]=1.0;

    initRandom();
}

double NoisyQuadratic::eval(Vector X, int *nerror)
{
    int i=n;
    double *x=X,r=0.0;
    while (i--) r+=sqr(x[i]-2.0);
    r+=rand1()*1e-5;
    updateCounter(r,X);
    return r;
}

NoisyQuadratic::NoisyQuadratic(int _t)
{
    int i;
    t=_t; strcpy(name,"NOQUAD");

    n=4;
    xOptimal.setSize(n);
    xStart.setSize(n);
    for (i=0; i<n; i++)
    {
        xOptimal[i]=2.0;
        xStart[i]=0.0;
    }
    valueOptimal=0.0;
    initRandom();
}

void ObjectiveFunction::initBounds()
{
    int dim=this->dim();
    bl.setSize(dim);
    bu.setSize(dim);
    double *dbl=bl,*dbu=bu;
    while (dim--)
    {
        *(dbl++)=-INF;
        *(dbu++)=INF;
    }
}

Vector ObjectiveFunction::evalGradNLConstraint(int j, $
$Vector v, int *nerror)
{
    Vector R(dim());
    evalGradNLConstraint(j, v, R, nerror);
    return R;
}

FletcherTest2::FletcherTest2(int _t)
{
    t=_t;
    strcpy(name,"FLETCHER");

    xOptimal.setSize(3);
    xStart.setSize(3);

    xOptimal[0]=0.0;
    xOptimal[1]=0.0;
    xOptimal[2]=2.0;
    valueOptimal=-2.0;

    xStart[0]=0.0;
    xStart[1]=0.22;
    xStart[2]=0.0;

    initBounds();
    nNLConstraints=0;

    bl[0]=0.0;
    bl[1]=0.0;
    bl[2]=0.0;

    bu[2]=2.0;

    endInit();
}

double FletcherTest2::eval(Vector v, int *nerror)
{
    double *x=v, r=0.75*pow(x[0]*x[0]-x[0]*x[1]+x[1]*x$
$[1],0.75)-x[2];
    updateCounter(r,v);
    return r;
//    return sqr(1.0-v[0])+sqr(1-v[1]);
}

FletcherTest::FletcherTest(int _t)
{
    t=_t;
    strcpy(name,"FLETCHER");

    xOptimal.setSize(2);
    xStart.setSize(2);

    xOptimal[0]=0.5*sqrt(2.0);
    xOptimal[1]=0.5*sqrt(2.0);
    valueOptimal=-sqrt(2.0);

    xStart[0]=0.0;
    xStart[1]=0.0;

    nNLConstraints=2;
    endInit();
}

double FletcherTest::eval(Vector v, int *nerror)
{
    double r=-v[0]-v[1];
    updateCounter(r,v);
    return r;
//    return sqr(1.0-v[0])+sqr(1-v[1]);
}

double FletcherTest::evalNLConstraint(int j, Vector vv, $
$int *nerror)
{
    double *v=vv;
    switch (j)
    {
    case 0: return 1-v[0]*v[0]-v[1]*v[1];
    case 1: return -v[0]*v[0]+v[1];
    }
    return 0;
}

void FletcherTest::evalGradNLConstraint(int j, Vector v, $
$Vector R, int *nerror)
{
    double *r=R;
    switch (j)
    {
    case 0: r[0]=-2*v[0];
            r[1]=-2*v[1];
            break;
    case 1: r[0]=-2*v[0];
            r[1]=1;
            break;
    }
}

SuperSimpleConstrainedObjectiveFunction::$
$SuperSimpleConstrainedObjectiveFunction(int _t)
{
    t=_t;
    strcpy(name,"SIMPLE");

    xStart.setSize(2);
    xOptimal.setSize(2);

    xStart[0]=0.0;
    xStart[1]=0.0;

    xOptimal[0]=0.0;
    xOptimal[1]=4.0;

    initBounds();
    nNLConstraints=0;

    bl[0]=-2.0;
    bl[1]=-3.0;
    bu[0]=4.0;
    bu[1]=4.0;

    A.setSize(1,2);
    b.setSize(1);

    A[0][0]=-1.0;
//    A[0][0]=0.0;
    A[0][1]=-1.0;
    b[0]=-4.0;

    endInit();
}

double SuperSimpleConstrainedObjectiveFunction::eval($
$Vector v, int *nerror)
{
    double r=sqr(v[0])+sqr(v[1]-5);
    updateCounter(r,v);
    return r;
```

```
//    return -v[0]-2*v[1];
//    return v[0]+v[1];
}

#define BADSCALING 1e3

double BADScaleRosenbrock::eval(Vector X, int *nerror)
{
    double *x=X,r=100*sqr(x[1]/BADSCALING-sqr(x[0]))+sqr$
$(1-x[0]);
    updateCounter(r,X);
    return r;
}

BADScaleRosenbrock::BADScaleRosenbrock(int _t)
{
    t=_t;
    strcpy(name,"BSROSEN");

    xOptimal.setSize(2);
    xStart.setSize(2);

    xOptimal[0]=1.0;
    xOptimal[1]=1.0*BADSCALING;
    valueOptimal=0.0;

    xStart[0]=-1.2;
    xStart[1]=1.0*BADSCALING;
}

double CorrectScaleOF::eval(Vector X, int *nerror)
{
    int i=dim();
    double *x=X, *xr=xTemp, *re=rescaling;
    while (i--) xr[i]=re[i]*x[i];
    double r=of->eval(xTemp,nerror);
    updateCounter(r,X);
    return r;
}

double CorrectScaleOF::evalNLConstraint(int j, Vector X, $
$int *nerror)
{
    int i=dim();
    double *x=X, *xr=xTemp, *re=rescaling;
    while (i--) xr[i]=re[i]*x[i];
    return of->evalNLConstraint(j,xTemp,nerror);
}

void CorrectScaleOF::evalGradNLConstraint(int j, Vector X,$
$ Vector result, int *nerror)
{
    int i=dim();
    double *x=X, *xr=xTemp, *re=rescaling;
    while (i--) xr[i]=re[i]*x[i];
    of->evalGradNLConstraint(j,xTemp,result,nerror);
}

CorrectScaleOF::CorrectScaleOF(int _t, ObjectiveFunction *$
$_of):
    of(_of)
{
    t=_t;
    rescaling=_of->xStart.clone();
    if (of->isConstrained)
    {
        double *bl=of->bl, *bu=of->bu,*r=rescaling;
        int i=of->dim();
        while (i--)
        {
            if ((bl[i]>INF)&&(bu[i]<INF)) { r[i]=(bl[i]+bu$
$[i])/2; continue; }
            if ((r[i]==0.0)&&(bu[i]<INF)) { r[i]=bu[i]; $
$          continue; }
            if (r[i]==0.0) r[i]=1.0;
        }
    }
    init();
}

CorrectScaleOF::CorrectScaleOF(int _t, ObjectiveFunction *$
$_of, Vector _rescaling):
    of(_of), rescaling(_rescaling)
{
    t=_t;
    init();
}

void CorrectScaleOF::init()
{
    double *xos=of->xOptimal, *xss=of->xStart, *xod, *xsd,
           *r=rescaling, **datas=of->data, **datad,
           *bls=of->bl, *bus=of->bu, *bld, *bud, **as=of->$
$A, **ad;
    int n=of->dim(), i=n,j;
    strcpy(name,"SCALING");

    xTemp.setSize(n);
    xOptimal.setSize(n);                  xod=xOptimal;
    xStart.setSize(n);                    xsd=xStart;
    data.setSize(of->data.nLine(),n+1); datad=data;

    while(i--)
    {
        if (xos) xod[i]=xos[i]/r[i];
        xsd[i]=xss[i]/r[i];
        j=data.nLine();
        while (j--) datad[j][i]=datas[j][i]/r[i];
    }
    j=data.nLine();
    while (j--) datad[j][n]=datas[j][n];

    valueOptimal=of->valueOptimal;
    noiseAbsolute=of->noiseAbsolute;
    noiseRelative=of->noiseRelative;
    objectiveConst=of->objectiveConst;

    if (of->isConstrained==0) { isConstrained=0; return; }

    // there are constraints: scale them !
    isConstrained=of->isConstrained;
    nNLConstraints=of->nNLConstraints;
    bl.setSize(n);                        bld=bl;
    bu.setSize(n);                        bud=bu;
    A.setSize(A.nLine(),n);               ad=A;

    i=n;
    while(i--)
    {
        bld[i]=bls[i]/r[i];
        bud[i]=bus[i]/r[i];
        j=A.nLine();
        while (j--) ad[j][i]/=as[j][i]*r[i];
    }
}

void CorrectScaleOF::finalize()
{
    int i=dim();
    of->xBest.setSize(i);
    double *xb=of->xBest, *xbr=xBest, *r=rescaling;

    while (i--) xb[i]=xbr[i]*r[i];

    of->valueBest=valueBest;
    of->nfe=nfe;
    of->nfe2=nfe2;
    of->finalize();
}


#ifdef __INCLUDE_SIF__

#include "sif/SIFFunction.h"

/* extern elfunType elfunPARKCH_; extern groupType $
$groupPARKCH_; */
extern elfunType elfunAkiva_;    extern groupType $
$groupAkiva_;
extern elfunType elfunRosen_;    extern groupType $
$groupRosen_;
extern elfunType elfunALLINITU_; extern groupType $
$groupALLINITU_;
extern elfunType elfunSTRATEC_;  extern groupType $
$groupSTRATEC_;
extern elfunType elfunTOINTGOR_; extern groupType $
$groupTOINTGOR_;
extern elfunType elfunTOINTPSP_; extern groupType $
$groupTOINTPSP_;
extern elfunType elfun3PK_;      extern groupType $
$group3PK_;
extern elfunType elfunBIGGS6_;   extern groupType $
$groupBIGGS6_;
extern elfunType elfunBROWNDEN_; extern groupType $
$groupBROWNDEN_;
extern elfunType elfunDECONVU_;  extern groupType $
$groupDECONVU_;
extern elfunType elfunHEART_;    extern groupType $
$groupHEART_;
extern elfunType elfunOSBORNEB_; extern groupType $
$groupOSBORNEB_;
extern elfunType elfunVIBRBEAM_; extern groupType $
$groupVIBRBEAM_;
extern elfunType elfunKOWOSB_;   extern groupType $
$groupKOWOSB_;
```

```
extern elfunType elfunHELIX_;    extern groupType $
$groupHELIX_;

extern elfunType elfunCRAGGLVY_; extern groupType $
$groupCRAGGLVY_;
extern elfunType elfunEIGENALS_; extern groupType $
$groupEIGENALS_;
extern elfunType elfunHAIRY_;    extern groupType $
$groupHAIRY_;
extern elfunType elfunPFIT1LS_;  extern groupType $
$groupPFIT1LS_;
extern elfunType elfunVARDIM_;   extern groupType $
$groupVARDIM_;
extern elfunType elfunMANCINO_;  extern groupType $
$groupMANCINO_;
extern elfunType elfunPOWER_;    extern groupType $
$groupPOWER_;
extern elfunType elfunHATFLDE_;  extern groupType $
$groupHATFLDE_;
extern elfunType elfunWATSON_;   extern groupType $
$groupWATSON_;
extern elfunType elfunFMINSURF_; extern groupType $
$groupFMINSURF_;
extern elfunType elfunDIXMAANK_; extern groupType $
$groupDIXMAANK_;
extern elfunType elfunMOREBV_;   extern groupType $
$groupMOREBV_;
extern elfunType elfunBRYBND_;   extern groupType $
$groupBRYBND_;
extern elfunType elfunSCHMVETT_; extern groupType $
$groupSCHMVETT_;
extern elfunType elfunHEART6LS_; extern groupType $
$groupHEART6LS_;
extern elfunType elfunBROWNAL_;  extern groupType $
$groupBROWNAL_;
extern elfunType elfunDQDRTIC_;  extern groupType $
$groupDQDRTIC_;
extern elfunType elfunGROWTHLS_; extern groupType $
$groupGROWTHLS_;
extern elfunType elfunSISSER_;   extern groupType $
$groupSISSER_;
extern elfunType elfunCLIFF_;    extern groupType $
$groupCLIFF_;
extern elfunType elfunGULF_;     extern groupType $
$groupGULF_;
extern elfunType elfunSNAIL_;    extern groupType $
$groupSNAIL_;
extern elfunType elfunHART6_;    extern groupType $
$groupHART6_;

#endif

#ifdef __INCLUDE_AMPL__
#include "ampl/AMPLof.h"
#endif

ObjectiveFunction *getObjectiveFunction(int i, double *rho$
$)
{
    int n=2;
    ObjectiveFunction *of;
    double rhoEnd=-1;

    switch (i)
    {
    // first choice: internally coded functions:
    case  1: of=new Rosenbrock(i); break; // n=2;
    case  2: of=new BADScaleRosenbrock(i); break; // n=2;
    case  3: of=new FletcherTest(i); break; // n=2;
    case  4: of=new $
$SuperSimpleConstrainedObjectiveFunction(i); break; //n=2;
    case  5: of=new FletcherTest2(i); break; // n=3;
    case  6: of=new NoisyRosenbrock(i); break; //n=2;
    case  7: of=new NoisyQuadratic(i); break; //n=2;
// second choice: create new random objective function
    case  20: of=new RandomOF(i+1,n); ((RandomOF *)of)->$
$save("test.dat"); break;

    // third choice : reload from disk previous random $
$objective function
    case  21: of=new RandomOF(i,"test.dat"); break;

#ifdef __INCLUDE_SIF__

    // fourth choice: use SIF file
    case 104: of= new SIFFunction(i,"sif/examples/akiva.d"$
$     ,elfunAkiva_    ,groupAkiva_   ); break; // 2
    case 105: of= new SIFFunction(i,"sif/examples/allinitu$
$.d"  ,elfunALLINITU_ ,groupALLINITU_); break; // 4
    case 106: of= new SIFFunction(i,"sif/examples/stratec.$
$d"   ,elfunSTRATEC_  ,groupSTRATEC_ ); break; // 10
    case 107: of= new SIFFunction(i,"sif/examples/heart.d"$
$     ,elfunHEART_    ,groupHEART_   ); break; // 8
    case 108: of= new SIFFunction(i,"sif/examples/osborneb$
$.d"  ,elfunOSBORNEB_ ,groupOSBORNEB_); break; // 11
    case 109: of= new SIFFunction(i,"sif/examples/vibrbeam$
$.d"  ,elfunVIBRBEAM_ ,groupVIBRBEAM_); break; // 8
    case 110: of= new SIFFunction(i,"sif/examples/kowosb.d$
$"    ,elfunKOWOSB_   ,groupKOWOSB_  ); break; // 4
    case 111: of= new SIFFunction(i,"sif/examples/helix.d"$
$     ,elfunHELIX_    ,groupHELIX_   ); break; // 3

    case 112: of= new SIFFunction(i,"sif/examples/$
$rosenbrock.d",elfunRosen_    ,groupRosen_   ); rhoEnd= 5e$
$-3; break; // 2
    case 114: of= new SIFFunction(i,"sif/examples/sisser.d$
$"    ,elfunSISSER_   ,groupSISSER_  ); rhoEnd= 1e-2; $
$break; // 2
    case 115: of= new SIFFunction(i,"sif/examples/cliff.d"$
$     ,elfunCLIFF_    ,groupCLIFF_   ); rhoEnd= 1e-3; $
$break; // 2
    case 116: of= new SIFFunction(i,"sif/examples/hairy.d"$
$     ,elfunHAIRY_    ,groupHAIRY_   ); rhoEnd= 2e-3; $
$break; // 2
    case 117: of= new SIFFunction(i,"sif/examples/pfit1ls.$
$d"   ,elfunPFIT1LS_  ,groupPFIT1LS_ ); rhoEnd= 1e-2; $
$break; // 3
    case 118: of= new SIFFunction(i,"sif/examples/hatflde.$
$d"   ,elfunHATFLDE_  ,groupHATFLDE_ ); rhoEnd=12e-3; $
$break; // 3
    case 119: of= new SIFFunction(i,"sif/examples/schmvett$
$.d"  ,elfunSCHMVETT_ ,groupSCHMVETT_); rhoEnd= 1e-2; $
$break; // 3
    case 120: of= new SIFFunction(i,"sif/examples/growthls$
$.d"  ,elfunGROWTHLS_ ,groupGROWTHLS_); rhoEnd= 5e-3; $
$break; // 3
    case 121: of= new SIFFunction(i,"sif/examples/gulf.d" $
$     ,elfunGULF_     ,groupGULF_    ); rhoEnd= 5e-2; $
$break; // 3
    case 122: of= new SIFFunction(i,"sif/examples/brownden$
$.d"  ,elfunBROWNDEN_ ,groupBROWNDEN_); rhoEnd=57e-2; $
$break; // 4
    case 123: of= new SIFFunction(i,"sif/examples/eigenals$
$.d"  ,elfunEIGENALS_ ,groupEIGENALS_); rhoEnd= 1e-2; $
$break; // 6
    case 124: of= new SIFFunction(i,"sif/examples/heart6ls$
$.d"  ,elfunHEART6LS_ ,groupHEART6LS_); rhoEnd= 5e-2; $
$break; // 6
    case 125: of= new SIFFunction(i,"sif/examples/biggs6.d$
$"    ,elfunBIGGS6_   ,groupBIGGS6_  ); rhoEnd= 6e-2; $
$break; // 6
    case 126: of= new SIFFunction(i,"sif/examples/hart6.d"$
$     ,elfunHART6_    ,groupHART6_   ); rhoEnd= 2e-1; $
$break; // 6
    case 127: of= new SIFFunction(i,"sif/examples/cragglvy$
$.d"  ,elfunCRAGGLVY_ ,groupCRAGGLVY_); rhoEnd= 6e-2; $
$break; // 10
    case 128: of= new SIFFunction(i,"sif/examples/vardim.d$
$"    ,elfunVARDIM_   ,groupVARDIM_  ); rhoEnd= 1e-3; $
$break; // 10
    case 129: of= new SIFFunction(i,"sif/examples/mancino.$
$d"   ,elfunMANCINO_  ,groupMANCINO_ ); rhoEnd= 1e-6; $
$break; // 10
    case 130: of= new SIFFunction(i,"sif/examples/power.d"$
$     ,elfunPOWER_    ,groupPOWER_   ); rhoEnd= 2e-2; $
$break; // 10
    case 131: of= new SIFFunction(i,"sif/examples/morebv.d$
$"    ,elfunMOREBV_   ,groupMOREBV_  ); rhoEnd= 1e-1; $
$break; // 10
    case 132: of= new SIFFunction(i,"sif/examples/brybnd.d$
$"    ,elfunBRYBND_   ,groupBRYBND_  ); rhoEnd= 6e-3; $
$break; // 10
    case 133: of= new SIFFunction(i,"sif/examples/brownal.$
$d"   ,elfunBROWNAL_  ,groupBROWNAL_ ); rhoEnd= 8e-3; $
$break; // 10
    case 134: of= new SIFFunction(i,"sif/examples/dqdrtic.$
$d"   ,elfunDQDRTIC_  ,groupDQDRTIC_ ); rhoEnd= 1e-3; $
$break; // 10
    case 135: of= new SIFFunction(i,"sif/examples/watson.d$
$"    ,elfunWATSON_   ,groupWATSON_  ); rhoEnd= 4e-2; $
$break; // 12
    case 137: of= new SIFFunction(i,"sif/examples/fminsurf$
$.d"  ,elfunFMINSURF_ ,groupFMINSURF_); rhoEnd= 1e-1; $
$break; // 16

    case 138: of= new SIFFunction(i,"sif/examples/tointgor$
$.d"  ,elfunTOINTGOR_ ,groupTOINTGOR_); break; // 50
    case 139: of= new SIFFunction(i,"sif/examples/tointpsp$
$.d"  ,elfunTOINTPSP_ ,groupTOINTPSP_); break; // 50
    case 140: of= new SIFFunction(i,"sif/examples/3pk.d" $
$       ,elfun3PK_       ,group3PK_     ); break; // 30
    case 141: of= new SIFFunction(i,"sif/examples/deconvu.$
$d"   ,elfunDECONVU_  ,groupDECONVU_ ); break; // 61
/*  case 142: of= new SIFFunction(i,"sif/examples/parkch.d$
$"    ,elfunPARKCH_   ,groupPARKCH_  ); break; // 15 */
#ifdef WIN32
    case 113: of= new SIFFunction(i,"sif/examples/snail.d"$
$     ,elfunSNAIL_    ,groupSNAIL_   ); rhoEnd= 2e-4; $
$break; // 2
    case 136: of= new SIFFunction(i,"sif/examples/dixmaank$
$.d"  ,elfunDIXMAANK_ ,groupDIXMAANK_); rhoEnd= 3e-1; $
$break; // 15
#else
    case 113: of= new SIFFunction(i,"sif/examples/snail.d"$
$     ,elfunSNAIL_    ,groupSNAIL_   ); rhoEnd= 7e-4; $
$break; // 2
    case 136: of= new SIFFunction(i,"sif/examples/dixmaank$
$.d"  ,elfunDIXMAANK_ ,groupDIXMAANK_); rhoEnd= 4e-1; $
$break; // 15
#endif

#endif
#ifdef __INCLUDE_AMPL__

    case 200: of= new AMPLObjectiveFunction(i,"ampl/$
$examples/hs022.nl",1.0); break;
    case 201: of= new AMPLObjectiveFunction(i,"ampl/$
$examples/hs023.nl"); break;
    case 202: of= new AMPLObjectiveFunction(i,"ampl/$
$examples/hs026.nl"); break;
    case 203: of= new AMPLObjectiveFunction(i,"ampl/$
$examples/hs034.nl"); break;
    case 204: of= new AMPLObjectiveFunction(i,"ampl/$
$examples/hs038.nl"); break;
    case 205: of= new AMPLObjectiveFunction(i,"ampl/$
$examples/hs044.nl"); break;
    case 206: of= new AMPLObjectiveFunction(i,"ampl/$
$examples/hs065.nl"); break;
    case 207: of= new AMPLObjectiveFunction(i,"ampl/$
$examples/hs076.nl"); break;
    case 208: of= new AMPLObjectiveFunction(i,"ampl/$
$examples/hs100.nl"); break;
    case 209: of= new AMPLObjectiveFunction(i,"ampl/$
$examples/hs106.nl"); break;
    case 210: of= new AMPLObjectiveFunction(i,"ampl/$
$examples/hs108.nl"); rhoEnd= 1e-5; break;
    case 211: of= new AMPLObjectiveFunction(i,"ampl/$
$examples/hs116.nl"); break;
    case 212: of= new AMPLObjectiveFunction(i,"ampl/$
$examples/hs268.nl"); break;

    case 250: of= new AMPLObjectiveFunction(i,"ampl/$
$examples/rosenbr.nl" ); rhoEnd= 5e-3; break; // 2
    case 251: of= new AMPLObjectiveFunction(i,"ampl/$
$examples/sisser.nl"  ); rhoEnd= 1e-2; break; // 2
    case 252: of= new AMPLObjectiveFunction(i,"ampl/$
$examples/cliff.nl"   ); rhoEnd= 1e-3; break; // 2
    case 253: of= new AMPLObjectiveFunction(i,"ampl/$
$examples/hairy.nl"   ); rhoEnd= 2e-2; break; // 2
    case 254: of= new AMPLObjectiveFunction(i,"ampl/$
$examples/pfit1ls.nl" ); rhoEnd= 1e-2; break; // 3
    case 255: of= new AMPLObjectiveFunction(i,"ampl/$
$examples/hatflde.nl" ); rhoEnd=12e-3; break; // 3
    case 256: of= new AMPLObjectiveFunction(i,"ampl/$
$examples/growthls.nl"); rhoEnd= 5e-3; break; // 3
    case 257: of= new AMPLObjectiveFunction(i,"ampl/$
$examples/gulf.nl"    ); rhoEnd= 5e-2; break; // 3
    case 258: of= new AMPLObjectiveFunction(i,"ampl/$
$examples/brownden.nl"); rhoEnd=57e-2; break; // 4
    case 259: of= new AMPLObjectiveFunction(i,"ampl/$
$examples/eigenals.nl"); rhoEnd= 1e-2; break; // 6
    case 260: of= new AMPLObjectiveFunction(i,"ampl/$
$examples/heart6ls.nl"); rhoEnd= 1e-2; break; // 6
    case 261: of= new AMPLObjectiveFunction(i,"ampl/$
$examples/biggs6.nl"  ); rhoEnd= 6e-2; break; // 6
    case 262: of= new AMPLObjectiveFunction(i,"ampl/$
$examples/hart6.nl"   ); rhoEnd= 2e-1; break; // 6
    case 263: of= new AMPLObjectiveFunction(i,"ampl/$
$examples/cragglvy.nl"); rhoEnd= 1e-2; break; // 10
    case 264: of= new AMPLObjectiveFunction(i,"ampl/$
$examples/vardim.nl"  ); rhoEnd= 1e-3; break; // 10
    case 265: of= new AMPLObjectiveFunction(i,"ampl/$
$examples/mancino.nl" ); rhoEnd= 1e-6; break; // 10
    case 266: of= new AMPLObjectiveFunction(i,"ampl/$
$examples/power.nl"   ); rhoEnd= 2e-2; break; // 10
    case 267: of= new AMPLObjectiveFunction(i,"ampl/$
$examples/morebv.nl"  ); rhoEnd= 1e-1; break; // 10
    case 268: of= new AMPLObjectiveFunction(i,"ampl/$
$examples/brybnd.nl"  ); rhoEnd= 3e-3; break; // 10
    case 269: of= new AMPLObjectiveFunction(i,"ampl/$
$examples/brownal.nl" ); rhoEnd= 8e-3; break; // 10
    case 270: of= new AMPLObjectiveFunction(i,"ampl/$
$examples/dqdrtic.nl" ); rhoEnd= 1e-3; break; // 10
    case 271: of= new AMPLObjectiveFunction(i,"ampl/$
$examples/watson.nl"  ); rhoEnd= 4e-2; break; // 12
    case 272: of= new AMPLObjectiveFunction(i,"ampl/$
$examples/dixmaank.nl"); rhoEnd= 3e-1; break; // 15
    case 273: of= new AMPLObjectiveFunction(i,"ampl/$
$examples/fminsurf.nl"); rhoEnd= 1e-1; break; // 16

#endif

    }
    if ((i>=250)&&(i<=273)) of->isConstrained=0;

    if ((rho)&&(rhoEnd!=-1)) *rho=rhoEnd;
    return of;
}
```

14.2.17 Tools.h

```
//
//      Header file for tools
//


#ifndef _MPI_TOOLS_H_
#define _MPI_TOOLS_H_

#include <math.h>

//#define SWAP(a,b) {tempr=(a); (a)=(b); (b)=tempr;}

#define maxDWORD 4294967295 //2^32-1;
#define INF 1.7E+308
#define EOL 10
#define PI 3.1415926535897932384626433832795
#define ROOT2 1.41421356

inline double abs( const double t1 )
{
        return t1 > 0.0 ? t1 : -t1;
}

inline double sign( const double a )
 // equivalent to sign(1,a)
{
    return a<0?-1:1;
}

inline double sign( const double t1, const double t2 )
{
    if(t2>=0) return abs(t1);
    return -abs(t1);
}

inline double isInt( const double a)
{
    return abs(a-floor( a + 0.5 ))<1e-4;
}

inline double mmin( const double t1, const double t2 )
{
        return t1 < t2 ? t1 : t2;
}

inline unsigned mmin( const unsigned t1, const unsigned t2$
$ )
{
        return t1 < t2 ? t1 : t2;
}

inline int mmin( const int t1, const int t2 )
{
        return t1 < t2 ? t1 : t2;
}

inline double mmax( const double t1, const double t2 )
{
        return t1 > t2 ? t1 : t2;
}
```

```
inline unsigned mmax( const unsigned t1, const unsigned t2$
$ )
{
        return t1 > t2 ? t1 : t2;
}

inline int mmax( int t1, int t2 )
{
        return t1 > t2 ? t1 : t2;
}

inline double sqr( const double& t )
{
        return t*t;
}

inline double round (double a)
{
    return (int)(a+.5);
}

unsigned long choose( unsigned n, unsigned k );
double rand1();
void initRandom(int i=0);
double euclidianNorm(int i, double *xp);

#include "Vector.h"
#include "Matrix.h"

void saveValue(Vector tmp,double valueOF, Matrix data);

#endif  /* _MPI_TOOLS_H_ */
```

14.2.18 Tools.cpp

```
//
//      Various tools and auxilary functions
//
#include <stdlib.h>
#include <stdio.h>
#include <sys/timeb.h>
#include "tools.h"

unsigned long choose( unsigned n, unsigned k )
{
        const unsigned long uupSize = 100;
        static unsigned long uup[uupSize];
        register unsigned long *up;
        static unsigned long Nold = 0;
        static unsigned long Kold = 0;
        register unsigned long l,m;
        register unsigned i,j;

        if ( (n < k) || !n ) return 0;

        if ( (n == k) || !k )   // includes n == 1
                return 1;

        if ( k > (n >> 1) )     // Only lower half
                k = n-k;

        if ( (Nold == n) && (k < Kold) )        // We did $
$it last time ...
                return *(uup + k - 1);

        if ( k > uupSize )
        {
                printf( "choose( unsigned, unsigned) :  $
$overflow\n");
                getchar(); exit(-1);
        }

        Nold=n; Kold=k;

        *(up=uup)=2;
        for (i=2; i<n; i++)             // Pascal's $
$triangle
        {
        // todo: remove next line:
                *(up+1)=1;
                l=1;
        m=*(up=uup);
                for (j=0; j<mmin(i,k); j++)
                {
                        *up=m+l;
                        l=m;
                        m=*(++up);
                }
        // todo: remove next line:
                *up=1;
        }

        return *(uup + k - 1);
}

unsigned long mysrand;
double rand1()
{
    mysrand=1664525*mysrand+1013904223L;
    return ((double)mysrand)/4294967297.0;
}

void initRandom(int i)
{
    if (i) { mysrand=i; return; }
     struct timeb t;
     ftime(&t);
     mysrand=t.millitm;
//     printf("seed for random number generation: %i\n",t.$
$millitm);
}

void error(char *s)
{
   printf("Error due to %s.", s);
   getchar(); exit(255);
};

double euclidianNorm(int i, double *xp)
{
// no tested
// same code for the Vector eucilidian norm and for the $
$Matrix Froebenis norm
/*
    double sum=0;
    while (i--) sum+=sqr(*(xp++));
    return sqrt(sum);
*/
    const double SMALL=5.422e-20, BIG=1.304e19/((double)i)$
$;
    double s1=0,s2=0,s3=0, x1max=0, x3max=0, xabs;

    while (i--)
    {
        xabs=abs(*(xp++));

        if (xabs>BIG)
        {
            if (xabs>x1max)
            {
                s1=1.0+s1*sqr(x1max/xabs);
                x1max=xabs;
                continue;
            }
            s1+=sqr(xabs/x1max);
            continue;
        }
        if (xabs<SMALL)
        {
            if (xabs>x3max)
            {
               s3=1.0+s3*sqr(x3max/xabs);
               x3max=xabs;
               continue;
            }
            if (xabs!=0) s3+=sqr(xabs/x3max);
            continue;
        }
        s2+=sqr(xabs);
    };
    if (s1!=0) return x1max*sqrt(s1+(s2/x1max)/x1max);
    if (s2!=0)
    {
        if (s2>=x3max) return sqrt(s2*(1.0+(x3max/s2)*($
$x3max*s3)));
        return sqrt(x3max*((s2/x3max)+(x3max*s3)));
    }
    return x3max*sqrt(s3);
}
```

14.2.19 MSSolver.cpp (LagMaxModified)

```
// model step solver

#include <stdio.h>
#include <memory.h>

#include "Matrix.h"
#include "tools.h"
#include "Poly.h"


//int lagmax_(int *n, double *g, double *h__, double *rho,
//                      double *d__, double *v, double *vmax$
$);

Vector LAGMAXModified(Vector G, Matrix H, double rho,$
$double &VMAX)
{
    //not tested in depth but running already quite good

//    SUBROUTINE LAGMAX (N,G,H,RHO,D,V,VMAX)
//    IMPLICIT REAL*8 (A-H,O-Z)
//    DIMENSION G(*),H(N,*),D(*),V(*)
//
//    N is the number of variables of a quadratic $
$objective function, Q say.
//    G is the gradient of Q at the origin.
//    H is the symmetric Hessian matrix of Q. Only the $
$upper triangular and
//      diagonal parts need be set.
//    RHO is the trust region radius, and has to be $
$positive.
//    D will be set to the calculated vector of variables.
//    The array V will be used for working space.
//    VMAX will be set to |Q(0)-Q(D)|.
//
//     Calculating the D that maximizes |Q(0)-Q(D)| $
$subject to ||D|| .EQ. RHO
//    requires of order N**3 operations, but sometimes it $
$is adequate if
//    |Q(0)-Q(D)| is within about 0.9 of its greatest $
$possible value. This
//    subroutine provides such a solution in only of order$
$ N**2 operations,
//    where the claim of accuracy has been tested by $
$numerical experiments.
/*
    int n=G.sz();
    Vector D(n), V(n);
    lagmax_(&n, (double *)G, *((double**)H), &rho,
                  (double*)D, (double*)V, &VMAX);
    return D;
*/
    int i,n=G.sz();
    Vector D(n);

    Vector V=H.getMaxColumn();
    D=H.multiply(V);
    double vv=V.square(),
           dd=D.square(),
           vd=V.scalarProduct(D),
           dhd=D.scalarProduct(H.multiply(D)),
           *d=D, *v=V, *g=G;

//
//     Set D to a vector in the subspace spanned by V and $
$HV that maximizes
//     |(D,HD)|/(D,D), except that we set D=HV if V and HV$
$ are nearly parallel.
//
    if (sqr(vd)<0.9999*vv*dd)
    {
        double a=dhd*vd-dd*dd,
               b=.5*(dhd*vv-dd*vd),
               c=dd*vv-vd*vd,
               tmp1=-b/a;
        if (b*b>a*c)
        {
            double tmp2=sqrt(b*b-a*c)/a, dd1, dd2, dhd1, $
$dhd2;
            Vector D1=D.clone();
            D1.multiply(tmp1+tmp2);
            D1+=V;
            dd1=D1.square();
            dhd1=D1.scalarProduct(H.multiply(D1));

            Vector D2=D.clone();
            D2.multiply(tmp1-tmp2);
            D2+=V;
            dd2=D2.square();
            dhd2=D2.scalarProduct(H.multiply(D2));

            if (abs(dhd1/dd1)>abs(dhd2/dd2)) { D=D1; dd=$
$dd1; dhd=dhd1; }
            else { D=D2; dd=dd2; dhd=dhd2; }
            d=(double*)D;
        }
    };


//
//     We now turn our attention to the subspace spanned $
$by G and D. A multiple
//     of the current D is returned if that choice seems $
$to be adequate.
//
    double gg=G.square(),
           normG=sqrt(gg),
           gd=G.scalarProduct(D),
           temp=gd/gg,
           scale=sign(rho/sqrt(dd), gd*dhd);

    i=n; while (i--) v[i]=d[i]-temp*g[i];
    vv=V.square();

    if ((normG*dd)<(0.5-2*rho*abs(dhd))||(vv/dd<1e-4))
    {
        D.multiply(scale);
        VMAX=abs(scale*(gd+0.5*scale*dhd));
        return D;
    }

//
//     G and V are now orthogonal in the subspace spanned $
$by G and D. Hence
//     we generate an orthonormal basis of this subspace $
$such that (D,HV) is
//     negligible or zero, where D and V will be the basis$
$ vectors.
//
    H.multiply(D,G); //  D=HG;
    double ghg=G.scalarProduct(D),
           vhg=V.scalarProduct(D),
           vhv=V.scalarProduct(H.multiply(V));
    double theta, cosTheta, sinTheta;

    if (abs(vhg)<0.01*mmax(abs(vhv),abs(ghg)))
    {
        cosTheta=1.0;
        sinTheta=0.0;
    } else
    {
        theta=0.5*atan(0.5*vhg/(vhv-ghg));
        cosTheta=cos(theta);
        sinTheta=sin(theta);
    }
    i=n;
    while(i--)
    {
        d[i]= cosTheta*g[i]+ sinTheta*v[i];
        v[i]=-sinTheta*g[i]+ cosTheta*v[i];
    };

//
//     The final D is a multiple of the current D, V, D+V $
$or D-V. We make the
//     choice from these possibilities that is optimal.
//

    double norm=rho/D.euclidianNorm();
    D.multiply(norm);
    dhd=(ghg*sqr(cosTheta)+vhv*sqr(sinTheta))*sqr(norm);

    norm=rho/V.euclidianNorm();
    V.multiply(norm);
    vhv=(ghg*sqr(sinTheta)+vhv*sqr(cosTheta)*sqr(norm));

    double halfRootTwo=sqrt(0.5),   // =sqrt(2)/2=cos(PI$
$/4)
           t1=normG*cosTheta*rho,   // t1=abs(D.$
$scalarProduct(G));
```

```
            t2=normG*sinTheta*rho,   // t2=abs(V.$
$scalarProduct(G));
            at1=abs(t1),
            at2=abs(t2),
            t3=0.25*(dhd+vhv),
            q1=abs(at1+0.5*dhd),
            q2=abs(at2+0.5*vhv),
            q3=abs(halfRootTwo*(at1+at2)+t3),
            q4=abs(halfRootTwo*(at1-at2)+t3);
    if ((q4>q3)&&(q4>q2)&&(q4>q1))
    {
        double st1=sign(t1*t3), st2=sign(t2*t3);
        i=n; while (i--) d[i]=halfRootTwo*(st1*d[i]-st2*v[$
$i]);
        VMAX=q4;
        return D;
    }
    if ((q3>q2)&&(q3>q1))
    {
        double st1=sign(t1*t3), st2=sign(t2*t3);
        i=n; while (i--) d[i]=halfRootTwo*(st1*d[i]+st2*v[$
$i]);
        VMAX=q3;
        return D;
    }
    if (q2>q1)
    {
        if (t2*vhv<0) V.multiply(-1);
        VMAX=q2;
        return V;
    }
    if (t1*dhd<0) D.multiply(-1);
    VMAX=q1;
    return D;
};

Vector LAGMAXModified(Polynomial q, Vector pointXk, double$
$ rho,double &VMAX)
{
    int n=q.dim();
    Matrix H(n,n);
    Vector G(n), D(n);
    q.gradientHessian(pointXk,G,H);
    return LAGMAXModified(G,H,rho,VMAX);
};

Vector LAGMAXModified(Polynomial q, double rho,double &$
$VMAX)
{
    return LAGMAXModified(q,Vector::emptyVector,rho,VMAX);
};
```

14.2.20 Parallel.h

```
#ifndef _PARALLEL_H_
#define _PARALLEL_H_

void parallelImprove(InterPolynomial *p, int *_k, double _rho, double *_valueFk, Vector _Base);
void parallelInit(int _nnode, int _dim, ObjectiveFunction *_of);
void parallelFinish();
void startParallelThread();

#endif
```

14.2.21 Parallel.cpp

```
#include <memory.h>
#include <stdio.h>

#include "Vector.h"
#include "IntPoly.h"
#include "tools.h"
#include "ObjectiveFunction.h"
#include "Method/METHODof.h"

#ifndef __ONE_CPU__

//#define NNODE 8
//char *nodeName[NNODE]={ "s07", "s06", "s05", "s04", "s03$
$", "s02", "s01", "master" };
//int   nodePort[NNODE]={  4321,  4321,  4321,  4321,  $
$4321,  4321,  4321,     4321 };

//#define NNODE 3
//char *nodeName[NNODE]={ "164.15.10.73", "164.15.10.73", $
$"164.15.10.73" };
//int   nodePort[NNODE]={           4321,           4322, $
$          4323 };

//#define NNODE 2
//char *nodeName[NNODE]={ "164.15.10.73", "164.15.10.73" $
$};
//int   nodePort[NNODE]={           4321,           4322 $
$};

//#define NNODE 2
//char *nodeName[NNODE]={ "127.0.0.1", "127.0.0.1" };
//int   nodePort[NNODE]={        4321,        4322 };

//#define NNODE 1
//char *nodeName[NNODE]={ "164.15.10.73" };
//int   nodePort[NNODE]={           4321 };

#define NNODE 1
char *nodeName[NNODE]={ "127.0.0.1"};
int   nodePort[NNODE]={        4321};

//#define NNODE 2
//char *nodeName[NNODE]={ "192.168.1.206", $
$"192.168.1.206"};
//int   nodePort[NNODE]={           4321,           $
$4322};

//#define NNODE 1
//char *nodeName[NNODE]={ "192.168.1.206"};
//int   nodePort[NNODE]={           4321};

#define REDUCTION 0.7

int localPort=4320;

int nnode;
int meanEvalTime=1; // in seconds

class Parallel_element
{
  public:
        Vector x;
        double vf;
        char isFinished,isFree;
    Parallel_element(): x(), isFinished(false), isFree($
$true) {};
    ~Parallel_element(){};
};
Parallel_element *pe;
int peiMax,kbest, interSitesI, nUrgent;
double rho, valueOF;
Matrix interpSites;
Vector Base;
int parallelElement[NNODE+1];
InterPolynomial *pcopy;
bool Allfinished;
ObjectiveFunction *of;

void mutexLock();
void mutexUnlock();
void sendVector(int nodeNumber, Vector v);
void sendInt(int nodeNumber, int i);
Vector rcvVector(int nodeNumber);
void sendDouble(int nodeNumber, double d);
double rcvDouble(int nodeNumber);
void myConnect();
void waitForCompletion();
void myDisconnect();
unsigned long ipaddress(char *name);


/*
void sortPE()
{
    int i;
    bool t=true;
    Parallel_element tmp;
    while (t)
    {
        t=false;
        for (i=0; i<pei-1; i++)
            if (pe[i].vf>pe[i+1].vf)
            {
                tmp=pe[i];
                pe[i]=pe[i+1];
                pe[i+1]=tmp;
                t=true;
            }
    }
}
*/

void parallelImprove(InterPolynomial *p, int *_k, double $
$_rho, double *_valueFk, Vector _Base)
{
        if (nnode==0) return;

    mutexLock();

    int i,j,t=0;

    if (!_Base.equals(Base))
    {
        Vector translation=Base-_Base;
        for (i=0; i<peiMax; i++) if (!pe[i].isFree) pe[i].$
$x+=translation;
    }

    double d=INF;
    for (j=0; j<peiMax; j++)
    {
        if (pe[j].isFinished)
        {
            t++;
            if (pe[j].vf<d) { i=j; d=pe[j].vf; }
        }
    }
    printf("We have calculated in parallel %i values of the$
$ objective function.\n", t);

    if (t>0)
    {
        if (d<*_valueFk)
        {
            t=p->findAGoodPointToReplace(-1, _rho, pe[i].x$
$);
            *_k=t;
            *_valueFk=pe[i].vf;
            p->replace(t, pe[i].x, pe[i].vf);
            pe[i].isFree=true;
            pe[i].isFinished=false;
            printf("we have a great parallel improvement\n"$
$);
        };

        for ( j=0 ; j<peiMax; j++)
            if (pe[j].isFinished)
            {
                if (pe[j].x.euclidianDistance(p->$
$NewtonPoints[*_k])<2*REDUCTION*_rho)
                {
                   t=p->findAGoodPointToReplace(*_k, _rho,$
$ pe[j].x);

                   p->replace(t, pe[j].x, pe[j].vf);
                }

//                p->maybeAdd(pe[j].x, *_k, _rho, pe[j].vf$
$);
                pe[j].isFree=true;
                pe[j].isFinished=false;
            };
    }

    pcopy->copyFrom(*p);
    kbest=*_k;
    nUrgent=pcopy->getGoodInterPolationSites(interpSites, $
$kbest, _rho*REDUCTION );
    interSitesI=0;
    rho=_rho;
    valueOF=*_valueFk;
```

```
    Base=_Base.clone();

    mutexUnlock();
}

void parallelInit(int _nnode, int _dim, ObjectiveFunction $
$*_of)
{
    int dim=_dim,n=choose( 2+dim,dim )-1,i;
        nnode=mmin(mmin(_nnode,n), NNODE);
        if (nnode==0) return;

    peiMax=2*n;
        pe=new Parallel_element[peiMax];
    for (i=0; i<nnode; i++) parallelElement[i]=-1;
    interpSites.setSize(n,dim);
    pcopy=new InterPolynomial(dim,2);

    of=_of;
    myConnect();
//    for (i=0; i<NNODE; i++) sendInt(i,_of->t);
}

void parallelFinish()
{
        if (nnode==0) return;
    myDisconnect();

    delete[] pe;
    delete(pcopy);
}

void startOneComputation()
{
    if (interSitesI==interpSites.nLine()) return;

    int i=0,pei=0;

    nUrgent++;
    while (nUrgent>0)
    {
        // search for a free node and use it!
        while ((i<nnode)&&(parallelElement[i]!=-1)) i++;
        if (i==nnode) return;

        while ((pei<peiMax)&&(!pe[pei].isFree)) pei++;
        if (pei==peiMax) return;

        pe[pei].x=interpSites.getLine(interSitesI); $
$interSitesI++; nUrgent--;
        pe[pei].isFree=false;
        parallelElement[i]=pei;
        sendVector(i, Base+pe[pei].x);
    }
    // if dist>2*rho continue to add jobs
}

void finishComputation(int ii)
{
    int t,i=parallelElement[ii];
    Vector pointToAdd=pe[i].x;
    double _valueOF=pe[i].vf=rcvDouble(ii);
    // saveValue(Base+pointToAdd,valueOF, data);
    printf("valueOF parallel=%e\n",_valueOF);
    if (_valueOF<valueOF)
    {
        valueOF=_valueOF;
        t=pcopy->findAGoodPointToReplace(-1, rho, $
$pointToAdd);
        kbest=t;
        pcopy->replace(t, pointToAdd, _valueOF);

        nUrgent=pcopy->getGoodInterPolationSites($
$interpSites, kbest, rho*REDUCTION );
        interSitesI=0;
    }
    else
    {
        t=pcopy->findAGoodPointToReplace(kbest, rho, $
$pointToAdd);
        pcopy->replace(t, pointToAdd, _valueOF);
//    }
//    if (pcopy->maybeAdd(pointToAdd, kbest, rho, _valueOF$
$))
//    {
        pcopy->getGoodInterPolationSites(interpSites, $
$kbest, rho*REDUCTION );
        interSitesI=0;
    }
    pe[i].isFinished=true;
    parallelElement[ii]=-1;
}
```

```
#ifdef WIN32

/*****************************************
 *                                       *
 *       windows specific code:          *
 *                                       *
 *****************************************/

#include <winsock.h>

// Define SD_BOTH in case it is not defined in winsock.h
#ifndef SD_BOTH
#define SD_BOTH 2 // to close tcp connection in both $
$directions
#endif

#include <process.h>

SOCKET sock[NNODE+1];
HANDLE hh;
HANDLE PolyMutex;

void mutexLock()
{
    WaitForSingleObject( PolyMutex, INFINITE );
}

void mutexUnlock()
{
    ReleaseMutex( PolyMutex );
}

DWORD WINAPI parallelMain( LPVOID lpParam )
{
    waitForCompletion();
    return 0;
}

SECURITY_ATTRIBUTES sa;
void startParallelThread()
{
        if (nnode==0) return;
    printf("starting parallel thread.\n");

    sa.nLength=sizeof(SECURITY_ATTRIBUTES);
    sa.lpSecurityDescriptor=NULL;
    sa.bInheritHandle=true;

    PolyMutex=CreateMutex( &sa, FALSE, NULL );

// for debug:
//    waitForCompletion();

    Allfinished=false;

    unsigned long fake;
        hh=CreateThread(NULL, 0,
        (unsigned long (__stdcall *)(void *))parallelMain,$
$ NULL, 0, &fake);
}

void sendVector(int nodeNumber, Vector v)
{
    int n=v.sz();
    send(sock[nodeNumber],(const char FAR *) &n,sizeof(int$
$),0);
    send(sock[nodeNumber],(const char FAR *)(double*)v,$
$sizeof(double)*n,0);
}

void sendDouble(int nodeNumber, double d)
{
    send(sock[nodeNumber],(const char FAR *)&d,sizeof($
$double),0);
}

void sendInt(int nodeNumber, int i)
{
    send(sock[nodeNumber],(const char FAR *)&i,sizeof(int)$
$,0);
}

void receive(SOCKET sock,void *cc,int n)
{
    char *c=(char*)cc;
    int i;
    while ((i=recv(sock,c,n,0))>0)
    {
        n-=i;
        c+=i;
        if (n<=0) return;
    }
    printf("Receive corrupted. Press ENTER\n");
    getchar();
    exit(254);
}

Vector rcvVector(int nodeNumber)
{
    int n;
    Vector v;
    receive(sock[nodeNumber],&n,sizeof(int));
    v.setSize(n);
    receive(sock[nodeNumber],(double *)v,n*sizeof(double))$
$;
    return v;
}

double rcvDouble(int nodeNumber)
{
    double d;
    receive(sock[nodeNumber],&d,sizeof(double));
    return d;
}

SOCKET createSocket(char *nodeName, int port, bool $
$resolveName=true)
{
    SOCKET sock;
        SOCKADDR_IN to;
    struct hostent* hostentry;

    if (resolveName)
    {
        hostentry = gethostbyname( nodeName );
        if ( !hostentry ) {
            printf("Unknown %s, error: %u!\n", nodeName, $
$WSAGetLastError());
            exit(-1);
        }
        printf("Resolving successful\n");

        if ( hostentry->h_addrtype != AF_INET || hostentry$
$->h_length != 4 ) {
            printf("No ip addresses for this host!\n");
            exit(-1);
        }
    }

    // Here we create a TCP socket
        sock = socket(
                            PF_INET, // Protocol family =$
$ Internet
                            SOCK_STREAM, // stream mode
                            IPPROTO_TCP // TCP protocol
                        );
        // Check if the socket was successfully created!!
        if ( sock == INVALID_SOCKET ) {
                printf("Could not create socket: %u\n", $
$WSAGetLastError());
                exit(-1);
        }

        // First, we need to specify the destination $
$address

        memset( &to, 0, sizeof(to) ); // init. entire $
$structure to zero
        to.sin_family = AF_INET;
        if (resolveName) memcpy(&to.sin_addr.s_addr, *$
$hostentry->h_addr_list, hostentry->h_length);
    else to.sin_addr.s_addr=ipaddress(nodeName);
    to.sin_port = htons(port);

        // Setup a connection to the remote host
        if ( connect(sock,(SOCKADDR*) &to, sizeof(to) ) < $
$0 )
    {
                printf("Failed to connect to %s\n", $
$nodeName);
                exit(-1);
        }
    send(sock,(const char FAR *)&of->t,sizeof(int),0);
    if (of->t==30)
    {
        METHODObjectiveFunction *mof=($
$METHODObjectiveFunction *)of;
        send(sock,(const char FAR *) &mof->fullDim,sizeof($
$int),0);
        send(sock,(const char FAR *)(double*)mof->$
$vFullStart,sizeof(double)*mof->vFullStart.sz(),0);
    }
    return sock;
}

SOCKET createUDPrcvSocket(int port)
```

```
{
    struct sockaddr_in localaddr;
    SOCKET sock;
        // Here we create a TCP socket
        sock = socket(
                            PF_INET, // Protocol family =$
$ Internet
                            SOCK_DGRAM, // stream mode
                            IPPROTO_UDP // UDP protocol
                        );
        // Check if the socket was successfully created!!
        if ( sock == INVALID_SOCKET ) {
                printf("Could not create socket: %u\n", $
$WSAGetLastError());
                exit(-1);
        }

        // First, we need to specify the destination $
$address

        memset( &localaddr, 0, sizeof(localaddr) ); // $
$init. entire structure to zero
        localaddr.sin_family = AF_INET;
        localaddr.sin_addr.s_addr = INADDR_ANY;
        localaddr.sin_port = htons(port);

        if ( bind( sock, (SOCKADDR*) &localaddr, sizeof($
$localaddr)  ))
    {
                printf("Error in bind: %u\n", $
$WSAGetLastError());
                exit(-1);
        }

        printf("Socket bound to local address\n");
    return sock;
}

void stopAll(int port)
{
    struct sockaddr_in to;
    SOCKET sock;
        // Here we create a TCP socket
        sock = socket(
                            PF_INET, // Protocol family =$
$ Internet
                            SOCK_DGRAM, // stream mode
                            IPPROTO_UDP // UDP protocol
                        );
        // Check if the socket was successfully created!!
        if ( sock == INVALID_SOCKET ) {
                printf("Could not create socket: %u\n", $
$WSAGetLastError());
                exit(-1);
        }

        // First, we need to specify the destination $
$address

        memset( &to, 0, sizeof(to) ); // init. entire $
$structure to zero
        to.sin_family = AF_INET;
        to.sin_addr.s_addr = inet_addr("127.0.0.1");
        to.sin_port = htons(port);

    int i=1;
    sendto(sock,(const char*)&i,sizeof(int),0,(const $
$struct sockaddr *)&to,sizeof(to));

    closesocket(sock);

    while (!Allfinished)
    {
        Sleep(500);
    }
}


void myConnect()
{
    int i,error;
        WSADATA wsaData;

        if ( error=WSAStartup(MAKEWORD(1,1),&wsaData) )
    {
                printf("WSAStartup failed: error = %u\n", $
$error);
                exit(-1); // Exit program
        }

    for (i=0; i<NNODE; i++)
    {
        sock[i]=createSocket(nodeName[i], nodePort[i]);
    }
}

SOCKET sockToStopAll;
void waitForCompletion()
{
    sockToStopAll=createUDPrcvSocket(localPort);
        // Now, the connection is setup and we can send $
$and receive data

    // directly start a computation
    mutexLock();
    startOneComputation();
    mutexUnlock();

    int i,r;
    fd_set fd;
    struct timeval tv;
    tv.tv_usec=((int)(((double)meanEvalTime)/nnode*1e6))$
$%1000000;
    tv.tv_sec=((int)meanEvalTime/nnode);

    while (1)
    {
        FD_ZERO(&fd);
        for (i=0; i<nnode; i++) if (parallelElement[i$
$]!=-1) FD_SET(sock[i],&fd);
        FD_SET(sockToStopAll, &fd);

        r=select(0, &fd, NULL, NULL, &tv);

        mutexLock();

        if (r)
        {
            // computation is finished on a node
            for (i=0; i<nnode; i++)
                if (FD_ISSET(sock[i],&fd))
                    finishComputation(i);
        }
        if (FD_ISSET(sockToStopAll,&fd)) break;
        startOneComputation();

        mutexUnlock();
    }
    Allfinished=true;
    ExitThread(0);
}

void myDisconnect()
{
    int error,i;
    // to stop the trhead:

    stopAll(localPort);

    closesocket(sockToStopAll);
    CloseHandle(hh);
    CloseHandle( PolyMutex );

    for (i=0; i<NNODE; i++)
    {
        error = shutdown( sock[i], SD_BOTH );
            if ( error < 0 ) {
                    printf("Error while closing: %u\n", $
$WSAGetLastError());
            }

            // When we are finished with the socket, $
$destroy it
            closesocket( sock[i] );
    }

        WSACleanup();
}

#include <errno.h>

void calculateNParallelJob(int n,double *vf,Vector *cp, $
$ObjectiveFunction *of)
{
    int i,r;
    if (nnode==0)
    {
        for (i=0; i<n; i++) vf[i]=of->eval(cp[i]);
        return;
    }

    int k=0, j=0, nn=mmin(NNODE+1,n);
    fd_set fd;

    char buf[200];
    sprintf(buf,"%i",localPort);
    i=(int)_spawnlp(_P_NOWAIT, "..\\client\\debug\\client.$
$exe", "..\\client\\debug\\client.exe", buf, NULL);
//    printf("%i %i %i %i | %i", E2BIG,EINVAL,ENOENT,$
$ENOEXEC,ENOMEM, i);

    Sleep(2000);

    parallelElement[NNODE]=0;
    sock[NNODE]=createSocket("127.0.0.1", localPort, false$
$);
    sendVector(NNODE,cp[0]);

    for (j=1; j<nn; j++)
    {
        parallelElement[j-1]=j;
        sendVector(j-1,cp[j]);
    }
    for (i=nn-1; i<NNODE; i++) parallelElement[i]=-1;

    while (k<n)
    {
        FD_ZERO(&fd);
        for (i=0; i<NNODE+1; i++) if (parallelElement[i$
$]!=-1) FD_SET(sock[i],&fd);

        r=select(0, &fd, NULL, NULL, NULL);

        if (r)
        {
            // computation is finished on a node
            for (i=0; i<NNODE+1; i++)
                if (FD_ISSET(sock[i],&fd))
                {
                    vf[parallelElement[i]]=rcvDouble(i);
                       k++;
                    if (j<n)
                    {
                        parallelElement[i]=j;
                        sendVector(i,cp[j]);
                        j++;
                    } else parallelElement[i]=-1;
                }
        }
    }

    sendInt(NNODE, -1);

    int error = shutdown( sock[NNODE], SD_BOTH );
        if ( error < 0 ) {
                printf("Error while closing: %u\n", $
$WSAGetLastError());
        }

        // When we are finished with the socket, destroy $
$it
        closesocket( sock[NNODE] );
}

#else

/******************************************
 *                                        *
 *         Linux specific code:           *
 *                                        *
 ******************************************/

#include <unistd.h>
#include <sys/time.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/socket.h>
#include <syslog.h>
#include <netinet/in.h>
#include <pthread.h>
#include <pthread.h>
#include <netdb.h>

typedef unsigned int SOCKET;

SOCKET sock[NNODE+1];
pthread_t  hh;
pthread_mutex_t PolyMutex;

void mutexLock()
{
    pthread_mutex_lock(&PolyMutex);
}

void mutexUnlock()
{
    pthread_mutex_unlock(&PolyMutex );
}

void *threadLinux(void *ThreadArgs)
{
    waitForCompletion();
    return NULL;
}

void startParallelThread()
{
        if (nnode==0) return;
    printf("starting parallel thread.\n");

    pthread_mutex_init(&PolyMutex,NULL);

// for debug:
//    waitForCompletion();

    Allfinished=false;

    pthread_create ( &hh, NULL,(void * (*)(void *))::$
$threadLinux, NULL  );
}

void sendVector(int nodeNumber, Vector v)
{
    int n=v.sz();
    send(sock[nodeNumber], &n,sizeof(int),0);
    send(sock[nodeNumber], (double*)v,sizeof(double)*n,0);
}

void sendDouble(int nodeNumber, double d)
{
    send(sock[nodeNumber],&d,sizeof(double),0);
}

void sendInt(int nodeNumber, int i)
{
    send(sock[nodeNumber],&i,sizeof(int),0);
}

void receive(int sock,void *cc,int n)
{
    char *c=(char*)cc;
    int i;
    while ((i=recv(sock,c,n,0))>0)
    {
        n-=i;
        c+=i;
        if (n<=0) return;
    }
    printf("Receive corrupted. Press ENTER\n");
    getchar();
    exit(254);
}

Vector rcvVector(int nodeNumber)
{
    int n;
    Vector v;
    receive(sock[nodeNumber],&n,sizeof(int));
    v.setSize(n);
    receive(sock[nodeNumber],(double *)v,n*sizeof(double))$
$;
    return v;
}

double rcvDouble(int nodeNumber)
{
    double d;
    receive(sock[nodeNumber],&d,sizeof(double));
    return d;
}

int createSocket(char *nodeName, int port, bool $
$resolveName=true)
{
    struct sockaddr_in toaddr,adresse;
    struct hostent *hp=NULL;     /* pour l'adresse de la $
$machine distante */
    int desc;                                      /* $
$descripteur socket */
    unsigned int longueur=sizeof(struct sockaddr_in);   /*$
$ taille adresse */

    if (resolveName)
    {
        if ((hp=gethostbyname(nodeName))==NULL)
        {
           fprintf(stderr,"computer %s unknown.\n",$
$nodeName);
           exit(2);
        };
    };
    /* creation socket */
    if ((desc=socket(AF_INET, SOCK_STREAM, 0))==-1)
    {
        fprintf(stderr,"socket creation impossible");
            return -1;
    };

    adresse.sin_family=AF_INET;
    adresse.sin_addr.s_addr=htonl(INADDR_ANY);
    adresse.sin_port=0; /*numero du port en format reseau $
$*/

    /* demande d'attachement du socket */
    if (bind(desc,(sockaddr *)&adresse,longueur)==-1)
    {
            fprintf(stderr,"Attachement socket impossible.\$
$n");
        close(desc);
            return -1;
    };
    printf("socket binded.\n");

    toaddr.sin_family=AF_INET;
    toaddr.sin_port=htons(port);
    if (resolveName) memcpy(&toaddr.sin_addr.s_addr, hp->$
$h_addr, hp->h_length);
    else toaddr.sin_addr.s_addr=ipaddress(nodeName);

    /* demande d'attachement du socket */
    if (connect(desc,(sockaddr *)&toaddr,longueur)==-1)
    {
        fprintf(stderr,"socket connect impossible.\n");
        exit(3);
    };
    printf("socket connected.\n");
    send(desc,&of->t,sizeof(int),0);
    if (of->t==30)
    {
        METHODObjectiveFunction *mof=($
$METHODObjectiveFunction *)of;
        send(sock,&mof->fullDim,sizeof(int),0);
        send(sock,(double*)mof->vFullStart,sizeof(double)*$
$mof->vFullStart.sz(),0);
    }
    return desc;
}

SOCKET createUDPrcvSocket(int port)
{
    struct sockaddr_in adresse;
    int desc;                                  /* $
$descripteur socket */
    unsigned int longueur=sizeof(struct sockaddr_in);   /*$
$ taille adresse */

    /* creation socket */
    if ((desc=socket(AF_INET, SOCK_DGRAM, 0))==-1)
    {
        fprintf(stderr,"socket Creation impossible");
        exit(255);
    };

    /* preparation de l'adresse d'attachement */
    adresse.sin_family=AF_INET;
    adresse.sin_addr.s_addr=htonl(INADDR_ANY);
    adresse.sin_port=htons(port); /*numero du port en $
$format reseau */

    /* demande d'attachement du socket */
    if (bind(desc,(sockaddr *)&adresse,longueur)==-1)
    {
        fprintf(stderr,"socket Attachement impossible.\n");
        exit(255);
    };
    printf("Socket bound to local address\n");
    return desc;
}

void stopAll(int port)
{
    struct sockaddr_in to;
    int sock;                                  /* $
$descripteur socket */

    /* creation socket */
    if ((sock=socket(AF_INET, SOCK_DGRAM, 0))==-1)
    {
        fprintf(stderr,"Creation socket impossible");
        exit(255);
    };

    /* preparation de l'adresse d'attachement */
    to.sin_family=AF_INET;
    to.sin_addr.s_addr=ipaddress("127.0.0.1");
    to.sin_port=htons(port); /*numero du port en format $
$reseau */

    int i=1;
    sendto(sock,(const char*)&i,sizeof(int),0,(const $
$struct sockaddr *)&to,sizeof(to));

    close(sock);

    pthread_join(hh,NULL);
    pthread_detach(hh);
}


void myConnect()
{
    int i;

    for (i=0; i<NNODE; i++)
    {
        sock[i]=createSocket(nodeName[i], nodePort[i]);
    }
}

SOCKET sockToStopAll;
void waitForCompletion()
{
    SOCKET maxS;
    sockToStopAll=createUDPrcvSocket(localPort);
        // Now, the connection is setup and we can send $
$and receive data

    // directly start a computation
    mutexLock();
    startOneComputation();
    mutexUnlock();

    int i,r;
    struct timeval tv;
    fd_set fd;

    tv.tv_sec=((int)meanEvalTime/nnode);
    tv.tv_usec=((int)(((double)meanEvalTime)/nnode*1e6))$
$%1000000;

    while (1)
    {
        FD_ZERO(&fd); maxS=0;
        for (i=0; i<nnode; i++)
            if (parallelElement[i]!=-1) { FD_SET(sock[i],&$
$fd); maxS=mmax(maxS,sock[i]); }
        FD_SET(sockToStopAll, &fd);
        maxS=mmax(maxS,sockToStopAll);

        r=select(maxS+1, &fd, NULL, NULL, &tv);

        mutexLock();

        if (r)
        {
            // computation is finished on a node
            for (i=0; i<nnode; i++)
                if (FD_ISSET(sock[i],&fd))
                    finishComputation(i);
        }
        if (FD_ISSET(sockToStopAll,&fd)) break;
        startOneComputation();

        mutexUnlock();
    }
    int factice=0;
    pthread_exit(&factice);

}
```

```
void myDisconnect()
{
    int i;
    // to stop the trhead:

    stopAll(localPort);

    close(sockToStopAll);

    pthread_mutex_destroy( &PolyMutex );

    for (i=0; i<NNODE; i++)
            close( sock[i] );
}

#include <errno.h>

void calculateNParallelJob(int n,double *vf,Vector *cp, $
$ObjectiveFunction *of)
{
    int i,r;
    if (nnode==0)
    {
        for (i=0; i<n; i++) vf[i]=of->eval(cp[i]);
        return;
    }

    int k=0, j=0, nn=mmin(NNODE+1,n);
    SOCKET maxS;
    fd_set fd;

    char buf[200];
    sprintf(buf,"%i",localPort);
        if (fork()==0)
        {
       // child process
       execlp("../client/Debug/clientOptimizer",  "../$
$client/Debug/clientOptimizer", buf, NULL);
    };
    sleep(2);

    parallelElement[NNODE]=0;
    sock[NNODE]=createSocket("127.0.0.1", localPort, false$
$);
    sendVector(NNODE,cp[0]);

    for (j=1; j<nn; j++)
    {
        parallelElement[j-1]=j;
        sendVector(j-1,cp[j]);
    }
    for (i=nn-1; i<NNODE; i++) parallelElement[i]=-1;

    while (k<n)
    {
        FD_ZERO(&fd); maxS=0;
        for (i=0; i<NNODE+1; i++)
           if (parallelElement[i]!=-1) { FD_SET(sock[i],&$
$fd); maxS=mmax(maxS,sock[i]); }
        r=select(maxS+1, &fd, NULL, NULL, NULL);

        if (r)
        {
            // computation is finished on a node
            for (i=0; i<NNODE+1; i++)
                if (FD_ISSET(sock[i],&fd))
                {
                    vf[parallelElement[i]]=rcvDouble(i);
                    k++;
                    if (j<n)
                    {
                        parallelElement[i]=j;
                        sendVector(i,cp[j]);
                        j++;
                    } else parallelElement[i]=-1;
                }
        }
    }

    sendInt(NNODE, -1);

        // When we are finished with the socket, destroy $
$it
        close( sock[NNODE] );
    printf("\n");
}

#endif

/*****************************************
 *                                       *
 *         common network code:          *
 *                                       *
 *****************************************/

unsigned long ipaddress(char *name)
{
    int i;
    unsigned long ip,t;
    for (i=0; i<4; i++)
    {
        t=0;
        while ((*name!='.')&&(*name!='\0')) { t=t*10+*name$
$-'0'; name++; }
        name++;
        switch (i)
        {
        case 0: ip=t<<24; break;
        case 1: ip|=t<<16; break;
        case 2: ip|=t<<8; break;
        case 3: ip|=t; break;
        }
    }
    return htonl(ip);
}

#else

void parallelImprove(InterPolynomial *p, int *_k, double $
$_rho, double *_valueFk, Vector _Base)
{}

void startParallelThread(){}
void parallelInit(int _nnode, int _dim, ObjectiveFunction $
$*_of){}
void parallelFinish(){}

void calculateNParallelJob(int n,double *vf,Vector *cp, $
$ObjectiveFunction *of)
{
    int i;
    for (i=0; i<n; i++) vf[i]=of->eval(cp[i]);
}

#endif
```

14.2.22 METHODof.h

```
#ifndef METHOD_OBJECTIVEFUNCTION_INCLUDE
#define METHOD_OBJECTIVEFUNCTION_INCLUDE

#include "../ObjectiveFunction.h"
#include "../Vector.h"
#include "../VectorChar.h"

class METHODObjectiveFunction : public ObjectiveFunction
{
public:
    int fullDim;
    Vector vScale,vFullStart;

    METHODObjectiveFunction(int _t,char *argv, double *$
$rhoEnd);
    ~METHODObjectiveFunction();

    double eval(Vector v, int *ner=NULL);
    double evalNLConstraint(int j, Vector v, int *ner=NULL$
$);
    void evalGradNLConstraint(int j, Vector v, Vector $
$result, int *ner=NULL);

    virtual void finalize();

    //    void printStats() {ConstrainedObjectiveFunction$
$::printStats();}
private:
    char *objFunction, *inputObjF, *outputObjF, $
$startPointIsGiven;
    Vector weight,center,exponent,vTmpOF,vLx;
    VectorChar vXToOptim;
    int nToOptim, timeToSleep;
    double rhoEnd,BADVALUE;

    void shortXToLongX(double *sx, double *llx);
    void loadconstraints(int nineq, FILE *stream,char *$
$line);
    void loadData(char *line);
    void init(char *filename);
    void reductionDimension();
};

class ClientMETHODObjectiveFunction : public $
$ObjectiveFunction
{
public:

    ClientMETHODObjectiveFunction(int _t, char *, Vector v$
$);
    ~ClientMETHODObjectiveFunction();

    double eval(Vector v, int *ner=NULL);
    double evalNLConstraint(int j, Vector v, int *ner=NULL$
$){return 0;};
    void evalGradNLConstraint(int j, Vector v, Vector $
$result, int *ner=NULL){};

private:
    char *objFunction, *inputObjF, *outputObjF;
    int fullDim;
    Vector weight,center,exponent,vTmpOF,vFullStart,vLx,$
$vXToOptim;
};

#endif
```

14.2.23 METHODof.cpp

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#ifdef WIN32
#include <windows.h>
#include <process.h>
#else
//#include <signal.h>
//#include <setjmp.h>
#include <sys/wait.h>
#include <unistd.h>
#endif

#include "METHODof.h"
#include "../tools.h"

void launchObjFunction(int t, char *objFunction,char *$
$inputObjF, char *outputObjF)
{
    int k_old=-1, k=0;
    FILE *stream;
#ifdef WIN32
    remove(outputObjF);
    int er=(int)_spawnlp(_P_NOWAIT,objFunction,
                                   objFunction,
                                   inputObjF,NULL);
    if (er==-1)
    {
        perror("evaluation of OF");
        exit(1);
    }
#else
    unlink(outputObjF);
    if (fork()==0)
    {
        execlp(objFunction,
               objFunction,
               inputObjF,NULL);
    }
#endif
    // wait until the whole file has been written to the $
$disk
    // (if the disk is a network drive (mounted drive) it $
$can take
    // some time)
    while (true)
    {
        #ifdef WIN32
        Sleep(t);
        #else
        sleep(t);
        #endif
        stream=fopen(outputObjF,"rb");
        if (stream==NULL) continue;
        fseek(stream, 0, SEEK_END);
        k=ftell(stream);
        fclose(stream);
        t=1;
        if (k==k_old) break;
        k_old=k;
    }
    #ifndef WIN32
    wait(NULL);
    #endif
}

double evalLarge(Vector vLx, int *nerr, char *objFunction,$
$ char *inputObjF,
                 char *outputObjF, Vector vTmpOF, int $
$timeToSleep,
                 Vector weight, Vector center, Vector $
$exponent)
{
    double *lx=vLx,*tmp=vTmpOF;
    int j;

    // todo: define inputObjF and outputObjF

    FILE *stream=fopen(inputObjF,"wb");
    fputs("f",stream);
    j=1;
    fwrite(&j,sizeof(int),1,stream); // number N of $
$evaluation to perform
                                     // the next three $
$items are repeated N times.
    j=-1;
    fwrite(&j,sizeof(int),1,stream); // index of closest $
$evaluation already computed
    fwrite(&j,sizeof(int),1,stream); // index of the new $
$evaluation
    fwrite(lx,vLx.sz()*sizeof(double),1,stream); // point $
$where to perform evaluation
    fclose(stream);

    launchObjFunction(timeToSleep, objFunction, inputObjF,$
$ outputObjF);

    stream=fopen(outputObjF,"rb");
    fread(tmp,vTmpOF.sz()*sizeof(double),1,stream); // we $
$only get back one value
    fclose(stream);

    // aggregation into one (and only one) value.
    double s=0;
    for (j=0; j<(int)vTmpOF.sz() ; j++) s+=weight[j]*pow($
$tmp[j]-center[j],exponent[j]);
    return s;
}


void METHODObjectiveFunction::shortXToLongX(double *sx, $
$double *llx)
{
    int i=fullDim,j=dim();
    double *xf=vFullStart;
    char *xToOptim=vXToOptim;
    while (i--)
        if (xToOptim[i]) llx[i]=sx[--j];
        else llx[i]=xf[i];
}

//void METHODObjectiveFunction::longXtoShortX(double *llx,$
$ double *sx)
//{
//    int i,j=0;
//    for (i=0; i<fullDim; i++)
//        if (xToOptim[i]) { sx[j]=llx[i]; j++; }
//}

double METHODObjectiveFunction::evalNLConstraint(int j, $
$Vector v, int *ner)
{
   double gj, *lx=vLx;
   shortXToLongX(v,lx);

   switch (j)
   {
      case 1: gj=      -lx[18]-lx[23]/lx[21]*(lx[19]-lx$
$[4]); break;
//            case 1: *gj= lx[0]*lx[0]-0.7; break;
      case 2: gj=-lx[0]+lx[18]+lx[23]/lx[21]*(lx[19]-lx$
$[4]); break;
   }
   return gj;
}

void METHODObjectiveFunction::evalGradNLConstraint(int j, $
$Vector v, Vector result, int *ner)
{
    result.zero();
    double *grad=result, *lx=vLx;
   shortXToLongX(v,lx);

// initially "grad" is filled with zeros
   switch (j)
   {
//            case 1: temp[0]=2*lx[0];
//                    temp[1]=0;
      case 1: grad[4]=lx[23]/lx[21];
              grad[18]=-1;
              grad[19]=-lx[23]/lx[21];
              grad[21]=lx[23]*(lx[19]-lx[4])/sqr(lx[21]);
              grad[23]=-(lx[19]-lx[4])/lx[21];
              break;
      case 2: grad[0]=-1;
              grad[4]=-lx[23]/lx[21];
              grad[18]=1;
              grad[19]=lx[23]/lx[21];
              grad[21]=-lx[23]*(lx[19]-lx[4])/sqr(lx[21]);
              grad[23]=(lx[19]-lx[4])/lx[21];
              break;
   }
}


//void userFonction(double *t, double s)
//{
//    FILE *stream2;
//    computationNumber++;
//    stream2=fopen("/home/fvandenb/L6/progress.txt","a");
//    fprintf(stream2,"%i %f %f %f %f %f %f %f\n",$
$computationNumber, t[18],t[19],t[23],t[16+25],t[18+25],t$
$[19+25],s);
//    fclose(stream2);
//}

void METHODObjectiveFunction::loadconstraints(int nineq, $
$FILE *stream,char *line)
{
    int j;
    A.setSize(nineq,fullDim);
    b.setSize(nineq);
        Vector vTmp(fullDim+1);
    double *pb=b,*tmp=vTmp;

        for (j=0; j<nineq; j++)
        {
        if (j!=0) fgets(line,300,stream);
        vTmp.getFromLine(line);
        pb[j]=tmp[fullDim];
        A.setLine(j,vTmp,fullDim);
    }
}

METHODObjectiveFunction::~METHODObjectiveFunction()
{
    free(objFunction);
    free(outputObjF);
    free(inputObjF);
}

#define EOL1 13
#define EOL2 10

char emptyline(char *line)
{
        int i=0;
        if (line[0]==';') return 1;
        while (line[i]==' ') i++;
        if ((line[i]==EOL1)||(line[i]==EOL2)||(line[i]=='$
$\0')) return 1;
        return 0;
}

char *GetRidOfTheEOL(char *tline)
{
   char *t=tline;
   while (*t==' ') t++;
   while ((*tline!=EOL1)&&(*tline!=EOL2)&&(*tline!=' ')$
$&&(*tline)) tline++;
   *tline='\0';
   return t;
}

void METHODObjectiveFunction::loadData(char *line)
{
    char *tline=GetRidOfTheEOL(line);
    FILE *f=fopen(tline,"rb");
    if (f==NULL) return;
    fclose(f);
    Matrix dataInter(tline), data(dataInter.nLine(),$
$dataInter.nColumn());
    Vector r(fullDim);
    int n=0,i,j;
    char isAllReadyThere;
    double **di=dataInter, **d=data;

    initTolLC(vFullStart);

    // eliminate doubles and infeasible points
    for (i=0; i<dataInter.nLine(); i++)
    {
        dataInter.getLine(i, r, fullDim);
        if (!isFeasible(r)) continue;
        isAllReadyThere=0;
        for (j=0; j<n; j++)
            if (memcmp(di[i],d[j],fullDim*sizeof(double))$
$==0) { isAllReadyThere=1; break; }
        if (isAllReadyThere) continue;
        data.setLine(n,r);
        n++;
    }
}

void METHODObjectiveFunction::init(char *filename)
{
        FILE *stream;
        int i=0,j,nineq;
        char line[30000];
    Vector vTmp;
    double *tmp;
    char *xToOptim;

    if ((stream=fopen(filename,"r"))==NULL)
        {
            printf("optimization config file not found.\n");$
$ exit(254);
        };

        while ((fgets(line,30000,stream)!=NULL))
        {
                if (emptyline(line)) continue;
                switch (i)
                {
                    case 0:  // name of blackbox flow $
$solver
                     objFunction=(char*)malloc(strlen(line$
$)+1);
                             strcpy(objFunction,line);
                             GetRidOfTheEOL(objFunction);
                    case 1:  inputObjF=(char*)malloc($
$strlen(line)+1);
                             strcpy(inputObjF,line);
                             GetRidOfTheEOL(inputObjF);
                    case 2:  outputObjF=(char*)malloc($
$strlen(line)+1);
                             strcpy(outputObjF,line);
                             GetRidOfTheEOL(outputObjF);
                        case 3:  // number of parameters
                                 fullDim=atol(line);
                                 break;
                        case 4:  // weight for each $
$component of the objective function
                                   weight=Vector(line,0);
                       vTmpOF.setSize(weight.sz()); break;
            case 5:    center=Vector(line,weight.sz()); $
$break;
            case 6:  exponent=Vector(line,weight.sz()); $
$break;
                        case 7:  vTmp=Vector(line,fullDim)$
$; tmp=vTmp;
```

```
                                 vXToOptim.setSize(fullDim$
$); xToOptim=vXToOptim;
                      nToOptim=0;
                                 for (j=0; j<fullDim; j++)
                                 {
                                     xToOptim[j]=(tmp[j$
$]!=0.0);
                                     if (xToOptim[j]) $
$nToOptim++;
                                 }
                                 if (nToOptim==0)
                                 {
                                     printf("no variables $
$to optimize.\n");
                                     exit(255);
                                 };
                         case 8:  vFullStart=Vector(line,$
$fullDim);
                                 break;
                         case 9:  startPointIsGiven=(char)$
$atol(line);
                                 break;
                         case 10:  bl=Vector(line,fullDim);
                                 break;
                         case 11:  bu=Vector(line,fullDim);
                                 break;
                         case 12: nineq=atol(line); $
$                    break;
                         case 13: if (nineq>0)
                                 {
                                 loadconstraints(nineq,$
$stream,line);
                                     break;
                                 }
                                 i++;
             case 14: vScale=Vector(line,fullDim); $
$            break;
                         case 15: rhoEnd=atof(line); $
$                      break;
                         case 16: BADVALUE=atof(line); $
$                 break;
                         case 17: if (atol(line)==0) $
$nNLConstraints=0;

                                 //if (lazyMode==0) {$
$fclose(stream); return;}
                                 break;
                         case 18: loadData(line); break;
             case 19: timeToSleep=atol(line); break;
                 };
                 i++;
         };
         fclose(stream);

};

void METHODObjectiveFunction::reductionDimension()
{
    vLx.setSize(fullDim);

    // get the starting point

    int i;
    double *dp,d;
    if (data.nLine()==0)
    {
        d=evalLarge(vFullStart,NULL,objFunction,inputObjF,$
$outputObjF,
            vTmpOF,timeToSleep,weight,center,exponent);
        ObjectiveFunction::saveValue(vFullStart,d);
        data.swapLines(0,data.nLine()-1);
    } else
    {
        if (startPointIsGiven)
        {
            dp=*((double**)data);
            for (i=0; i<data.nLine(); i++)
                if (memcmp(dp,vFullStart.d->p,fullDim*$
$sizeof(double))==0) break;
            if (i<data.nLine())
            {
                data.swapLines(0,i);
            } else
            {
                double d=evalLarge(vFullStart,NULL,$
$objFunction,
                    inputObjF,outputObjF,vTmpOF,$
$timeToSleep,weight,center,exponent);

                ObjectiveFunction::saveValue(vFullStart,d)$
$;
                data.swapLines(0,data.nLine()-1);
            }
        } else
        {
            dp=(*((double**)data)+fullDim);
            double best=*dp;
            int n=0;
            for (i=1; i<data.nLine(); i++)
            {
                if (best>*dp) {best=*dp; n=i;}
                dp+=fullDim+1;
            }
            data.swapLines(0,n);
            data.getLine(0,vFullStart,fullDim);
        }
    }

    int ib,l,nl=0,j;
    char *xToOptim=vXToOptim;
    double *dbl=bl, *dbu=bu,sum, *pb=b;

    // reduce linear constraints

    double **p=A, *x=vFullStart;
    for (j=0; j<A.nLine(); j++)
        {
        // count number of non-null on the current line
        l=0; sum=pb[j]; ib=0;
            for (i=0; i<fullDim; i++)
                if ((xToOptim[i]) && (p[j][i]!=0.0))
            {
                l++; ib=i;
                if (l==2) break;
            }
            else sum-=p[j][i]*x[i];
            if (l==0) continue;
        if (l==2)
        {
            ib=0; sum=0;
                for (i=0; i<fullDim; i++)
                    if (xToOptim[i])
                {
                    p[nl][ib]=p[j][i]; ib++;
                }
                else sum-=p[j][i]*x[i];
            pb[j]+=sum;
            nl++;
            continue;
        }
            if (l==1)
            {
            d=p[j][ib];
                    if (d>0)
                    {
                        dbu[ib]=mmin(dbu[ib],sum/d);
                        if (x[ib]>dbu[ib])
                        {
                        fprintf(stderr,"error(2) on linear$
$ constraints %i.\n",j+1);
                        exit(254);
                        }
                    }
                    else
                    {
                        dbl[ib]=mmax(dbl[ib],sum/d);
                        if (x[ib]<dbl[ib])
                        {
                        fprintf(stderr,"error(3) on linear$
$ constraints %i.\n",j+1);
                        exit(254);
                        }
            }
                    continue;
                }
    }
    if (nl) A.setSize(nl,nToOptim);


    // reduce upper and lower bound, vScale
    // compute xStart
    xStart.setSize(nToOptim);
    ib=0;
    double *xs=xStart, *xf=vFullStart, *s=vScale;
    for (i=0; i<fullDim; i++)
        if (xToOptim[i])
        {
            dbl[ib]=dbl[i];
            dbu[ib]=dbu[i];
            xs[ib]=xf[i];
            s[ib]=s[i];
            ib++;
        }
    bl.setSize(nToOptim);
    bu.setSize(nToOptim);
    vScale.setSize(nToOptim);
    // reduce data
    double **ddata=data;
    for (j=0; j<data.nLine(); j++)
    {
        dp=ddata[j];
        ib=0;
        for (i=0; i<fullDim; i++)
            if (xToOptim[i])
            {
                dp[ib]=dp[i];
                ib++;
            }
        dp[ib]=dp[fullDim];
    }
    data.setNColumn(nToOptim);
}

#ifdef WIN32
void initLinux(){};
#else
void action(int) { wait(NULL); }
void action2(int) {}
void initLinux()
{
        // to prevent zombie processus:
    struct sigaction maction;
    maction.sa_handler=action;
    sigemptyset(&maction.sa_mask);
    sigaction(SIGCHLD,&maction,NULL);

/*      signal(   SIGHUP, action2);
        signal(   SIGINT, action2);
        signal(  SIGQUIT, action2);
        signal(   SIGILL, action2);
        signal(  SIGTRAP, action2);
        signal(  SIGABRT, action2);
        signal(   SIGIOT, action2);
        signal(   SIGBUS, action2);
        signal(   SIGFPE, action2);
        signal(  SIGKILL, action2);
        signal(  SIGUSR1, action2);
        signal(  SIGSEGV, action2);
        signal(  SIGUSR2, action2);
        signal(  SIGPIPE, action2);
        signal(  SIGALRM, action2);
        signal(  SIGTERM, action2);
        signal(SIGSTKFLT, action2);
        signal(  SIGCONT, action2);
        signal(  SIGTSTP, action2);
        signal(  SIGTTIN, action2);
        signal(  SIGTTOU, action2);
        signal(  SIGXCPU, action2);
        signal(  SIGXFSZ, action2);
        signal(SIGVTALRM, action2);
        signal(  SIGPROF, action2);
        signal(    SIGIO, action2);
        signal(   SIGPWR, action2);
        signal(   SIGSYS, action2);
*/
}
#endif

METHODObjectiveFunction::METHODObjectiveFunction(int _t,$
$char *argv, double *r)
{
    setName("METHOD");
    t=30;
    nNLConstraints=2;
    init(argv);
    reductionDimension();
    *r=rhoEnd;
    isConstrained=1;
    initLinux();
}

double METHODObjectiveFunction::eval(Vector v, int *nerr)
{
    shortXToLongX(v,vLx);
    double r=evalLarge(vLx,nerr,objFunction,inputObjF,$
$outputObjF,
            vTmpOF,timeToSleep,weight,center,exponent);
    updateCounter(r,vLx);
    if ((r>=BADVALUE)&&(nerr)) *nerr=1;
    return r;
}

void METHODObjectiveFunction::finalize()
{
    // to do: convert vBest to long form
    shortXToLongX(xBest,xBest);
}
```

```
ClientMETHODObjectiveFunction::$
$ClientMETHODObjectiveFunction(int _t, char *filename, $
$Vector v)
{
    t=_t;
    vFullStart=v;
        FILE *stream;
        int i=0;
        char line[30000];

    if ((stream=fopen(filename,"r"))==NULL)
        {
           printf("optimization config file not found.\n");$
$ exit(254);
        };

        while ((fgets(line,30000,stream)!=NULL))
        {
                if (emptyline(line)) continue;
                switch (i)
                {
                    case 0:  // name of blackbox flow $
$solver
                     objFunction=(char*)malloc(strlen(line$
$)+1);
                             strcpy(objFunction,line);
                             GetRidOfTheEOL(objFunction);
                    case 1:  inputObjF=(char*)malloc($
$strlen(line)+1);
                             strcpy(inputObjF,line);
                             GetRidOfTheEOL(inputObjF);
                    case 2:  outputObjF=(char*)malloc($
$strlen(line)+1);
                             strcpy(outputObjF,line);
                             GetRidOfTheEOL(outputObjF);
                        case 3:  // number of parameters
                                 fullDim=atol(line);
                                 break;
                        case 4:  // weight for each $
$component of the objective function
                                   weight=Vector(line,0);
                       vTmpOF.setSize(weight.sz()); break;
            case 5:    center=Vector(line,weight.sz()); $
$break;
            case 6:  exponent=Vector(line,weight.sz()); $
$break;
            case 7:  vXToOptim=Vector(line,fullDim); break$
$;
                };
                i++;
        };
        fclose(stream);
    initLinux();
};

ClientMETHODObjectiveFunction::~$
$ClientMETHODObjectiveFunction()
{
    free(objFunction);
    free(outputObjF);
    free(inputObjF);
}

double ClientMETHODObjectiveFunction::eval(Vector v, int *$
$nerr)
{
    int i,j=0;
    double *xf=vFullStart,*xToOptim=vXToOptim,*sx=v,*llx=$
$vLx;
    for (i=0; i<fullDim; i++)
        if (xToOptim[i]!=0.0) { llx[i]=sx[j]; j++; }
        else llx[i]=xf[i];

    double r=evalLarge(vLx,nerr,objFunction,inputObjF,$
$outputObjF,
            vTmpOF,3,weight,center,exponent);
    return r;
}
```

14.2.24 QPSolver.cpp

```
#include <stdio.h>
#include <memory.h>

//#include <crtdbg.h>

#include "Matrix.h"
#include "tools.h"
#include "VectorChar.h"


// #define POWELLQP
#ifdef POWELLQP

int ql0001_(int *m,int *me,int *mmax,int *n,int *nmax,int *mnn,
            double *c,double *d,double *a,double *b,double *xl,
            double *xu,double *x,double *u,int *iout,int *ifail,
            int *iprint,double *war,int *lwar,int *iwar,int *liwar,
            double *eps1);

void simpleQPSolve(Matrix G, Vector vd, Matrix Astar, Vector vBstar,   // in
                   Vector vXtmp, Vector vLambda)                        // out
{
    Astar.setNLine(Astar.nLine()+1);
    Matrix thisG, At=Astar.transpose();
    Astar.setNLine(Astar.nLine()-1);
    Vector minusvd=vd.clone();
    minusvd.multiply(-1.0); vBstar.multiply(-1.0);
    int m=Astar.nLine(), me=0, mmax=Astar.nLine()+1, n=vd.sz(), nmax=n,
        mnn=m+n+n, iout=0, ifail, iprint=0, lwar=3*nmax*nmax/2 + 10*nmax+ 2*mmax+1,
        liwar=n;
    if (G==Matrix::emptyMatrix) {
        thisG.setSize(n,n); thisG.diagonal(1.0); }
    else thisG=G;
    double *c=*((double**)thisG), *d=minusvd, *a=*((double**)At), *b=vBstar;
    Vector vxl(n),vxu(n), temp(lwar);
    VectorInt itemp(liwar);
    vLambda.setSize(mnn);
    double *xl=vxl, *xu=vxu, *x=vXtmp, *u=vLambda, *war=temp, eps1=1e-20;
    int *iwar=itemp;

    int dim=n; while (dim--) { xl[dim]=-INF; xu[dim]=INF; }
    iwar[0]=0;

    ql0001_(&m,&me,&mmax,&n,&nmax,&mnn,c,d,a,b,xl,xu,x,u,&iout,&ifail,
            &iprint,war,&lwar,iwar,&liwar,&eps1);

    vLambda.setSize(m);
}

#else

Matrix mQ_QP;
MatrixTriangle mR_QP;
VectorInt vi_QP, viPermut_QP;
Vector vLastLambda_QP;
char bQRFailed_QP;

void QPReconstructLambda(Vector vLambda, Vector vLambdaScale)
{
    int n=vi_QP.sz()-1, nc=vLambda.sz();
    int *ii=vi_QP;
    double *l=vLambda, *s=vLambdaScale;

    if (n>=0)
        while (nc--)
        {
            if (ii[n]==nc)
            {
//                 l[nc]=mmax(0.0,l[n]/s[n]);
                l[nc]=l[n]/s[n];
                n--;
                if (n<0) break;
            }
            else l[nc]=0.0;
        }
    if (nc) memset(l,0,nc*sizeof(double));
}

void simpleQPSolve(Matrix mH, Vector vG, Matrix mA, Vector vB,   // in
                   Vector vP, Vector vLambda, int *info)         // out
{
    const double tolRelFeasibility=1e-6;
//     int *info=NULL;
    int dim=mA.nColumn(), nc=mA.nLine(), ncr, i,j,k, lastJ=-2, *ii;
    MatrixTriangle M(dim);
    Matrix mAR, mZ(dim,dim-1), mHZ(dim,dim-1), mZHZ(dim-1,dim-1);
    Vector vTmp(mmax(nc,dim)), vYB(dim), vD(dim), vTmp2(dim), vTmp3(nc), vLast(dim),
           vBR_QP, vLambdaScale(nc);
    VectorChar vLB(nc);
    double *lambda=vLambda, *br, *b=vB, violationMax, violationMax2, activeLambdaMin,
           *rlambda,ax,al,dviolation, **a=mA, **ar=mAR, mymin, mymax, maxb, *llambda, delta,
           **r,t, *scaleLambda=vLambdaScale;
    char finished=0, feasible=0, *lb=vLB;

    if (info) *info=0;

    // remove lines which are null
    k=0; vi_QP.setSize(nc); maxb=0.0;
    for (i=0; i<nc; i++)
    {
        mymin=INF; mymax=-INF;
        for (j=0; j<dim; j++)
        {
            mymin=mmin(mymin,a[i][j]);
            mymax=mmax(mymax,a[i][j]);
            if ((mymin!=mymax)||(mymin!=0.0)||(mymax!=0.0)) break;
        }
        if ((mymin!=mymax)||(mymin!=0.0)||(mymax!=0.0))
        {
            lambda[k]=lambda[i];
            maxb=mmax(maxb,abs(b[i]));
            scaleLambda[k]=mA.euclidianNorm(i);
            vi_QP[k]=i;
            k++;
        }
    }
    nc=k; vi_QP.setSize(nc); ii=vi_QP; maxb=(1.0+maxb)*tolRelFeasibility;
    vLast.zero();

    for (i=0; i<nc; i++) if (lambda[i]!=0.0) lb[i]=2; else lb[i]=1;

    while (!finished)
    {
        finished=1;
        mAR.setSize(dim,dim); mAR.zero(); ar=mAR;
        vBR_QP.setSize(mmin(nc,dim)); br=vBR_QP;
        ncr=0;
        for (i=0; i<nc; i++)
            if (lambda[i]!=0.0)
            {
//                mAR.setLines(ncr,mA,ii[i],1);
                k=ii[i];
                t=scaleLambda[ncr];
                for (j=0; j<dim; j++) ar[ncr][j]=a[k][j]*t;
                br[ncr]=b[ii[i]]*t;
                ncr++;
            }
        mAR.setSize(ncr,dim);
        vBR_QP.setSize(ncr);
        vLastLambda_QP.copyFrom(vLambda); llambda=vLastLambda_QP;

        if (ncr==0)
        {
            // compute step
            vYB.copyFrom(vG);
            vYB.multiply(-1.0);
            if (mH.cholesky(M))
            {
                M.solveInPlace(vYB);
                M.solveTransposInPlace(vYB);
            } else
            {
                printf("warning: cholesky factorisation failed.\n");
                if (info) *info=2;
            }
            vLambda.zero();
            activeLambdaMin=0.0;
        } else
        {
            Matrix mAR2=mAR.clone(), mQ2;
            MatrixTriangle mR2;
            mAR2.QR(mQ2,mR2);

            mAR.QR(mQ_QP,mR_QP, viPermut_QP); // content of mAR is destroyed here !
            bQRFailed_QP=0;

            r=mR_QP;
            for (i=0; i<ncr; i++)
                if (r[i][i]==0.0)
                {
                    // one constraint has been wrongly added.
                    bQRFailed_QP=1;
                    QPReconstructLambda(vLambda,vLambdaScale); vP.copyFrom(vLast); return;
                }

            for (i=0; i<ncr; i++)
                if (viPermut_QP[i]!=i)
                {
                  //  printf("whoups.\n");
                }
            if (ncr<dim)
            {
                mQ_QP.getSubMatrix(mZ,0,ncr);
```

```
                    // Z^t H Z
                    mH.multiply(mHZ,mZ);
                    mZ.transposeAndMultiply(mZHZ,mHZ);
                    mQ_QP.setSize(dim,ncr);
                }

                // form Yb
                vBR_QP.permutIn(vTmp,viPermut_QP);
                mR_QP.solveInPlace(vTmp);
                mQ_QP.multiply(vYB,vTmp);

                if (ncr<dim)
                {

                    // ( vG + H vYB )^t Z : result in vD

                    mH.multiply(vTmp,vYB);
                    vTmp+=vG;
                    vTmp.transposeAndMultiply(vD,mZ);

                    // calculate current delta (result in vD)
                    vD.multiply(-1.0);
                    if (mZHZ.cholesky(M))
                    {
                        M.solveInPlace(vD);
                        M.solveTransposInPlace(vD);
                    }
                    else
                    {
                        printf("warning: cholesky factorisation failed.\n");
                        if (info) *info=2;
                    };

                    // evaluate vX* (result in vYB):
                    mZ.multiply(vTmp, vD);
                    vYB+=vTmp;
                }

                // evaluate vG* (result in vTmp2)
                mH.multiply(vTmp2,vYB);
                vTmp2+=vG;

                // evaluate lambda star (result in vTmp):
                mQ2.transposeAndMultiply(vTmp,vTmp2);
                mR2.solveTransposInPlace(vTmp);

                // evaluate lambda star (result in vTmp):
                mQ_QP.transposeAndMultiply(vTmp3,vTmp2);
                mR_QP.solveTransposInPlace(vTmp3);
                vTmp3.permutOut(vTmp,viPermut_QP);
                rlambda=vTmp;

                ncr=0;
                for (i=0; i<nc; i++)
                    if (lambda[i]!=0.0)
                    {
                        lambda[i]=rlambda[ncr];
                        ncr++;
                    }
            } // end of test on ncr==0

            delta=vG.scalarProduct(vYB)+0.5*vYB.scalarProduct(mH.multiply(vYB));
            // find the most violated constraint j among non-active Linear constraints:
            j=-1;
            if (nc>0)
            {
                k=-1; violationMax=-INF; violationMax2=-INF;
                for (i=0; i<nc; i++)
                {
                    if (lambda[i]<=0.0) // test to see if this constraint is not active
                    {
                        ax=mA.scalarProduct(ii[i],vYB);
                        dviolation=b[ii[i]]-ax;
                        if (llambda[i]==0.0)
                        {
                            // the constraint was not active this round
                            // thus, it can enter the competition for the next
                            // active constraint

                            if (dviolation>maxb)
                            {
                                // the constraint should be activated
                                if (dviolation>violationMax2)
                                    { k=i; violationMax2=dviolation; }
                                al=mA.scalarProduct(ii[i],vLast)-ax;
                                if (al>0.0) // test to see if we are going closer
                                {
                                    dviolation/=al;
                                    if (dviolation>violationMax )
                                        { j=i; violationMax =dviolation; }
                                }
                            }
                    } else
                    {
                        lb[i]--;
                        if (feasible)
                        {
                            if (lb[i]==0)
                            {
                                vLambda.copyFrom(vLastLambda_QP);
                                if (lastJ>=0) lambda[lastJ]=0.0;
                                QPReconstructLambda(vLambda,vLambdaScale);
                                vP.copyFrom(vYB);
                                return;
                            }
                        } else
                        {
                            if (lb[i]==0)
                            {
                                if (info) *info=1;
                                QPReconstructLambda(vLambda,vLambdaScale);
                                vP.zero();
                                return;
                            }
                        }
                        finished=0;  // this constraint was wrongly activated.
                        lambda[i]=0.0;
                    }
                }
            }

            // !!! the order the tests is important here !!!
            if ((j==-1)&&(!feasible))
            {
                feasible=1;
                for (i=0; i<nc; i++) if (llambda[i]!=0.0) lb[i]=2; else lb[i]=1;
            }
            if (j==-1) { j=k; violationMax=violationMax2; } // change j to k after feasible is set
            if (ncr==mmin(dim,nc)) {
                if (feasible) { // feasible must have been checked before
                    QPReconstructLambda(vLambda,vLambdaScale); vP.copyFrom(vYB); return;
                } else
                {
                    if (info) *info=1;
                    QPReconstructLambda(vLambda,vLambdaScale); vP.zero(); return;
                }
            }
            // activation of constraint only if ncr<mmin(dim,nc)
            if (j>=0) { lambda[j]=1e-5; finished=0; } // we need to activate a new constraint
//            else if (ncr==dim){
//                QPReconstructLambda(vLambda); vP.copyFrom(vYB); return; }
        }

        // to prevent rounding error
        if (j==lastJ) {
//        if (0) {
            QPReconstructLambda(vLambda,vLambdaScale); vP.copyFrom(vYB); return; }
        lastJ=j;

        vLast.copyFrom(vYB);
    }
    QPReconstructLambda(vLambda,vLambdaScale); vP.copyFrom(vYB); return;
}

void restartSimpleQPSolve(Vector vB0,   // in
                          Vector vP)                    // out
{
    if (bQRFailed_QP) { vP.zero(); return; }
    int i,k=0, *ii=vi_QP, nc2=vi_QP.sz();
    double *lambda=vLastLambda_QP, *b=vB0;
    Vector vTmp(nc2);
    for (i=0; i<nc2; i++)
    {
        if (lambda[i]!=0.0)
        {
            b[k]=b[ii[i]];
            k++;
        }
    }
    vB0.setSize(k);
    vB0.permutIn(vTmp,viPermut_QP);
    mR_QP.solveInPlace(vTmp);
    mQ_QP.multiply(vP,vTmp);
}

#endif
```

14.2.25 CTRSSolver.cpp (ConstrainedL2NormMinimizer)

```
#include <stdio.h>
#include <memory.h>

//#include <crtdbg.h>
```

```
#include "ObjectiveFunction.h"
#include "Matrix.h"
#include "IntPoly.h"
#include "tools.h"
#include "VectorChar.h"

// from QPsolver:
void simpleQPSolve(Matrix mH, Vector vG, Matrix mA, Vector vB,   // in
                   Vector vP, Vector vLambda, int *info);         // out
void restartSimpleQPSolve(Vector vBO,  // in
                          Vector vP);  // out

// from TRSSolver:
Vector L2NormMinimizer(Polynomial q, double delta,
                       int *infoOut=NULL, int maxIter=1000, double *lambda1=NULL);
Vector L2NormMinimizer(Polynomial q, Vector pointXk, double delta,
                       int *infoOut=NULL, int maxIter=1000, double *lambda1=NULL);
Vector L2NormMinimizer(Polynomial q, Vector pointXk, double delta,
                       int *infoOut, int maxIter, double *lambda1, Vector minusG, Matrix H);

double updateMu(double mu, Vector Lambda)
{
    const double delta=1.;
    double sigma=Lambda.LnftyNorm();
    if (1/mu<sigma+delta)
        mu=1/(sigma+2*delta);
    return mu;
}

void updatePositiveHessianOfLagrangian(Matrix mB, Vector vS, Vector vY, Matrix mH)
{
    int i,j, dim=vS.sz();

    if (vS.euclidianNorm()<1e-9) return;

    // take into account curvature of the quadratic
    vY+=mH.multiply(vS);

    Vector vBS=mB.multiply(vS);
    double sy=vY.scalarProduct(vS), sBs=vS.scalarProduct(vBS), theta=1.0;

    if (sy<0.2*sBs) theta=0.8*sBs/(sBs-sy);

    Vector vR;
    vBS.multiply(vR,1-theta);
    vR.addInPlace(theta, vY);

    double *r=vR, *bs=vBS, sr=1/vS.scalarProduct(vR), **p=mB;

    sBs=1/sBs;
    for (i=0; i<dim; i++)
        for (j=0; j<dim; j++)
            p[i][j]+=-sBs*bs[i]*bs[j]+sr*r[i]*r[j];
}

void calculateVX(Vector vX,Vector vBase, Matrix mZ, Vector vP)
{
    if (mZ==Matrix::emptyMatrix)
    {
        vX.copyFrom(vBase);
        vX+=vP;
        return;
    }
    mZ.multiply(vX, vP);
    vX+=vBase;
}

double mu=1.0, maxcc=0.0;
Vector nlSQPMinimizer(ObjectiveFunction *of, double delta, Vector vLambda, Matrix mZ,
                    Vector vYB, Vector vBase2, Matrix mH, Vector vG, Vector vC, double minStepSize=1e-16)
{
    const int nIterMax=700;
    const double ethaWolfCondition=0.5, //0.9,
                 alphaContraction=0.75, epsilonGradient=1e-5;
    char finished=0, bfeasible;
    int dim=vG.sz(),i, nc=of->nNLConstraints, ncPlus1=nc+1, info, ntry=2*nc, nerror;
    if (delta>=INF) ncPlus1=nc;
    Vector vBase=vBase2.clone(), vX(dim), vTmp(dim), vXtmp2(of->dim()), vP(dim), vCur(dim), vGCur(dim), vY, vB(ncPlus1$
$),vBest(dim);
    Matrix mHPositive(dim,dim),mA(ncPlus1, dim);
    double **a=mA, *b=vB, *lambda,
           *c=vC, alpha, phi1, phi0,dphi,t, mminstep=mmin(minStepSize*1e6,delta*1e-5*dim),
           dist, vPNorm,r, *vcx=vCur, *p=vP, minNormG;

    if (!(vYB==Vector::emptyVector)) vBase+=vYB;

    of->initTolNLC(vC,delta);
    minNormG=(1+vG.euclidianNorm())*epsilonGradient;

    vLambda.setSize(ncPlus1); lambda=vLambda;

    // construct non-linear part of Astar, vBstar and check if we already have a solution
    // calculateVX(vX,vBase,mZ,vP);

    mHPositive.diagonal(1.0);

    vCur.zero(); dist=minStepSize*.1;
    vY.setSize(dim);
    vP.zero();
    vBest.zero();
    int niter=nIterMax;

    while (!finished)
    {
        niter--;
        if (niter==0) break;
        finished=1;

        if (dist>minStepSize) vCur.copyFrom(vXtmp2);

        // update Atar, vBstar

        // vY is a temporary variable used for the update of the Inverse of the Hessian Of the Lagrangian
        vY.zero();
        calculateVX(vX,vBase,mZ,vCur);
        vXtmp2.setSize(of->dim());
        for (i=0; i<nc; i++)
        {
            vY.addInPlace(lambda[i],i,mA);

            t=b[i]=-of->evalNLConstraint(i,vX);

            // the gradient is in dimension of->dim which is different from dim: todo: maybe bug$
$!!!!!!!!!!!!!!!!!!!!!!!
            of->evalGradNLConstraint(i,vX,vXtmp2);
            if (!(mZ==Matrix::emptyMatrix))
            {
//                mZ.transposeAndMultiply(vTmp, vXtmp2);
                vXtmp2.transposeAndMultiply(vTmp, mZ);
                mA.setLine(i,vTmp);
            } else
                mA.setLine(i,vXtmp2);

            // termination test
            if (t>of->tolNLC) finished=0;

            vY.addInPlace(-lambda[i],vXtmp2);
        }

        if (delta<INF)
        {
            // trust region bound
            vY.addInPlace(lambda[nc],nc,mA);
            t=b[nc]=vCur.square()-delta*delta;
            for (i=0; i<dim; i++) a[nc][i]=-2.0*vcx[i];
            if (t>of->tolNLC) finished=0;
            vY.addInPlace(-lambda[nc],nc,mA);
        }
        if (finished) vBest.copyFrom(vCur);

        updatePositiveHessianOfLagrangian(mHPositive, vP, vY, mH);

        // find step direction
        mH.multiply(vGCur,vCur); vGCur+=vG;

        t=vGCur.euclidianNorm();
        r=mHPositive.LnftyNorm();
        if (t<minNormG*r)
        {
            for (i=0; i<dim; i++) p[i]=rand1()-0.5;
            dist=vPNorm=vP.euclidianNorm(); finished=0; info=1;
        } else
        {
            simpleQPSolve(mHPositive, vGCur, mA, vB,  // in
                          vP, vLambda, &info);               // out
            mu=updateMu(mu, vLambda);

            /*for (i=0; i<(int)vLambda.sz(); i++)
            {
                if (lambda[i]<0.0)
                {
                    printf("warning: a lambda is negative!.\n");
                }
            }*/

    // for debug puposes:
    //        calculateVX(vX,vBase,mZ,vCur);
    //        t2=isFeasible(vX,(ConstrainedObjectiveFunction *)of);

            // update penalty parameter
            if (info==1)
            {
                for (i=0; i<dim; i++) p[i]=rand1()-0.5;
                dist=vPNorm=vP.euclidianNorm(); finished=0;
            } else
            {
                dist=vPNorm=vP.euclidianNorm();
                if ((vPNorm==0.0)||(vPNorm>delta*100.0))
                {
                    vP.copyFrom(vGCur);
                    vP.multiply(-1.0);
                    dist=vPNorm=vP.euclidianNorm();
                }

//                mu=mmin(1.0,1.0-(1.0-mu)*0.5);
//                mu=1.0;
            }
        }

        //mu=updateMu(mu, vLambda);
        alpha=1.0;

        // start of line search to find step length
        // evaluation of the merit function at vCur (result in phi0):
        mH.multiply(vTmp,vCur);
        phi0=vCur.scalarProduct(vG)+0.5*vCur.scalarProduct(vTmp);
        r=0; for (i=0; i<ncPlus1; i++) r+=-mmin(-b[i], 0.0);
        phi0+=r/mu;

        // evaluation of the directional derivative of the merit function at vCur (result in dphi)
        // equation 12.3.3. page 298
        dphi=vP.scalarProduct(vGCur); r=0;
        for (i=0; i<ncPlus1; i++)
        {
            t=-b[i];
            if (t==0.0) r+=-mmin( mA.scalarProduct(i,vP), 0.0 );
            else if (lambda[i]!=0.0) r-=mA.scalarProduct(i,vP);
        }
        dphi+=r/mu;
        dphi=mmin(0.0,dphi);

        while (dist>minStepSize)
        {
            vXtmp2.copyFrom(vCur);
            vXtmp2.addInPlace(alpha,vP);

            // eval merit function at point vXtmp2 (result in phi1)
            mH.multiply(vTmp,vXtmp2);
            phi1=vG.scalarProduct(vXtmp2)+0.5*vTmp.scalarProduct(vXtmp2);
            calculateVX(vX,vBase,mZ,vXtmp2); r=0.0; bfeasible=1;
            for (i=0; i<nc; i++)
            {
                t=of->evalNLConstraint(i,vX,&nerror);
                if (nerror>0) break;
                b[i]=-t; c[i]=abs(t);
                r+=-mmin(t, 0.0);
                if (-t>of->tolNLC) bfeasible=0;
            }
            if (nerror>0) { alpha*=alphaContraction; dist=alpha*vPNorm; continue; }
            if (delta<INF) r+=-mmin(delta*delta-vXtmp2.square(), 0.0);
            phi1+=r/mu;

            if (phi1<=phi0+ethaWolfCondition*alpha*dphi)
            {
                if (bfeasible) vBest.copyFrom(vXtmp2);
                vP.multiply(alpha);
                break;
            }

            if ((vLambda.mmax()==0.0)||(info!=0)) { alpha*=alphaContraction; dist=alpha*vPNorm; continue; }

            if (delta<INF) b[nc]=vXtmp2.square()-delta*delta;

            // due to the linearization of the non-linear constraint, xtmp may not be feasible.
                // (second order correction step)

            restartSimpleQPSolve(vB,vTmp);
            vXtmp2+=vTmp;

            // end of SOC

            // eval merit function at point vXtmp2 (result in phi1)
            mH.multiply(vTmp,vXtmp2);
            phi1=vG.scalarProduct(vXtmp2)+0.5*vTmp.scalarProduct(vXtmp2);
            calculateVX(vX,vBase,mZ,vXtmp2); r=0; bfeasible=1;
            for (i=0; i<nc; i++)
            {
                t=-mmin(of->evalNLConstraint(i,vX,&nerror), 0.0);
                if (nerror>0) break;
                r+=t; c[i]=abs(t);
                if (-t>of->tolNLC) bfeasible=0;
            }
            if (nerror>0) { alpha*=alphaContraction; dist=alpha*vPNorm; continue; }
            if (delta<INF) r+=-mmin(delta*delta-vXtmp2.square(), 0.0);
            phi1+=r/mu;

            if (phi1<=phi0+ethaWolfCondition*alpha*dphi)
            {
                if (bfeasible) vBest.copyFrom(vXtmp2);
                vP.copyFrom(vXtmp2); vP-=vCur;
                dist=vP.euclidianNorm();
                break;
            }

            alpha*=alphaContraction;
            dist=alpha*vPNorm;
        }
        if (dist<=minStepSize)
        {
            ntry--;
            if (ntry==0) return vBest;
            vLambda.zero(); dist=0.0;
            finished=0;
        }
//        if ((dist>1e-6)||(dist>epsilonStep*distold)) finished=0;
//        distold=dist;

        if (dist>minStepSize)
        {
            if (dist>mminstep) finished=0;
//            vCur.copyFrom(vXtmp2);
        } // vCur+vBase is the current best point
    };
    if (niter==0)
    {
        printf("Warning: max number of iteration reached in SQP algorithm.\n");
        return vBest;
    }
    // calculate d
    if (vBest.euclidianNorm()>mminstep) return vBest;
    return vCur;
//    return vBest;
}

Vector FullLambda;

Vector ConstrainedL2NormMinimizer( Matrix mH, Vector vG, double delta,
                                   int *info, int iterMax, double *lambda1, Vector vBase,
                                   ObjectiveFunction *of, double minStepSize=1e-14)
{
    // the starting point x=0 is always good => simplified version of the algorithm

    int dim=vG.sz();
    Matrix mA(dim,dim),mQ, mZ,mHZ,mZHZ;
    MatrixTriangle mR;
    Vector vB(dim), vD, vTmp(dim), vTmp2(dim), vYB,vX(dim), vC(of->nNLConstraints),
           lastFullLambda(FullLambda.sz()), vBest(dim);
    double *flambda=FullLambda, *base=vBase, **a, *b, dviolation, violationMax,
           violationMax2, *x=vX, *s, vYBsq,cdelta, *lambda, *c=vC, *bl=of->bl,
           *bu=of->bu, *ofb=of->b, *lflambda=lastFullLambda, t, *vt, value, valueBest=INF;
    int nc=0, i,j,k, nLC=dim*2+of->A.nLine(), lastJ=-2;
    VectorChar vLB(nLC);
    char bNLCActive, finished=0, *lb=vLB, feasible=1, bIsCycling=0,bNLCWasActive;
    // nc is 'number of (active,linear) constraints'
    // nLC is 'number of (linear) constraints'

    of->initTolLC(vBase);

    for (i=0; i<dim; i++)
        if ((t=bl[i]-base[i])>of->tolLC) { feasible=0; break; }
    if (feasible)
    {
        for (i=0; i<dim; i++)
            if ((t=base[i]-bu[i])>of->tolLC) { feasible=0; break; }
        if (feasible)
            for (i=0; i<of->A.nLine(); i++)
                if ((t=ofb[i]-of->A.scalarProduct(i,vBase))>of->tolLC) { feasible=0; break; }
    }

    bNLCActive=0;
    // is there non-linear constraints active ?
    for (i=0; i<of->nNLConstraints; i++)
        if (flambda[i+nLC]!=0.0) bNLCActive=1;
    bNLCWasActive=bNLCActive;

    vBest.zero();
    for (i=0; i<nLC; i++) if (flambda[i]!=0.0) lb[i]=2; else lb[i]=1;

    while(!finished)
    {
        finished=1;
        mA.setSize(dim,dim); mA.zero(); nc=0; vB.setSize(dim);
        a=mA; b=vB;
        for (i=0; i<dim; i++)
            if (flambda[i]!=0.0)
            {
                a[nc][i]=1.0;
                b[nc]=bl[i]-base[i];
                nc++;
            }
        for (i=0; i<dim; i++)
            if (flambda[i+dim]!=0.0)
            {
                a[nc][i]=-1.0;
                b[nc]=-bu[i]+base[i];
                nc++;
            }

        if (of->A.nLine()>0)
        {
            double t1;
            for (i=0; i<of->A.nLine(); i++)
                if (flambda[i+2*dim]!=0.0)
                {
                    t1=of->A.scalarProduct(i,vBase);
                    mA.setLines(nc,of->A,i,1);
                    b[nc]=ofb[i]-t1;
                    nc++;
                }
        }

        lastFullLambda.copyFrom(FullLambda);
        mA.setSize(nc,dim); vB.setSize(nc);

        if (nc>dim)
        {
            printf("strange things happening...\n");
            getchar(); exit(255);
        }

        if (nc==0)
        {
            if (bNLCActive)
            {
                vYBsq=0.0;
                vTmp.setSize(of->nNLConstraints+1); vTmp.setPart(0,FullLambda,0,nLC);
                vD=nlSQPMinimizer(of, delta, vTmp, Matrix::emptyMatrix, Vector::emptyVector, vBase, mH, vG, vC, $
$minStepSize);
                vYB.copyFrom(vD);
                if (lambda1) *lambda1=0.0;
                bNLCActive=0; vt=vTmp;
                for (i=0; i<of->nNLConstraints; i++)
                {
//                    if (vt[i]<1e-8) flambda[i+nLC]=0.0; else
                      flambda[i+nLC]=vTmp[i];
                    if (flambda[i+nLC]!=0.0) bNLCActive=1;
                }
                //if (bNLCActive==0) finished=0;
            } else
            {
                vYBsq=0.0;
                vTmp2.copyFrom(vG);
                vD=L2NormMinimizer(Polynomial::emptyPolynomial,Vector::emptyVector,delta,info,iterMax,lambda1,vTmp2,mH$
$);
                vYB.copyFrom(vD);
//                vTmp.setSize(0);
                FullLambda.zero();
            }
            // evaluate vG* (result in vTmp2)
            mH.multiply(vTmp2,vYB);
            vTmp2+=vG;
        } else
        {
            mA.QR(mQ,mR); // content of mA is destroyed here !

            for (i=0; i<nc; i++)
                if (mR[i][i]==0.0)
                {
                    // the last constraint has been added erroneously
                    flambda[j]=0.0;
                    return vYB;
                }

            if (nc<dim)
            {
                mQ.getSubMatrix(mZ,0,nc);

                // Z^t H Z
                mH.multiply(mHZ,mZ);
                mZ.transposeAndMultiply(mZHZ,mHZ);
                mQ.setSize(dim,nc);
            }

            // form Yb
            vTmp.copyFrom(vB);
            mR.solveInPlace(vTmp);
            mQ.multiply(vYB,vTmp);
            vYBsq=vYB.square();

            if (nc<dim)
            {

                // calculate ( vG + H vYB )^t Z    : result in vTmp2
                mH.multiply(vTmp,vYB);
                vTmp+=vG;
                vTmp.transposeAndMultiply(vTmp2,mZ);

                if (bNLCActive)
                {
                    cdelta=delta*delta-vYBsq;
                    if (cdelta>0.0)
                    {
                        cdelta=sqrt(cdelta);
                        vTmp.setSize(of->nNLConstraints+1); vTmp.setPart(0,FullLambda,0,nLC);
                        vD=nlSQPMinimizer(of, cdelta, vTmp, mZ, vYB, vBase, mZHZ, vTmp2, vC, minStepSize);
                        bNLCActive=0; vt=vTmp;
                        for (i=0; i<of->nNLConstraints; i++)
                        {
                        //    if (vt[i]<1e-8) flambda[i+nLC]=0.0; else
                                flambda[i+nLC]=vTmp[i];
                            if (flambda[i+nLC]!=0.0) bNLCActive=1;
                        }
                        //if (bNLCActive==0) finished=0;
                    } else
                    {
                        vD.setSize(vTmp2.sz());
                        vD.zero();
                        FullLambda.zero(nLC);
                    }
                    if (lambda1) *lambda1=0.0;
                } else
                {
                    // calculate current delta
                    if (vYBsq==0.0)
                    {
                        vD=L2NormMinimizer(Polynomial::emptyPolynomial,Vector::emptyVector,delta,info,iterMax,lambda1,$
$vTmp2,mZHZ);
                    }
                    else
                    {

                        cdelta=delta*delta-vYBsq;
                        if (cdelta>0.0)
                        {
                            cdelta=sqrt(cdelta);
                            vD=L2NormMinimizer(Polynomial::emptyPolynomial,Vector::emptyVector,cdelta,info,iterMax,$
$NULL,vTmp2,mZHZ);
                        } else
                        {
                            vD.setSize(vTmp2.sz());
                            vD.zero();
                        }
                        if (lambda1) *lambda1=0.0;
                    }
                    FullLambda.zero(nLC); // set NLC to null
                }

                // evaluate vX* (result in vYB):
                mZ.multiply(vTmp, vD);
                vYB+=vTmp;
            }

            // evaluate vG* (result in vTmp2)
            mH.multiply(vTmp2,vYB);
            vTmp2+=vG;

            // evaluate lambda* (result in vTmp):
            mQ.transposeAndMultiply(vTmp,vTmp2);
            mR.solveTransposInPlace(vTmp);
            lambda=vTmp;

        /*
        // search for most inactive contraint (to be removed)
            i=-1; minLambda=INF;
            for (j=0; j<nc; j++)
                if (lambda[j]<minLambda)
                {
                    i=j;
                    minLambda=lambda[j];
                }

            // termination test
            if (i<0)
            {
                return vYB;
            }
        */
            // update fullLambda and remove all inactive constraints
            nc=0; j=0;
            for (i=0; i<nLC; i++)
                if (flambda[i]!=0.0)
                {
                    if (lambda[nc]==0.0) flambda[i]=-1e-20; // only to prepare the next tests
                    else flambda[i]=lambda[nc];
                    nc++;
            }
    } // end of test on nc==0

    // find the most violated constraint j among non-active Linear constraints:
    vX.copyFrom(vYB); vX+=vBase;
    violationMax=-INF; violationMax2=-INF; s=vYB;
    j=-1; k=-1;

    for (i=0; i<dim; i++)
        if (flambda[i]<=0.0)
        {
            if (lflambda[i]==0.0)
            {
                dviolation=bl[i]-x[i];
                if (dviolation>of->tolLC)
                {
                    if (dviolation>violationMax2) { k=i; violationMax2=dviolation; }
                    if (s[i]<0.0)
                    {
                        dviolation/=-s[i];
                        if (dviolation>violationMax) { j=i; violationMax=dviolation; }
                    }
                }
            } else
            {
                lb[i]--;
                flambda[i]=0.0;
                if (feasible)
                {
                    if (lb[i]==0) bIsCycling=1;
                } else
                {
                    if (lb[i]==0) {
                        vYB.zero(); bIsCycling=1; }
                }
                finished=0;
            }
        }

    for (i=0; i<dim; i++)
        if (flambda[i+dim]<=0.0)
        {
            if (lflambda[i+dim]==0.0)
            {
                dviolation=x[i]-bu[i];
                if (dviolation>of->tolLC)
                {
                    if (dviolation>violationMax2) { k=i+dim; violationMax2=dviolation; }
                    if (s[i]>0.0)
                    {
                        dviolation/=s[i];
                        if (dviolation>violationMax) { j=i+dim; violationMax=dviolation; }
                    }
                }
            } else
            {
                lb[i+dim]--;
                flambda[i+dim]=0.0;
                if (feasible)
                {
                    if (lb[i+dim]==0) bIsCycling=1;
                } else
                {
                    if (lb[i+dim]==0) {
                        vYB.zero(); bIsCycling=1; }
                }
                finished=0;
            }
        }

    if (of->A.nLine()>0)
    {
        double ax,al;
        for (i=0; i<of->A.nLine(); i++)
        {
            ax=of->A.scalarProduct(i,vX);
            if (flambda[i+2*dim]<=0.0)
            {
                if (lflambda[i+2*dim]==0.0)
                {
                    dviolation=ofb[i]-ax;
                    if (dviolation>of->tolLC)
                    {
                        if (dviolation>violationMax2) { k=i+2*dim; violationMax2=dviolation; }
                        al=of->A.scalarProduct(i,vYB);
                        if (al>0.0)
                        {
                            dviolation/=al;
                            if (dviolation>violationMax) { j=i+2*dim; violationMax=dviolation; }
                        }
                    }
                } else
                {
                        lb[i+2*dim]--;
                        flambda[i+2*dim]=0.0;
                        if (feasible)
                        {
                            if (lb[i+2*dim]==0) bIsCycling=1;
                        } else
                        {
                            if (lb[i+2*dim]==0) {
                                vYB.zero(); bIsCycling=1; }
                        }
                        finished=0;
                    }
                }
            }
        }
        if ((!bNLCWasActive)&&(!bNLCActive))
        {
            // test if a new NL contraint has just turned active.
            bNLCActive=0;
            for (i=0; i<of->nNLConstraints; i++)
                if ((c[i]=-of->evalNLConstraint(i,vX))>of->tolNLC) bNLCActive=1;
        }
        if ((j==-1)&&(k==-1)&&((bNLCWasActive)||(!bNLCActive)))
        {
            value=vTmp2.scalarProduct(vYB);
            if (value<valueBest)
            {
                vBest.copyFrom(vYB);
                valueBest=value;
            }
        }
        if ((bIsCycling)&&((bNLCWasActive)||(!bNLCActive)))
        {
            if (delta<INF) return vBest;
            return vYB;
        }
        if (j==-1)
        {
            j=k; violationMax=violationMax2;
            if (!feasible)
            {
                feasible=1;
                for (i=0; i<nLC; i++) if (flambda[i]!=0.0) lb[i]=2; else lb[i]=1;
            }
        }
        if (nc==mmin(dim,nLC)) {
//        if (0) {
            if (!feasible) vYB.zero();
            return vYB;
        }
        if ((bNLCActive)&&(!bNLCWasActive)) { finished=0; j=-3; }
        bNLCWasActive=bNLCActive;

//        if ((bNLCActive)&&(minLambda>-1e-12)&&(lastJ>=0)&&(j==-1))
//        {
//            flambda[lastJ]=1.0;
//            continue;
//        }

        // termination test
//        if ((!bNLCActive)&&
//            ((vTmp.sz()==0)||
//             (minLambda>=0.0)||
//             ((vD.euclidianNorm()<1e-8)&&(minLambda>-1e-8))
//            )&&
//            ((j<0)||(violationMax<1e-8))
//           ) return vYB;

        // to prevent rounding error
        if ((j==lastJ))
            return vYB;
        lastJ=j;

        // add linear constraint j to the active set:
        if (j>=0) { finished=0; flambda[j]=1.0; }
        // we are in a totally diffeent space => lambda for NLC are useless => reset
       // FullLambda.zero(nLC);
    }
    return vYB;
}

Vector ConstrainedL2NormMinimizer(InterPolynomial poly, int k, double delta,
                                   int *info, int iterMax, double *lambda1, Vector vOBase, ObjectiveFunction *of)
{
    int dim=poly.dim();
    Matrix mH(dim,dim);
    Vector vG(dim);
    poly.gradientHessian(poly.NewtonPoints[k],vG,mH);

    if (!of->isConstrained)
        return L2NormMinimizer(poly, poly.NewtonPoints[k], delta, info, iterMax, lambda1, vG, mH);

    return ConstrainedL2NormMinimizer(mH,vG,delta,info,iterMax,lambda1,vOBase+poly.NewtonPoints[k],of);
```

```
}

void projectionIntoFeasibleSpace(Vector vFrom, Vector vBase, ObjectiveFunction *of)
// result in vBase
{
    double epsilonStart=1e-1, epsilonStop=1e-10;

    int dim=vFrom.sz(), info;
    Matrix mH(dim,dim);
    Vector vG=vFrom.clone(), vD(dim);
    vBase.setSize(dim);

    vBase.zero();
    vG.multiply(-1.0);
    mH.diagonal(1.0);

    printf("Feasibility restoration phase \n");
    while (epsilonStart>epsilonStop)
    {
        vD=ConstrainedL2NormMinimizer(mH, vG, INF, &info, 1000, NULL, vBase, of, epsilonStart);
        vBase+=vD; vG+=vD;
        epsilonStart*=0.1;
        printf(".");
    }
    printf("\n");

    // loosen a bit the constraints:
    // int i;
    //for (i=0; i<of->nNLConstraints; i++)
    //    maxc=mmax(maxc,-of->evalNLConstraint(i,vBase));
}

Vector FullLambdaOld, vOldPos;
int standstill;
char checkForTermination(Vector d, Vector Base, double rhoEnd)
{
//    int i=FullLambda.sz();
//    double *fl=FullLambda; //, *flo;
    Vector vPos=d+Base;

    if ((vOldPos.sz()!=0)&&(vPos.euclidianDistance(vOldPos)>rhoEnd))
    {
        standstill=d.sz();
        vOldPos.copyFrom(vPos);
        return 0;
    }
    vOldPos.copyFrom(vPos);

    //if (FullLambda.mmax()<=0.0) return 0;
    //if (FullLambdaOld.sz()==0)
    //{
    //    standstill=d.sz();
    //    FullLambdaOld.setSize(FullLambda.sz());
    //    FullLambdaOld.zero();
    //}
    //
    //flo=FullLambdaOld;
    //while (i--)
    //{
    //    if (((flo[i]<=0.0)&&(fl[i]>0.0))||
    //        ((flo[i]>0.0)&&(fl[i]<=0.0)))
    //    {
    //        standstill=d.sz();
    //        FullLambdaOld.copyFrom(FullLambda);
    //        return 0;
    //    }
    //}
    if (FullLambda.mmax()>0.0)
    {
        standstill--;
        if (standstill==0) return 1;
    } else standstill=d.sz();
    return 0;
}

void initConstrainedStep(ObjectiveFunction *of)
{
    if (!of->isConstrained) return;

    FullLambda.setSize(of->dim()*2+of->A.nLine()+of->nNLConstraints);
    FullLambda.zero();
    mu=0.5;
    FullLambdaOld.setSize(0);
    vOldPos.setSize(0);
}
```

14.2.26 UTRSSolver.cpp (L2NormMinimizer)

```
// trust region step solver

#include <stdio.h>
```

```
#include <memory.h>

#include "Matrix.h"
#include "tools.h"
#include "Poly.h"

double findAlpha(Vector s,Vector u, double delta, Polynomial &q, Vector pointXk,Vector &output, Vector minusG, Matrix $
$H)
// find root (apha*) of equation L2norm(s+alpha u)=delta
// which makes q(s)=<g,s>+.5*<s,Hs> smallest
// output is (s+alpha* u)
{
    static Vector v1, v2, tmp;
    double a=0,b=0,c=-sqr(delta), *sp=s, *up=u;
    int n=s.sz();
    while (n--)
    {
        a+=sqr(*up); b+=*up * *sp; c+=sqr(*sp);
        sp++; up++;
    }
    double tmp1=-b/a, tmp2= sqrt(b*b-a*c)/a, q1, q2;

    n=s.sz();
    v1.setSize(n); v2.setSize(n); tmp.setSize(n);
    if (!(q==Polynomial::emptyPolynomial))
    {
        v1.copyFrom(u);
        v1.multiply(tmp1+tmp2);
        v1+=s;
        tmp.copyFrom(v1);
        tmp+=pointXk;
        q1=q(tmp);

    // !!! don't do this:
    //    output=v1;
    //    return tmp1+tmp2;

        v2.copyFrom(u);
        v2.multiply(tmp1-tmp2);
        v2+=s;
        tmp.copyFrom(v2);
        tmp+=pointXk;
        q2=q(tmp);

    } else
    {
        v1.copyFrom(u);
        v1.multiply(tmp1+tmp2);
        v1+=s;
        H.multiply(tmp,v1);
        q1=-minusG.scalarProduct(v1)+0.5*v1.scalarProduct(tmp);

        v2.copyFrom(u);
        v2.multiply(tmp1-tmp2);
        v2+=s;
        H.multiply(tmp,v2);
        q2=-minusG.scalarProduct(v2)+0.5*v2.scalarProduct(tmp);
    }
    if (q1>q2) { output=v1; return tmp1+tmp2; }
    output=v2; return tmp1-tmp2;
}

double initLambdaL(double normG,double delta, Matrix H)
{
    int n=H.nLine(),i,j;
    double **h=H, sum,l,a=INF;

    for (i=0; i<n; i++) a=mmin(a,h[i][i]);
    l=mmax(0.0,-a);

    a=0;
    for (i=0; i<n; i++)
    {
        sum=h[i][i];
        for (j=0; j<n; j++) if (j!=i) sum+=abs(h[i][j]);
        a=mmax(a,sum);
    }
    a=mmin(a,H.frobeniusNorm());
    a=mmin(a,H.LnftyNorm());

    l=mmax(l,normG/delta-a);
    return l;
}

double initLambdaU(double normG,double delta, Matrix H)
{
    int n=H.nLine(),i,j;
    double **h=H, sum,l,a=-INF;

    for (i=0; i<n; i++)
    {
        sum=-h[i][i];
        for (j=0; j<n; j++) if (j!=i) sum+=abs(h[i][j]);
        a=mmax(a,sum);
    }
    a=mmin(a,H.frobeniusNorm());
    a=mmin(a,H.LnftyNorm());

    l=mmax(0.0,normG/delta+a);
    return l;
}

double initLambdaU2(Matrix H)
{
    int n=H.nLine(),i,j;
    double **h=H, sum,a=-INF;

    for (i=0; i<n; i++)
    {
        sum=h[i][i];
        for (j=0; j<n; j++) if (j!=i) sum+=abs(h[i][j]);
        a=mmax(a,sum);
    }
    a=mmin(a,H.frobeniusNorm());
    return mmin(a,H.LnftyNorm());
}

// #define POWEL_TERMINATION 1

Vector L2NormMinimizer(Polynomial q, Vector pointXk, double delta,
                                        int *infoOut, int maxIter, double *lambda1, Vector minusG, Matrix H)
{
// lambda1>0.0 if interior convergence

    const double theta=0.01;
//    const double kappaEasy=0.1, kappaHard=0.2;
    const double kappaEasy=0.01, kappaHard=0.02;

    double normG,lambda,lambdaCorrection, lambdaPlus, lambdaL, lambdaU,
           uHu, alpha, normS;
    int info=0, n=minusG.sz();
    Matrix HLambda(n,n);
    MatrixTriangle L(n);
    Vector s(n), omega(n), u(n), sFinal;
    bool gIsNull, choleskyFactorAlreadyComputed=false;

//    printf("\nG= "); minusG.print();
//    printf("\nH=\n"); H.print();

    gIsNull=minusG.isNull();
    normG=minusG.euclidianNorm();
    lambda=normG/delta;
    minusG.multiply(-1.0);

    lambdaL=initLambdaL(normG,delta,H);
    lambdaU=initLambdaU(normG,delta,H);

    //    Special case: parl = paru.
    lambdaU= mmax(lambdaU,(1+kappaEasy)*lambdaL);

    lambda=mmax(lambda, lambdaL);
    lambda=mmin(lambda, lambdaU);

    while (maxIter--)
    {
        if (!choleskyFactorAlreadyComputed)
        {
            if (!H.cholesky(L, lambda, &lambdaCorrection))
            {
       //         lambdaL=mmax(mmax(lambdaL,lambda),lambdaCorrection);
                lambdaL=mmax(lambdaL,lambda+lambdaCorrection);
                lambda=mmax(sqrt(lambdaL*lambdaU), lambdaL+theta*(lambdaU-lambdaL));
                continue;
            }
        } else choleskyFactorAlreadyComputed=false;


        // cholesky factorization successfull : solve Hlambda * s = -G
        s.copyFrom(minusG);
        L.solveInPlace(s);
        L.solveTransposInPlace(s);
        normS=s.euclidianNorm();

        // check for termination
#ifndef POWEL_TERMINATION
        if (abs(normS-delta)<kappaEasy*delta)
        {
            s.multiply(delta/normS);
            info=1;
            break;
        }
#else
//      powell check !!!
        HLambda.copyFrom(H);
        HLambda.addUnityInPlace(lambda);
        double sHs=s.scalarProduct(HLambda.multiply(s));
        if (sqr(delta/normS-1)<kappaEasy*(1+lambda*delta*delta/sHs))
        {
            s.multiply(delta/normS);
            info=1;
            break;
        }
#endif

        if (normS<delta)
        {
            // check for termination
            // interior convergence; maybe break;
            if (lambda==0) { info=1; break; }
            lambdaU=mmin(lambdaU,lambda);
        } else lambdaL=mmax(lambdaL,lambda);

//         if (lambdaU-lambdaL<kappaEasy*(2-kappaEasy)*lambdaL) { info=3; break; };

        omega.copyFrom(s);
        L.solveInPlace(omega);
        lambdaPlus=lambda+(normS-delta)/delta*sqr(normS)/sqr(omega.euclidianNorm());
        lambdaPlus=mmax(lambdaPlus, lambdaL);
        lambdaPlus=mmin(lambdaPlus, lambdaU);

        if (normS<delta)
        {
            L.LINPACK(u);
#ifndef POWEL_TERMINATION
            HLambda.copyFrom(H);
            HLambda.addUnityInPlace(lambda);
#endif
            uHu=u.scalarProduct(HLambda.multiply(u));
            lambdaL=mmax(lambdaL,lambda-uHu);

            alpha=findAlpha(s,u,delta,q,pointXk,sFinal,minusG,H);
            // check for termination
#ifndef POWEL_TERMINATION
            if (sqr(alpha)*uHu<
                kappaHard*(s.scalarProduct(HLambda.multiply(s))   ))//  +lambda*sqr(delta)))
#else
            if (sqr(alpha)*uHu+sHs<
                kappaHard*(sHs+lambda*sqr(delta)))
#endif
            {
                s=sFinal; info=2; break;
            }
        }
        if ((normS>delta)&&(!gIsNull)) { lambda=lambdaPlus; continue; };

        if (H.cholesky(L, lambdaPlus, &lambdaCorrection))
        {
            lambda=lambdaPlus;
            choleskyFactorAlreadyComputed=true;
            continue;
        }

        lambdaL=mmax(lambdaL,lambdaPlus);
        // check lambdaL for interior convergence
//      if (lambdaL==0) return s;
        lambda=mmax(sqrt(lambdaL*lambdaU), lambdaL+theta*(lambdaU-lambdaL));
    }

    if (infoOut) *infoOut=info;
    if (lambda1)
    {
        if (lambda==0.0)
        {
            // calculate the value of the lowest eigenvalue of H
            // to check
            lambdaL=0; lambdaU=initLambdaU2(H);
            while (lambdaL<0.99*lambdaU)
            {
                lambda=0.5*(lambdaL+lambdaU);
                if (H.cholesky(L,-lambda)) lambdaL=lambda;
//                if (H.cholesky(L,-lambda,&lambdaCorrection)) lambdaL=lambda+lambdaCorrection;
                else lambdaU=lambda;
            }
            *lambda1=lambdaL;
        } else *lambda1=0.0;
    }
    return s;
}

Vector L2NormMinimizer(Polynomial q, Vector pointXk, double delta,
                                         int *infoOut, int maxIter, double *lambda1)
{
    int n=q.dim();
    Matrix H(n,n);
    Vector vG(n);
    q.gradientHessian(pointXk,vG,H);
    return L2NormMinimizer(q,pointXk,delta,infoOut,maxIter,lambda1,vG,H);
}
```

```
Vector L2NormMinimizer(Polynomial q, double delta,
                                        int *infoOut, int maxIter, double *lambda1)
{
    return L2NormMinimizer(q, Vector::emptyVector, delta, infoOut, maxIter, lambda1);
}
```

14.2.27 CNLSolver.cpp (QPOptim)

```
#include <stdio.h>
#include <memory.h>

//#include <crtdbg.h>

#include "ObjectiveFunction.h"
#include "Solver.h"
#include "Matrix.h"
#include "tools.h"
#include "KeepBests.h"
#include "IntPoly.h"
#include "parallel.h"
#include "MultInd.h"
#include "VectorChar.h"

// from CTRSSolver:
Vector ConstrainedL2NormMinimizer(InterPolynomial poly, int k,
                                  double delta, int *info, int iterMax, double *lambda1,
                                  Vector vOBase, ObjectiveFunction *of);
void projectionIntoFeasibleSpace(Vector vFrom, Vector vBase, ObjectiveFunction *of);
char checkForTermination(Vector d, Vector Base, double rhoEnd);
void initConstrainedStep(ObjectiveFunction *of);

int findBest(Matrix data, ObjectiveFunction *of)
{
    // find THE best point in the datas.
    int i=data.nLine(), k=-1, dim=data.nColumn()-1;
    if (i==0)
    {
        Vector b(dim);
        if (of->isConstrained)
        {
            Vector a(dim); a.zero();
            projectionIntoFeasibleSpace(a, b, of);
        }
        of->saveValue(b,of->eval(b));
        return 0;
    }

    double *p=*((double**)(data))+dim-1, best=INF;
    if (of->isConstrained)
    {
        Vector r(dim);
        int j;
        double best2=INF;
        while (i--)
        {
            data.getLine(i,r,dim);
            if (*p<best2) {j=i; best2=*p; }
            if (of->isFeasible(r)&&(*p<best)) { k=i; best=*p; }
            p+=dim+1;
        }

        if (k==-1)
        {
            data.getLine(j,r,dim);
            Vector b(dim);
            projectionIntoFeasibleSpace(r, b,of);
            if (!of->isFeasible(b,&best))
            {
                printf("unable to start (violation=%e).\n",best);
            }
            of->saveValue(b,of->eval(b));
            return data.nLine()-1;
        }
        return k;
    }

    while (i--)
    {
        if (*p<best) { k=i; best=*p; }
        p+=dim+1;
    }
    return k;
}

Vector *getFirstPoints(double **ValuesFF, int *np, double rho,
                       ObjectiveFunction *of)
{
    Matrix data=of->data;
    int k=findBest(data, of);
    if (k==-1)
    {
        printf("Matrix Data must at least contains one line.\n"); getchar(); exit(255);
    }
    int dim=data.nColumn()-1, n=(dim+1)*(dim+2)/2, nl=data.nLine(), i=nl,j=0;
    Vector Base=data.getLine(k,dim);
    double vBase=((double**)data)[k][dim];
    double *p,norm, *pb=Base;
    KeepBests kb(n*2,dim);
    Vector *points;
    double *valuesF;

    fprintf(stderr,"Value Objective=%e\n", vBase);

    while (j<n)
    {
        i=data.nLine(); kb.reset(); j=0;
        while (i--)
        {
            p=data[i];
            norm=0; k=dim;
            while (k--) norm+=sqr(p[k]-pb[k]);
            norm=sqrt(norm);
            if (norm<=2.001*rho) { kb.add(norm,p[dim], p); j++; }
        }
        if (j>=n)
        {
            // we have retained only the 2*n best points:
            j=mmin(j,2*n);
            points=new Vector[j];
            valuesF=(double*)malloc(j*sizeof(double));
            for (i=0; i<j; i++)
            {
                valuesF[i]=kb.getValue(i);
                points[i]=Vector(dim, kb.getOptValue(i));
            }
        } else
        {
            points=GenerateData(&valuesF, rho, Base, vBase, of);
            for (i=0; i<n-1; i++) of->saveValue(points[i],valuesF[i]);
            delete[] points;
            free(valuesF);
        }
    }
    *np=j;
    *ValuesFF=valuesF;
    return points;
}

int findK(double *ValuesF, int n, ObjectiveFunction *of, Vector *points)
{
    if (of->isConstrained) return 0;
    // find index k of the best value of the function
    double minimumValueF=INF;
    int i,k=-1;
    for (i=0; i<n; i++)
        if ((ValuesF[i]<minimumValueF)
            &&(of->isFeasible(points[i])))
        { k=i; minimumValueF=ValuesF[i]; }
    if (k==-1) k=0;
    return k;
}

void QPOptim( double rhoStart, double rhoEnd, int niter,
              ObjectiveFunction *of, int nnode)
{
    rhoStart=mmax(rhoStart,rhoEnd);
    int dim=of->dim(), n=(dim+1)*(dim+2)/2, info, k, t, nPtsTotal;
    double rho=rhoStart, delta=rhoStart, rhoNew,
           lambda1, normD=rhoEnd+1.0, modelStep, reduction, r, valueOF, valueFk, bound, noise;
    double *ValuesF;
    Vector Base, d, tmp, *points;
    bool improvement, forceTRStep=true, evalNeeded;

    initConstrainedStep(of);

    // pre-create the MultInd indexes to prevent multi-thread problems:
    cacheMultInd.get(dim,1); cacheMultInd.get(dim,2);

    parallelInit(nnode, dim, of);

    of->initDataFromXStart();
    if (of->isConstrained) of->initTolLC(of->xStart);

    points=getFirstPoints(&ValuesF, &nPtsTotal, rhoStart,of);

    fprintf(stderr,"init part 1 finished.\n");

    // find index k of the best (lowest) value of the function
    k=findK(ValuesF, nPtsTotal, of, points);
    Base=points[k].clone();
//    Base=of->xStart.clone();
    valueFk=ValuesF[k];
    // translation:
    t=nPtsTotal; while (t--) points[t]-=Base;
    // exchange index 0 and index k (to be sure best point is inside poly):
    tmp=points[k];
    points[k]=points[0];
    points[0]=tmp;
    ValuesF[k]=ValuesF[0];
    ValuesF[0]=valueFk;
    k=0;

    InterPolynomial poly(2, nPtsTotal, points, ValuesF );

    // update M:
    for (t=n; t<nPtsTotal; t++) poly.updateM(points[t], ValuesF[t]);

    fprintf(stderr,"init part 2 finished.\n");
    fprintf(stderr,"init finished.\n");

    // first of init all variables:
    parallelImprove(&poly, &k, rho, &valueFk, Base);

    // really start in parallel:
    startParallelThread();

    while (true)
    {
//        fprintf(stderr,"rho=%e; fo=%e; NF=%i\n", rho,valueFk,QP_NF);
        while (true)
        {
            // trust region step
            while (true)
            {
//                poly.print();
                parallelImprove(&poly, &k, rho, &valueFk, Base);

                niter--;
                if ((niter==0)
                   ||(of->isConstrained&&checkForTermination(poly.NewtonPoints[k], Base, rhoEnd)))
                {
                    Base+=poly.NewtonPoints[k];
                    fprintf(stderr,"rho=%e; fo=%e; NF=%i\n", rho,valueFk,of->nfe);
                    of->valueBest=valueFk;
                    of->xBest=Base;
                    return;
                }

                // to debug:
                fprintf(stderr,"Best Value Objective=%e (nfe=%i)\n", valueFk, of->nfe);

                d=ConstrainedL2NormMinimizer(poly,k,delta,&info,1000,&lambda1,Base,of);

//                if (d.euclidianNorm()>delta)
//                {
//                    printf("Warning d to long: (%e > %e)\n", d.euclidianNorm(), delta);
//                }

                normD=mmin(d.euclidianNorm(), delta);
                d+=poly.NewtonPoints[k];

//              next line is equivalent to reduction=valueFk-poly(d);
//              BUT is more precise (no rounding error)
                reduction=-poly.shiftedEval(d,valueFk);

                //if (normD<0.5*rho) { evalNeeded=true; break; }
                if ((normD<0.5*rho)&&(!forceTRStep)) { evalNeeded=true; break; }

                //  IF THE MODEL REDUCTION IS SMALL, THEN WE DO NOT SAMPLE FUNCTION
                //  AT THE NEW POINT. WE THEN WILL TRY TO IMPROVE THE MODEL.

                noise=0.5*mmax(of->noiseAbsolute*(1+of->noiseRelative), abs(valueFk)*of->noiseRelative);
                if ((reduction<noise)&&(!forceTRStep)) { evalNeeded=true; break; }
                forceTRStep=false; evalNeeded=false;

                tmp=Base+d; valueOF=of->eval(tmp); of->saveValue(tmp,valueOF); // of->updateCounter(valueFk);
                if (!of->isFeasible(tmp, &r))
                {
                    printf("violation: %e\n",r);
                }

                // update of delta:
                r=(valueFk-valueOF)/reduction;
                if (r<=0.1) delta=0.5*normD;
                else if (r<0.7) delta=mmax(0.5*delta, normD);
                     else delta=mmax(rho+ normD, mmax(1.25*normD, delta));
            // powell's heuristics:
                if (delta<1.5*rho) delta=rho;

                if (valueOF<valueFk)
                {
                    t=poly.findAGoodPointToReplace(-1, rho, d,&modelStep);
                    k=t; valueFk=valueOF;
                    improvement=true;
//                    fprintf(stderr,"Value Objective=%e\n", valueOF);
                } else
                {
                    t=poly.findAGoodPointToReplace(k, rho, d,&modelStep);
                    improvement=false;
//                    fprintf(stderr,".");
                };

                if (t<0) { poly.updateM(d, valueOF); break; }

                // If we are along constraints, it's more important to update
                // the polynomial with points which increase its quality.
                // Thus, we will skip this update to use only points coming
                // from checkIfValidityIsInBound

                if ((!of->isConstrained)||(improvement)||(reduction>0.0)||(normD<rho)) poly.replace(t, d, valueOF);

                if (improvement) continue;
//                if (modelStep>4*rho*rho) continue;
                if (modelStep>2*rho) continue;
                if (normD>=2*rho) continue;
                break;
            }
            // model improvement step
            forceTRStep=true;

//            fprintf(stderr,"improvement step\n");
            bound=0.0;
            if (normD<0.5*rho)
            {
                bound=0.5*sqr(rho)*lambda1;
                if (poly.nUpdateOfM<10) bound=0.0;
            }

            parallelImprove(&poly, &k, rho, &valueFk, Base);

            // !! change d (if needed):
            t=poly.checkIfValidityIsInBound(d, k, bound, rho );
            if (t>=0)
            {
                tmp=Base+d; valueOF=of->eval(tmp); of->saveValue(tmp,valueOF); // of->updateCounter(valueFk);
                poly.replace(t, d, valueOF);
                if ((valueOF<valueFk)&&
                    (of->isFeasible(tmp))) { k=t; valueFk=valueOF; };
                continue;
            }

            // the model is perfect for this value of rho:
            // OR
            // we have crossed a non_linear constraint which prevent us to advance
            if ((normD<=rho)||(reduction<0.0)) break;
        }

        // change rho because no improvement can now be made:
        if (rho<=rhoEnd) break;

        fprintf(stderr,"rho=%e; fo=%e; NF=%i\n", rho,valueFk,of->nfe);

        if (rho<16*rhoEnd) rhoNew=rhoEnd;
        else if (rho<250*rhoEnd) rhoNew=sqrt(rho*rhoEnd);
             else rhoNew=0.1*rho;
        delta=mmax(0.5*rho,rhoNew);
        rho=rhoNew;


        // update of the polynomial: translation of x[k].
        // replace BASE by BASE+x[k]
        if (!poly.NewtonPoints[k].equals(Vector::emptyVector))
        {
            Base+=poly.NewtonPoints[k];
            poly.translate(poly.NewtonPoints[k]);
        }
    }
    parallelFinish();

    if (evalNeeded)
    {
        tmp=Base+d; valueOF=of->eval(tmp); of->saveValue(tmp,valueOF); // of->updateCounter(valueFk);
        if (valueOF<valueFk) { valueFk=valueOF; Base=tmp; }
        else Base+=poly.NewtonPoints[k];
    } else Base+=poly.NewtonPoints[k];


//    delete[] points; :necessary: not done in destructor of poly which is called automatically:
    fprintf(stderr,"rho=%e; fo=%e; NF=%i\n", rho,valueFk,of->nfe);

    of->valueBest=valueFk;
    of->xBest=Base;
    of->finalize();
}
```

14.3 AMPL files

These files were NOT written be me.

14.3.1 hs022

```
var x {1..2};

minimize obj:
  (x[1] - 2)^2 + (x[2] - 1)^2
  ;

subject to constr1: -x[1]^2 + x[2] >= 0;
subject to constr2: x[1] + x[2] <= 2;

let x[1] := 2;
let x[2] := 2;

#printf "optimal solution as starting point \n";
#let x[1] := 1;
#let x[2] := 1;

#display obj - 1;
write ghs022;
```

14.3.2 hs023

```
var x {1..2} <= 50, >= -50;

minimize obj:
  x[1]^2 + x[2]^2
  ;

subject to constr1: x[1] + x[2] >= 1;
subject to constr2: x[1]^2 + x[2]^2 >= 1;
subject to constr3: 9*x[1]^2 + x[2]^2 >= 9;
subject to constr4: x[1]^2 - x[2] >= 0;
subject to constr5: x[2]^2 - x[1] >= 0;

let x[1] := 3;
let x[2] := 1;

#printf "optimal solution as starting point \n";
#let x[1] := 1;
#let x[2] := 1;

#display obj - 2;
write ghs023;
```

14.3.3 hs026

```
var x {1..3};

minimize obj:
  (x[1] - x[2])^2 + (x[2] - x[3])^4
  ;

subject to constr1: (1 + x[2]^2)*x[1] + x[3]^4 >= 3;

let x[1] := -2.6;
let x[2] :=  2;
let x[3] :=  2;

#printf "optimal solution as starting point \n";
#let x[1] :=  1;
#let x[2] :=  1;
#let x[3] :=  1;

#display obj;
#solve;
#display x;
#display obj;
#display obj - 0;

write ghs026;
```

14.3.4 hs034

```
var x {1..3} >= 0;

minimize obj:
  -x[1]
  ;
```

```
subject to constr1: x[2] >= exp(x[1]);
subject to constr2: x[3] >= exp(x[2]);
subject to constr3: x[1] <= 100;
subject to constr4: x[2] <= 100;
subject to constr5: x[3] <= 10;

let x[1] := 0;
let x[2] := 1.05;
let x[3] := 2.9;

#printf "optimal solution as starting point \n";
#let x[1] := 0.83403;
#let x[2] := 2.30258;
#let x[3] := 10;

#display obj + log(log(10));

write ghs034;
```

14.3.5 hs038

```
var x {1..4} >= -10, <= 10;

minimize obj:
  100*(x[2]-x[1]^2)^2 + (1-x[1])^2 + 90*(x[4]-x[3]^2)^2 + (1-x[3])^2
  + 10.1*( (x[2]-1)^2 + (x[4]-1)^2 ) + 19.8*(x[2]-1)*(x[4]-1)
  ;

subject to constr1: x[1] + 2*x[2] + 2*x[3] <= 72;
subject to constr2: x[1] + 2*x[2] + 2*x[3] >= 0;

let x[1] := -3;
let x[2] := -1;
let x[3] := -3;
let x[4] := -1;

#printf "optimal solution as starting point \n";
#let x[1] := 1;
#let x[2] := 1;
#let x[3] := 1;
#let x[4] := 1;

#display obj - 0;

write ghs038;
```

14.3.6 hs044

```
var x {1..4} >= 0;

minimize obj:
  x[1] - x[2] - x[3] - x[1]*x[3] + x[1]*x[4] + x[2]*x[3] - x[2]*x[4]
  ;

subject to constr1: x[1] + 2*x[2] <= 8;
subject to constr2: 4*x[1] + x[2] <= 12;
subject to constr3: 3*x[1] + 4*x[2] <= 12;
subject to constr4: 2*x[3] + x[4] <= 8;
subject to constr5: x[3] + 2*x[4] <= 8;
subject to constr6: x[3] + x[4] <= 5;

let x[1] := 0;
let x[2] := 0;
let x[3] := 0;
let x[4] := 0;

#printf "optimal solution as starting point \n";
#let x[1] := 0;
#let x[2] := 3;
#let x[3] := 0;
#let x[4] := 4;

#display obj + 15;

write ghs044;
```

14.3.7 hs065

```
var x {1..3};

minimize obj:
  (x[1] - x[2])^2 + (x[1] + x[2] - 10)^2/9 + (x[3] - 5)^2
  ;

subject to constr1: x[1]^2 + x[2]^2 + x[3]^2 <= 48;
subject to constr2: -4.5 <= x[1] <= 4.5;
subject to constr3: -4.5 <= x[2] <= 4.5;
subject to constr4:   -5 <= x[3] <=   5;

let x[1] := -5;
let x[2] :=  5;
let x[3] :=  0;

#printf "optimal solution as starting point \n";
#let x[1] := 3.650461821;
#let x[2] := 3.65046168;
#let x[3] := 4.6204170507;

#display obj - 0.9535288567;

write ghs065;
```

14.3.8 hs076

```
var x {j in 1..4} >= 0;

minimize obj:
  x[1]^2 + 0.5*x[2]^2 + x[3]^2 + 0.5*x[4]^2 - x[1]*x[3] + x[3]*x[4]
  - x[1] - 3*x[2] + x[3] - x[4]
  ;

subject to constr1: x[1] + 2*x[2] + x[3] + x[4] <= 5;
subject to constr2: 3*x[1] + x[2] + 2*x[3] - x[4] <= 4;
subject to constr3: x[2] + 4*x[3] >= 1.5;

data;

let x[1] := 0.5;
let x[2] := 0.5;
let x[3] := 0.5;
let x[4] := 0.5;

#printf "optimal solution as starting point \n";
#let x[1] := 0.2727273;
#let x[2] := 2.090909;
#let x[3] :=  -0.26e-10;
#let x[4] := 0.5454545;

data;

#display obj + 4.681818181;

write ghs076;
```

14.3.9 hs100

```
var x {1..7};

minimize obj:
  (x[1]-10)^2 + 5*(x[2]-12)^2 + x[3]^4 + 3*(x[4]-11)^2 + 10*x[5]^6
  + 7*x[6]^2 + x[7]^4 - 4*x[6]*x[7] - 10*x[6] - 8*x[7]
  ;

subject to constr1: 2*x[1]^2 + 3*x[2]^4 + x[3] + 4*x[4]^2 + 5*x[5] <= 127;
subject to constr2: 7*x[1] + 3*x[2] + 10*x[3]^2 + x[4] - x[5] <= 282;
subject to constr3: 23*x[1] + x[2]^2 + 6*x[6]^2 - 8*x[7] <= 196;
subject to constr4: -4*x[1]^2 - x[2]^2 + 3*x[1]*x[2] -2*x[3]^2 - 5*x[6]
                        +11*x[7] >= 0;

data;

let x[1] := 1;
let x[2] := 2;
let x[3] := 0;
let x[4] := 4;
let x[5] := 0;
let x[6] := 1;
let x[7] := 1;

#printf "optimal solution as starting point \n";
#let x[1] := 2.330499;
#let x[2] := 1.951372;
#let x[3] := -0.4775414;
#let x[4] := 4.365726;
#let x[5] := 1.038131;
#let x[6] := -0.6244870;
#let x[7] := 1.594227;

#display obj - 680.6300573;

write ghs100;
```

14.3.10 hs106

```
# hs106.mod     LQR2-MN-8-22
# Original AMPL coding by Elena Bobrovnikova (summer 1996 at Bell Labs).

# Heat exchanger design

# Ref.: W. Hock and K. Schittkowski, Test Examples for Nonlinear Programming
# Codes.  Lecture Notes in Economics and Mathematical Systems, v. 187,
# Springer-Verlag, New York, 1981, p. 115.

# Number of variables: 8
# Number of constraints:  22
# Objective linear
# Nonlinear constraints

param N integer, := 8;
set I := 1..N;

param a >= 0;
param b >= 0;
param c >= 0;
param d >= 0;
param e >= 0;
param f >= 0;
param g >= 0;
param h >= 0;

var x{I};

minimize obj:
     x[1] + x[2] + x[3];

s.t. c1: 1 - a * (x[4] + x[6]) >= 0;
s.t. c2: 1 - a * (x[5] + x[7] - x[4]) >= 0;
s.t. c3: 1 - b * (x[8] - x[5]) >= 0;
s.t. c4: x[1] * x[6] - c * x[4] - d * x[1] + e >= 0;
s.t. c5: x[2] * x[7] - f * x[5] - x[2] * x[4] + f * x[4] >= 0;
s.t. c6: x[3] * x[8] - g - x[3] * x[5] + h * x[5] >= 0;
s.t. c7: 100 <= x[1] <= 10000;
s.t. c8 {i in {2,3}}: 1000 <= x[i] <= 10000;
s.t. c9 {i in 4..8}: 10 <= x[i] <= 1000;


data;
param a := 0.0025;
param b := 0.01;
param c := 833.3325;
param d := 100;
param e := 83333.33;
param f := 1250;
param g := 1250000;
param h := 2500;

var x :=
    1  5000   2  5000   3  5000   4  200   5  350   6  150   7  225   8  425;

#printf "optimal solution as starting point \n";
#var x :=
#    1  579.3167
#    2  1359.943
#    3  5110.071
#    4  182.0174
#    5  295.5985
#    6  217.9799
#    7  286.4162
#    8  395.5979
#   ;

#display obj - 7049.330923;

write ghs106;
```

14.3.11 hs108

```
var x {1..9};

minimize obj:
   -.5*(x[1]*x[4]-x[2]*x[3]+x[3]*x[9]-x[5]*x[9]+x[5]*x[8]-x[6]*x[7]);

s.t. c1:  1-x[3]^2-x[4]^2>=0;
s.t. c2:  1-x[5]^2-x[6]^2>=0;
s.t. c3:  1-x[9]^2>=0;
s.t. c4:  1-x[1]^2-(x[2]-x[9])^2>=0;
s.t. c5:  1-(x[1]-x[5])^2-(x[2]-x[6])^2>=0;
s.t. c6:  1-(x[1]-x[7])^2-(x[2]-x[8])^2>=0;
s.t. c7:  1-(x[3]-x[7])^2-(x[4]-x[8])^2>=0;
s.t. c8:  1-(x[3]-x[5])^2-(x[4]-x[6])^2>=0;
s.t. c9:  1-x[7]^2-(x[8]-x[9])^2>=0;
s.t. c10: x[1]*x[4]-x[2]*x[3]>=0;
s.t. c11: x[3]*x[9]>=0;
s.t. c12: -x[5]*x[9]>=0;
s.t. c13: x[5]*x[8]-x[6]*x[7]>=0;
s.t. c14: x[9]>=0;
```

```
data;

let x[1] := 1;
let x[2] := 1;
let x[3] := 1;
let x[4] := 1;
let x[5] := 1;
let x[6] := 1;
let x[7] := 1;
let x[8] := 1;
let x[9] := 1;

#let x[1] := 0.8841292;
#let x[2] := 0.4672425;
#let x[3] := 0.03742076;
#let x[4] := 0.9992996;
#let x[5] := 0.8841292;
#let x[6] := 0.4672425;
#let x[7] := 0.03742076;
#let x[8] := 0.9992996;
#let x[9] := 0;

#display obj+.8660254038;

write ghs108;
```

14.3.12 hs116

```
# hs116.mod     LQR2-MN-13-41
# Original AMPL coding by Elena Bobrovnikova (summer 1996 at Bell Labs).

# 3-stage membrane separation

# Ref.: W. Hock and K. Schittkowski, Test Examples for Nonlinear Programming
# Codes.  Lecture Notes in Economics and Mathematical Systems, v. 187,
# Springer-Verlag, New York, 1981, p. 124.

# Number of variables: 13
# Number of constraints: 41
# Objective linear
# Nonlinear constraints


param N > 0 integer, := 13;
set I := 1 .. N;

var x {i in I} >= 0;

param a > 0;
param b > 0;
param c > 0;
param d > 0;
param e > 0;
param f > 0;


minimize obj:
     x[11] + x[12] + x[13];

s.t. c1: x[3] - x[2] >= 0;
s.t. c2: x[2] - x[1] >= 0;
s.t. c3: 1 - a * x[7] + a * x[8] >= 0;
s.t. c4: x[11] + x[12] + x[13] >= 50;
s.t. c5: x[13] - b * x[10] + c * x[3] * x[10] >= 0;
s.t. c6: x[5] - d * x[2] - e * x[2] * x[5] + f * x[2]^2 >= 0;
s.t. c7: x[6] - d * x[3] - e * x[3] * x[6] + f * x[3]^2 >= 0;
s.t. c8: x[4] - d * x[1] - e * x[1] * x[4] + f * x[1]^2 >= 0;
s.t. c9: x[12] - b * x[9] + c * x[2] * x[9] >= 0;
s.t. c10: x[11] - b * x[8] + c * x[1] * x[8] >= 0;
s.t. c11: x[5] * x[7] - x[1] * x[8] - x[4] * x[7] + x[4] * x[8] >= 0;
s.t. c12: 1 - a * (x[2] * x[9] + x[5] * x[8] - x[1] * x[8] - x[6] * x[9]) -
          x[5] - x[6] >= 0;
s.t. c13: x[2] * x[9] - x[3] * x[10] - x[6] * x[9] - 500 * x[2] +
          500 * x[6] + x[2] * x[10] >= 0;
s.t. c14: x[2] - 0.9 - a * (x[2] * x[10] - x[3] * x[10]) >= 0;
s.t. c15: x[11] + x[12] + x[13] <= 250;

s.t. b1: 0.1 <= x[1] <= 1;
s.t. b2: 0.1 <= x[2] <= 1;
s.t. b3: 0.1 <= x[3] <= 1;
s.t. b4: 0.0001 <= x[4] <= 0.1;
s.t. b5: 0.1 <= x[5] <= 0.9;
s.t. b6: 0.1 <= x[6] <= 0.9;
s.t. b7: 0.1 <= x[7] <= 1000;
s.t. b8: 0.1 <= x[8] <= 1000;
s.t. b9: 500 <= x[9] <= 1000;
s.t. b10: 0.1 <= x[10] <= 500;
s.t. b11: 1 <= x[11] <= 150;
s.t. b12: 0.0001 <= x[12] <= 150;
s.t. b13: 0.0001 <= x[13] <= 150;
```

```
data;
param a := 0.002;
param b := 1.262626;
param c := 1.231059;
param d := 0.03475;
param e := 0.975;
param f := 0.00975;
var x :=
    1 0.5  2 0.8  3 0.9  4 0.1  5 0.14  6 0.5  7 489  8 80  9 650
   10 450  11 150  12 150  13 150;

#display obj - 97.588409;

write ghs116;
```

14.3.13 hs268

```
# AMPL Model by Hande Y. Benson
#
# Copyright (C) 2001 Princeton University
# All Rights Reserved
#
# Permission to use, copy, modify, and distribute this software and
# its documentation for any purpose and without fee is hereby
# granted, provided that the above copyright notice appear in all
# copies and that the copyright notice and this
# permission notice appear in all supporting documentation.

#   Source:
#   K. Schittkowski
#   "More Test Examples for Nonlinear Programming Codes"
#   Springer Verlag, Berlin, Lecture notes in economics and
#   mathematical systems, volume 282, 1987

#   SIF input: Michel Bierlaire and Annick Sartenaer, October 1992.
#              minor correction by Ph. Shott, Jan 1995.

#   classification QLR2-AN-5-5

param D{1..5, 1..5};
param B{1..5};
var x{1..5} := 1.0;

minimize f:
        14463.0 + sum {i in 1..5, j in 1..5} D[i,j]*x[i]*x[j]
        + -2*sum {i in 1..5} (B[i]*x[i]);
subject to cons1:
        -sum {i in 1..5} x[i] + 5>= 0;
subject to cons2:
        10*x[1]+10*x[2]-3*x[3]+5*x[4]+4*x[5] -20 >= 0;
subject to cons3:
        -8*x[1]+x[2]-2*x[3]-5*x[4]+3*x[5] + 40 >= 0;
subject to cons4:
        8*x[1]-x[2]+2*x[3]+5*x[4]-3*x[5] -11>= 0;
subject to cons5:
        -4*x[1]-2*x[2]+3*x[3]-5*x[4]+x[5] +30>= 0;

data;
param B:=
1       -9170
2       17099
3       -2271
4       -4336
5       -43;
param D:
        1       2       3       4       5:=
1       10197   -12454  -1013   1948    329
2       -12454  20909   -1733   -4914   -186
3       -1013   -1733   1755    1089    -174
4       1948    -4914   1089    1515    -22
5       329     -186    -174    -22     27;

#
# optimal solution:
# x1= 1
# x2= 2
# x3= -1
# x4= 3
# x5= -4
#

write ghs268;
```

Bibliography

[BCD$^+$95] Andrew J. Booker, A.R. Conn, J.E. Dennis Jr., Paul D. Frank, Michael Trosset, Virginia Torczon, and Michael W. Trosset. Global modeling for optimization: Boeing/ibm/rice collaborative project 1995 final report. Technical Report ISSTECH-95-032, Boeing Information Support Services, Research and technology, Box 3707, M/S 7L-68, Seattle, Washington 98124, December 1995.

[BDBVB00] Edy Bertolissi, Antoine Duchâteau, Hugues Bersini, and Frank Vanden Berghen. Direct Adaptive Fuzzy Control for MIMO Processes. In *FUZZ-IEEE 2000 conference*, San Antonio, Texas, May 2000.

[BDF$^+$98] Andrew J. Booker, J.E. Dennis Jr., Paul D. Frank, David B. Serafini, Virginia Torczon, and Michael W. Trosset. Optimization using surrogate objectives on a helicopter test example. *Computational Methods in Optimal Design and Control*, pages 49–58, 1998.

[BDF$^+$99] Andrew J. Booker, J.E. Dennis Jr., Paul D. Frank, David B. Serafini, Virginia Torczon, and Michael W. Trosset. A rigorous framework for optimization of expensive functions by surrogates. *Structural Optimization*, 17, No. 1:1–13, February 1999.

[BK97] D. M. Bortz and C. T. Kelley. The Simplex Gradient and Noisy Optimization Problems. Technical Report CRSC-TR97-27, North Carolina State University, Department of Mathematics, Center for Research in Scientific Computation Box 8205, Raleigh, N. C. 27695-8205, September 1997.

[BSM93] Mokhtar S. Bazaraa, Hanif D. Sherali, and Shetty C. M. *Nonlinear Programming: Theory and Algorithms, 2nd Edition.* Weiley Text Books, 1993.

[BT96] Paul T. Boggs and Jon W. Tolle. Sequential Quadratic Programming. *Acta Numerica*, pages 1–000, 1996.

[BV04] Stephen Boyd and Lieven Vandenberghe. *Convex Optimization.* Press syndicate of the University of Cambridge, cambridge university press edition, 2004.

[CAVDB01] R. Cosentino, Z. Alsalihi, and R. Van Den Braembussche. Expert System for Radial Impeller Optimisation. In *Fourth European Conference on Turbomachinery, ATI-CST-039/01*, Florence,Italy, 2001.

[CGP$^+$01] R. G. Carter, J. M. Gablonsky, A. Patrick, C. T. Kelley, and O. J. Eslinger. Algorithms for Noisy Problems in Gas Transmission Pipeline Optimization. *Optimization and Engineering*, 2:139–157, 2001.

[CGT92] Andrew R. Conn, Nicholas I.M. Gould, and Philippe L. Toint. *LANCELOT: a Fortran package for large-scale non-linear optimization (Release A)*. Springer Verlag, HeidelBerg, Berlin, New York, springer series in computational mathematics edition, 1992.

[CGT00a] Andrew R. Conn, Nicholas I.M. Gould, and Philippe L. Toint. *Trust-region Methods*. SIAM Society for Industrial & Applied Mathematics, Englewood Cliffs, New Jersey, mps-siam series on optimization edition, 2000.

[CGT00b] Andrew R. Conn, Nicholas I.M. Gould, and Philippe L. Toint. *Trust-region Methods*. SIAM Society for Industrial & Applied Mathematics, Englewood Cliffs, New Jersey, mps-siam series on optimization edition, 2000. Chapter 9: conditional model, pp. 307–323.

[CGT00c] Andrew R. Conn, Nicholas I.M. Gould, and Philippe L. Toint. *Trust-region Methods*. SIAM Society for Industrial & Applied Mathematics, Englewood Cliffs, New Jersey, mps-siam series on optimization edition, 2000. The ideal Trust Region: pp. 236–237.

[CGT00d] Andrew R. Conn, Nicholas I.M. Gould, and Philippe L. Toint. *Trust-region Methods*. SIAM Society for Industrial & Applied Mathematics, Englewood Cliffs, New Jersey, mps-siam series on optimization edition, 2000. Note on convex models, pp. 324–337.

[CGT00e] Andrew R. Conn, Nicholas I.M. Gould, and Philippe L. Toint. *Trust-region Methods*. SIAM Society for Industrial & Applied Mathematics, Englewood Cliffs, New Jersey, mps-siam series on optimization edition, 2000. Chapter 12: Projection Methods for Convex Constraints, pp441–489.

[CGT00f] Andrew R. Conn, Nicholas I.M. Gould, and Philippe L. Toint. *Trust-region Methods*. SIAM Society for Industrial & Applied Mathematics, Englewood Cliffs, New Jersey, mps-siam series on optimization edition, 2000. Chapter 13: Barrier Methods for Inequality Constraints.

[CGT00g] Andrew R. Conn, Nicholas I.M. Gould, and Philippe L. Toint. *Trust-region Methods*. SIAM Society for Industrial & Applied Mathematics, Englewood Cliffs, New Jersey, mps-siam series on optimization edition, 2000. Chapter 7.7: Norms that reflect the underlying Geometry, pp. 236–242.

[CGT98] Andrew R. Conn, Nicholas I.M. Gould, and Philippe L. Toint. A Derivative Free Optimization Algorithm in Practice. Technical report, Department of Mathematics, University of Namur, Belgium, 98. Report No. 98/11.

[CGT99] Andrew R. Conn, Nicholas I.M. Gould, and Philippe L. Toint. Sqp methods for large-scale nonlinear programming. Technical report, Department of Mathematics, University of Namur, Belgium, 99. Report No. 1999/05.

[Col73] A.R. Colville. A comparative study of nonlinear progamming code. Technical report, IBM, New York, 1973. Scientific center report 320-2949.

[CST97] Andrew R. Conn, K. Scheinberg, and Philippe L. Toint. Recent progress in unconstrained nonlinear optimization without derivatives. *Mathematical Programming*, 79:397–414, 1997.

[DB98] Carl De Boor. *A Practical Guide to Splines (revised edition).* Springer-Verlag, 1998.

[DBAR90] Carl De Boor and A. A Ron. On multivariate polynomial interpolation. *Constr. Approx.*, 6:287–302, 1990.

[DBAR98] Carl De Boor and A. A Ron. Box Splines. *Applied Mathematical Sciences*, 1998.

[DS96] J.E. Dennis Jr. and Robert B. Schnabel. *Numerical Methods for unconstrained Optimization and nonlinear Equations.* SIAM Society for Industrial & Applied Mathematics, Englewood Cliffs, New Jersey, classics in applied mathematics, 16 edition, 1996.

[DT91] J.E. Dennis Jr. and V. Torczon. Direct search methods on parallel machines. *SIAM J. Optimization*, 1(4):448–474, 1991.

[FGK02] Robert Fourer, David M. Gay, and Brian W. Kernighan. *AMPL: A Modeling Language for Mathematical Programming.* Duxbury Press / Brooks/Cole Publishing Company, 2002.

[Fle87] R. Fletcher. *Practical Methods of optimization.* a Wiley-Interscience publication, Great Britain, 1987.

[GK95] P. Gilmore and C. T. Kelley. An implicit filtering algorithm for optimization of functions with many local minima. *SIAM Journal of Optimization*, 5:269–285, 1995.

[GMSM86] P.E. Gill, W. Murray, M.A. Saunders, and Wright M.H. Users's guide for npsol (version 4.0): A fortran package for non-linear programming. Technical report, Department of Operations Research, Stanford University, Stanford, CA94305, USA, 1986. Report SOL 862.

[GOT01] Nicholas I. M. Gould, Dominique Orban, and Philippe L. Toint. CUTEr (and SifDec), a Constrained and Unconstrained Testing Environment, revisited*. Technical report, Cerfacs, 2001. Report No. TR/PA/01/04.

[GVL96] Gene H. Golub and Charles F. Van Loan. *Matrix Computations, third edition.* Johns Hopkins University Press, Baltimore, USA, 1996.

[HS81] W. Hock and K. Schittkowski. Test Examples for Nonlinear Programming Codes. *Lecture Notes en Economics and Mathematical Systems*, 187, 1981.

[Kel99] C. T. Kelley. *Iterative Methods for Optimization*, volume 18 of *Frontiers in Applied Mathematics.* SIAM, Philadelphia, 1999.

[KLT97] Tamara G. Kolda, Rober Michael Lewis, and Virginia Torczon. Optimization by Direct Search: New Perspectives on Some Classical and Model Methods. *Siam Review*, 45 , N°3:385–482, 1997.

[Lor00] R.A. Lorentz. Multivariate Hermite interpolation by algebraic polynomials: A survey. *Journal of computation and Applied Mathematics*, 12:167–201, 2000.

[MS83] J.J. Moré and D.C. Sorensen. Computing a trust region step. *SIAM journal on scientif and statistical Computing*, 4(3):553–572, 1983.

[Mye90] Raymond H. Myers. *Classical and Modern regression with applications.* PWS-Kent Publishing Company, Boston, the duxbury advanced series in statistics and decision sciences edition, 1990.

[Noc92] Jorge Nocedal. Theory of Algorithm for Unconstrained Optimization. *Acta Numerica*, pages 199–242, 1992.

[NW99] Jorge Nocedal and Stephen J. Wright. *Numerical Optimization.* Springer Verlag, spinger series in operations research edition, 1999.

[Pal69] J.R. Palmer. An improved procedure for orthogonalising the search vectors in rosenbrock's and swann's direct search optimisation methods. *The Computer Journal*, 12, Issue 1:69–71, 1969.

[PMM$^+$03] S. Pazzi, F. Martelli, V. Michelassi, Frank Vanden Berghen, and Hugues Bersini. Intelligent Performance CFD Optimisation of a Centrifugal Impeller. In *Fifth European Conference on Turbomachinery*, Prague, CZ, March 2003.

[Pol00] C. Poloni. Multi Objective Optimisation Examples: Design of a Laminar Airfoil and of a Composite Rectangular Wing. *Genetic Algorithms for Optimisation in Aeronautics and Turbomachinery*, 2000. von Karman Institute for Fluid Dynamics.

[Pow77] M.J.D. Powell. A fast algorithm for nonlinearly constrained optimization calculations. *Numerical Analysis, Dundee*, 630:33–41, 1977. Lecture Notes in Mathematics, Springer Verlag, Berlin.

[Pow94] M.J.D. Powell. A direct search optimization method that models the objective and constraint functions by linar interpolation. In *Advances in Optimization and Numerical Analysis, Proceedings of the sixth Workshop on Optimization and Numerical Analysis, Oaxaca, Mexico*, volume 275, pages 51–67, Dordrecht, NL, 1994. Kluwer Academic Publishers.

[Pow97] M.J.D. Powell. The use of band matrices for second derivative approximations in trust region algorithms. Technical report, Department of Applied Mathematics and Theoretical Physics, University of Cambridge, England, 1997. Report No. DAMTP1997/NA12.

[Pow00] M.J.D. Powell. UOBYQA: Unconstrained Optimization By Quadratic Approximation. Technical report, Department of Applied Mathematics and Theoretical Physics, University of Cambridge, England, 2000. Report No. DAMTP2000/14.

[Pow02] M.J.D. Powell. Least Frobenius norm updating of quadratic models that satisfy interpolation conditions. Technical report, Department of Applied Mathematics and Theoretical Physics, University of Cambridge, England, 2002. Report No. DAMTP2002/NA08.

[Pow04] M.J.D. Powell. On updating the inverse of a KKT matrix. Technical report, Department of Applied Mathematics and Theoretical Physics, University of Cambridge, England, 2004. Report No. DAMTP2004/01.

[PT93] E.R. Panier and A.L. Tits. On Combining Feasibility, Descent and Superlinear convergence in Inequality Constrained Optimization. *Math. Programming*, 59:261–276, 1993.

[PT95] Eliane R. Panier and André L. Tits. On combining feasibility, Descent and Superlinear Convergence in Inequality Contrained Optimization. *Mathematical Programming*, 59:261–276, 1995.

[PTVF99] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C (second edition)*. Cambridge University Press, 1999.

[PVdB98] Stéphane Pierret and René Van den Braembussche. Turbomachinery blade design using a Navier-Stokes solver and artificial neural network. *Journal of Turbomachinery*, ASME 98-GT-4, 1998. publication in the transactions of the ASME: " Journal of Turbomachinery ".

[Ros60] H.H. Rosenbrock. An automatic method for finding the greatest or least value of a function. *The Computer Journal*, 3, Issue 3:175–184, 1960.

[RP63] Fletcher R. and M.J.D. Powell. A rapidly convergent descent method for minimization. *Comput. J.*, 8:33–41, 1963.

[Sau95] Thomas Sauer. Computational aspects of multivariate polynomial interpolation. *Advances Comput. Math*, 3:219–238, 1995.

[SBT$^+$92] D. E. Stoneking, G. L. Bilbro, R. J. Trew, P. Gilmore, and C. T. Kelley. Yield optimization Using a gaAs Process Simulator Coupled to a Physical Device Model. *IEEE Transactions on Microwave Theory and Techniques*, 40:1353–1363, 1992.

[Sch77] Hans-Paul Schwefel. *Numerische Optimierung von Computer–Modellen mittels der Evolutionsstrategie*, volume 26 of *Interdisciplinary Systems Research*. Birkh'́auser, Basle, 1977.

[SP99] Thomas Sauer and J.M. Pena. On the multivariate Horner scheme. *SIAM J. Numer. Anal.*, 1999.

[SX95] Thomas Sauer and Yuan Xu. On multivariate lagrange interpolation. *Math. Comp.*, 64:1147–1170, 1995.

[VB04] Frank Vanden Berghen. Optimization algorithm for Non-Linear, Constrained, Derivative-free optimization of Continuous, High-computing-load Functions. Technical report, IRIDIA, Université Libre de Bruxelles, Belgium, 2004. Available at http://iridia.ulb.ac.be/∼fvandenb/work/thesis/.