Université Libre de Bruxelles
Faculté des Sciences Appliquées
Service: IRIDIA

Année académique
2002-2003

Intermediate Report on the development of an optimization code for smooth, continuous objective functions when derivatives are not available

Directeur de Mémoire :
Pr. H. Bersini

Travail de Spécialisation présenté par
Frank Vanden Berghen en vue de l'obtention
du DEA en Sciences Appliquées.

I first want to thank Pr. Dr. Ir. Hugues Bersini, the coordinator of my research and head of the IRIDIA department at the University of Brussels.

I also thank the members of the jury for all the attention they will put in the lecture of my work.

I also want to thank my lovely fiancée, Sabrina Wenig and Frédéric Schoepp for their patience and their re-lecture and correction of my work.

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

– Albert Einstein

Contents

1 Introduction **7**
- 1.1 Motivations . . . 8
- 1.2 Formal description . . . 8
- 1.3 Trust Region and Line-search Methods. . . . 10
  - 1.3.1 Conventions . . . 10
  - 1.3.2 General principle . . . 10
  - 1.3.3 Notion of Speed of convergence. . . . 11
  - 1.3.4 A simple line-search method: The Newton's method. . . . 11
  - 1.3.5 B_k must be positive definite for Line-search methods. . . . 12
  - 1.3.6 Why is Newton's method crucial: Dennis-Moré theorem . . . 12
- 1.4 A simple trust-region algorithm. . . . 17
- 1.5 The basic trust-region algorithm (BTR). . . . 18
- 1.6 About the QP_TR algorithm. . . . 19
- Bibliography . . . 21

2 Multivariate Lagrange Interpolation **22**
- 2.1 Introduction . . . 22
- 2.2 A small reminder about univariate interpolation. . . . 23
  - 2.2.1 Lagrange interpolation . . . 23
  - 2.2.2 Newton interpolation . . . 24
  - 2.2.3 The divided difference for the Newton form. . . . 24
  - 2.2.4 The Horner scheme . . . 26
- 2.3 Multivariate Lagrange interpolation. . . . 27
  - 2.3.1 The Lagrange polynomial basis $\{P_1(x), \ldots, P_N(x)\}$. . . . 27
  - 2.3.2 The Lagrange interpolation polynomial $L(x)$. . . . 28
  - 2.3.3 The multivariate Horner scheme . . . 28
- 2.4 The Lagrange Interpolation inside the optimization loop. . . . 30
  - 2.4.1 A bound on the interpolation error. . . . 31
  - 2.4.2 Validity of the interpolation in a radius of ρ around $\boldsymbol{x}_{(k)}$. . . . 32
  - 2.4.3 Find a good point to replace in the interpolation. . . . 33
  - 2.4.4 Replace the interpolation point $\boldsymbol{x}_{(t)}$ by a new point X. . . . 33
  - 2.4.5 Generation of the first set of point $\{\boldsymbol{x}_{(1)}, \ldots, \boldsymbol{x}_{(N)}\}$. . . . 34
  - 2.4.6 Translation of a polynomial. . . . 34
- Bibliography . . . 35

3 The Trust-Region subproblem **37**
3.1 $H(\lambda^*)$ must be positive definite. . . . 37
3.2 Explanation of the Hard case. . . . 39
3.2.1 Convex example. . . . 40
3.2.2 Non-Convex example. . . . 41
3.2.3 The hard case. . . . 41
3.3 Finding the root of $\|s(\lambda)\|_2 - \Delta = 0$. . . . 42
3.4 Starting and safe-guarding Newton's method 43
3.5 How to pick λ inside $[\lambda_L \lambda_U]$? 44
3.6 Initial values of λ_L and λ_U 44
3.7 How to find a good approximation of u_1 : LINPACK METHOD 45
3.8 The Rayleigh quotient trick 46
3.9 Termination Test. . . . 47
3.9.1 $s(\lambda)$ is near the boundary of the trust region: normal case 48
3.9.2 $s(\lambda)$ is inside the trust region: hard case 48
3.10 An estimation of the slope of $q(x)$ at the origin. . . . 48
Bibliography 49

4 The secondary Trust-Region subproblem **50**
4.1 Generating $\tilde{s}$. . . . 51
4.2 Generating $\hat{u}$ and $\tilde{u}$ from $\hat{s}$ and $\tilde{s}$ 52
4.3 Generating the final s from $\hat{u}$ and $\tilde{u}$. . . . 52
4.4 About the choice of $\tilde{s}$ 53
Bibliography 53

5 The QP_TR algorithm. **54**
5.1 The bound ϵ. . . . 56
5.2 Note about the validity check. . . . 56
Bibliography 56

6 Numerical Results of QP_TR algorithm. **57**
Bibliography 58

7 Conclusions **59**
7.1 The H-norm 59
Bibliography 60

8 Annexes **61**
8.1 Line-Search addenda. . . . 61
8.1.1 Speed of convergence of Newton's method. . . . 61
8.1.2 How to improve Newton's method : Zoutendijk Theorem. . . . 62
8.2 Gram-Schmidt orthogonalization procedure. . . . 65
8.3 Notions of constrained optimization 65
8.4 The secant equation 66
8.5 1D Newton's search 67
8.6 Cholesky decomposition. . . . 68
8.6.1 Performing LU decomposition. . . . 68
8.6.2 Performing Cholesky decomposition. . . . 69
Bibliography 70

9 Code of the optimizer. 71
9.1 Main . . . 71
9.2 Matrix manipulation . . . 72
9.2.1 Header file . . . 72
9.2.2 Code file . . . 73
9.3 Triangular Matrix manipulation . . . 79
9.3.1 Header file . . . 79
9.3.2 Code file . . . 80
9.4 Vector manipulation . . . 82
9.4.1 Header file . . . 82
9.4.2 Code file . . . 82
9.5 Vector of integer manipulation . . . 86
9.5.1 Header file . . . 86
9.5.2 Code file . . . 86
9.6 MultiIndex manipulation . . . 87
9.6.1 Header file . . . 87
9.6.2 Code file . . . 88
9.7 Simple polynomial manipulation . . . 91
9.7.1 Header file . . . 91
9.7.2 Code file . . . 92
9.8 Lagrange Interpolaton polynomial manipulation . . . 98
9.8.1 Header file . . . 98
9.8.2 Code file . . . 99
9.9 Various simple tools . . . 103
9.9.1 Header file . . . 103
9.9.2 Code file . . . 104
9.10 Sort tool . . . 106
9.10.1 Header file . . . 106
9.10.2 Code file . . . 106
9.11 The optimizers. . . . 108
9.11.1 Header file . . . 108
9.11.2 Code file - Chapter 3 . . . 109
9.11.3 Code file - Chapter 4 . . . 111
9.11.4 Code file - Chapter 5 . . . 113

Chapter 1

Introduction

Abstract

We will present an algorithm which optimizes a non-linear function $y = \mathcal{F}(x),\ x \in \Re^n\ y \in \Re$. That is the algorithm will, hopefully, find the value of x for which y is the lowest. 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). 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 contribution of this work is:

- The implementation in C++ of the whole algorithm.
- The assembly of different well-known algorithms in one code. (In particular, the code of chapter 2 has never been assembled to the code of chapter 3, but rather to a different but equivalent code).
- The bibliographical work needed to:
  - Understand all the parts of the algorithm.
  - Acquire a huge amount of background knowledge in continuous optimization. This is necessary to be able to review all the available techniques in continuous optimization and choose the best promising technique. I will however 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)
- The comparison of the new algorithm QP_TR with a famous one: CFSQP. The result of this comparison is that QP_TR requires one fifth to one half of the number of evaluations of $\mathcal{F}$ that CFSQP requires, depending on the complexity of the objective function. This result is the conclusion of the experimental study done in chapter 6.

This new algorithm is mainly based on the recent work of M.J.D.Powell [8].

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 takes as much as one or two hours to finish.

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 $\mathcal{F}(x)$.

Optimization code usually requires the derivatives of $\mathcal{F}(x)$ to be available. Unfortunately, we usually don't have them. Very often, there is also some noise on $\mathcal{F}(x)$ due to rounding errors. To overcome these limitations, I present here a new algorithm: QP_TR. This algorithm is mainly based on the recent work of M.J.D.Powell [8].

Here are the assumptions needed to use this new algorithm:

- The dimension n of the search space must be lower than 50.
- 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 converges but in a greater time.
- The algorithm tries to minimize the number of evaluation of $\mathcal{F}$, 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.

1.2 Formal description

This treatise is about optimization of non-linear **continuous** functions. We want to find $x^* \in \mathcal{R}^n$ which satisfies:

$$\mathcal{F}(x^*) = \min_x \mathcal{F}(x) \tag{1.1}$$

$\mathcal{F}(x) : \Re^n \to \Re$ is called the objective function.

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 $\mathcal{F}$.

In this book, we will not tackle the more complicated subject of constrained optimization when only a part of the space is allowed for x.

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

- The dimension n of the search space.
- Wether 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 $\mathcal{F}(x)$.
- Wether the Objective function is smooth or not.
- Wether the search space is continuous (there is no discrete variable like variable which can take the following values: red, green, blue.).

If there is 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).

In the rest of the book, we will make the assumption that the objective function is smooth and that we want only a local minimum.

Otherwise,

dimension n	algorithm	
	only $\mathcal{F}(x)$ is available	$g(x)$ is also available
$2 \leq n < 50$	QP_TR	Broyden_TR
$30 \leq n < 10000$	Broyden_TR (with approximative derivatives)	

"QP_TR" and "Broyden_TR" are both Trust Region methods. The main difference between the two methods is the way the curvature information (the matrix H) will be generated.

In "Broyden_TR", we will construct incrementally H using a BFGS update.

In "QP_TR", we will construct, using multivariate Lagrange interpolation, a quadratic polynomial which is a local approximation of the function. The matrix H will be extracted from this polynomial. The description of this algorithm is the main goal of this book.

Another category of algorithm are "Line-search methods". These algorithms are still used very often in constrained optimization. We will see their relationship with the "Trust Region methods".

For higher dimension, the choice of the algorithm is not clear but, if the hessian matrix H is directly available, a good choice will be a Conjugate-gradient/Lanczos method.

1.3 Trust Region and Line-search Methods.

1.3.1 Conventions

$\mathcal{F}(x) : \Re^n \to \Re$ 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 quadratic 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.

1.3.2 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":

$$s_k = -g_k \tag{1.2}$$

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

$$x_{k+1} = x_k - g_k \tag{1.3}$$

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

1.3.3 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:

$$\lim_{k \to \infty} \frac{\|x_{k+1} - x^*\|}{\|x_k - x^*\|} = 0 \tag{1.4}$$

1.3.4 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:

$$\begin{aligned} \nabla \mathcal{Q}(\delta_k) = g_k + B_k \delta_k &= 0 \\ \Longleftrightarrow B_k \delta_k &= -g_k \end{aligned} \tag{1.5}$$

Equation 1.5 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 quadratic 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 8.1.2 for conditions on α: the Wolf conditions)
Near the optimum, we must always take $\alpha = 1$, to allow a "full step of one" to occur: see chapter 1.3.6 for more information.

Newton's method has quadratic convergence: see section 8.1.1 for proof.

1.3.5 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.

$$\Longrightarrow \delta^T g < 0 \tag{1.6}$$

Taking the value of g from 1.5 and putting it in 1.6, we have:

$$\begin{aligned} & -\delta^T B\delta < 0 \\ \Leftrightarrow \quad & \delta^T B\delta > 0 \end{aligned} \tag{1.7}$$

The equation 1.7 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 1.5) $B_{new}\delta_k = -g_k$. 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 point 2 above is important. It can be proofed 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 algorithm are based on this principle. ($\|\Delta_k\|$ 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.

1.3.6 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:

$A_k \to B_k$ (and $B_k \to H_k$)). 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*

$$A_k \delta_k = -g_k \tag{1.8}$$

where

$$\lim_{k \to \infty} \frac{\|(A_k - H^*)\delta_k\|}{\|\delta_k\|} = 0 \tag{1.9}$$

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

$$\|g(x) - g(y)\| \leq \gamma \|x - y\| \tag{1.10}$$

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

Lemma 1.

We will proof that $\|a\|^2 - \|b\|^2 \leq \|a - b\|(\|a\| + \|b\|)$

Using the cauchy-swartz inequality $\|u\ v\| < \|u\|\|v\|$ with $u = a - b$ and $v = a + b$, we can write:

$$\begin{aligned} \|(a-b)(a+b)\| &< \|a-b\|\|a+b\| \\ \|a^2 - b^2\| &< \|a-b\|\|a+b\| \end{aligned}$$

Since a^2 and b^2 are two real numbers:

$$\|a\|^2 - \|b\|^2 < \|a - b\|\|a + b\|$$

Using triangular inequality:

$$\|a\|^2 - \|b\|^2 < \|a - b\|(\|a\| + \|b\|) \tag{1.11}$$

Lemma 2.

We will prove that if H *is Lipchitz continuous then*
$\|g(v) - g(u) - H(x)(v - u)\| \leq \frac{\gamma}{2}\|v - u\|(\|v - x\| + \|u - x\|)$ *hold.*

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 $H(x)\delta$:

$$g(x+\delta) - g(x) - H(x)\delta = \int_0^1 (H(x+t\delta) - H(x))\delta dt \tag{1.12}$$

In the previous equation we pose $\delta = v - x$:

$$g(v) - g(x) - H(x)(v-x) = \int_0^1 (H(x+t(v-x)) - H(x))(v-x)dt \tag{1.13}$$

In equation 1.12, we pose $\delta = u - x$:

$$g(u) - g(x) - H(x)(u-x) = \int_0^1 (H(x+t(u-x)) - H(x))(u-x)dt \tag{1.14}$$

Equation 1.13 minus equation 1.14 equals:

$$\begin{aligned} g(v) - g(u) - H(x)(v-u) &= \int_0^1 (H(x+t(v-x)) - H(x))(v-x)dt \\ &- \int_0^1 (H(x+t(u-x)) - H(x))(u-x)dt \end{aligned}$$

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

$$\begin{aligned} \|g(v) - g(u) - H(x)(v-u)\| &\leq \int_0^1 \|H(x+t(v-x)) - H(x)\|\|v-x\|dt \\ &- \int_0^1 \|H(x+t(u-x)) - H(x)\|\|u-x\|dt \end{aligned}$$

Using the fact that the Hessian H is Lipchitz continuous (see equation 1.10):

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

Using equation 1.11 with $a = v - x$ and $b = x - u$, we obtain:

$$\|g(v) - g(u) - H(x)(v-u)\| \leq \frac{\gamma}{2}\|v-u\|(\|v-x\| + \|u-x\|) \tag{1.15}$$

Lemma 3.

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

$$\alpha\|v-u\| \leq \|g(v) - g(u)\| \tag{1.16}$$

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 $\|a\ b\| < \|a\|\|b\|$ with $a = H^{-1}$ and $b = v - u$, and using equation 1.15:

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

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 1.8:

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

Using lemma 2: equation 1.15, 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 1.9:

$$\lim_{k\to\infty} \frac{\|g_{k+1}\|}{\|\delta_k\|} = 0 \tag{1.17}$$

Using lemma 3: equation 1.16, there exists $\alpha > 0, k_0 \geq 0$, such that, for all $k > k_0$, we have (using $g(x^*) = 0$):

$$\|g_{k+1}\| = \|g_{k+1} - g(x^*)\| \geq \alpha \|e_{k+1}\| \tag{1.18}$$

Combing equation 1.17 and 1.18,

$$\begin{aligned} 0 &= \lim_{k\to\infty} \frac{\|g_{k+1}\|}{\|\delta_k\|} \geq \lim_{k\to\infty} \alpha \frac{\|e_{k+1}\|}{\|\delta_k\|} \\ &\geq \lim_{k\to\infty} \alpha \frac{\|e_{k+1}\|}{\|e_k\| + \|e_{k+1}\|} = \lim_{k\to\infty} \frac{\alpha r_k}{1 + r_k} \end{aligned} \tag{1.19}$$

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

$$\lim_{k\to\infty} r_k = 0 \tag{1.20}$$

which completes the proof of superlinear convergence.

Since $g(x)$ is Lipschitz continuous, it's easy to show that the Dennis-morè theorem remains true if equation 1.9 is replaced by:

$$\lim_{k\to\infty} \frac{\|(A_k - H_k)\delta_k\|}{\|\delta_k\|} = \lim_{k\to\infty} \frac{\|(A_k - B_k)\delta_k\|}{\|\delta_k\|} = 0 \tag{1.21}$$

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 1.5) 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 equation 8.4 and 8.5 in section 8.1.2) always allow a "full step of one".

When we will deal with constraints, it's some time 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.

1.4 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 quadratic 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 $\mathcal{Q}$:
$$r_k = \frac{f(x_k) - f(x_k + \delta_k)}{\mathcal{Q}_k(0) - \mathcal{Q}_k(\delta_k)} \tag{1.22}$$
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)
$x_{k+1} = x_k$ $\Delta_{k+1} = \frac{\Delta_k}{2}$	$x_{k+1} = x_k + \delta_k$ $\Delta_{k+1} = \Delta_k$	$x_{k+1} = x_k + \delta_k$ $\Delta_{k+1} = 2\Delta_k$

(1.23)

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 of 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 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:

$$\frac{\partial \mathcal{F}}{\partial x_i} = g_i = \frac{\mathcal{F}(\overline{x} + \epsilon_i \overline{e_i}) - \mathcal{F}(\overline{x})}{\epsilon_i} \qquad i = 1, \ldots, n \tag{1.24}$$

  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 $n+1$ workstations at our disposal, we can distribute the 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 in this book.

  Unfortunately, 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 5.2 how to cope with this difficulty.

There are different ways of obtaining B_k. Many are unpractical. Here are some reasonable one:

- 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 quadratic 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 could be 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.

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

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

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

1.5 The basic trust-region algorithm (BTR).

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

$$\mathcal{B}_k = \{x \in \Re^n | \|x - x_k\|_k \leq \Delta_k\} \tag{1.25}$$

The simple algorithm described in the section 1.4 can be generalized by the following one:

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:

$$0 < \eta_1 \leq \eta_2 < 1 \text{ and } 0 < \gamma_1 \leq \gamma_2 < 1 \tag{1.26}$$

   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:

$$r_k = \frac{f(x_k) - f(x_k + s_k)}{m_k(x_k) - m_k(x_k + s_k)} \tag{1.27}$$

   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

$$\Delta_{k+1} \in \begin{cases} [\Delta_k, \infty) & \text{if } r_k \geq \eta_2, \\ [\gamma_2 \Delta_k, \Delta_k) & \text{if } r_k \in [\eta_1, \eta_2), \\ [\gamma_1 \Delta_k, \gamma_2 \Delta_k) & \text{if } r_k < \eta_1. \end{cases} \tag{1.28}$$

   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. The proof will be skipped.

1.6 About the QP_TR algorithm.

To start the QP_TR 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 points when the algorithm stops.

We will approximatively use the following algorithm (for a complete description, see chapter 5):

1. Create an interpolation polynomial $q(x)$ of degree 2 which intercepts the objective function around x_{start} (see chapter 2). All the points in the interpolation set (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 $\rho = \rho_{start}$.

2. Outer loop.

(a) Set $\Delta_k = \rho$

(b) Inner loop: solve the problem for a given precision ρ.

i. Solve $\delta_k = \min_{\delta \in \Re^n} q(x_k + \delta)$ subject to $\|\delta\|_2 < \Delta_k$ (see chapter 3)

ii. Update The trust region radius Δ_k and the current best point x_k (using a scheme similar to equation 1.23). Include the new x_k inside the interpolation set of $q(x)$.

iii. If some progress has been accomplished (for example, $\|\delta_k\| > 2\rho$ or there was a reduction $f(x_{k+1}) < f(x_k)$), increment k and go back to step **i**.

iv. Try to increase the precision of $q(x)$ in a radius of ρ around x_k: remove the worst point of the interpolation set and replace it with a new point x_{new} such that:

- $\|x_{new} - x_k\| < \rho$
- The precision of $q(x)$ increases substantially.

If all the points in the interpolation set are already good and if $\|\delta_k\| < \rho$ go to step **(c)** otherwise, increment k and go back to step **i**.

(c) No more progress is possible with this value of ρ. The steps are becoming too small. **Reduce** ρ.

(d) If $\rho = \rho_{end}$ then stop, otherwise increment k and go back to step **(a)**.

From this description, we can say that ρ and Δ_k are two trust region radius (global and local). It is possible to proof that the algorithm is globally convergent to a local optimum. The proof will be skipped.

This book is divided as follow:

- **Chapter 1:** Introduction. You are reading it.
- **Chapter 2:** How to construct and use $q(x)$? Inside the QP_TR algorithm we need a polynomial approximation of the objective function. How do we build it? How do we use it?
- **Chapter 3:** 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 (2)(b)i. of the QP_TR algorithm.
- **Chapter 4:** 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 5:** The precise description of the QP_TR algorithm.
- **Chapter 6:** Numerical results of the QP_TR algorithm.
- **Chapter 7:** Some Conclusions.
- **Chapter 8:** Annexes: All the basic mathematical background knowledge needed to understand the book (just in case you need it).
- **Chapter 9:** Code.

Bibliography

[1] R.Fletcher
"Practical Methods of optimization"
"a Wiley-Interscience pblication" ISBN 0 471 91547 5.

[2] Eliane R. Panier and André L. Tits
" On combining feasibility, Descent and Superlinear Convergence in Inequality Contrained Optimization"
Mathematical Programming 59 (1993) 261-276

[3] Paul T. Boggs , Jon W. Tolle
"Sequential Quadratic Programming"
Acta Numerica (1996), pp. 1-000

[4] Jorge Nocedal
"Theory of Algorithm for Unconstraned Optimization"
Acta Numerica (1992), pp. 199-242

[5] N.I.M. Gould and Ph.L.Toint "SQP methods for large-scale nonlinear programming"
ftp://thales.math.fundp.ac.be/pub/reports, report 99/05

[6] J.E.Dennis, Jr, Robert B.Schnabel
"Numerical Methods for unconstrained Optimization and nonlinear Equations"
Classics in applied mathematics 16, SIAM

[7] Andrew R. Conn, Nicholas I.M.Gould, Philippe L.Toint
"Trust-region Methods"
MPS-SIAM series on Optimization

[8] M.J.D. Powell
"UOBYQA: unconstrained optimization by quadratic approximation"
Internal report Department of Applied Mathematics and Theoretical Physics, University of Cambridge, England

Chapter 2

Multivariate Lagrange Interpolation

2.1 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 $L : \Re^2 -> \Re : z = c_1 + c_2x + c_3y$ of degree 1 (a plane), which interpolates locally a function $F : \Re^2 \to \Re$, we need exactly 3 points A, B and C. Why do we need 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:

$$\begin{pmatrix} 1 & A_x & A_y \\ 1 & B_x & B_y \\ 1 & C_x & C_y \end{pmatrix} \begin{pmatrix} c_1 \\ c_2 \\ c_3 \end{pmatrix} = \begin{pmatrix} f(A) \\ f(B) \\ f(C) \end{pmatrix} \tag{2.1}$$

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 $F : \Re^n \to \Re$, 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 $\mathcal{Q} : \Re^2 -> \Re : z = c_1 + c_2x + c_3y + c_4x^2 + c_5xy + c_6y^2$ of degree 2, which interpolates locally a function $F : \Re^2 \to \Re$, at points $\{A, B, C, D, E, F\}$we will have the following Vandermonde system:

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

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"**.

2.2 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 $i \neq j$.

2.2.1 Lagrange interpolation

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

$$P_i(x) := \prod_{\substack{j=1 \\ j \neq i}}^{N} \frac{x - \boldsymbol{x}_{(j)}}{\boldsymbol{x}_{(i)} - \boldsymbol{x}_{(j)}} \tag{2.2}$$

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

$$\delta_{(i,j)} = \begin{cases} 0, & i \neq j; \\ 1, & i = j. \end{cases} \tag{2.3}$$

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

$$L(x) = \sum_{i=1}^{N} f(\boldsymbol{x}_{(i)})\ P_i(x) \tag{2.4}$$

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.

2.2.2 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:

$$P_{k+1}(x) = P_k(x) + \quad (x - \boldsymbol{x}_{(1)}) \cdots (x - \boldsymbol{x}_{(k)}) \quad [\boldsymbol{x}_{(1)}, \ldots, \boldsymbol{x}_{(k+1)}]f \tag{2.5}$$

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 coefficient (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:

$$P(x) = P_N(x) = \sum_{k=1}^{N} \quad (x - \boldsymbol{x}_{(1)}) \cdots (x - \boldsymbol{x}_{(k-1)}) \quad [\boldsymbol{x}_{(1)}, \ldots, \boldsymbol{x}_{(k)}]f \tag{2.6}$$

The final interpolation polynomial is a sum of polynomials of degree varying from 0 to $N-1$ (see equation 2.5). The manipulation of the Newton polynomials (of equation 2.5) is faster than the Lagrange polynomials (of equation 2.2) 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:

$$[\boldsymbol{x}_{(k)}]f = f(\boldsymbol{x}_{(k)}) \tag{2.7}$$

$$[\boldsymbol{x}_{(1)}, \boldsymbol{x}_{(2)}]f = \frac{f(\boldsymbol{x}_{(2)}) - f(\boldsymbol{x}_{(1)})}{\boldsymbol{x}_{(2)} - \boldsymbol{x}_{(1)}} \tag{2.8}$$

2.2.3 The divided difference for the Newton form.

Lemma 1

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

$$f(x) - P_N(x) = (x - \boldsymbol{x}_{(1)}) \cdots (x - \boldsymbol{x}_{(N)}) \quad [\boldsymbol{x}_{(1)}, \ldots, \boldsymbol{x}_{(N)}, x]f \tag{2.9}$$

We will proof this by induction. First, we rewrite equation 2.9, for N=1, using 2.7:

$$f(x) = f(\boldsymbol{x}_{(1)}) + (x - \boldsymbol{x}_{(1)})[\boldsymbol{x}_{(1)}, x]f \tag{2.10}$$

Using equation 2.8 inside 2.10, we obtain:

$$f(x) = f(\boldsymbol{x}_{(1)}) + (x - \boldsymbol{x}_{(1)})\frac{f(x) - f(\boldsymbol{x}_{(1)})}{x - \boldsymbol{x}_{(1)}} \tag{2.11}$$

This equation is readily verified. The case for $N = 1$ is solved. Suppose the equation 2.9 verified for $N = k$ and proof that it will also be true for $N = k + 1$. First, let us rewrite equation 2.10, replacing $\boldsymbol{x}_{(1)}$ by $\boldsymbol{x}_{(k+1)}$ and replacing $f(x)$ by $[\boldsymbol{x}_{(1)}, \ldots, \boldsymbol{x}_{(k)}, x]f$ *as a function of* x. (In other word, we will interpolate the function $f(x) \equiv [\boldsymbol{x}_{(1)}, \ldots, \boldsymbol{x}_{(k)}, x]f$ at the point $\boldsymbol{x}_{(k+1)}$ using 2.10.) We obtain:

$$[\boldsymbol{x}_{(1)}, \ldots, \boldsymbol{x}_{(k)}, x]f = [\boldsymbol{x}_{(1)}, \ldots, \boldsymbol{x}_{(k+1)}]f + (x - \boldsymbol{x}_{(k+1)})[\boldsymbol{x}_{(k+1)}, x]\Big([\boldsymbol{x}_{(1)}, \ldots, \boldsymbol{x}_{(k)}, \cdot]f\Big) \tag{2.12}$$

Let us rewrite equation 2.9 with $N = k$.

$$f(x) = P_k(x) + (x - \boldsymbol{x}_{(1)}) \cdots (x - \boldsymbol{x}_{(k)}) \quad [x_1, \ldots, x_k, x]f \tag{2.13}$$

Using equation 2.12 inside equation 2.13:

$$f(x) = P_{k+1}(x) + (x - \boldsymbol{x}_{(1)}) \cdots (x - \boldsymbol{x}_{(k+1)})[\boldsymbol{x}_{(k+1)}, x]\Big([\boldsymbol{x}_{(1)}, \ldots, \boldsymbol{x}_{(k)}, \cdot]f\Big) \tag{2.14}$$

Let us rewrite equation 2.5 changing index $k + 1$ to $k + 2$.

$$P_{k+2}(x) = P_{k+1}(x) + \quad (x - \boldsymbol{x}_{(1)}) \cdots (x - \boldsymbol{x}_{(k+1)}) \quad [\boldsymbol{x}_{(1)}, \ldots, \boldsymbol{x}_{(k+2)}]f \tag{2.15}$$

Recalling the definition of the divided difference: $[x_1, \ldots, x_{k+2}]f$ is the unique leading coefficient of the polynomial of degree $k + 1$ that agree with f at the sequence $\{\boldsymbol{x}_{(1)}, \ldots, \boldsymbol{x}_{(k+2)}\}$. Because of the uniqueness and comparing equation 2.14 and 2.15, we see that (replacing x by $\boldsymbol{x}_{(k+2)}$ in 2.14):

$$[\boldsymbol{x}_{(k+1)}, \boldsymbol{x}_{(k+2)}]\Big([\boldsymbol{x}_{(1)}, \ldots, \boldsymbol{x}_{(k)}, \cdot]f\Big) = [\boldsymbol{x}_{(1)}, \ldots, \boldsymbol{x}_{(k+2)}]f \tag{2.16}$$

Using equation 2.16 inside 2.14:

$$f(x) = P_{k+1}(x) + (x - \boldsymbol{x}_{(1)}) \cdots (x - \boldsymbol{x}_{(k+1)})[\boldsymbol{x}_{(1)}, \ldots, \boldsymbol{x}_{(k+2)}]f \tag{2.17}$$

Recalling the discussion of the paragraph after equation 2.11, we can say then this last equation complete the proof for $N = k + 1$. The lemma 1 is now proofed.

Lemma 2

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

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

Lemma 3

A useful formula to compute finite differences.

$$[\boldsymbol{x}_{(i)},\ldots,\boldsymbol{x}_{(i+k)}]f = \frac{[\boldsymbol{x}_{(i+1)},\ldots,\boldsymbol{x}_{(i+k)}]f - [\boldsymbol{x}_{(i)},\ldots,\boldsymbol{x}_{(i+k-1)}]f}{\boldsymbol{x}_{(i+k)} - \boldsymbol{x}_{(i)}} \tag{2.18}$$

Combining equation 2.16 and equation 2.8, we obtain:

$$\frac{[\boldsymbol{x}_{(1)},\ldots,\boldsymbol{x}_{(k)},\boldsymbol{x}_{(k+2)}]f - [\boldsymbol{x}_{(1)},\ldots,\boldsymbol{x}_{(k)},\boldsymbol{x}_{(k+1)}]f}{\boldsymbol{x}_{(k+2)} - \boldsymbol{x}_{(k+1)}} = [\boldsymbol{x}_{(1)},\ldots,\boldsymbol{x}_{(k+2)}]f \tag{2.19}$$

Using equation 2.19 and lemma 2, we obtain directly 2.18. The lemma is proved.

Equation 2.18 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.
$\boldsymbol{x}_{(1)}$	$f(\boldsymbol{x}_{(1)})$					
		$[\boldsymbol{x}_{(1)},\boldsymbol{x}_{(2)}]f$				
$\boldsymbol{x}_{(2)}$	$f(\boldsymbol{x}_{(2)})$		$[\boldsymbol{x}_{(1)},\boldsymbol{x}_{(2)},\boldsymbol{x}_{(3)}]f$			
		$[\boldsymbol{x}_{(2)},\boldsymbol{x}_{(3)}]f$				
$\boldsymbol{x}_{(3)}$	$f(\boldsymbol{x}_{(3)})$		$[\boldsymbol{x}_{(2)},\boldsymbol{x}_{(3)},\boldsymbol{x}_{(4)}]f$	$\cdot$	$[\boldsymbol{x}_{(1)},\ldots\boldsymbol{x}_{(N-1)}]f$	
		$[\boldsymbol{x}_{(3)},\boldsymbol{x}_{(4)}]f$				$[\boldsymbol{x}_{(1)},\ldots\boldsymbol{x}_{(N)}]f$
$\boldsymbol{x}_{(4)}$			$\cdot$	$\cdot$	$[\boldsymbol{x}_{(2)},\ldots\boldsymbol{x}_{(N)}]f$	
$\vdots$	$\vdots$	$\cdot$				
$\boldsymbol{x}_{(N-1)}$	$f(\boldsymbol{x}_{(N-1)})$		$[\boldsymbol{x}_{(N-2)},\boldsymbol{x}_{(N-1)},\boldsymbol{x}_{(N)}]f$			
		$[\boldsymbol{x}_{(N-1)},\boldsymbol{x}_{(N)}]f$				
$\boldsymbol{x}_{(N)}$	$f(\boldsymbol{x}_{(N)})$					

We can generate the entries of the divided difference table *column by column* from the given dataset using equation 2.18. The top diagonal then contains the desired coefficients $[\boldsymbol{x}_{(1)}]f$, $[\boldsymbol{x}_{(1)},\boldsymbol{x}_{(2)}]f, [\boldsymbol{x}_{(1)},\boldsymbol{x}_{(2)},\boldsymbol{x}_{(3)}]f,\ldots,[\boldsymbol{x}_{(1)},\ldots,\boldsymbol{x}_{(N)}]f$ of the final Newton form of equation 2.6.

2.2.4 The Horner scheme

Suppose we want to evaluate the following polynomial:

$$P(x) = c_0 + c_1 x + c_2 x^2 + c_3 x^3 + c_4 x^4 \tag{2.20}$$

We will certainly NOT use the following algorithm:

1. **Initialisation** $r = c_0$
2. **For** $k = 2,\ldots,N$
   (a) Set $r := r + c_k x^k$

3. **Return** r

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

The Horner scheme uses the following representation of the polynomial 2.20: $P(x) = c_0 + x(c_1 + x(c_2 + x(c_3 + xc_4)$, to construct a very efficient evaluation algorithm:

1. **Initialisation** $r = c_N$
2. **For** $k = N-1, \ldots, 0$
   (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.

2.3 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 $(\boldsymbol{x}_{(i)}, f(\boldsymbol{x}_{(i)})), \quad i = 1, \ldots, N \quad \boldsymbol{x}_{(i)} \in \Re^n$ on the surface. We must be assure that the problem is well poised: The number N of points is $N = r_n^d$ and the Vandermonde determinant is not null.

2.3.1 The Lagrange polynomial basis $\{P_1(x), \ldots, P_N(x)\}$.

We will construct our polynomial basis $P_i(x) \quad i = 1, \ldots, N$ 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} of degree $k+1$ which doesn't destruct all what we have already done. That is the value of $P_{k+1}(x)$ must be zero for $x = \boldsymbol{x}_{(1)}, \boldsymbol{x}_{(2)}, \ldots, \boldsymbol{x}_{(k)}$. In other words, $P_i(\boldsymbol{x}_{(j)}) = \delta_{(i,j)}$. This is easily done in the univariate case: $P_k(x) = (x - \boldsymbol{x}_{(1)}) \ldots (x - \boldsymbol{x}_{(k)})$, but in the multivariate case, it becomes difficult.

We must find a new polynomial P_{k+1} which is somewhat "perpendicular" to the $P_i \quad i = 1, \ldots, k$ with respect to the k points $\boldsymbol{x}_{(1)}, \ldots, \boldsymbol{x}_{(k)}$. 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 $\boldsymbol{x}_{(1)}, \ldots, \boldsymbol{x}_{(k)}$. 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 8.2.

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

$$< P, Q >_{\mathcal{K}} = \sum_{j=1}^{k} P(\boldsymbol{x}_{(j)}) Q(\boldsymbol{x}_{(j)}) \tag{2.21}$$

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**

$$P_k(x) = \frac{P_{k\_old}(x)}{|P_{k\_old}(\boldsymbol{x}_{(k)})|} \tag{2.22}$$

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

$$P_{j\_old}(x) = P_{j\_old}(x) - P_{j\_old}(\boldsymbol{x}_{(k)})P_k(x) \quad j = 1, \ldots, k-1, k+1, \ldots, N \tag{2.23}$$

   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 2.22, 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 2.22). We should choose the $\boldsymbol{x}_{(k)}$ (among the points which are still left inside the dataset) so that the denominator of equation 2.22 is far from zero:

$$|P_{k\_old}(x)| \text{ as great as possible.} \tag{2.24}$$

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:

$$P_i(\boldsymbol{x}_{(j)}) = \delta_{(i,j)} \quad i, j = 1, \ldots, N \tag{2.25}$$

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

Using equation 2.25, we can write:

$$L(x) = \sum_{j=1}^{N} f(\boldsymbol{x}_{(j)})P_j(x) \tag{2.26}$$

2.3.3 The multivariate Horner scheme

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

$$P(x) = \sum_{i=1}^{N} c_i \prod_{j=1}^{n} x_j^{\alpha(i,j)} \quad \text{with } \max_{i,j} \alpha(i,j) = d \tag{2.27}$$

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") :

$\alpha'($ for $n = 2$ and $d = 2) =$

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

$\alpha'($ for $n = 3$ 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 $r_2 := c_5 + x_2r_2;$
- Return $c_6 + x_1r_1 + x_2r_2$

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 orderer 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.
   $r_0, \ldots, r_n$: registers for summation ($\in \Re$)
   $a_1, \ldots, a_n$: counters ($\in N_0$)
   $c_i, \quad i = 1, \ldots, N$: the coefficients of the polynomial ordered by degree.

2. **Initialization**
   Set $r_0 := c_{Tr(1)}$
   set $a_j := \alpha'(1, j) \quad j = 1, \ldots, n$

3. **For** $i = 2, \ldots, N$

   (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 $a_j := \alpha(i, j) \quad j = 1, \ldots, k-1$

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.

2.4 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.

2.4.1 A bound on the interpolation error.

We assume in this section that the objective function $f(x), x \in \Re^n$, has 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

$$\phi(\alpha) = f(y + \alpha\bar{d}), \quad \alpha \in \Re, \tag{2.28}$$

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 function of this form has the property

$$|\phi'''(\alpha)| \leq M, \quad \alpha \in \Re \tag{2.29}$$

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

$$\textit{Interpolation Error} = |L(x) - f(x)| < \frac{1}{6}M \sum_{j=1}^{N} |P_j(x)| \|x - \boldsymbol{x}_{(j)}\|^3 \tag{2.30}$$

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

$$\bar{d} = \frac{\boldsymbol{x}_{(j)} - y}{\|\boldsymbol{x}_{(j)} - y\|} \tag{2.31}$$

and let $\phi(\alpha), \alpha \in \Re$, be the function 2.28. The Taylor series with explicit remainder formula gives:

$$\phi(\alpha) = \phi(0) + \alpha\phi'(0) + \frac{1}{3}\alpha^2\phi''(0) + \frac{1}{6}\alpha^3\phi'''(\varepsilon), \qquad \alpha \geq 0, \tag{2.32}$$

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 2.31, 2.28, 2.32, 2.29 provide the bound

$$|f(\boldsymbol{x}_{(j)})| = \frac{1}{6}\alpha^3|\phi'''(\varepsilon)| \leq \frac{1}{6}M\|y - \boldsymbol{x}_{(j)}\|^3, \tag{2.33}$$

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 2.26 and from inequality 2.33, 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 2.30 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:

$$M_{new} = \max \left[M_{old}, \frac{|L(x) - f(x)|}{\frac{1}{6} \sum_{j=1}^{N} |P_j(x)| \|x - \boldsymbol{x}_{(j)}\|^3} \right] \tag{2.34}$$

2.4.2 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 2.30):

$$\frac{1}{6} M \max_x \{|P_j(x)| \|x - \boldsymbol{x}_{(k)}\|^3 : \|x - \boldsymbol{x}_{(k)}\| \leq \rho\} \tag{2.35}$$

$$\approx \frac{1}{6} M \|\boldsymbol{x}_{(j)} - \boldsymbol{x}_{(k)}\|^3 \max_d \{|P_j(\boldsymbol{x}_{(k)} + d)| : \|d\| \leq \rho\} \tag{2.36}$$

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

$$\frac{1}{6} M \|\boldsymbol{x}_{(j)} - \boldsymbol{x}_{(k)}\|^3 \max_d \{|P_j(\boldsymbol{x}_{(k)} + d)| : \|d\| \leq \rho\} \leq \epsilon \tag{2.37}$$

ϵ is a bound on the error which must be given to the procedure which checks the validity of the interpolation. See section 5.1 to know how to compute the bound ϵ.

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

$$\max_d \{|P_j(\boldsymbol{x}_{(k)} + d)| : \|d\| \leq \rho\} \tag{2.38}$$

is described in chapter 4.

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 2.37 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 2.37 and continue until the dataset $\mathcal{J}$ is empty.

If the test 2.37 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 2.38): see section 2.4.4, to know how to do.

2.4.3 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 2.24, we must have:

$$|P_t(X)| \text{ as great as possible} \tag{2.39}$$

We also wish to remove a point which seems to be making a relatively large contribution to the bound 2.30 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:

$$\begin{cases} |P_i(X)| \max\left[1, \frac{\|\boldsymbol{x}_{(i)} - X\|^3}{\rho^3}\right], \quad i = 1, \ldots, N & \text{if } f(X) < f(\boldsymbol{x}_{(k)}) \\ |P_i(X)| \max\left[1, \frac{\|\boldsymbol{x}_{(i)} - \boldsymbol{x}_{(k)}\|^3}{\rho^3}\right], \quad i = 1, \ldots, k-1, k+1, \ldots, N & \text{if } f(X) > f(\boldsymbol{x}_{(k)}) \end{cases} \tag{2.40}$$

2.4.4 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:

$$\tilde{P}_t(x) = \frac{P_t(x)}{P_t(X)} \tag{2.41}$$

$$\tilde{P}_i(x) = P_i(x) - P_i(X)\tilde{P}_t(x), \quad i \neq t \tag{2.42}$$

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

$$L_{new}(x) = L_{old}(x) + [f(X) - L_{old}(X)]\tilde{P}_t(x) \tag{2.43}$$

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

To be able to generate the first set of interpolation point $\{\boldsymbol{x}_{(1)}, \ldots, \boldsymbol{x}_{(N)}\}$, 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 ρ 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 $\boldsymbol{x}_{(1+n)}$:

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

Let us define σ_j:

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

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

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

- From $\boldsymbol{x}_{(2+2n)}$ to $\boldsymbol{x}_{(N)}$:
  Set $k = 2 + 2n$.
  For $j = 1, \ldots, n$
  1. For $i = 1, \ldots, j-1$
     (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.

2.4.6 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 8.23, we can write:

$$\begin{cases} a_2 := P_1(x_{(base)}) \\ g_2 := g_1 + H_1 x_{(base)} \\ H_2 := H_1 \end{cases} \tag{2.44}$$

Bibliography

[1] Carl De Boor, K Höllig, S Riemenshneider
"Box splines"
"Applied Mathematical Sciences, 98" Springer-Verlag

[2] Carl De Boor
"A Practical Guide to Splines" (revised edition)
"Applied Mathematical Sciences, 27" Springer-Verlag

[3] Carl De Boor, A Ron.
"On multivariate polynomial interpolation"
Constr. Approx. 6 (1990), pp. 287-302

[4] Thomas Sauer, Yuan Xu
"On multivariate Lagrange interpolation"
Math. Comp. 64 (1995) pp. 1147-1170

[5] Thomas Sauer
"Computational aspects of multivariate Polynomial interpolation"
Advances Comput. Math. 3 (1995), pp. 219-238

[6] J.M. Pena, Thomas Sauer
"On the multivariate Horner scheme"
SIAM J. Numer. Anal., 1999

[7] R.A. Lorentz
"Multivariate Hermite interpolation by algebraic polynomials: A survey"
Journal of computation and Applied Mathematics 122 (2000) pp. 167-201

[8] William H. Press, Saul A.Teukolsky, William T.Vetterling, Brian P.Flannery
"Numerical recipes in C" (second edition)
Cambridge University Press

[9] M.J.D. Powell
"UOBYQA: unconstrained optimization by quadratic approximation"
Internal report Department of Applied Mathematics and Theoretical Physics, University of Cambridge, England

[10] M.J.D. Powell
"On the Lagrange function of quadratic models that are defined by interpolation"
Internal report Department of Applied Mathematics and Theoretical Physics, University of Cambridge, England

Chapter 3

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 2.4.6 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 points which respects the constraint $\|s\|_2 \leq \Delta$.

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

$$q(s^*) \leq 0 \tag{3.1}$$

3.1 $H(\lambda^*)$ 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 1.5 ($Hs = -g$) how to find it. We have seen in equation 1.7 that

$$H \text{ must be definite positive} \quad (s^T H s > 0 \quad \forall s) \tag{3.2}$$

in order to be able to apply 1.5.

If we found a value of s^* using 1.5 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:

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

$$H(\lambda^*)s^* = -g, \tag{3.3}$$

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

Theorem 1

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 8.3). This gives us:

$$\mathcal{L}(s, \lambda) = q(s) - \lambda c(s) \tag{3.4}$$

Using first part of equation 8.22,we have

$$\nabla_s \mathcal{L}(s^*, \lambda^*) = \nabla q(s^*) - \lambda^* \nabla_s c(s^*) = Hs^* + g + \lambda^* s^* = 0 \tag{3.5}$$

which is 3.3.

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

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

$$q(s^F) = q(s^*) + \langle s^F - s^*, g(s^*)\rangle + \frac{1}{2}\langle s^F - s^*, H(s^F - s^*)\rangle \tag{3.6}$$

Using the secant equation (see Annexes, section 8.4), $g'' - g' = H(x'' - x') = Hs \quad (s = x'' - x')$, we can rewrite 3.5 into $g(s^*) = -\lambda^* s^*$. This and the restriction that ($\|s^F\| = \|s^*\| = \Delta$) implies that:

$$\begin{aligned}
\langle s^F - s^*, g(s^*)\rangle =& \langle s^* - s^F, s^*\rangle \lambda^* \\
=& (\Delta^2 - \langle s^F, s^*\rangle)\lambda^* \\
=& [\frac{1}{2}(\langle s^F, s^F\rangle + \langle s^*, s^*\rangle) - \langle s^F, s^*\rangle]\lambda^* \\
=& [\frac{1}{2}(\langle s^F, s^F\rangle + \langle s^*, s^*\rangle) - \frac{1}{2}\langle s^F, s^*\rangle - \frac{1}{2}\langle s^F, s^*\rangle]\lambda^* \\
=& \frac{1}{2}[\langle s^F, s^F\rangle - \langle s^F, s^*\rangle + \langle s^*, s^*\rangle - \langle s^F, s^*\rangle]\lambda^* \\
=& \frac{1}{2}[\langle s^F, s^F - s^*\rangle + \langle s^* - s^F, s^*\rangle]\lambda^* \\
=& \frac{1}{2}[\langle s^F, s^F - s^*\rangle - \langle s^*, s^F - s^*\rangle]\lambda^* \\
\langle s^F - s^*, g(s^*)\rangle =& \frac{1}{2}\langle s^F - s^*, s^F - s^*\rangle \lambda^*
\end{aligned} \tag{3.7}$$

Combining 3.6 and 3.7

$$\begin{aligned} q(s^F) =& q(s^*) + \frac{1}{2}\langle s^F - s^*, s^F - s^*\rangle\lambda^* + \frac{1}{2}\langle s^F - s^*, H(s^F - s^*)\rangle \\ =& q(s^*) + \frac{1}{2}\langle s^F - s^*, (H + \lambda^* I)(s^F - s^*)\rangle \\ =& q(s^*) + \frac{1}{2}\langle s^F - s^*, H(\lambda^*)(s^F - s^*)\rangle \end{aligned} \tag{3.8}$$

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 3.8, 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 3.8 shows that $q(s^F) > q(s^*)$ whenever s^F is feasible. Thus s^* is the unique global minimizer.

Using 3.2 (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

$$H(\lambda^*)s^* = -g, \tag{3.9}$$

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 8.3 for explanation, equation 8.22).

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.

3.2 Explanation of the Hard case.

Theorem 2 tells us that we should be looking for solutions to 3.9 and implicitly tells us what value of λ we need. Suppose that H has an eigendecomposition:

$$H = U^T \Lambda U \tag{3.10}$$

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

$$H(\lambda) = U^T(\Lambda + \lambda I)U \tag{3.11}$$

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:

$$s(\lambda) = -H(\lambda)^{-1}g = -U^T(\Lambda + \lambda I)^{-1}Ug \tag{3.12}$$

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:

$$\psi(\lambda) = \|U^T(\Lambda + \lambda I)^{-1}Ug\|_2^2 = \|(\Lambda + \lambda I)^{-1}Ug\|_2^2 = \sum_{i=1}^{n} \frac{\gamma_i^2}{\lambda_i + \lambda} \tag{3.13}$$

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

3.2.1 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 3.1. 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 3.9. If $\Delta^2 < 1.5$, there exist a unique value of $\lambda = \lambda^*$ (given in the figure and by

$$\|s(\lambda)\|_2 - \Delta = 0 \tag{3.14}$$

which, used inside 3.9, give the optimal s^*.

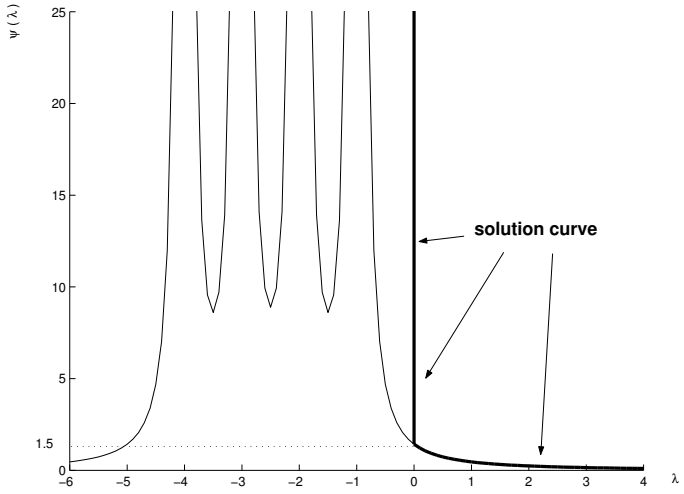

Figure 3.1: A plot of $\psi(\lambda)$ for H positive definite.

3.2.2 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 3.2. 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 3.14, we obtain the optimal s^* using 3.9.

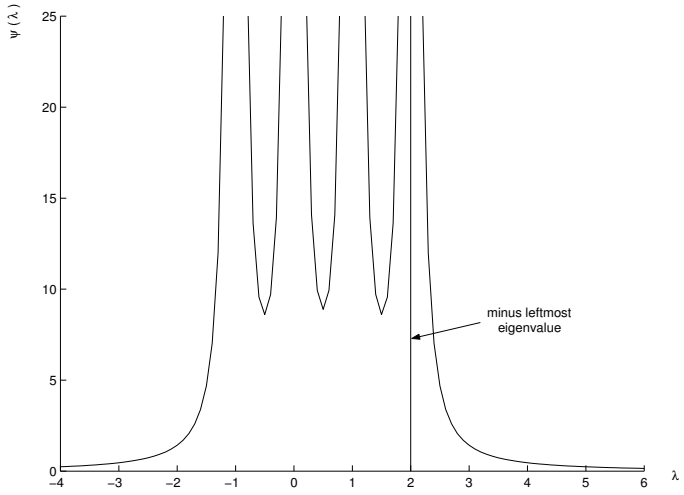

Figure 3.2: A plot of $\psi(\lambda)$ for H indefinite.

3.2.3 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 3.3. 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 3.9 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 3.9 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:

$$H(-\lambda_1)(s_{cri} + \alpha u_1) = -g \tag{3.15}$$

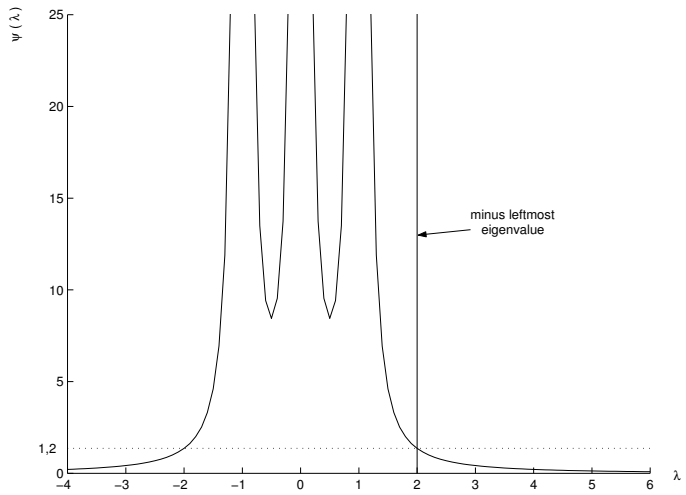

Figure 3.3: A plot of $\psi(\lambda)$ for H semi-definite and singular(hard case).

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.

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

We will apply the 1D-optimization algorithm called "1D Newton's search" (see Annexes, section 8.5) to the *secular equation*:

$$\phi(\lambda) = \frac{1}{\|s(\lambda)\|_2} - \frac{1}{\Delta} \tag{3.16}$$

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:

$$\phi'(\lambda) = -\frac{\langle s(\lambda), \nabla_\lambda s(\lambda) \rangle}{\|s(\lambda)\|_2^3} \tag{3.17}$$

where

$$\nabla_\lambda s(\lambda) = -H(\lambda)^{-1} s(\lambda) \tag{3.18}$$

The proof of these properties will be skipped.

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

$$\lambda_{k+1} = \lambda_k - \frac{\phi(\lambda)}{\phi'(\lambda)} \tag{3.19}$$

we need to evaluate the function $\phi(\lambda)$ and $\phi'(\lambda)$. The value of $\phi(\lambda)$ can be obtained by solving the equation 3.9 to obtain $s(\lambda)$. The value of $\phi'(\lambda)$ is available from 3.17 once $\nabla_\lambda s(\lambda)$ has been found using 3.18. 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 8.6 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 3.17. The simple relationship

$$\langle s(\lambda), H(\lambda)^{-1}s(\lambda)\rangle = \langle s(\lambda), L^{-T}L^{-1}s(\lambda)\rangle = \langle L^{-1}s(\lambda), L^{-1}s(\lambda)\rangle = \|\omega\|^2 \tag{3.20}$$

explains why we compute ω in step 3 of the following algorithm. Step 4 of the algorithm follows directly from 3.17 and 3.19. 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. $$\text{Replace } \lambda \text{ by } \lambda + (\frac{\|s(\lambda)\|_2 - \Delta}{\Delta})(\frac{\|s(\lambda)\|_2^2}{\|\omega\|_2^2}) \tag{3.21}$$
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.

3.4 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 3.21. 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 paragraph, 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 3.15 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 3.5).

- **Failure:** Improve λ_L using Rayleigh's quotient trick (see section 3.8). Use the "alternative" algorithm: pick λ inside $[\lambda_L \lambda_U]$ (see section 3.5).

4. go back to step 3.

3.5 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 ($\theta = 0.01$):

$$\lambda = max(\sqrt{\lambda_L \lambda_U}, \lambda_L + \theta(\lambda_U - \lambda_L)) \tag{3.22}$$

3.6 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:

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

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

3.7 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 3.15). 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

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

3.8 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:

$$\lambda_{min}[H] \leq \frac{\langle p, Hp\rangle}{\langle p, p\rangle} \leq \lambda_{max}[H] \tag{3.23}$$

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

$$H(\lambda + \delta e_k e_k^T)v = 0 \tag{3.24}$$

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 3.24 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 $\lambda_L = \max[\lambda_L, \lambda + \frac{\delta}{\|v\|_2^2}]$

3.9 Termination Test.

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

$$\langle v, H(\lambda)v\rangle \leq \kappa\,(\langle s(\lambda), H(\lambda)s(\lambda)\rangle + \lambda\Delta^2) \tag{3.25}$$

from some $\kappa \in (0,1)$ then $\hat{s} = s(\lambda)+v$ achieves the condition (see equation 3.1):

$$q(\hat{s}) \leq (1-\kappa)q(s^*) \tag{3.26}$$

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

$$\begin{aligned} &= -\langle H(\lambda)s(\lambda), (s(\lambda)+v)\rangle + \frac{1}{2}\langle s(\lambda)+v, H(\lambda)(s(\lambda)+v)\rangle - \frac{1}{2}\lambda\|s(\lambda)+v\|_2^2 \\ &= \frac{1}{2}\langle v, H(\lambda)v\rangle - \frac{1}{2}\langle s(\lambda), H(\lambda)s(\lambda)\rangle - \frac{1}{2}\lambda\|s(\lambda)+v\|_2^2 \end{aligned} \tag{3.27}$$

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

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

$$\Rightarrow \quad -\frac{1}{2}(\langle s(\lambda), H(\lambda)s(\lambda)\rangle + \lambda\Delta^2) \leq q(s^*) \tag{3.28}$$

From 3.27, 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 3.25:)

$$\begin{aligned} &\leq \frac{1}{2}\,\kappa\,(\langle s(\lambda), H(\lambda)s(\lambda)\rangle + \lambda\Delta^2) - \frac{1}{2}\langle s(\lambda), H(\lambda)s(\lambda)\rangle - \frac{1}{2}\lambda\Delta^2 \\ &\leq -\frac{1}{2}\,(1-\kappa)\,(\langle s(\lambda), H(\lambda)s(\lambda)\rangle + \lambda\Delta^2) \end{aligned} \tag{3.29}$$

Combining 3.28 and 3.29, we obtain finally 3.26.

3.9.1 $s(\lambda)$ is near the boundary of the trust region: normal case

Lemma

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

$$q(s(\lambda)) < (1 - \kappa_{easy}^2) q(s^*) \tag{3.30}$$

From the hypothesis:

$$\begin{aligned} |\|s(\lambda)\|_2 - \Delta| &\leq \kappa_{easy}\Delta \\ \|s(\lambda)\|_2 &\geq (1 - \kappa_{easy})\Delta \end{aligned} \tag{3.31}$$

Combining 3.31 and 3.27 when $v = 0$ reveals that:

$$\begin{aligned} q(s(\lambda)) &= -\frac{1}{2}(\langle s(\lambda), H(\lambda)s(\lambda)\rangle + \lambda\|s(\lambda)\|_2^2) \\ &\leq -\frac{1}{2}(\langle s(\lambda), H(\lambda)s(\lambda)\rangle + \lambda(1 - \kappa_{easy})^2\Delta^2) \\ &\leq -\frac{1}{2}(1 - \kappa_{easy})^2(\langle s(\lambda), H(\lambda)s(\lambda)\rangle + \lambda\Delta^2) \end{aligned} \tag{3.32}$$

The required inequality 3.30 is immediate from 3.28 and 3.32.

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

3.9.2 $s(\lambda)$ is inside the trust region: hard case

We will choose $\hat{s}$ as (see paragraph containing equation 3.15 for the meaning of α^* and u_1):

$$\hat{s} = s(\lambda) + \alpha^* u_1 \tag{3.33}$$

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

$$\alpha^2\langle u, H(\lambda)u\rangle < \kappa_{hard}(s(\lambda)^T H(\lambda)s(\lambda) + \lambda\Delta^2) \tag{3.34}$$

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

3.10 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

Bibliography

[1] Andrew R. Conn, Nicholas I.M.Gould, Philippe L.Toint
"Trust-region Methods"
MPS-SIAM series on Optimization

Chapter 4

The secondary Trust-Region subproblem

We seek an approximation of the solution s^*, of the minimization 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 minimization problem is equivalent (after a translation) and will be discussed in the chapter:

$$\max_{s\in\Re^n} |q(s)| \equiv |\langle g, s\rangle + \frac{1}{2}\langle s, Hs\rangle| \tag{4.1}$$
$$\text{subject to } \|s\|_2 \leq \Delta$$

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 $\|s\|_2 \leq \Delta$.

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

$$\max_{s\in\Re^n} |\langle g, s\rangle| + \frac{1}{2}|\langle s, Hs\rangle| \tag{4.2}$$
$$\text{subject to } \|s\|_2 < \Delta$$

Now, if $\hat{s}$ and $\tilde{s}$ are 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 4.1, 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

$$\begin{aligned} |\langle g, s\rangle| + \frac{1}{2}|\langle s, Hs\rangle| &\leq \left(|\langle g, \hat{s}\rangle| + \frac{1}{2}|\langle \hat{s}, H\hat{s}\rangle|\right) + \left(|\langle g, \tilde{s}\rangle| + \frac{1}{2}|\langle \tilde{s}, H\tilde{s}\rangle|\right) \\ &\leq 2\max\left[|\langle g, \hat{s}\rangle| + \frac{1}{2}|\langle \hat{s}, H\hat{s}\rangle|,\ \ |\langle g, \tilde{s}\rangle| + \frac{1}{2}|\langle \tilde{s}, H\tilde{s}\rangle|\right] \end{aligned} \tag{4.3}$$

(4.4)

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

$$\begin{aligned}
\|H\omega\| &\geq \|\omega\|^2 = \max_k\{\|v_k\| : k = 1, \ldots, n\}\|\omega\| \\
&\geq \|\omega\|\sqrt{\frac{1}{n}\sum_{k=1}^{n}\|v_k\|^2} \qquad (4.5) \\
&\geq \frac{\|\omega\|}{\sqrt{n}}\,\sigma(H) \qquad (4.6)
\end{aligned}$$

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:

$$H = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & -2/3 & -2/3 \\ 1 & -2/3 & -1 & -2/3 \\ 1 & -2/3 & -2/3 & -1 \end{pmatrix} \qquad (4.7)$$

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

$$\frac{|\langle\alpha\omega + \beta H\omega, H(\alpha\omega + \beta H\omega)\rangle|}{\|\alpha\omega + \beta H\omega\|^2} \qquad (4.8)$$

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.

4.1 Generating $\tilde{s}$.

Let us define $V := \omega$, $D := H\omega$, $r := \beta/\alpha$, equation 4.8, 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 4.8. $\tilde{s}$ is thus $V + r^* D$.

4.2 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

$$\begin{cases} \tilde{u} = \cos(\theta)G + \sin(\theta)V \\ \hat{u} = -\sin(\theta)G + \cos(\theta)V \end{cases} \qquad \text{we have directly } \hat{u}^T\tilde{u} = 0 \tag{4.9}$$

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

$$\begin{aligned} &(V^T HG)\cos(2\theta) + \frac{G^T HG - V^T HV}{2}\sin(2\theta) = 0 \\ \Leftrightarrow &\theta = \frac{1}{2} arctg(\frac{2V^T HG}{V^T HV - G^T HG}) \end{aligned} \tag{4.10}$$

Using the value of θ from equation 4.10 in equation 4.9 give the required $\hat{u}$ and $\tilde{u}$.

4.3 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 4.2.

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

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

$$|\tilde{s}^T H \tilde{s}| \geq \frac{1}{2}\frac{1}{\sqrt{n}}\sigma(H)\rho^2 \tag{4.11}$$

The proof will be skipped.

Bibliography

[1] M .J.D. Powell
"UOBYQA: unconstrained optimization by quadratic approximation"
Internal report Department of Applied Mathematics and Theoretical Physics, University of Cambridge, England

Chapter 5

The QP_TR 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 error 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 2.4.5.
2. Choose the index k of the best (lowest) point of the set $\{\boldsymbol{x}_{(1)}, \ldots, \boldsymbol{x}_{(N)}\}$. 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 2.3.2).
3. Find 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 3.
4. If $\|s\| < \dfrac{\rho}{2}$, then break and go to step 14: we need to be sure that the model is valid before doing a step so small.
5. Let $R := q(\boldsymbol{x}_{(k)}) - q(\boldsymbol{x}_{(k)} + s^*) \geq 0$, the predicted reduction of the objective function.
6. 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 14.
7. 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}.
8. Compute the agreement r between $f(x)$ and the model $q(x)$:
$$r = \frac{F_{old} - F_{new}}{R} \tag{5.1}$$
9. Update the local trust region radius: change Δ to:
$$\begin{cases} \max[\Delta, \frac{5}{4}\|s\|, \rho + \|s\|] & \text{if } 0.7 \leq r, \\ \max[\frac{1}{2}\Delta, \|s\|] & \text{if } 0.1 \leq r < 0.7, \\ \frac{1}{2}\|s\| & \text{if } r < 0.1 \end{cases} \tag{5.2}$$

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

10. Store $\boldsymbol{x}_{(k)} + s^*$ inside the interpolation dataset: choose the point $\boldsymbol{x}_{(t)}$ to remove using technique of section 2.4.3 and replace it by $\boldsymbol{x}_{(k)} + s^*$ using the technique of section 2.4.4. Let us define the $ModelStep := \|\boldsymbol{x}_{(t)} - (\boldsymbol{x}_{(k)} + s^*)\|$

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

12. Update the value of M which is used during the check of the validity of the polynomial around $\boldsymbol{x}_{(k)}$ (see section 2.4.1 and more precisely equation 2.34).

13. 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 3.

14. We must now check the validity of our model using the technique of section 2.4.2. We will need, to check this validity, a parameter ϵ: see section 5.1 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 2.4.2. Eventually, we will have to update the index k of the best point in the dataset and F_{old}. Once this is finished, go back to step 3.
    - **Model is valid** If $\|s^*\| > \rho$ go back to step 3, otherwise continue.

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

16. Update of the global trust region radius.

$$\rho_{new} = \begin{cases} \rho_{end} & \text{if } \rho_{end} < \rho \leq 16\rho_{end} \\ \sqrt{\rho_{end}\,\rho} & \text{if } 16\rho_{end} < \rho \leq 250\rho_{end} \\ 0.1\rho & \text{if } 250\rho_{end} < \rho \end{cases} \tag{5.3}$$

    Set $\Delta := \max[\frac{\rho}{2}, \rho_{new}]$. Set $\rho := \rho_{new}$. Go back to step 3.

17. 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 2.26) and to the whole dataset $\{\boldsymbol{x}_{(1)}, \ldots, \boldsymbol{x}_{(N)}\}$.

18. The iteration are now complete but one more value of $f(x)$ may be required before termination. Indeed, we recall from step 4 and step 6 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 $\delta_k \leq \Delta_k$.

5.1 The bound ϵ.

See section 2.4.2 to know about ϵ.

If we have updated the value of M less than 10 times, we will set $\epsilon := 0$ (see section 2.4.1 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 QP_TR 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 3.10 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.

5.2 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:

$$\|\boldsymbol{x}_{(j)} - \boldsymbol{x}_{(k)}\| \leq 2\rho \tag{5.4}$$

$$\frac{1}{6}M\|\boldsymbol{x}_{(j)} - \boldsymbol{x}_{(k)}\|^3 \max_d\{|P_j(\boldsymbol{x}_{(k)} + d)| : \|d\| \leq \rho\} \leq \epsilon \tag{5.5}$$

must hold for every points in the dataset. When ρ is reduced by formula 5.3, the equation 5.4 is very often NOT verified. Only equation 5.5, 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.

Bibliography

[1] Andrew R. Conn, Nicholas I.M.Gould, Philippe L.Toint
"Trust-region Methods"
MPS-SIAM series on Optimization

[2] M.J.D. Powell
"UOBYQA: unconstrained optimization by quadratic approximation"
Internal report Department of Applied Mathematics and Theoretical Physics, University of Cambridge, England

Chapter 6

Numerical Results of QP_TR algorithm.

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

$$f(x) = \sum_{i=1}^{n} \left[a_i - \sum_{j=1}^{n} (S_{ij} \sin x_j + C_{ij} \cos x_j) \right]^2, x \in \Re^n \tag{6.1}$$

The way of generating the parameters of $f(x)$ is taken from Fletcher and Powell (1963), 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 $a_i, i = 1, \ldots, n$ 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 QP_TR 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 QP_TR: $\rho_{start} = 0.1 \quad \rho_{end} = 10^{-8}$.
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	
	QP_TR	CFSQP	QP_TR	CFSQP	QP_TR	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 5.2:

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 as before except for the starting point which is $(-1.2\ ;\ 1.0)$.

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

Bibliography

[1] R. Fletcher and M.J.D. Powell (1963), "A Rapidly convergent descent method for minimization", Comput. J., Vol. 8, pp 33-41.

Chapter 7

Conclusions

The algorithm seems to be very powerful compared to older algorithms using BGFS update. This is mainly due to the better approximation of the Hessian matrix (see introduction).

However, 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 ($= (n+1)(n+2)/2$). It would be interesting to build the first polynomial using another technique which requires less points (for example, see the paper "Least frobenius norm updating of quadratic models that satisfy interpolation conditions" from M.J.D.Powell).
- 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".

A 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.

Another improvement, is to parallelize the algorithm.

One last improvement, would be to use the H-norm instead of the $L2$-norm during the calculation of the trust region step (the algorithm of chapter 3 is only for $\|\cdot\|_2 = L2$-norm).

7.1 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

$$\|s\|^2_{|H|} = \langle s, |H|s\rangle \tag{7.1}$$

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.

Bibliography

[1] Andrew R. Conn, Nicholas I.M.Gould, Philippe L.Toint
"Trust-region Methods"
MPS-SIAM series on Optimization

[2] M.J.D.Powell
"Least frobenius norm updating of quadratic models that satisfy interpolation conditions"
Internal report Department of Applied Mathematics and Theoretical Physics, University of Cambridge, England, october 2002

Chapter 8

Annexes

8.1 Line-Search addenda.

8.1.1 Speed of convergence of Newton's method.

We have:

$$B_k \delta_k = -g_k \tag{8.1}$$

The Taylor series of g around x_k is:

$$\begin{aligned} g(x_k + h) &= g(x_k) + B_k h + o(\|h\|^2) \\ \iff g(x_k - h) &= g(x_k) - B_k h + o(\|h\|^2) \end{aligned} \tag{8.2}$$

If we set in equation 8.2 $h = h_k = x_k - x^*$, 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 8.1:

$$\begin{aligned} & & 0 &= -\delta_k - h_k + o(\|h_k\|^2) \\ \iff & & \delta_k + h_k &= o(\|h_k\|^2) \\ \iff & (x_{k+1} - x_k) + (x_k - x^*) &= o(\|h_k\|^2) \\ \iff & x_{k+1} - x^* &= o(\|h_k\|^2) \\ \iff & h_{k+1} &= o(\|h_k\|^2) \end{aligned}$$

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.

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 of the algorithm before going in "one step" to the optimum. In fact, the algorithm becomes "only" super-linearly convergent.

8.1.2 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

$$x_{k+1} = x_k + \alpha_k s_k \tag{8.3}$$

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:

$$f(\alpha) \leq f(0) + \rho\alpha f'(0) \qquad \rho \in (0, \frac{1}{2}) \tag{8.4}$$

$$f'(\alpha) > \sigma f'(0) \qquad \sigma \in (\rho, 1) \tag{8.5}$$

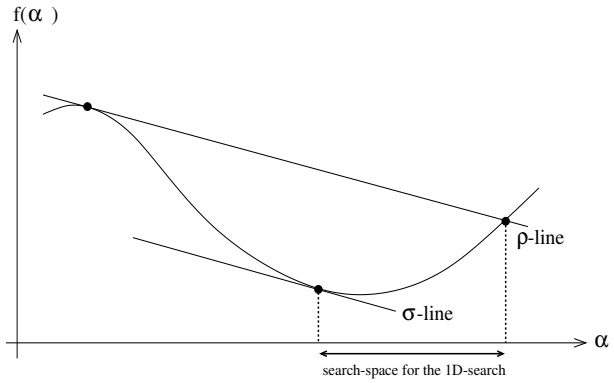

Figure 8.1: bounds on α: wolf conditions

The objective of the wolf conditions is to give a lower bound (equation 8.5) and upper bound (equation 8.4) on the value of α, such that the 1D-search algorithm is easier: see figure 8.1. Equation 8.4 expresses that the objective function $\mathcal{F}$ must be reduced sufficiently. Equation 8.5 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 ($= s_k$):

$$\cos(\theta_k) = \frac{-g_k^T s_k}{\|g_k\|\|s_k\|} \tag{8.6}$$

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:

$$\exists L > 0 : \ \|g(x) - g(\tilde{x})\| < L\|x - \tilde{x}\| \quad \forall x, \tilde{x} \in \mathcal{N} \tag{8.7}$$

We have:

$$\textbf{Zoutendijk Theorem: } \sum_{k>1} \cos^2(\theta_k)\|g_k\|^2 < \infty \tag{8.8}$$

From equation 8.5, we have:

$$\begin{aligned} f'(\alpha) &> \sigma f'(0) \\ \iff g_{k+1}^T s_k &> \sigma g_k^T s_k \\ \text{we add } -g_k^T s_k \text{ on both side:} & \\ \iff (g_{k+1}^T - g_k^T)s_k &> (\sigma - 1)g_k^T s_k \end{aligned} \tag{8.9}$$

From equation 8.7, we have:

$$\begin{aligned} \|g_{k+1} - g_k\| &< L\|x_{k+1} - x_k\| \\ \text{using equation 8.3 :} & \\ \iff \|g_{k+1} - g_k\| &< \alpha_k L\|s_k\| \\ \text{we multiply by } \|s_k\| \text{ both sides:} & \\ \iff (g_{k+1} - gk^T)s_k < \|g_{k+1} - g_k\|\|s_k\| &< \alpha_k L\|s_k\|^2 \end{aligned} \tag{8.10}$$

Combining equation 8.9 and 8.10 we obtain:

$$\begin{aligned} (\sigma - 1)g_k^T s_k &< (g_{k+1} - gk^T)s_k < \alpha_k L\|s_k\|^2 \\ \iff \frac{(\sigma - 1)g_k^T s_k}{L\|s_k\|^2} &< \alpha_k \end{aligned} \tag{8.11}$$

We can replace in equation 8.4:

$$f(\alpha) \leq f(0) + \underbrace{\underbrace{\underbrace{\rho}_{>0} \quad \underbrace{f'(0)}_{=g_k^T s_k<0}}_{<0} \alpha_k}$$

the α_k by its lower bound from equation 8.11. We obtain:

$$f_{k+1} \leq fk + \frac{\rho(\sigma-1)(g_k^T s_k)^2}{L\|s_k\|^2} \frac{\|g_k\|^2}{\|g_k\|^2}$$

If we define $c = \frac{\rho(\sigma-1)}{L} < 0$ and if we use the definition of θ_k (see eq. 8.6), we have:

$$f_{k+1} \leq f_k + \underbrace{\underbrace{c}_{<0} \underbrace{\cos^2(\theta_k)\|g_k\|^2}_{>0}}_{<0} \tag{8.12}$$

$$\frac{f_{k+1} - f_k}{c} \geq \cos^2(\theta_k)\|g_k\|^2 \tag{8.13}$$

Summing equation 8.13 on k, we have

$$\frac{1}{c}\sum_k (f_{k+1} - f_k) \geq \sum_k \cos^2(\theta_k)\|g_k\|^2 \tag{8.14}$$

We know that $\mathcal{F}$ is bounded below, we also know from equation 8.12, that $f_{k+1} \leq f_k$. 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 8.14 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 8.8), 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.

8.2 Gram-Schmidt orthogonalization procedure.

We have a set of independent vectors $\{a_1, a_2, \ldots, a_n\}$. We want to convert it into a set of orthonormal vectors $\{b_1, b_2, \ldots, b_n\}$ 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**

$$\tilde{b_k} = a_k - \sum_{j=1}^{k} < a_k, b_j > b_j \tag{8.15}$$

   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**

$$b_k = \frac{\tilde{b_k}}{\|\tilde{b_k}\|} \tag{8.16}$$

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

Algorithm 2.

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

$$b_k = \frac{a_k}{\|a_k\|} \tag{8.17}$$

3. **Orthogonalisation** for $j = k + 1$ to n do:

$$a_j = a_j - < a_j, b_k > b_k \quad j = k + 1, \ldots, n \tag{8.18}$$

   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.

8.3 Notions of constrained optimization

Let us define the problem:

Find the minimum of $f(x)$ subject to r constraints $c_j(x) \leq 0 (j = 1, ..., r)$.

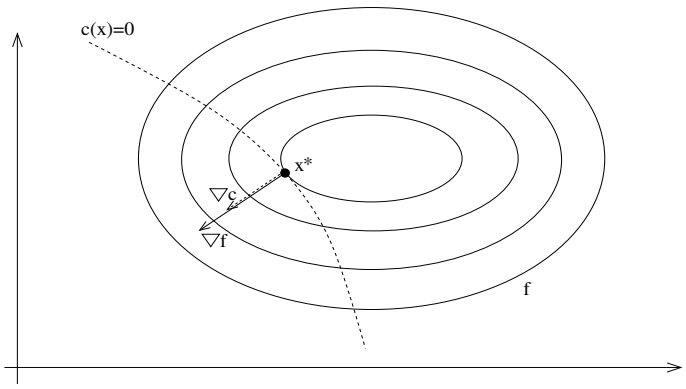

Figure 8.2: Existence of Lagrange Multiplier λ

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 8.2) (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:

$$\nabla f = g = \sum_{j \in E} \lambda_j \nabla(c_j) \qquad\qquad c_j(x) = 0,\ j \in E \tag{8.19}$$

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:

$$\mathcal{L}(x, \lambda) = f(x) - \sum_i \lambda_i c_i(x). \tag{8.20}$$

The equation 8.19 is then equivalent to:

$$\blacktriangledown \mathcal{L}(x^*, \lambda^*) = 0 \text{ where } \blacktriangledown = \begin{pmatrix} \nabla_x \\ \nabla_\lambda \end{pmatrix} \tag{8.21}$$

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:

$$(x^*, \lambda^*) \text{ is a KKT point} \iff \begin{array}{rcl} \nabla_x \mathcal{L}(x^*, \lambda^*) & = & 0 \\ \lambda_j^*\, c_j(x^*) & = & 0,\ \ i = 1, ..., r \end{array} \tag{8.22}$$

The second equation of 8.22 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 8.3.

8.4 The secant equation

Let us define a general polynomial of degree 2:

$$q(x) = q(0) + < g(0), x > + \frac{1}{2} < x, H(0)x > \tag{8.23}$$

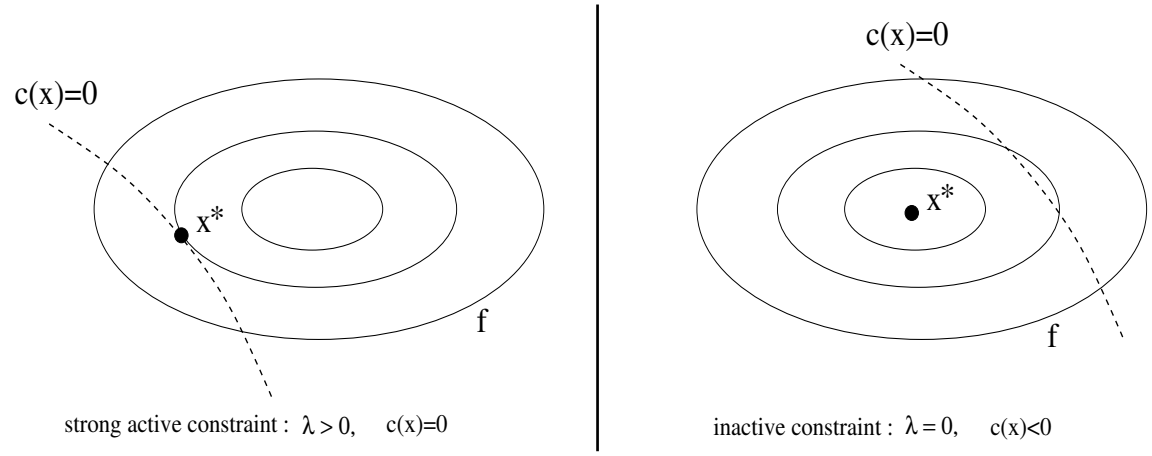

Figure 8.3: complementarity condition

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 8.23 (using $u = x, v = H(0)x$) that

$$\nabla q(x) = g(x) = H(0)x + g(0) \tag{8.24}$$

$$\nabla^2 q(x) = H(0)$$

A consequence of 8.24 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

$$g_{(2)} - g_{(1)} = H(x_{(2)} - x_{(1)}) \tag{8.25}$$

This is called the "*Secant Equation*". That is the Hessian matrix maps the differences in position into differences in gradient.

8.5 1D Newton's search

Suppose we want to find the root of $f(x) = x^2 - 3$ (see figure 8.4). 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

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)} \tag{8.26}$$

which gives $x_{k+1} = 2 - \frac{1}{4} = 1.75$. We apply the same process and iterate on k.

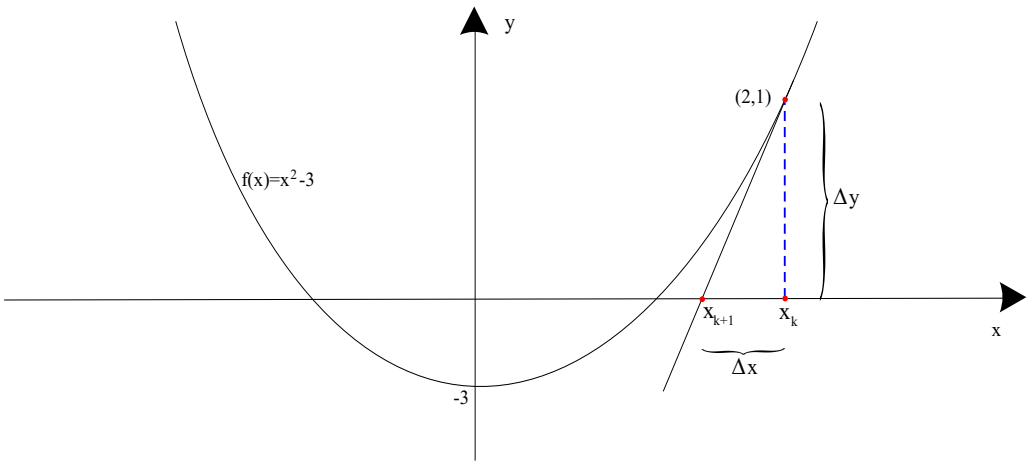

Figure 8.4: A plot of $\psi(\lambda)$ for H indefinite.

8.6 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:

$$L \cdot L^T = A \tag{8.27}$$

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:

$$L \cdot U = A \tag{8.28}$$

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:

$$\begin{pmatrix} \alpha_{11} & 0 & 0 & 0 \\ \alpha_{21} & \alpha_{22} & 0 & 0 \\ \alpha_{31} & \alpha_{32} & \alpha_{33} & 0 \\ \alpha_{41} & \alpha_{42} & \alpha_{43} & \alpha_{44} \end{pmatrix} \cdot \begin{pmatrix} \beta_{11} & \beta_{12} & \beta_{13} & \beta_{14} \\ 0 & \beta_{22} & \beta_{23} & \beta_{24} \\ 0 & 0 & \beta_{33} & \beta_{34} \\ 0 & 0 & 0 & \beta_{44} \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix} \tag{8.29}$$

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 2 systems are trivial to solve because they are triangular.

8.6.1 Performing LU decomposition.

First let us rewrite the component a_{ij} of A from the equation 8.28 or 8.29. 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:

$$i < j: \qquad a_{ij} = \alpha_{i1}\beta_{1j} + \alpha_{i2}\beta_{2j} + \cdots + \alpha_{ii}\beta_{ij} \tag{8.30}$$

$$i = j: \qquad a_{ii} = \alpha_{i1}\beta_{1i} + \alpha_{i2}\beta_{2i} + \cdots + \alpha_{ii}\beta_{ii} \tag{8.31}$$

$$i > j: \qquad a_{ij} = \alpha_{i1}\beta_{1j} + \alpha_{i2}\beta_{2j} + \cdots + \alpha_{ij}\beta_{jj} \tag{8.32}$$

Equations 8.30 - 8.32, 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 are invited 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:

$$\alpha_{ii} \equiv 1 \quad i = 1, \ldots, n \tag{8.33}$$

A surprising procedure, now, is *Crout's algorithm*, which, quite trivially, solves the set of $n^2 + n$ equations 8.30 - 8.32 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 8.30, 8.31 and 8.33 to solve for β_{ij}, namely

  $$\beta_{ij} = a_{ij} - \sum_{k=1}^{i-1} \alpha_{ik}\beta_{kj} \tag{8.34}$$

  Second, for $i = j + 1, \ldots, n$, use 8.32 to solve for α_{ij}, namely

  $$\alpha_{ij} = \frac{1}{\beta_{jj}} \left(a_{ij} - \sum_{k=1}^{j-1} \alpha_{ik}\beta_{kj} \right) \tag{8.35}$$

  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 8.34 and 8.35 are already determined by the time they are needed.

8.6.2 Performing Cholesky decomposition.

We can obtain the analogs of equations 8.34 and 8.35 for the Cholesky decomposition:

$$L_{ii} = \sqrt{a_{ii} - \sum_{k=1}^{i-1} L_{ik}^2} \tag{8.36}$$

and

$$L_{ji} = \frac{1}{L_{ii}} \left(a_{ij} - \sum_{k=1}^{i-1} L_{ik}L_{jk} \right) \quad j = i + 1, \ldots, n \tag{8.37}$$

If you apply equation 8.36 and 8.37 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 8.36.

What about pivoting? Pivoting (that is: the selection of a salubrious pivot element for the division in equation 8.37) 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.

Bibliography

[1] R.Fletcher
"Practical Methods of optimization"
"a Wiley-Interscience pblication" ISBN 0 471 91547 5.

[2] J.E.Dennis, Jr, Robert B.Schnabel
"Numerical Methods for unconstrained Optimization and nonlinear Equations"
Classics in applied mathematics 16, SIAM

[3] Andrew R. Conn, Nicholas I.M.Gould, Philippe L.Toint
"Trust-region Methods"
MPS-SIAM series on Optimization

[4] William H. Press, Saul A.Teukolsky, William T.Vetterling, Brian P.Flannery
"Numerical recipes in C" (second edition)
Cambridge University Press

Chapter 9

Code of the optimizer.

The code is made 100% by me.

9.1 Main

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include "Solver.h"
#include "tools.h"

Vector xOptimal,A, xStart;
Matrix S,C;

void initF(int n)
{
    initRandom();
    xOptimal.setSize(n);
    xStart.setSize(n);
    A.setSize(n);
    S.setSize(n,n);
    C.setSize(n,n);

    double *xo=xOptimal, *xs=xStart, *a=A, **s=S, **c=C, sum;
    int i,j;
    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;
    }
}

void saveF(char *filename)
{
    char buf[300], *t;
    strcpy(buf, filename);
    int l=strlen(buf);
    t=buf+l;

    strcpy(t,"_xo.dat"); xOptimal.save(buf);
    strcpy(t,"_xs.dat");   xStart.save(buf);
    strcpy(t,"_a.dat" );        A.save(buf);
    strcpy(t,"_s.dat" );        S.save(buf,0);
    strcpy(t,"_c.dat" );        C.save(buf,0);
}

void loadF(char *filename)
{
    char buf[300], *t;
    strcpy(buf, filename);
```

```
    int l=strlen(buf);
    t=buf+l;

    strcpy(t,"_xo.dat"); xOptimal=Vector(buf);
    strcpy(t,"_xs.dat");   xStart=Vector(buf);
    strcpy(t,"_a.dat" );        A=Vector(buf);
    strcpy(t,"_s.dat" );        S=Matrix(buf);
    strcpy(t,"_c.dat" );        C=Matrix(buf);
}

double f(Vector X)
{
    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);
    }

    return r;
}

double f2(Vector X)
{
    return 100*sqr(X[1]-sqr(X[0]))+sqr(1-X[0]);
}

void initF2(int n)
{
    if (n!=2) { printf("wrong dimension.\n"); exit(255); }
    xOptimal.setSize(2);
    xStart.setSize(2);
    xOptimal[0]=1.0;
    xOptimal[1]=1.0;
    xStart[0]=5.0;
    xStart[1]=-5.0;
}

void testL2()
{
    int n=3;
    double bestValueOF, rhoStart=1e-1, rhoEnd=1e-8;
    int niter=1000;
    initF(n);    fprintf(stderr,"Fmin=%e\n", f(xOptimal));
//    initF2(n);    fprintf(stderr,"Fmin=%e\n", f2(xOptimal));
    Vector S=QPOptim(&bestValueOF,rhoStart, rhoEnd, niter, f, xStart);
    printf("\n\nDimension of the search space: %i\n",n);
    printf("Optimal value found: %e\n", bestValueOF);
    printf("distance to the optimum: %e\n", S.euclidianDistance(xOptimal));
    printf("finished\n");
}

int main()
{
 //   while (1)
    {
        system("cls");
        testL2();
        printf("\npress return"); getchar();
    }
    return 0;
}
```

9.2 Matrix manipulation

9.2.1 Header file

```
#ifndef _MPI_MATRIX_H
#define _MPI_MATRIX_H

#include "Vector.h"
#include "MatrixTriangle.h"

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(MatrixTriangle A);
    Matrix(char *filename);
    void save(char *filename,char ascii);
    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 setLine(int i, Vector v, int n=0);

// simple math tools:
    void zero();
    void multiply(Matrix R, Matrix Bplus); // result in R
    Matrix multiply(Matrix B);
    void multiplyInPlace(double d);
    void multiply(Vector R, Vector v);  // result in R
    Vector multiply(Vector v);
    void addInPlace(Matrix B);
    void addUnityInPlace(double d);
    void transposeInPlace();
    bool cholesky(MatrixTriangle matL, double lambda=0, double *lambdaCorrection=NULL);
    void choleskySolveInPlace(Vector b);
    double frobeniusNorm();
    double infinitumNorm();
    Vector getMaxColumn();

// default return matrix in case of problem in a function
    static Matrix emptyMatrix;
};

#endif
```

9.2.2 Code file

```
#include <memory.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include "matrix.h"
#include "tools.h"

Matrix Matrix::emptyMatrix;
MatrixTriangle MatrixTriangle::emptyMatrixTriangle(0);

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);
};

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");
    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;
    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;
    setExtSize(d->extLine,_nColumn);
}

void Matrix::setSize(int _nLine,int _nColumn)
{
    d->nLine=_nLine;
    d->nColumn=_nColumn;
    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);
        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");
            exit(255);
        }

        i=_extLine;
        while (i--)
        {
            *(tmp3++)=tmp2;
            tmp2+=_extColumn;
        };

        if ((nc)&&(d->nLine))
        {
            tmp2=oldBuffer;
            i=d->nLine;
            nc*=sizeof(double);
            while(i--)
            {
                memcpy(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");
            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; j++)
                fprintf(f,"%1.10f ",p[i][j]);
            fprintf(f,"\n");
        }
    } 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::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");
        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;

    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]);
        printf("\n");
    }
}

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");
        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;
    };
}

Matrix::Matrix(MatrixTriangle A)
{
    int n=A.nLine(),i,j;
    init(n,n,n,n);
    double **pD=(*this), **pS=A;
    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;
}

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);
    for (i=0; i<nl; i++)
        for (j=0; j<nc; j++) dp[j][i]=sp[i][j];
}

//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()*nColumn()*sizeof(double));
};

void Matrix::multiply(Matrix R, Matrix Bplus)
{
    if (Bplus.nLine()!=nColumn())
    {
        printf("(matrix * matrix) error");
        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;
        }
}

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 (nl!=(int)v.sz())
    {
        printf("matrix multiply error");
        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;
    }
}

Vector Matrix::multiply(Vector v)
{
    Vector r(nLine());
    multiply(r,v);
```

```
    return r;
}

void Matrix::addInPlace(Matrix B)
{
    if ((B.nLine()!=nLine())||
        (B.nColumn()!=nColumn()))
    {
        printf("matrix addition error");
        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];
}

//inline double sqr(double a){return a*a;};

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;
               while(i--)
               {
                   sum=x[i];
                   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::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::infinitumNorm()
{
// 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=::max(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];
    return rr;
}

Vector Matrix::getLine(int i, int n)
{
    if (n==0) n=nColumn();
    Vector r(n,d->p[i]);
    return r;
}

void Matrix::setLine(int i, Vector v, int n)
{
    if (n==0) n=nColumn();
    memcpy(d->p[i], (double*)v, n*sizeof(double));
}
```

9.3 Triangular Matrix manipulation

9.3.1 Header file

```
#ifndef _MPI_MATRIXTRIANGLE_H
#define _MPI_MATRIXTRIANGLE_H

#include "Vector.h"

class Matrix;

class MatrixTriangle // lower triangular
{
  friend Matrix;
  protected:
    void destroyCurrentBuffer();
    typedef struct MatrixTriangleDataTag
    {
       int n;
       int ext;
       int ref_count;
    } MatrixTriangleData;
    MatrixTriangleData *d;
    double **p;

  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 int nLine() { return d->n; };
    inline double *operator [](int i) { return p[i]; };
    inline operator double**() const { return 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
```

9.3.2 Code file

```
#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=p=(double**)malloc(_n*sizeof(double*));
        t2=(double*)malloc((_n+1)*_n/2*sizeof(double));
        while(_n--)
        {
            *(t++)=t2; t2+=i; i++;
        }
    } else p=NULL;
}

void MatrixTriangle::setSize(int _n)
{
    d->n=_n;
    if (_n>d->ext)
    {
        d->ext=_n;
        double **t,*t2;
        t2=(double*)realloc(*p,(_n+1)*_n/2*sizeof(double));
        t=p=(double**)realloc(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;

    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)
    {
        free(d);
        if (p) { free(*p); free(p); }
    };
}

MatrixTriangle::MatrixTriangle(const MatrixTriangle &A)
{
    // shallow copy
    p=A.p;
    d=A.d;
    (d->ref_count)++ ;
}

MatrixTriangle& MatrixTriangle::operator=( const MatrixTriangle& A )
{
    // shallow copy
    if (this != &A)
    {
        destroyCurrentBuffer();
        p=A.p;
        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(*p,*(r.p),(n+1)*n/2*sizeof(double));
}
```

9.4 Vector manipulation

9.4.1 Header file

```
#ifndef _MPI_VECTOR_H
#define _MPI_VECTOR_H

#include <stdlib.h> // for the declaration of NULL

class Point;

class Vector
{
  friend Point;
  protected:
    // 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;
    } VectorData;
    VectorData *d;

  public:
// creation & management of Vector:
    Vector(int _n=0);
    Vector(int _n, int _ext);
    Vector(int _n, double *dd);
    Vector(char *filename);
    void extend();
    void setSize(int _n);
    void exactshape();
    void print();
    void save(char *filename);

// allow shallow copy:
    Vector clone();
    void copyFrom(Vector r);
    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 { return d->p; };
    //double &operator[]( unsigned i) {return p[i];};

// simple math tools:
    double euclidianNorm();
    double euclidianDistance(Vector v);
    double square();
    void multiply(double a);
    void zero();
    double scalarProduct(Vector v);
    double min();
    double max();
    bool isNull();
    Vector operator-( Vector v);
    Vector operator+( Vector v);
    Vector operator-=( Vector v);
    Vector operator+=( Vector v);

// default return Vector in case of problem in a function
    static Vector emptyVector;

};

#endif
```

9.4.2 Code file

```
#include <stdio.h>
#include <memory.h>
#include "Vector.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"); 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)
{
    alloc(n,n);
    if (dd) memcpy(d->p,dd,n*sizeof(double));
    else zero();
}

void Vector::zero()
{
    if (d->p) memset(d->p,0,d->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"); exit(253); }
        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; 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"); 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("]"); return; }

    double *up=d->p;
    while (--N) printf("%f,",*(up++));
    printf("%f]",*up);
}

Vector::~Vector()
{
    destroyCurrentBuffer();
};

void Vector::destroyCurrentBuffer()
{
    if (!d) return;
    (d->ref_count) --;
    if (d->ref_count==0)
    {
        if (d->p) 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)
{
    unsigned n=r.sz();
    if (n==0) return;
    if (n!=sz())
    {
        printf("Vector: copyFrom: size do not agree");
        exit(254);
    }
    memcpy(d->p,r.d->p,n*sizeof(double));
}

double Vector::euclidianNorm()
{
    return ::euclidianNorm(sz(), d->p);
}

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);
}

void Vector::multiply(double a)
{
    double *xp=d->p;
```

```
    int ni=sz();
    while (ni--) *(xp++)*=a;
}

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::min()
{
    if (sz()==0) return 0;
    double *xp=d->p, m=INF;
    int ni=sz();
    while (ni--) m=::min(m,*(xp++));
    return m;
}

double Vector::max()
{
    if (sz()==0) return 0;
    double *xp=d->p, m=-INF;
    int ni=sz();
    while (ni--) m=::max(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;
}

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);
}
```

9.5 Vector of integer manipulation

9.5.1 Header file

```
#ifndef _MPI_VectorIntINT_H
#define _MPI_VectorIntINT_H

#include <stdlib.h>

class VectorInt
{
  public:
    int *p;
    const int &n;

    VectorInt(): np(0), n(np), extention(0), p(NULL) {};
    VectorInt(int _n);
    VectorInt(int _n, int _ext);
    VectorInt(int _n, int *d);
    VectorInt( const VectorInt& P );
    VectorInt(VectorInt *v);
    ~VectorInt();

    void extend();
    void setN(int _n);
    void exactshape();
    void print();

    // only use the following method at your own risks!
    void prepareExtend(int new_extention);

//    int &operator [](int i) { return p[i]; };
    int operator==( const VectorInt& );
    VectorInt& operator=( const VectorInt& P );
    operator int*() const { return p; };
    operator unsigned*() const { return (unsigned*)p; };
    int &operator[]( unsigned i) {return p[i];};

  protected:
    int np,extention;

  private:
    void alloc();

};
#endif
```

9.5.2 Code file

```
#include <stdio.h>
#include <memory.h>
#include "VectorInt.h"

#define CHECK(p) if ((p)==NULL) { printf("memory allocation error\n"); exit(253); }

void VectorInt::alloc()
{
    CHECK(p=(int*)malloc(extention*sizeof(int)));
}

VectorInt::VectorInt(int _n): np(_n), n(np), extention(_n)
{
    alloc();
    memset(p,0,extention*sizeof(int));
};

VectorInt::VectorInt(int _n, int _ext): np(_n), n(np), extention(_ext)
{
    alloc();
    memset(p,0,extention*sizeof(int));
};

VectorInt::VectorInt(VectorInt *v): np(v->n), n(np), extention(v->n)
{
    alloc();
    memcpy(p,v->p,n*sizeof(int));
}

VectorInt::VectorInt(int _n, int *d): np(_n), n(np), extention(_n)
{
    alloc();
    if (d) memcpy(p,d,_n*sizeof(int));
    else memset(p,0,extention*sizeof(int));
}

void VectorInt::prepareExtend(int new_extention)
{
    if (extention<new_extention)
    {
```

```
        CHECK(p=(int*)realloc(p,new_extention*sizeof(int)));
        memset(p+extention,0,(new_extention-extention)*sizeof(int));
        extention=new_extention;
    };
};

void VectorInt::setN(int _n)
{
    np=_n;
    if (_n==0) { free(p); p=NULL; return; }
    prepareExtend(_n);
}

void VectorInt::extend()
{
    np++;
    if (n>extention) prepareExtend(extention+100);
}

void VectorInt::exactshape()
{
    if (extention!=0)
    {
        CHECK(p=(int*)realloc(p,n*sizeof(int)));
        extention=0;
    };
};

VectorInt::~VectorInt()
{
    if (p) free(p);
};

int VectorInt::operator==( const VectorInt& Q )
{
  if (n != Q.n) return 0;

  int *cP = p, *cQ = Q.p;
  int i = n;

  while( i-- )
  {
    if (*cP!=*cQ) return 0;
    cP++; cQ++;
  }

  return 1;
}

VectorInt& VectorInt::operator=( const VectorInt& P )
{
    if (extention<P.n)
    {
        extention=P.n;
        free(p); alloc();
    }
    np=P.n;
    memcpy(p,P.p,n*sizeof(int));
    return *this;
}

VectorInt::VectorInt( const VectorInt& P ): np(P.n), n(np), extention(P.n)
{
    alloc();
    memcpy(p,P.p,n*sizeof(int));
}

//ostream& VectorInt::PrintToStream( ostream& out ) const
void VectorInt::print()
{
    printf("[");
    if (!n || !p) { printf("]"); return; }

    int N=n,*up=p;
    while (--N) printf("%i,",*(up++));
    printf("%i]",*up);
}
```

9.6 MultiIndex manipulation

A multiindex is a line inside the α matrix (see equation 2.27).

9.6.1 Header file

```
//
```

```
//  class Multiindex
//
#ifndef _MPI_MULTIND_H_
#define _MPI_MULTIND_H_

#include <iostream.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 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_ */
```

9.6.2 Code file

```
//
//  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.setN(n-1);
    indexesOfCoefInLexOrderV.setN(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");
        exit(252);
    }
    return (unsigned*)lastChangesV.p;
}

unsigned *MultInd::indexesOfCoefInLexOrder()
{
    if (deg==0)
    {
        printf("use MultIndCache to instanciate MultInd");
        exit(252);
    }
    return (unsigned*)indexesOfCoefInLexOrderV.p;
}

MultIndCache::MultIndCache(): head(NULL)
{
    MultInd::maxDim=100;
    MultInd::buffer=(unsigned*)malloc(200*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");
        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;
}
```

9.7 Simple polynomial manipulation

9.7.1 Header file

```
//
//  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 vecor 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:
    virtual ~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 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

    int operator==( Polynomial );

    // 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; }
};

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_ */
```

9.7.2 Code file

```
//
//  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;

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"); 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");
        exit(254);
    }

    d->deg = max(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)
    {
        cerr << "op/(Poly,double): Division by zero\n";
        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)
    {
        cerr << "Poly::op+ : Different dimension\n";
        exit(-1);
    }

    Polynomial r(d->dim,max(d->deg,q.d->deg));
    unsigned N1=sz(), N2=q.sz(), Ni=min(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)
    {
        cerr << "Poly::op- : Different dimension\n";
        exit(-1);
    }

    Polynomial r(d->dim,max(d->deg,q.d->deg));
    unsigned N1=sz(), N2=q.sz(), Ni=min(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 ...

    double *tq = (*this);
    while( Ni-- ) *(tq++) = -(*(tp++));
    return *this;
}


Polynomial Polynomial::operator+=( Polynomial p )
{
    if (d->dim != p.d->dim)
    {
        cerr << "Poly::op+= : Different dimension\n";
        exit(-1);
    }

    d->deg = max(d->deg,p.d->deg);  // New degree
    unsigned N1=sz(), N2=p.sz(), Ni=min(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)
    {
        cerr << "Poly::op-= : Different dimension\n";
        exit(-1);
    }

    d->deg = max(d->deg,p.d->deg);  // New degree
    unsigned N1=sz(), N2=p.sz(), Ni=min(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)
    {
        cerr << "Poly::op/= : Division by zero\n";
        exit(-1);
    }

    int i=sz();
    double *tp = (*this);

    while (i--) *(tp++) /=t;
    return *this;
}


int Polynomial::operator==( Polynomial q )
{
    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 (*tt != 0)
        {
            if (IsFirst)
            {
                if (queryFlag( NicePrint ))
                {
                    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;
}
*/

// Evaluation oprator
// According to Pena, Sauer, "On the multivariate Horner scheme",
//   SIAM J. Numer. Anal., to appear

double Polynomial::operator()( Vector Point )
{
  unsigned dim=d->dim, deg=d->deg;
  static double r,r0;
  static double rbuf[100];  // That should suffice
  static double *rbufp = rbuf;
  static unsigned lsize = 100;
  static double *rptr;
  int i,j;

  if (Point==Vector.emptyVector) return *d->coeff;

  if ( dim != (unsigned)Point.sz() )
  {
    cerr << "Polynomial::operator()( Vector& ) : Improper size\n";
    #ifdef VERBOSE
    cerr << "dim = " << dim << ", P = " << P << "\n";
    #endif
    exit(-1);
  }

  if ( !sz() )
  {
    if ( queryFlag( Warning ) )
    {
      cerr << "Polynomial::operator()( Vector& ) : evaluating void polynomial\n";
    }
    return 0;
  }

  if ( dim > lsize )    // Someone must be crazy !!!
  {
    if ( queryFlag( Warning ) )
    {
    cerr << "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 )
    {
      cerr << "Polynomial::operator()( Vector& ) : Cannot allocate "
       << "<rbufp>\n" << flush;
      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;
  static 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;
    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 *hp=*h,*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");
        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_ */
```

9.8 Lagrange Interpolaton polynomial manipulation

9.8.1 Header file

```
//
//  Multivariate Interpolating Polynomials
//  Application header
//  ...
//  V 0.0
```

```
#include "Poly.h"
#include "Vector.h"

#ifndef _MPI_INTPOLY_H_
#define _MPI_INTPOLY_H_


class InterPolynomial : public Polynomial
{
public:

    double M;
    unsigned nPtsUsed, nPtsTotal;

    // (*this) = sum_i newBasis[i]*NewtonCoefPoly[i]
    Polynomial *NewtonBasis;
    // double *NewtonCoefPoly;

    // data:
    Vector *NewtonPoints;

    double *NewtonCoefficient(double *);
    void ComputeNewtonBasis(double *);

/*
    InterPolynomial() : Polynomial() {}
    InterPolynomial( const Polynomial& p ) : Polynomial( p ) {};
    InterPolynomial( const InterPolynomial& p ) : Polynomial( p ) {};
*/

    InterPolynomial( unsigned _deg, unsigned nPtsTotal, Vector *_Pp, double *_Yp );
    ~InterPolynomial();

    unsigned findAGoodPointToReplace(int excludeFromT,double rho,
                                                          Vector pointToAdd);
    void replace(unsigned t, Vector pointToAdd, double valueF);
    void updateM(Vector newPoint, double valueF);
    int checkIfValidityIsInBound(Vector dd, unsigned k, double bound, double rho);

    void translate(int k);

    void test();
    void check(Vector Base, double (*f)(  Vector ) );
};

Vector *GenerateData(double **valuesF, double rho,
                     Vector Base, double vBase, double (*f)( Vector ) );


#endif  /* _MPI_INTPOLY_H_ */
```

9.8.2 Code file

```
//
//    Multivariate Interpolating Polynomials
//    Private header
//    ...
//    V 0.3

#include <stdio.h>
#include "Poly.h"
#include "Vector.h"
#include "tools.h"
#include "Solver.h"

#ifndef _MPI_INTPOLYP_H_
#define _MPI_INTPOLYP_H_

#include "IntPoly.h"

// 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");
      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;
   static 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)
{
   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");
      exit(251);
   }

   MultInd I(dim);
   for (i=0; i<nPtsUsed; i++)
   {
      *(qq++)=Polynomial(I);
      I++;
   }

   unsigned k, kmax;
   double v, vmax, vabs;

   for (i=0; i<nPtsUsed; i++)
   {
      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 (abs(v) > vabs)
          {
              vmax = v;
              vabs = abs(v);
              kmax = k;
          }
          if ( abs(v) > good ) break;
      }

      // Now, check ...
      if (abs(vmax) < eps)
      {
          printf("Cannot construct newton basis");
          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
   }
}

InterPolynomial::InterPolynomial( unsigned _deg, unsigned _nPtsTotal, Vector *_Pp,
    double *_Yp ) : Polynomial(_Pp->sz(), _deg),
       nPtsTotal(_nPtsTotal), NewtonPoints(_Pp),
```

```
        M(0.0)
{
    nPtsUsed=choose( _deg+d->dim,d->dim );
    if (!_Pp)
    {
        cerr << "InterPolynomial::InterPolynomial( double *) : No Vectors\n";
        exit(-1);
    }
    if (!_Yp)
    {
        cerr << "InterPolynomial::InterPolynomial( double *) : No data\n";
        exit(-1);
    }
    if (_nPtsTotal<nPtsUsed)
    {
        cerr << "InterPolynomial::InterPolynomial( double *) : Not enough data\n";
        exit(-1);
    }

    // Generate basis
    ComputeNewtonBasis(_Yp);
//    test();

    // Compute Interpolant
//    double *NewtonCoefPoly=NewtonCoefficient(_Yp);
    double *NewtonCoefPoly=_Yp;

    double *coc= NewtonCoefPoly+nPtsUsed-1;
    Polynomial *ppc= NewtonBasis+nPtsUsed-1;
    this->copyFrom(*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);
}

InterPolynomial::~InterPolynomial()
{
    delete[] NewtonBasis;
    delete[] NewtonPoints;
}

Vector *GenerateData(double **valuesF, double rho,
                     Vector Base, double vBase, double (*f)( Vector ) )
// 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)), vCur;

    for (j=0; j<dim;j++)
    {
        cur=Base.clone();
        cur[j]+=rho;

        *(cp++)=cur;
        vCur=*(vf++)=f(cur);

        cur=Base.clone();
        if (vCur<vBase) { cur[j]+=2*rho; sigma[j]=rho; }
        else { cur[j]-=rho; sigma[j]=-rho; }

        *(cp++)=cur;
        vCur=*(vf++)=f(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;
            vCur=*(vf++)=f(cur);
        }
    }

    free(sigma);
    return ap;
}

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=max(M, abs((*this)(newPoint)-valueF)/sum);
}

unsigned InterPolynomial::findAGoodPointToReplace(int excludeFromT,
                                double rho, Vector pointToAdd)
{
    //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, i, N=nPtsUsed;
    double a, maxa=0;
    Polynomial *pp=NewtonBasis;

    for (i=0; i<N; i++)
    {
        if (i==excludeFromT) continue;
        a=XkHat.euclidianDistance(xx[i])/rho;
        // because of the next line, rho is important:
        a=::max(a*a*a,1);
        a*=abs(pp[i] (pointToAdd));
        if (a>maxa)
        {
            t=i; maxa=a;
        }
    }
    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(unsigned t, Vector pointToAdd, double valueF)
{
    //not tested
    updateM(pointToAdd, valueF);

    Vector *xx=NewtonPoints;
    Polynomial *pp=NewtonBasis, t1;
    unsigned i, N=nPtsUsed;

    t1=pp[t]/=(pp[t]( pointToAdd ));

    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.

    (*this)+=(valueF-(*this)(pointToAdd))*pp[t];

//    test();
}

int InterPolynomial::checkIfValidityIsInBound(Vector ddv, unsigned k, double bound, double rho)
// input: k,bound,rho
// output: j,ddv
{
    //not tested

    // 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=LAGMAXModified(GXk,H,rho,vmax);
            // to debug:
            //  this line was difficult to find:
            vd+=xk;
            vmax=pp[j](vd);
            if (tmp*vmax>=bound) break;
        }
    }
    if (j>=0) ddv.copyFrom(vd);
    return j;
}

void InterPolynomial::translate(int k)
{
    Vector translation=NewtonPoints[k];
    Polynomial::translate(translation);
    int i=nPtsUsed;
    while (i--) NewtonBasis[i].translate(translation);
    i=nPtsUsed;
    while (i--) if (i!=k) NewtonPoints[i]-=translation;
    NewtonPoints[k].zero();
}

#endif     /* _MPI_INTPOLYP_H_ */
```

9.9 Various simple tools

9.9.1 Header file

```
//
//  Header file for tools
//

#include <iostream.h>
```

```
#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 3.14159265358979323846264338327795


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 min( const double t1, const double t2 )
{
    return t1 < t2 ? t1 : t2;
}

inline unsigned min( const unsigned t1, const unsigned t2 )
{
    return t1 < t2 ? t1 : t2;
}

inline int min( const int t1, const int t2 )
{
    return t1 < t2 ? t1 : t2;
}

inline double max( const double t1, const double t2 )
{
    return t1 > t2 ? t1 : t2;
}

inline unsigned max( const unsigned t1, const unsigned 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();
double euclidianNorm(int i, double *xp);

#endif  /* _MPI_TOOLS_H_ */
```

9.9.2 Code file

```
//
//  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 )
    {
        cerr << "choose( unsigned, unsigned) :  overflow\n";
        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<min(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()
{
//    timeval tv;
//    gettimeofday(&tv,NULL);
//    srand(tv.tv_usec);
     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);
   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);
}
```

9.10 Sort tool

The goal of this tool is to take inside a giant matrix, the best lines, according to a given criteria. This tool is used at the beginning of the optimization algorithm to search for the best points in the database of known points.

9.10.1 Header file

```
#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);
private:
    void init();
    cell *ctable,*end,*_local_getOptValueC;
    int n,optionalN,_local_getOptValueI;
};

#endif
```

9.10.2 Code file

```
#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;
}
```

9.11 The optimizers.

9.11.1 Header file

```
#ifndef _MPI_SOLVER_H
#define _MPI_SOLVER_H

#include "Poly.h"
#include "Vector.h"

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, int maxIter, double *lambda1);

Vector LAGMAXModified(Polynomial q, double rho,double &VMAX);
Vector LAGMAXModified(Polynomial q, Vector pointXk, double rho,double &VMAX);
Vector LAGMAXModified(Vector G, Matrix H, double rho,double &VMAX);

Vector QPOptim(double *bestValueOF,
            double rhoStart, double rhoEnd, int niter,
            double (*f)(  Vector ), Matrix data,
            double noiseAbsolute=0.0, double noiseRelative=0.0);

Vector QPOptim(double *bestValueOF,
            double rhoStart, double rhoEnd, int niter,
            double (*f)(  Vector ), Vector start, double vOF, char *name=NULL,
            double noiseAbsolute=0.0, double noiseRelative=0.0);

Vector QPOptim(double *bestValueOF,
            double rhoStart, double rhoEnd, int niter,
            double (*f)(  Vector ), Vector start, char *name=NULL,
            double noiseAbsolute=0.0, double noiseRelative=0.0);

#endif
```

9.11.2 Code file - Chapter 3

```
#include <stdio.h>
#include <memory.h>

#include "Solver.h"
#include "Matrix.h"
#include "tools.h"
#include "KeepBests.h"
#include "IntPoly.h"

double findAlpha(Vector s,Vector u, double delta, Polynomial &q, Vector pointXk,Vector &output)
// find root (apha*) of equation L2norm(s+alpha u)=delta
// which makes q(s)=<g,s>+<s,Hs> smallest
// output is (s+alpha* u)
{
    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;

    Vector v1=u.clone();
    v1.multiply(tmp1+tmp2);
    v1+=s;
    double q1=q(v1+pointXk);

    Vector v2=u.clone();
    v2.multiply(tmp1-tmp2);
    v2+=s;
    double q2=q(v2+pointXk);

    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=min(a,h[i][i]);
    l=max(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=max(a,sum);
    }
    a=min(a,H.frobeniusNorm());
    a=min(a,H.infinitumNorm());

    l=max(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=max(a,sum);
    }
    a=min(a,H.frobeniusNorm());
    a=min(a,H.infinitumNorm());

    l=max(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=max(a,sum);
    }
    a=min(a,H.frobeniusNorm());
    return min(a,H.infinitumNorm());
}
```

```
Vector L2NormMinimizer(Polynomial q, double delta,
                                       int *infoOut, int maxIter, double *lambda1)
{
    return L2NormMinimizer(q, Vector.emptyVector, delta, infoOut, maxIter, lambda1);
}

// #define POWEL_TERMINATION 1

Vector L2NormMinimizer(Polynomial q, Vector pointXk, double delta,
                                       int *infoOut, int maxIter, double *lambda1)
{
// 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,
           sHs, uHu, alpha, normS;
    int info=0, n=q.dim();
    Matrix HLambda(n,n),H(n,n);
    MatrixTriangle L(n);
    Vector s(n), omega(n), u(n), sFinal, minusG(n);
    bool gIsNull, choleskyFactorAlreadyComputed=false;

    q.gradientHessian(pointXk,minusG,H);

//    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= max(lambdaU,(1+kappaEasy)*lambdaL);

    lambda=max(lambda, lambdaL);
    lambda=min(lambda, lambdaU);

    while (maxIter--)
    {
        if (!choleskyFactorAlreadyComputed)
        {
            if (!H.cholesky(L, lambda, &lambdaCorrection))
            {
                lambdaL=max(lambdaL,lambda+lambdaCorrection);
                lambda=max(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);
        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=min(lambdaU,lambda);
        } else lambdaL=max(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=max(lambdaPlus, lambdaL);
        lambdaPlus=min(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=max(lambdaL,lambda-uHu);

            alpha=findAlpha(s,u,delta,q,pointXk,sFinal);
            // check for termination
#ifndef POWEL_TERMINATION
            if (sqr(alpha)*uHu<
                kappaHard*s.scalarProduct(HLambda.multiply(s)))
#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=max(lambdaL,lambdaPlus);
        // check lambdaL for interior convergence
//      if (lambdaL==0) return s;
        lambda=max(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;
}
```

9.11.3 Code file - Chapter 4

```
Vector LAGMAXModified(Polynomial q, double rho,double &VMAX)
{
    return LAGMAXModified(q,Vector.emptyVector,rho,VMAX);
};

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(Vector G, Matrix H, double rho,double &VMAX)
{
    //not tested

//    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|| .LEQ. 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 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.
//     The vector that has the name D at label 60 used to be the vector W.
//
    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*::max(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;
};
```

9.11.4 Code file - Chapter 5

```
int findBest(Matrix data)
{
    // find the best point in the datas.
    unsigned i=data.nLine(), k, nc=data.nColumn();
    if (i==0) return -1;
    double *p=*((double**)(data))+nc-1, best=INF;
    while (i--)
    {
        if (*p<best)
        {
            k=i;
            best=*p;
        }
        p+=nc;
    }
    return k;
}

void saveValue(Vector tmp,double valueOF, Matrix data)
{
    int nl=data.nLine(), dim=data.nColumn()-1;
    data.extendLine();
    data.setLine(nl,tmp,dim);
    ((double**)data)[nl][dim]=valueOF;
```

```
}

Vector *getFirstPoints(double **ValuesFF, int *np, double rho, Matrix data,
                       double (*f)(  Vector ))
{
    int k=findBest(data);
    if (k==-1)
    {
        printf("Matrix Data must at least contains one line.\n"); 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=::min(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, f);
            for (i=0; i<n-1; i++) saveValue(points[i],valuesF[i], data);
            delete[] points;
            free(valuesF);
        }
    }
    *np=j;
    *ValuesFF=valuesF;
    return points;
}

int findK(double *ValuesF, int n)
{
    // 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) { k=i; minimumValueF=ValuesF[i]; }
    return k;
}

int QP_NF;
Vector QPOptim(double *bestValueOF,
            double rhoStart, double rhoEnd, int niter,
            double (*f)(  Vector ), Matrix data,
            double noiseAbsolute, double noiseRelative)
{

    int dim=data.nColumn()-1, n=(dim+1)*(dim+2)/2, info, k, t, nPtsTotal, nUpdateOfM=0;
    double rho=rhoStart, delta=rhoStart, rhoNew,
           lambda1, normD, modelStep, reduction, r, valueOF, valueFk, bound, noise;
    double *ValuesF;
    Vector *points=getFirstPoints(&ValuesF, &nPtsTotal, rhoStart, data, f);
    Vector Base, d, tmp;
    bool improvement, forceTRStep=true, evalNeeded;

    QP_NF=n;
    fprintf(stderr,"init part 1 finished.\n");

    // find index k of the best (lowest) value of the function
    k=findK(ValuesF, nPtsTotal);
    Base=points[k].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]);
    nUpdateOfM+=nPtsTotal-n;

    free(ValuesF);

    fprintf(stderr,"init part 2 finished.\n");
    fprintf(stderr,"init finished.\n");

    while (true)
    {
//       fprintf(stderr,"rho=%e; fo=%e; NF=%i\n", rho,valueFk,QP_NF);
        while (true)
        {
            // trust region step
            while (true)
            {
//                poly.print();
                d=L2NormMinimizer(poly, poly.NewtonPoints[k], delta, &info, 1000, &lambda1);
                normD=::min(d.euclidianNorm(), delta);
                d+=poly.NewtonPoints[k];

                reduction=valueFk-poly(d);

                //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*max(noiseAbsolute*(1+noiseRelative), abs(valueFk)*noiseRelative);
                if ((reduction<noise)&&(!forceTRStep)) { evalNeeded=true; break; }
                forceTRStep=false; evalNeeded=false;

                tmp=Base+d; valueOF=(*f)(tmp); saveValue(tmp,valueOF, data); QP_NF++;

                // update of delta:
                r=(valueFk-valueOF)/reduction;
                if (r<=0.1) delta=0.5*normD;
                else if (r<0.7) delta=max(0.5*delta, normD);
                     else delta=max(delta, max(1.25*normD, rho+ normD));
            // powell's heuristics:
                if (delta<1.5*rho) delta=rho;

                if (valueOF<valueFk)
                {
                    t=poly.findAGoodPointToReplace(-1, rho, d);
                    k=t; valueFk=valueOF;
                    improvement=true;
//                    fprintf(stderr,"Value Objective=%e\n", valueOF);
                } else
                {
                    t=poly.findAGoodPointToReplace(k, rho, d);
                    improvement=false;
                };

                modelStep=poly.NewtonPoints[t].euclidianDistance(d);
                poly.replace(t, d, valueOF); nUpdateOfM++;
                //points[t]=Base.clone();

                if (improvement) continue;
                if (modelStep>2*rho) continue;
                if (normD>2*rho) continue;
                break;
            }
            // model improvement step
            forceTRStep=true;

//            fprintf(stderr,"improvement step\n");
            if (normD<0.5*rho) bound=0.5*sqr(rho)*lambda1;
            else bound=0;
            if (nUpdateOfM<10) bound=0.0;

            // !! change d (if needed):
            t=poly.checkIfValidityIsInBound(d, k, bound, rho );
            if (t>=0)
            {
                tmp=Base+d; valueOF=(*f)(tmp); saveValue(tmp,valueOF, data); QP_NF++;

                poly.replace(t, d, valueOF); nUpdateOfM++;
                if (valueOF<valueFk) { k=t; valueFk=valueOF; };
                continue;
            }

            // the model is perfect for this value of rho:
            if (normD<=rho) 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,QP_NF);

        if (rho<16*rhoEnd) rhoNew=rhoEnd;
        else if (rho<250*rhoEnd) rhoNew=sqrt(rho*rhoEnd);
             else rhoNew=0.1*rho;
        delta=max(0.5*rho,rhoNew);
        rho=rhoNew;


        // update of the polynomial: translation of x[k].
        // replace BASE by BASE+x[k]
        Base+=poly.NewtonPoints[k];
        poly.translate(k);
    }

    if (evalNeeded)
    {
        tmp=Base+d; valueOF=(*f)(tmp); saveValue(tmp,valueOF, data); QP_NF++;
        if (valueOF<valueFk) { valueFk=valueOF; Base=tmp; }
        else Base+=poly.NewtonPoints[k];
    } else Base+=poly.NewtonPoints[k];

//    delete[] points; :not necessary: done in destructor of poly which is called automatically:
    fprintf(stderr,"rho=%e; fo=%e; NF=%i\n", rho,valueFk,QP_NF);

    *bestValueOF=valueFk;
    return Base;
}


Vector QPOptim(double *bestValueOF,
            double rhoStart, double rhoEnd, int niter,
            double (*f)(  Vector ), Vector start, double vOF, char *name,
            double noiseAbsolute, double noiseRelative)
{
    int dim=start.sz();
    Matrix data(1,dim+1);
    data.setLine(0,start,dim);
    ((double**)data)[0][dim]=vOF;
    Vector r=QPOptim(bestValueOF,rhoStart, rhoEnd, niter, f, data, noiseAbsolute, noiseRelative);
    if (name) data.save(name,0); // binary save
    return r;
}

Vector QPOptim(double *bestValueOF,
            double rhoStart, double rhoEnd, int niter,
            double (*f)(  Vector ), Vector start, char *name,
            double noiseAbsolute, double noiseRelative)
{
    return QPOptim(bestValueOF,rhoStart, rhoEnd, niter, f, start,
           f(start), name, noiseAbsolute, noiseRelative);
}
```