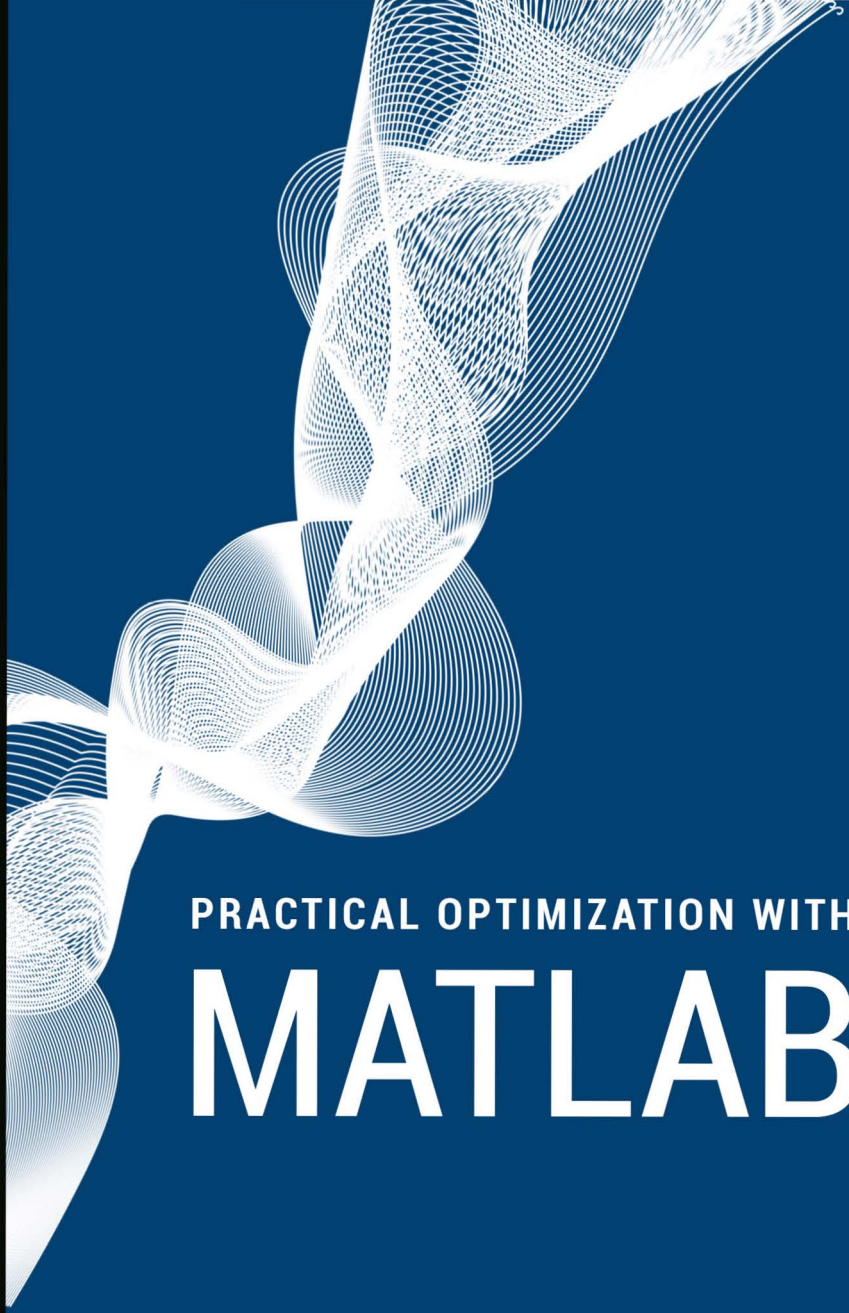

PRACTICAL OPTIMIZATION WITH MATLAB

Mircea Ancău

Practical Optimization with MATLAB

Practical Optimization with MATLAB

By
Mircea Ancău

Cambridge Scholars
Publishing

Practical Optimization with MATLAB

By Mircea Ancău

This book first published 2019

Cambridge Scholars Publishing

Lady Stephenson Library, Newcastle upon Tyne, NE6 2PA, UK

British Library Cataloguing in Publication Data
A catalogue record for this book is available from the British Library

ISBN (10): 1-5275-3849-4
ISBN (13): 978-1-5275-3849-8

PREFACE

This book is a brief introduction to the theory and practice of the numerical optimization techniques. It addresses all those interested in determining extreme values of functions. Because the book does not delve into mathematical demonstrations, it is accessible even to users without a theoretical background of optimization theory or detailed knowledge of computer programming. The concepts are presented in a simple way, starting with optimization methods for single variable functions without constraints and finishing with optimization methods for multivariable functions with constraints. Each of the methods presented is accompanied by its source code written in Matlab. The programs in the book are written as simply as possible. To make things even easier to follow, the first chapter of the book makes a brief overview of all the commands and programming instructions used in the source codes. Explanations accompanying each of the source codes within the book allow for the adaptation of programs to users' needs either by changing functions and restrictions expressions, or by including these programs only as simple functions in other, larger applications.

There is no method able to solve any type of optimization problem. Matlab possesses the *Optimization toolbox*, capable of solving a multitude of problems. Before grasping Matlab functions, you need to have enough knowledge to allow you to choose the right optimization methods for your problems. This book can help you take this first step. In addition to other similar works, this book also initiates in multicriteria optimization and combinatorial optimization problems. Because several of the source codes are derived from other source codes explained in the previous chapters, it is advisable for the scholarly reader to study each chapter of the book. However, it is also possible to use the source codes without reading the previous chapters, by providing the appropriate functions in the folder containing the program source code.

The book is structured in ten chapters. The first chapter addresses beginner programmers and reviews the basic Matlab programming knowledge. Fundamental concepts of reading and saving data in a specific format are explained. At the same time, with simple examples, the main programming syntax is explained, which is, anyway, similar to that encountered in other programming languages. Also presented are ways to

CONTENTS

plot the functions, their specific implementation detailed in the following chapters.

The second chapter is called *Basic concepts* and its role is to familiarize the reader with some optimization concepts such as objective function, decision variables, explicit or implicit constraints, optimization problems without or with constraints. The conditions of existence and uniqueness of the optimization solution, in the absence or presence of restrictions, are explained. The general mathematical model of an optimization problem is defined. Despite its lack of importance at first glance, the role of this chapter is to familiarize the reader with certain rules, starting with the way in which an optimization problem is formulated, the conditions in which the problem can be solved, and how the solution can be presented graphically.

In the third chapter, entitled *Optimization techniques for one variable unconstrained functions*, some optimization methods for functions that are dependent on a single variable without constraints are presented. Although the mathematical model of the optimization problem to be solved does not imply any constraint, the search must be limited to a specific domain. Most often, the interval containing the solution of the problem is not known. This is why, for the beginning, a simple method to find the limits of the interval that includes the solution to the problem is presented. The chapter continues with the grid optimization methods, the golden section method, the Fibonacci method, the quadratic approximation and the cubic approximation methods. At the end of the chapter, a method to solve the optimization problem that contains constraints is presented, based on an unconstrained optimization method, only by appropriately modifying the objective function. Each method explains the applied technique for step-by-step reduction of the length of the domain containing the optimal solution to a length less than or equal to an initially imposed precision factor. Each method is tested on the same optimization problem, so that the reader can compare the performance of each method. At first sight, solving an optimization problem whose objective function depends on a single variable, without constraints, may seem to have less practical applicability. However, the importance of these methods is significant. As will be seen in the following chapters, these optimization methods are the foundation of the most sophisticated optimization methods.

The fourth chapter, titled *Optimization techniques for n variables unconstrained functions*, takes a step further than the previous chapter. The optimization methods in this chapter are able to solve any optimization problems without restrictions, but optimization functions depend this time on n decision variables. From the multitude of

optimization methods of this type, the *random search method, the random path method, the relaxation method, the gradient method and the conjugate gradient method* are presented. All the optimization methods presented are iterative. This means that the search technique is applied in a repetitive or recursive way until some ending conditions are met. Criteria that stop the search for the solution are called convergence criteria. The end of the chapter illustrates some of these convergence criteria, which set the end time of the solution search procedure, when the initially set precision is reached.

The fifth chapter, titled *Optimization techniques for n variables constrained functions*, introduces the reader to some techniques of solving a more general optimization problem. These are the optimization problems with functions dependent on n variables, with constraints. There are presented *the random search method with constraints, the exterior penalty function method* and *the interior penalty function method*. The *random search method* with constraints is designed on the structure of *the random search method without constraints*, seen in the fourth chapter. The search method is very simple and once its principle understood, the transition to solving the optimization problem with constraints becomes a simple exercise. At the same time, even if the program is designed for functions dependent on just two variables, changing it to more than two variables is done by simply adding some source code lines. The following two search techniques, *the exterior penalty function method* and *the interior penalty function method*, are first-order methods that make use of the information given by the the first order derivative of the objective function. It is explained how to define the pseudo-objective function and how the expression of this new objective function influences the search process. The results of the search process are illustrated both numerically and graphically. Just like in case of *the random search method with constraints*, the source program is designed for functions that depend on just two variables and a single restriction. However, the transition from two to several decision variables, from one restriction to several, is very simple, via replication of certain source code lines.

Often, the methods presented in these three chapters are capable of solving convex optimization problems only. When this condition is no longer fulfilled, it may happen that the search procedure stops at a so-called local optimum point. For this reason, it is recommended to restart the search from different starting points and if the search method converges to the same point, it becomes likely that the solution thus determined represents the optimal point sought. However, there are optimization methods capable of managing such situations, in which there

are many local optimum points, but only one is absolute or overall optimum. One of these methods is *the global optimization algorithm* from the sixth chapter, designed to find an absolute optimum, hidden among many optimal local points. This algorithm searches for the optimum solution in two steps. The first step is based on a general Monte Carlo search process. The second step is a local search process, in the neighborhood of the general solution determined at the first step. The simple way of presenting the source code of the program makes it easy to modify it from two variables without constraints, to several variables. In addition, the experience gained with the study of optimization methods with constraints, in the fifth chapter, makes the transition to solving constrained optimization problems simpler.

The seventh chapter titled *Multicriteria optimization*, aims to solve optimization problems with multiple objective functions. The mathematical bases of multicriterial optimization are briefly presented. The significance of the *Pareto-optimal* set is also explained, as well as the way of determining the points belonging to it. A source program for determining the *Pareto-optimal* set is provided. The chapter illustrates the method of global multi-criteria optimization criterion along with its relative norm variant.

The eighth chapter, called *Traveling Salesman Problem*, belongs to the combinatorial optimization field. Traveling salesman is a problem with many practical applications in very different fields, from industrial engineering, applied physics, astronomy, medicine etc. For the beginning, some simple methods both in terms of working principle and difficulty of computer programming are presented. Next, a heuristic insertion algorithm is also explained. For all the methods described in the chapter, the source code is given. The program of the insertion heuristic algorithm is designed so that the user can add additional functions to the existing one, so that one can increase the degree of diversification of the solution search, or increase the performance of the local search.

The ninth chapter is entitled *Optimal nesting* and addresses, as well, a problem from the field of combinatorial optimization, which is the problem of optimal cutting of materials. Different solutions are suggested to address this problem of material cutting, for different possible situations.

The tenth chapter is titled *Flowshop scheduling problem* and is an illustration of the problem concerning the optimal launching of products in manufacturing. It is shown how to calculate the total inactivity time, according to the productive time values, in accordance with the methodology developed by Johnson (1954), in its well-known paper.

Johnson's algorithm is presented for the optimal flowshop of n jobs on two machines. Two heuristic methods to solve the problem are designed, one heuristic constructive and one improvement method respectively.

This book is partly based on the *Numerical Methods* and *Numerical Optimization Techniques* lectures, taught by the author at the Technical University of Cluj-Napoca. For this reason, the book can also be used as a course or laboratory teaching material to illustrate different concepts or methods. Thus, the book becomes a valuable tool not only for researchers, but also for students or professors from technical faculties.

Mircea Ancău
Cluj-Napoca, 29th of June 2019

1.

BRIEF INTRODUCTION TO MATLAB PROGRAMMING

1.1 Introduction

The present chapter is addressed to those who possess a symbolic knowledge of programming, not necessarily in Matlab. For this reason, this chapter is not a detailed Matlab tutorial in the true sense of the word. The chapter is rather a guide to understanding the programming functions used within the source codes appearing in the book. Only the programming notions used within the programs are exposed, focusing on input/output functions, operation with matrices, and programming functions, graphical representation functions etc.

1.2 Format to display

In Matlab there are a multitude of formats for displaying numerical data, from *short*, to *long*, *short e*, *long e*, *short g*, *long g* and so on. For example, the *short* format has 4 significant digits while the *long* format contains 15 significant digits. The entire list of these formats can be easily obtained with the command:

```
help format
```

1.3 Scalar variables

A scalar variable (a numerical value) must have a name that will necessarily begin with a letter, followed by any other combination of letters or numbers, or letters and numbers. To avoid overlapping a name with a pre-defined function name, check the '*proposed_name*' with the command:

```
exist('proposed_name')
```

This command returns zero if the proposed name does not exist among the predefined Matlab function names or the reserved words etc. If the proposed name exists within the above mentioned list, the command returns a non-zero value. Because Matlab is case sensitive, a variable name *Sin* will not conflict with the name of the trigonometric function *sin*. Matlab operates with a set of reserved words or keywords. To prevent these keywords from being used as variable names, they are displayed in blue. The full list of keywords can be displayed with the command:

```
iskeyword
```

1.4 Matrices and operations

A matrix is a collection of numbers arranged into a fixed number of m rows and n columns. A particular case of the matrix is a vector. A vector is an array having either a single line (line vector) or a single column (column vector). From this point of view, even a scalar variable can be interpreted as a matrix with a single line and a single column.

A matrix can be defined by including its elements in square brackets. Within square brackets, the elements of each line are separated by spaces, while the lines are separated by semicolons. The elements of a matrix can be accessed by indicating, within brackets, the line and column number at which the referenced item is located. If we define for example the matrix t:

```
t = [1 2 3 4;5 6 7 8;9 10 11 12];
```

the command $t(3,1)$ will refer to the element located in the third line, first column, and returns $t(3,1) = 9$. To find the size of a matrix it is enough to type:

```
[m,n] = size(t);
```

This command will return values $m = 3$ and $n = 4$. By the command:

```
help elmat
```

you can find the list of all types of elementary matrices and modes of operation with them. Within the source codes presented in the following chapters, often is used the command:

```
name = zeros(m,n);
```

in order to reserve memory space to a matrix called *name*, with ***m*** lines and *n* columns. All elements of this matrix are initialized with zero.
Concerning the division between two matrices, besides the normal division (with symbol - */*) or right division, Matlab has the division to the left (with symbol - \) or left division. Used usually to solve the matrix equations of the form $Ax = B$, which requires the calculation of the inverse of the matrix (i.e. A^{-1}), the left division solves this equation by the command $x = A\backslash B$, equivalent to $x = A^{-1}B$. It is recommended to use left division as an alternative of the inverse matrix calculation for reasons of stability in terms of numerical calculation and calculation speed, respectively. Details concerning these operations can be found by command:

```
help slash
```

The command:

```
help ops
```

will return the list of all operations, starting with the list of arithmetic operators (i.e. +, -, *, /, ^), relational operators (i.e. ==, ~=, <, >, <=, >=), list of logical operators (i.e. &&, ||, &, |, ~, ...), list of special characters (i.e. :, ( ), [ ], { }, @, ...), list of bitwise operators and set operators. For details about arithmetic operations involving matrices, execute the command:

```
help arith
```

For details on relational operations involving matrices, use the command:

```
help relop
```

Matlab contains a multitude of predefined mathematical functions that operate with matrices. The complete list of these functions can be found with the command:

```
help elfun
```

If one is interested in all primary help topics, one needs just type the command `help` in the *Command Window* of Matlab. To avoid a long list

scrolling on the screen, without having time to read it, even if you can scroll back, before the `help` command, just type `more on`. This command will lead to page by page scrolling. If someone is interested in something in particular, for instance about the role of a function, use the command `lookfor` followed by the topic (i.e. the function name) of your search.

To make the multiplication of two matrices A and B, the main condition is the equality between the number of colunms of the first matrix A with the number of lines of the second matrix B.
Let us consider two matrices A and B:

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{bmatrix}; \quad B = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \\ b_{31} & b_{32} \end{bmatrix}; \tag{1.1}$$

The result of the multiplication of these two matrices is also a matrix denoted C:

$$C = \begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix}; \tag{1.2}$$

whose elements are:

$$\begin{aligned} c_{11} &= a_{11} \cdot b_{11} + a_{12} \cdot b_{21} + a_{13} \cdot b_{31}; \\ c_{12} &= a_{11} \cdot b_{12} + a_{12} \cdot b_{22} + a_{13} \cdot b_{32}; \\ c_{21} &= a_{21} \cdot b_{11} + a_{22} \cdot b_{21} + a_{23} \cdot b_{31}; \\ c_{22} &= a_{21} \cdot b_{12} + a_{22} \cdot b_{22} + a_{23} \cdot b_{32}; \end{aligned} \tag{1.3}$$

When using a compiler of a programming language (i.e. C/C++, Pascal etc.), programming the multiplication of these two matrices requires several lines of source code (see Code 1.1).

```
1   %            matrixMultiplication.m
2   %
3   % this program calculates the product of two
4   % matrices: a(m,n), b(n,p).
5   % the matrix c must have the size: (m,p)
6   %
7   A = [2 4 6; 1 3 5];
8   B = [1 2; 3 4; 5 6];
9   %
10  % get the size of matrices a and b
11  [m,n] = size(A);
```

```
12  [~,p] = size(B);
13  %
14  C = zeros(m,p);
15  for i = 1:m
16      for j = 1:p
17          C(i,j) = 0;
18          for k = 1:n
19              C(i,j) = C(i,j)+A(i,k)*B(k,j);
20          end
21      end
22  end
23  %
```

Code 1.1 *matrixMultiplication.m*.

Running the program in Code 1.1 for the matrices A and B defined at lines 6 and 7, will provide:

$$A \cdot B = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{bmatrix} \cdot \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \\ b_{31} & b_{32} \end{bmatrix} = \begin{bmatrix} 44 & 56 \\ 35 & 44 \end{bmatrix}; \tag{1.4}$$

To get the same result, in Matlab it is sufficient to type:

```
C = A*B;
```

Let us suppose the matrices A and B are of the same size, and we want to perform their multiplication, element by element. It means that:

$$A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}; \quad B = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}; \tag{1.5}$$

$$C = \begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix}; \tag{1.6}$$

where

$$\begin{aligned} c_{11} &= a_{11} \cdot b_{11}; \\ c_{12} &= a_{12} \cdot b_{12}; \\ c_{21} &= a_{21} \cdot b_{21}; \\ c_{22} &= a_{22} \cdot b_{22}; \end{aligned} \tag{1.7}$$

This sort of multiplication is exemplified by the program in Code 1.2.

```
1    %       elementByElementMultiplication.m
2    %
3    % this program calculates the product of two
4    % matrices, element by element.
5    % the matrices a and b must have the same size
6    %
7    A = [2 4 6; 1 3 5];
8    B = [1 2 3; 4 5 6];
9    %
10   [m,n] = size(A);
11   % memory allocation for matrix c
12   C = zeros(m,n);
13   %
14   for i = 1:m
15       for j = 1:n
16           C(i,j) = A(i,j)*B(i,j);
17       end
18   end
19   %
```

Code 1.2 *elementByElementMultiplication.m*.

Running the program in Code 1.2 for the matrices A and B defined at lines 6 and 7, will get:

$$A \cdot B = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{bmatrix} \cdot \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \end{bmatrix} = \begin{bmatrix} 2 & 8 & 18 \\ 4 & 15 & 30 \end{bmatrix}; \qquad (1.8)$$

To get the same result, in Matlab it is enough to type:

```
C = A.*B;
```

A number of other operations are possible with the elements of the data vectors or the matrices. Usually, operations applicable to scalars can also be applied, element by element, to strings of data.

1.5 Input and output operations

For the opening of a new or already existing file, the following syntax can be used:

```
fileID = fopen(filename,permission);
```

This syntax allows one to create a new file or open an existing one with the name *filename*, under the conditions specified by *permission*. *Filename* has the format *name.extension*, name being the file name, and the extension specifying the type of data within the file. The data type in the file can be a string in ASCII or binary format. The file can be created or opened in both ways. We can write data in ASCII format in a text file using the following syntax:

```
fprintf(fileID,formatSpec,A1,A2,…,An);
```

formatSpec specifies the format in which the data will be written in the file. Thus, if we have to write the values of three variables x_1, x_2, x_3, of which x_1 is an integer, x_2 of real type, and x_3 a string, then we need to write:

```
formatSpec = 'x1=%d    x2 = %f    x3 = %s'
```

Thus, the entire instruction becomes:

```
fprintf(fileID,'x1 = %d x2 = %f x3 = %s',x1,x2,x3);
```

or we may explicitly specify the format descriptors as follows:

```
fprintf(fileID,'x1 = %3d x2 = %10.6f x3 = %6s',x1,x2,x3);
```

In this way, the value of x_1 will be written to the file with maximally 3 digits, the value of x_2 with no more than 6 significant digits after the comma and the string x_3 with no more than six characters. The file is closed with the command:

```
fclose(fileID);
```

Table 1.1 contains the list of possible types of access permissions, when opening a file.

Table 1.1 Type of permited access.

Permission	Type of access
'r'	Open file for reading.
'w'	Open or create new file for writing. Discard existing contents, if any.
'a'	Open or create new file for writing. Append data to the end of the file.
'r+'	Open file for reading and writing.
'w+'	Open or create new file for reading and writing. Discard existing contents, if any.
'a+'	Open or create new file for reading and writing. Append data to the end of the file.
'A'	Open file for appending without automatic flushing of the current output buffer.
'W'	Open file for writing without automatic flushing of the current output buffer.

1.6 Programming guidelines

Within a program, many times, a certain set of some instructions is repeated several times. Sometimes the number of these recurrences is known in advance, sometimes not. Such sets of instructions, or blocks of instructions, can be handled by control-flow statements. Sometimes it is necessary to execute an instruction only if a certain condition is fulfilled. These situations are managed by means of conditional statements.

1.6.1 The `for` statement

The `for` statement is used in instructions that must repeat for a known number of steps. At the beginning, there must be a specified start value, an increment and the final value of the variable that controls the recurring process. All the values of the counter can be chosen as positive or negative, integers or real numbers. The syntax of the `for` statement is:

```
for counter = startValue:increment:finalValue
   do something;
end
```

All statements included between the start line of the `for` loop and the `end` statement are executed recursivelly until the `counter` reaches the

`finalValue`. The program *simpleForLoop.m* in Code 1.3, is an example of using this cycle.

```
%           simpleForLoop.m
%
% for-loop whose increment is a positive integer
a = zeros(1,10);
for i = 1:10
    a(1,i) = sqrt(i);
end
%
% for-loop whose increment is a negative integer
b = zeros(1,10);
j = 1;
for i = 10:-1:1
    b(1,j) = sqrt(i);
    j = j + 1;
end
%
% for-loop whose increment is a positive non-integer
c = zeros(1,10);
j = 1;
for i = 1:0.1:1.9
    c(1,j) = sqrt(i);
    j = j + 1;
end
%
% for-loop whose increment is a negative non-integer
d = zeros(1,10);
j = 1;
for i = 1.9:-0.1:1
    d(1,j) = sqrt(i);
    j = j + 1;
end
%
```

Code 1.3 *simpleForLoop.m*.

The first `for` loop in Code 1.3 calculates the square root of the first ten natural numbers starting from 1 to 10. Each of these ten values is stored in the line vector called *a*. Because inside this loop there is no specified increment, the Matlab compiler assumes it as one. On the second `for` loop, the increment is still an integer but negative. So, the same calculations as in the first loop apply, but in reverse order. The values of the square roots of the first ten integers are stored in a line vector called *b*.

The next two `for` loops make similar calculations, but, this time, the increment is a non-integer. It is important to note that starting with the second `for` loop, the index j denoting the line vector location differs from the counter i of the loop. This difference is required because the location index in a vector or a matrix is always an integer, while the loop increment may be an integer or not.

1.6.2 The `while` statement

With this type of statement a cycle can be defined, even if we do not know how many times the instructions in the cycle need to be repeated. The syntax of the while statement is:

```
while (condition holds TRUE)
    do statement 1;
    do statement 2
    ...
    do statement k;
end
```

This cycle is terminated by the `end` statement. All statements inside the cycle are repeated, until the condition introduced after the `while` statement is accomplished. Code 1.4 displays a very simple example of while statement use.

```
1   %          whileStatement.m
2   %
3   % This is an example of a cycle with an
4   % unknown number of steps
5   %
6   % open a text file to save results
7   fp = fopen('results.txt','w');
8   x = 0;
9   while x<=5
10      fprintf(fp,'x = %2d\n',x);
11      % increment x
12      x = x + 1;
13  end
14  % close the text file
15  fclose(fp);
16  %
```

Code 1.4 *whileStatement.m*.

This program will print into *results.txt* file the first six values of x, starting with zero. Always take much care in what way you formulate the condition that define the `while` statement. If in Code 1.4, at line 8, we assign, say $x = 10$ and at line 9, instead of $x <= 5$ we put $x >= 5$, the `while` block will loop forever.

1.6.3 The `if-elseif-else` statement

In a program, there is often the question of the execution of certain statements only when certain conditions are fulfiled. Typically, these situations can be solved using the `if-elseif-else` instructions. The syntax of these instructions is:

```
if condition 1 is TRUE
    statement 1;
elseif condition 2 is TRUE
    statement 2 (or group of statements);
elseif condition 3 is TRUE
    statement 3;
    ...
elseif condition k is TRUE
    statement k;
else
    do something else;
end
```

The program in Code 1.5 illustrates the way this instruction operates in the simple case of solving a second degree equation. The user must assign the numerical values of the coefficients of the equation $ax^2+bx+c = 0$, and the program will indicate the type of roots of that equation (i.e. distinct real roots, real equal roots or complex roots).

```
1   % define equation coefficients
2   a = 1;  b = 0;  c = 1;
3   % calculation of the equation discriminant
4   delta = b^2 - 4*a*c;
5   % decide the type of equation roots
6   if delta >0
7       message  = ('real and distinct roots');
8       disp(message);
9   elseif delta == 0
10      message = ('real and equal roots');
```

```
11        disp(message);
12    else
13        message = ('complex roots');
14        disp(message);
15    end
16    %
```

Code 1.5 *secDegrEq1.*

To test the condition inside `if-elseif-else` statements, any relational or logical operators can be used (see `help ops` in Matlab Command Window), and multiple conditions can be used in the same time.

1.6.4 The `break` statement

Sometimes it is necessary to execute a `for` loop only until a certain condition is fulfilled. To exit the cycle, an `if` statement should be inserted, indicating the moment of exit. The exit is made by the `break` statement. The syntax used in such cases is:

```
for counter = startValue:increment:finalValue
    statement 1;
    if condition is TRUE
        break;
    statement 2;
    statement 3;
    ...
    statement k;
end
```

Please note that when the Matlab compiler meets the `break` statement, all statements that follow the `break` statement are omitted, and control is transferred outside the `for` loop.

1.6.5 The `switch-case-otherwise` statemens

Another way to branch off a program is provided by the `switch-case-otherwise` statement, whose syntax is:

```
switch var
    case varValue 1
        list 1 of statements;
```

```
    case varValue 2
        list 2 of statements;
        ...
    case varValue k2
        list k of statements;
    otherwise
        another list of statements;
end
```

The variable `var` may be of any type, according to the algorithm. There may be any `case` branches and a single otherwise, in case any of the above `case` situations are not realized. Code 1.6 extends the case of solving the second-degree equation and indicates their values besides the type of solutions, using a `switch-case-otherwise` statement.

```
% define equation coefficients
a = 1;  b = 0;  c = 1;
% calculation of the equation discriminant
delta = b^2 - 4*a*c;
% decide the type of equation roots
if delta > 0
    message  = ('real and distinct roots');
    disp(message);
elseif delta == 0
    message = ('real and equal roots');
    disp(message);
else
    message = ('complex roots');
    disp(message);
end
%
% the calculation of roots
switch message
    case 'real and distinct roots'
        x1 = (-b - sqrt(delta))/(2*a);
        x2 = (-b + sqrt(delta))/(2*a);
        mess = sprintf('x1 = %f and x2 = %f ',x1,x2);
        disp(mess);
    case 'real and equal roots'
        x = -b/(2*a);
        mess = sprintf('x1 = x2 = %f ',x);
        disp(mess);
    otherwise
        realPart = -b/(2*a);
```

```
30          imagPart = sqrt(-delta)/(2*a);
31          mess = sprintf('x1=%f-i*%f\nx2=%f+i*%f',...
32  realPart,imagPart,realPart,imagPart);
33          disp(mess);
34  end
35  %
```

Code 1.6 *secDegrEq2.m*.

1.6.6 The `continue` statement

If the program finds a `continue` statement inside a loop of type `for` or `while`, it passes the control to the next iteration in which it appears, and skips any remaining statements that follow. Code 1.7 and Code 1.8 exemplifies the use of `continue` statements inside a `for` loop and a `while` loop, respectively.

```
1   %        continueStatement1.m
2   %
3   % 'continue' is inside a 'for' loop
4   % open the file ,results.txt'
5   fp = fopen('results.txt','w');
6   for i = 1:3
7       for j = 1:3
8           for k = 1:3
9               if i==2
10                  continue
11              end
12              fprintf(fp,'i=%2d    j=%2d    k=%2d\n',...
13                  i,j,k);
14          end
15      end
16  end
17  % close the file 'results.txt'
18  fclose(fp);
19  %
```

Code 1.7 *continueStatement1.m*.

```
1   %        continueStatement2.m
2   %
3   % 'continue' is inside a 'while' loop
4   % open the file ,results.txt'
5   fp = fopen('results.txt','w');
6   i = 0;
```

```
while i<=3
    j = 0;
    while j<3
        j = j + 1;
        if i==2
            continue
        end
        fprintf(fp,'i=%2d    j=%2d\n',i,j);
    end
    i = i + 1;
end
% close the file 'results.txt'
fclose(fp);
%
```

Code 1.8 *contimueStatement2.m.*

1.6.7 The `return` statement

Typically, at the end of a function, the compiler returns control to the program that called it, without the need to insert a `return` instruction. The `return` instruction is required, if a premature termination of the function and the return of control within the program is desired.

1.7 Scripts and functions

In each of the chapters that follow, two different types of programs, namely scripts and functions will appear. As a set of instructions that respects a specific syntax, the script is a Matlab program that does not accept input arguments and does not return output arguments. The script is designed by the user to operate with some data, so that it can generate new data. All data are located in the Matlab workspace. On the other hand, a function is also a set of instructions. The function is designed to accept some input arguments, or actual parameters, and return output parameters. All internal variables a function uses are considered as local, meaning these variables are not visible outside the function. As programs grow in complexity, they become increasingly difficult to understand and debug. For this reason, programs are divided into smaller pieces, each piece having its own well-defined role. Thus, the script keeps its role of program coordinator and allocates most of its attributes to functions. In this way, the programs become easy to track, understand and correct in case of error. Code 1.5 calculates the discriminant of the second degree algebraic

equation and determines the type of roots of this equation. Code 1.6 goes further and, based on the coefficients values of this equation, calculates the roots and displays them in the *Command Window*.

We will resume the problem of solving the second degree equation by delegating work to several functions. Thus, we will have a script program, called *solveEqByFuncCall.m* working as a coordinator. From this program, we call some functions with different attributes. First of all, we call the function *equationDiscriminant.m*. This function takes the equation coefficients as arguments and returns the value of the equation discriminant *delta*. The second function called by *solveEqByFuncCall.m* is *decideRootsType.m*. This function takes as argument the value of the equation discriminant *delta* and, based on this value, the function takes a decision concerning the type of equation roots. The third function called from *solveEqByFuncCall.m* is *rootsCalculation.m*. It can be seen that this time, the function does not return a result. It just takes as arguments *a, b, delta* and *message* and, based on this information, it calculates the equation roots and displays them in the *Command Window*.

```
%        solveEqByFuncCall.m
%
% define equation coefficients
a = 1;  b = 0;  c = 1;
% calculation of the equation discriminant
delta = equationDiscriminant(a,b,c);
% decide the type of equation roots
[message] = decideRootsType(delta);
% display the type of equation roots
% into the Command Window
disp(message);
% calculate the roots of the equation based on
% equation coefficients, and display them into
% the Command Window
rootsCalculation(a,b,delta,message);
%
```

Code 1.9 *solveEqByFuncCall.m*.

```
%       equationDiscriminant.m
function [delta] = equationDiscriminant(a,b,c)
%
delta = b^2 - 4*a*c;
%
```

Code 1.10 *equationDiscriminant.m*.

```
%    decideRootsType.m
function [message] = decideRootsType(delta)
```

```
%
if delta >0
    message  = ('real and distinct roots');
elseif delta == 0
    message = ('real and equal roots');
else
    message = ('complex roots');
end
%
```

Code 1.11 *decideRootsType.m*.

```
%       rootsCalculation.m
%
function rootsCalculation(a,b,delta,message)
switch message
    case 'real and distinct roots'
        x1 = (-b - sqrt(delta))/(2*a);
        x2 = (-b + sqrt(delta))/(2*a);
        mess = sprintf('x1 = %f and x2 = %f ',x1,x2);
        disp(mess);
    case 'real and equal roots'
        x = -b/(2*a);
        mess = sprintf('x1 = x2 = %f ',x);
        disp(mess);
    otherwise
        realPart = -b/(2*a);
        imagPart = sqrt(-delta)/(2*a);
        mess = sprintf('x1=%f-i*%f\nx2=%f+i*%f',...
            realPart,imagPart,realPart,imagPart);
        disp(mess);
end
%
```

Code 1.12 *rootsCalculation.m*.

The function *equationDiscriminant.m* in Code 1.10 is a very simple function and can be easily replaced by anonymous functions. These functions have the syntax:

```
f = @ (list of arguments) mathematical formula
```

So, instead of a new function file, it is enough to define the calculation of *delta* as an anonymous function:

```
delta @(a,b,c) b^2 - 4*a*c;
```

Now, the calculation of parameter *delta* can be made simply, by the command `delta(a,b,c)`. The anonymous function is not the only variant to use in such case. There is another very similar variant of `inline` function, whose syntax is:

```
f = inline('mathematical formula');
```

For the case of parameter *delta*, the `inline` function definition will be:

```
delta = inline('b^2 - 4*a*c');
```

and the call will be `delta(a,b,a)`, exactly as in the above case of calling the anonymous function.

1.8 Graphic representations

Although Matlab has a multitude of graphic representation possibilities, only a few were used in this book. Of these, one part is represented by 2D graphics and the other by 3D graphics. We may start by representing the variation of a parabolic function, whose equation is $f(x) = ax^2+bx+c$. In order to do this, we must describe the definition domain, or range, of variable x and then specify the values a, b and c of the function coefficients. In Code 1.13 the source code of the *graphic2D.m* program that draws the parabolic variation is listed.

```
1   %               graphic2D.m
2   %
3   % define the limits of the variable domain
4   leftBound = -3;
5   rightBound = 1;
6   % define the increment of the variable
7   step = 0.05;
8   %
9   % calculate the number of variable values
10  n = (rightBound-leftBound)/step+1;
11  %
12  % define variable 'x'
13  x = leftBound:step:rightBound;
14  %
15  % assign memory
16  y = zeros(1,n);
17  % set up the parabola coefficients
```

```
a = 1;
b = 2;
c = 1;
%
% define the function f(x) = a*x^2+b*x+c
f = @(x) a*x^2+b*x+c;
% calculate the values y = f(x)
for i = 1:n
    y(1,i) = f(x(1,i));
end
%
% plot the parabola graphic
plot(x,y), xlabel('x'), ylabel('y = a*x^2+b*x+c'),...
 title('parabola'), grid on, axis equal,...
 axis([-4 2 -1 5]);
% draw the horizontal axis
line([-4,2],[0,0], 'Color','k');
% draw the vertical axis
line([0,0],[-2,5],'Color','k');
%
```

Code 1.13 *graphic2D.m*.

If we execute the program *graphic2D.m* it will result in the graph from Fig. 1.1.

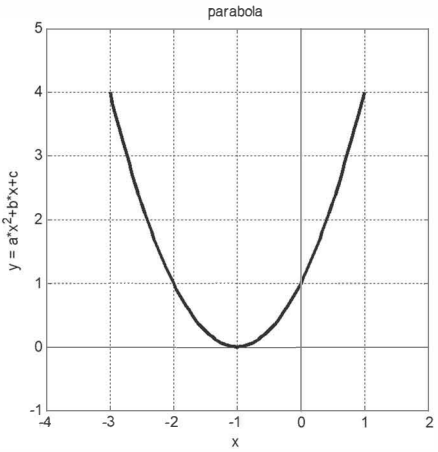

Fig. 1.1 The parabolic variation.

Lines 4 and 5 in Code 1.13 defines the range of variable *x*. We want to calculate the values of the parabola function in this range with a defined step. The smaller the step value is, the finer the trace of the variation curve. Lines 18 to 20 assign the values of the parabola function coefficients *a*, *b* and *c*. In line 23 the expression of the parabola $f(x) = ax^2+bx+c$ is defined, as an anonymous function. Lines 30, 31 and 32 draw the parabola graphic, by using the `plot` command. To a line one may add specifications regarding line type, the type of line marker used, its color and the line width. So, in line 30, instead of `plot(x,y)`, `plot(x,y, ':r','LineWidth',2)` can be written. It means that the parabola in Fig.1.1 will be drawn as dotted line and red color (see Fig. 1.2), using a line width of 2 (in points - their size depends on the screen resolution).

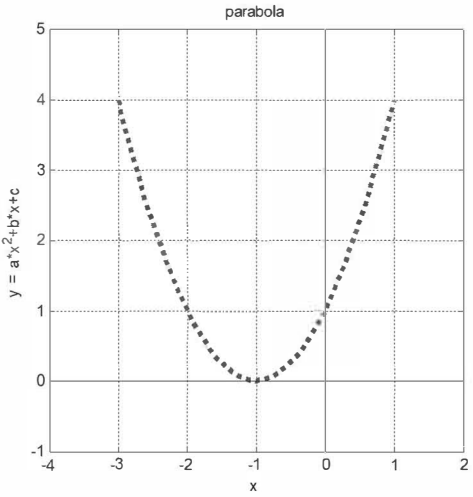

Fig. 1.2 Second instance of the parabolic variation.

In the `plot` command, after a comma the symbols `':r'` appear, which means that this time two specifiers were added only. However three specifiers can be added, and their meaning is `'Line Style Marker Color'`. Tables 1.1 shows line specifiers that can be used in the `plot` command, Table 1.2 shows types of markers and Table 1.3 shows basic color specifiers.

Table 1.2 Line style specifiers.

Line style	Specifier
'-'	Solid line (default)
'- -'	Dashed line
':'	Dotted line
'- .'	Dash-dot line

Table 1.3 Marker specifiers.

The marker	Specifier
'+'	Plus sign
'o'	Circle
'*'	Asterisk
'.'	Point
'x'	Cross
's'	Square
'd'	Diamond
'^'	Upward-pointing triangle
'v'	Downward-pointing triangle
'>'	Right-pointing triangle
'<'	Left-pointing triangle
'p'	Pentagram
'h'	Hexagram

Table 1.4 Color specifiers.

Color	Specifier
'r'	Red
'g'	Green
'b'	Blue
'c'	Cyan
'm'	Magenta
'y'	Yellow
'k'	Black
'w'	White

There are several other properties that can be specified, such as the color of marker edge and face, or the marker size.

Sometime more than one function needs to be drawn on the same graphic. This can be done in different ways. For this purpose, the Code 1.13 can be modified by defining two different functions (see Code 1.14).

```
%               twoFunctions2D.m
%
% define the limits of the variable domain
leftBound = -3;
rightBound = 1.5;
% define the increment of the variable
step = 0.05;
% calculate the number of variable values
n = (rightBound-leftBound)/step+1;
% define the variable 'x'
x = leftBound:step:rightBound;
% assign memory
y1 = zeros(1,n);
y2 = zeros(1,n);
% setup the function coefficients
a1 = 1;          a2 = -2;
b1 = 2;          b2 = 1;
c1 = 1;          c2 = 2;
% define the function f1(x) = a1*x^2+b1*x+c1
f1 = @(x) a1*x^2+b1*x+c1;
% define the function f2(x) = a2*x^2+b2*x+c2
f2 = @(x) a2*x^2+b2*x+c2;
% calculate the values y1 = f1(x); y2 = f2(x);
for i = 1:n
    y1(1,i) = f1(x(1,i));
    y2(1,i) = f2(x(1,i));
end
% plot the functions' graphs
h = plot(x,y1,'r',x,y2,'b');
set(h(1),'linewidth',2);
set(h(2),'linewidth',2);
legend('y1','y2');
title('parabolas');
xlabel('x');
ylabel('y1 = a1*x^2+b1*x+c1; y2 = a2*x^2+b2*x+c2');
axis equal;
axis([-5 5 -8 8]);
% draw the horizontal axis
line([-5,5],[0,0], 'Color','k');
% draw the vertical axis
line([0,0],[-8,8],'Color','k');
%
```

Code 1.14 *twoFunctions2D.m*.

In Code 1.14, two parabolas were defined and drawn within the same graph, using the `plot` command at line 29. All specifications for this graph are listed by lines 30 to 37. The graph generated by Code 1.14 is presented in Fig. 1.3. A similar graph might be obtained by drawing each of these two parabolas separately. To do this, the `hold on`, `hold off` commands need to be used (see Code 1.15).

```
1   %        tan(x)_cot(x).m
2   %
3   % setup the range of variable x
4   x = 0.1 : 0.1: 5;
5   % define the functions
6   y1 = tan(x);
7   y2 = cot(x);
8   % draw y1 = tan(x);
9   plot(x,y1,'r','linewidth',2);
10  hold on;
11  % draw y2 = cot(x)
12  plot(x,y2,'b','linewidth',2);
13  hold off;
14  legend('tan(x)', 'cot(x)');
15  title('the graphic of tan(x) and cot(x)');
16  xlabel('x');
17  ylabel('y1 = tan(x); y2 = cot(x)');
18  %
```

Code 1.15 *tan(x)_cot(x).m*.

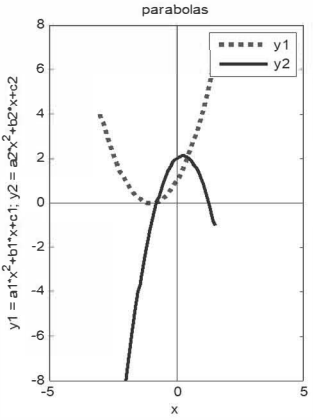

Fig. 1.3 Two parabolas drawn on the same graph.

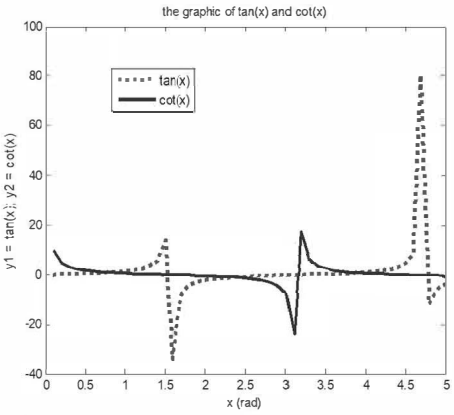

Fig. 1.4 The graphs of functions tan(x) and cot(x).

If a function depends on two variables, a case that will often be encountered in the following chapters, its graph is represented by a surface. When the expression of the function is linear, the function graph is a plane. When the expression of the function is nonlinear, its graphic is represented by a certain surface. There are several ways to visualize a surface. Those interested to know in detail the elementary list of functions that can be used for the graphical representation of surfaces need only use the command:

```
help graph3d
```

Besides the list of elementary functions of graphical representation of surfaces, this command provides additional information about how colors, lighting and transparency can be controlled, about axis, viewpoint and camera control, graph annotation and so on.
Although the graphs in this book do not constitute its main purpose, they are a particularly valuable tool for illustrating concepts. For this reason, mainly 3D surfaces presentations using the functions `mesh`, `contour` and `surf` (see Code 1.16) were employed.

```
1   %          surface3D.m
2   % setup the variables range
3   [x1,x2] = meshgrid(-5:0.5:5,-5:0.5:5);
4   f = x1.^2/3 - x2.^2/15+8;
5   % display the surface as a wireframe
6   mesh(f,'LineWidth',1.5);
7   legend('f(x1,x2)');
8   title('the mesh of f(x1,x2)');
9   xlabel('x1');
10  ylabel('x2');
11  figure;
12  % display the surface as contours
13  subplot(1,2,1);
14  contour (x1,x2,f,20); % number of contours = 20
15  legend('f(x1,x2)');
16  title('the contours of f(x1,x2)');
17  xlabel('x1');
18  ylabel('x2');
19  subplot(1,2,2);
20  % display again the surface as contours
21  [C,h] = contour(x1,x2,f,10); % number of contours = 20
22  % display the contours values
23  clabel (C,h);
```

```
24  legend('f(x1,x2)');
25  title('the contours of f(x1,x2)');
26  xlabel('x1');
27  ylabel('x2');
28  figure;
29  surf(x1,x2,f);
30  % remove edges on the surface
31  shading interp;
32  legend('f(x1,x2)');
33  title('the surface of f(x1,x2)');
34  xlabel('x1');
35  ylabel('x2');
36  %
```

Code 1.16 *surface3D.m*.

Let us consider, for example, a function f that depends on two variables, whose expression is:

$$f(x_1, x_2) = \frac{x_1^2}{3} - \frac{x_2^2}{15} + 8; \tag{1.9}$$

We can graphically represent the function given by Eq. 1.9 as a wireframe (see Fig. 1.5), using the `mesh` command in line 6 from Code 1.16. Various specifications can be added to the graph, such as line width, legend, title, x and y labels etc.

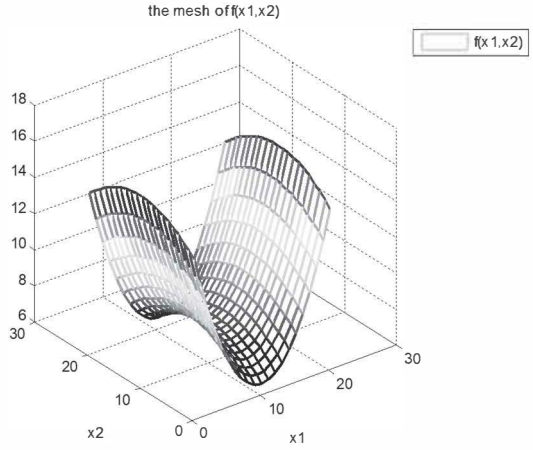

Fig. 1.5 The mesh of the function from Eq. 1.9.

The second alternative to display the surface by Eq. 1.9, is by using the `contour` command. Contours are the result of the intersection between the surface and several planes parallels to the horizontal plane, the resulted contours being projected on the xOy plane. This kind of surface representation is very convenient in numerous situations, so that it wil be used in the following chapters. Lines 14 to 27 in Code 1.16 create contours in two ways. Initially, contours are made without labeling them and following by inserting numerical values on each contour (see Fig. 1.6).

Matlab allows the user to display multiple drawings in the same window. This type of display use the `subplot` command. We can imagine several figures arranged in a table with m lines and n columns. To place a drawing in a certain position in the table, we must specify its location as `subplot(m,n,location)`, because locations are read from left to right and from top to bottom. For instance `subplot(2,3,3)` means the last position in the first row, while `subplot(2,3,4)` means the first position in the second row.

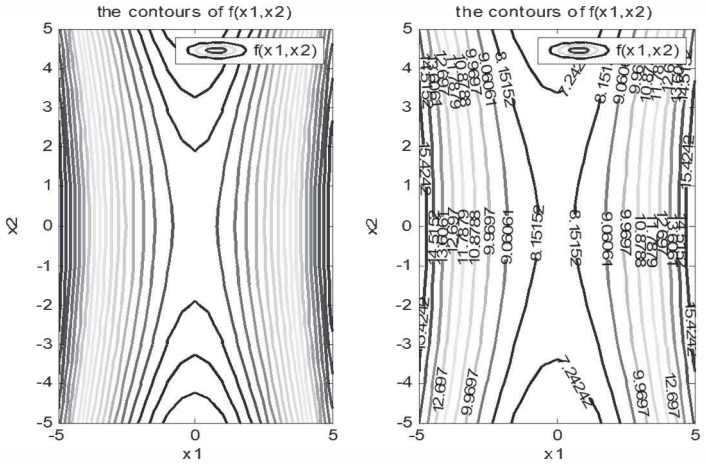

Fig. 1.6 The contours of the function in Eq. 1.9 without labels (left) and with labels (right).

Fig. 1.6 was made using `subplot` command (see lines 14 and 20 in Code 1.16) and illustrates the same surface made by `contour` command, on the left side, the contours being without labels, while on the right side being with labels. Fig. 1.7 shows a continuous surface made using `surf` command.

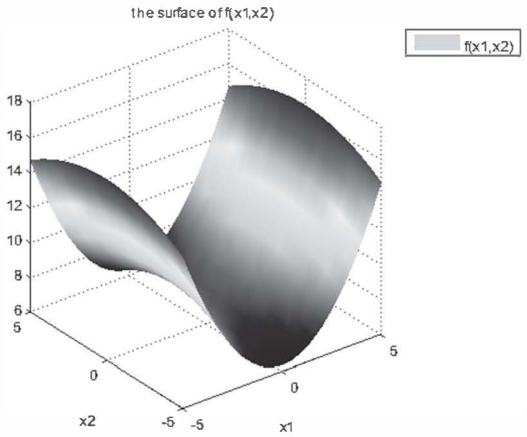

Fig. 1.7 The drawing of Eq. 1.9 made by the `surf` command.

There are various surface properties that control the appearance of the surface, regarding the faces, the edges, the markers, the colors and transparency, the lighting and so on.

1.9 Conclusion

The source programs presented in the following chapters are not complicated, they only require some minimal programming knowledge. This does not imply that the reader must limit oneself to this level of programming knowledge. Programs illustrating different optimization methods only facilitate understanding of the concepts presented and open the way for researchers to develop or include them in their own research or study programs.

The specialized literature abounds in Matlab language studies, from beginner works to advanced studies. The recommended bibliographic references are the manuals developed by MathWorks. All the programming instructions used in this book can be found in the *Matlab Getting Started Guide*, *Matlab Mathematics*, *Matlab Graphics*, and *Matlab Programming Fundamentals*. But there are other books that show a lot of information, accompanied by examples, in fewer pages. Among these books, we mention *Getting Started with MATLAB 7 - A Quick Introduction for Scientists and Engineers* written by Pratap (2006) or *MATLAB: An Introduction With Applications* of Gilat (2008). It's a good

idea to start with a Matlab tutorial, always accompanied by the MathWorks manual for the details that are beyond the scope of the tutorial.

2.

BASIC CONCEPTS

2.1 Introduction

Optimization is a branch of mathematics that provides us with the tools to determine the values of one or more parameters corresponding to the maximum or minimum of a function. Optimization problems were formulated as early as the time of Euclid, but the mathematical foundations were implemented only after the emergence of differential and variational calculus in the 17th and 18th centuries (Mihăileanu,1974). The importance of optimization is reflected by its many applications in virtually all branches of technology. A lot of optimization methods have been developed, each of which is dedicated to a particular type of problem. The fact that we still do not have an optimization method to solve any type of problem should not be seen as a shortcoming or a limitation. The continuous increase in computer performance makes the multitude of numerical optimization algorithms differentiated not only by the number of iterations or the number of steps required to reach the solution with a certain accuracy, but also by the time it takes to determine it. The researcher must carefully formulate the mathematical model of the optimization problem. It is the first and foremost step in solving the problem, because the final result will depend to the fullest extent on the accuracy of the mathematical model. The next step is choosing the right way to solve the problem and solve the problem effectively. The final step, just as important as the previous ones, consists of testing the numerical results.

2.2 The concept of optimization

In a complete mathematical formulation, the optimization problem has three basic components:

- an objective function that must be minimized or maximized. For example, in a manufacturing process we aim to maximize productivity or minimize cost.
- a set of unknown or decision variables that determines the value of the objective function. In manufacturing processes, decision variables may include set up parameters of technological processes, or the size of the different resources used, or the time involved in different actions, etc.
- a set of restrictions that allow unknown variables to take certain values and exclude others. For example, in some processes, decision variables can only take values within certain intervals.

Definition: *The optimization problem consists in determining the values of the decision variables that minimize or maximize the objective function and at the same time, fulfill all the restrictions.*

Although the vast majority of optimization problems have only one objective function, two exceptions are of note:

- lack of objective function; in certain situations (when designing integrated circuits), the purpose is to determine a set of variables that satisfy the restrictions. If it is not desirable to optimize something, then an objective function need not be defined. This type of problem is called the *problem of possibility*.
- more objective functions; it is often desirable to optimize a number of different objective functions simultaneously. Typically, these objective functions are not compatible with each other and the values of the decision variables for which one of the objective functions is optimal are far from optimal for the rest of the objective functions.

In practice, optimization problems with multiple objective functions are reduced to a single objective function, or some objective functions are considered to be restrictions. All these aspects will be fully described in the chapter about multi-criteria optimization.

Decision variables are essential. Their absence would make it impossible to mathematically define the objective function and the restrictions, respectively. Restrictions are not essential. The optimization without restrictions is one of the most important fields in optimization and boosts an impressive number of algorithms and computer programs.

We will introduce the basic concepts of numerical optimization theory by means of two simple examples. Let us consider the following algebraic function:

$$F(x_1, x_2) = 2x_1 + x_2 + (x_1^2 - x_2^2) + (x_1 - x_2^2)^2; \quad (2.1)$$

This function depends on two variables x_1 and x_2. The goal is to determine the values of x_1 and x_2 that correspond to the minimum value of the function $F(x_1,x_2)$. According to this goal, the function (1.1) is called *the objective function* or simply *the objective*. The variables x_1 and x_2 are called *decision variables*. So, our problem depends on two decision variables only. However, there are no limits on the number of decision variables. Because there are no additional conditions imposed on the decision variables, the problem of determining the minimum of the algebraic function (2.1) is called *optimization problem without constraints*. Fig.2.1 shows the three-dimensional image of the objective function (2.1).

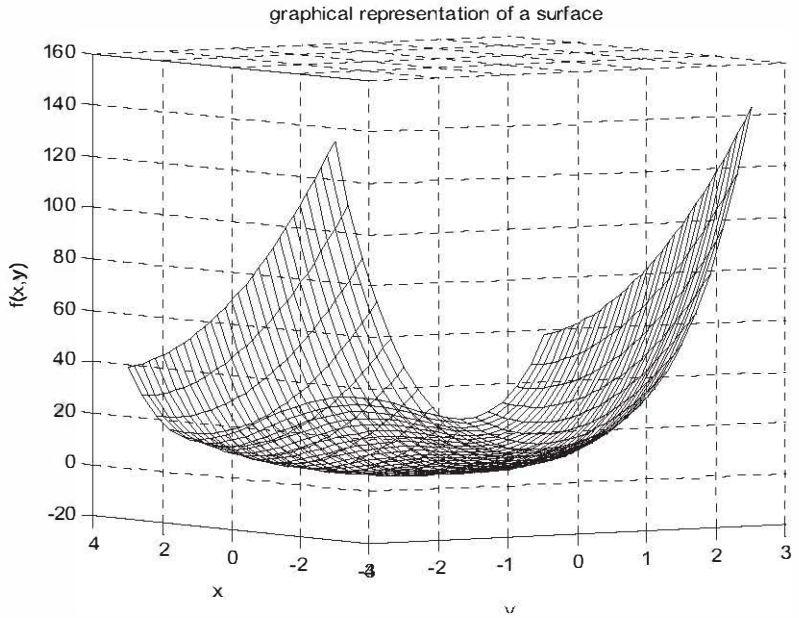

Fig. 2.1 The graphical representation of the function (2.1).

Code 2.1 illustrates the Matlab program used to make the graphical representation in Fig. 2.1. This source code makes a call to the function *f.m* (see Code 2.2) whose expression belongs to Eq. 2.1.

```
%            fig2_1graphic.m
% we define the ranges of the variables x,y
[x,y] = meshgrid(-3:0.2:3,-3:0.2:3);
z = f(x,y);
% plot the surface from Fig2.1;
% some specifications are added
surf(x,y,z,'LineWidth',1.0, 'FaceColor', 'none',...
'EdgeColor', 'black');
% axes labels and title
xlabel 'x';
ylabel 'y';
zlabel 'f(x,y)';
title 'graphical representation of a surface';
%
```

Code 2.1 *fig2_1graphic.m*.

```
%                    f.m
% the evaluation of function f in a point
function [val] = f(x,y)
val = 2*x+y+(x.^2-y.^2)+(x-y.^2).^2;
end
%
```

Code 2.2 *f.m*.

In order to easily appreciate the location of the minimum point of this function, this graphical representation can be made by contours like in Fig. 2.2. For this, let us imagine that we are cutting the surface defined by the objective function in Eq. 2.1 with planes Γ_i that are parallel to the plane x_1Ox_2 (the horizontal plane). The intersection of any of the planes Γ_i with the surface in Fig. 2.1 is a closed curve. The vertical projection of all these closed curves on a horizontal plane gets the contours displayed in Fig. 2.2. In this picture, it is easy enough to assume that the position of the minimum point is very close to the origin of the Cartesian axis of coordinate system. The Code 2.3 contains the commands list required to make the drawing of the surface contours from Fig. 2.2.

```
%           fig2_2graphic.m
%
% we define the ranges of the variables x,y
[x,y] = meshgrid(-3:0.2:3,-3:0.2:3);
% calculation of z values
z = f(x,y);
% draw the cotours of the surface
% there are 30 contours specified
```

```
9    [c,h] = contour(x,y,z,30);
10   % label manually desired contours
11   clabel(c,h,'manual');
12   %
```

Code 2.3 *fig2_2graphic.m*.

In the drawing window you will see a reticle that moves with the mouse. Place the reticle on the desired contour in a desired position and left click with the mouse. The value of that contour will be displayed in the specified position.

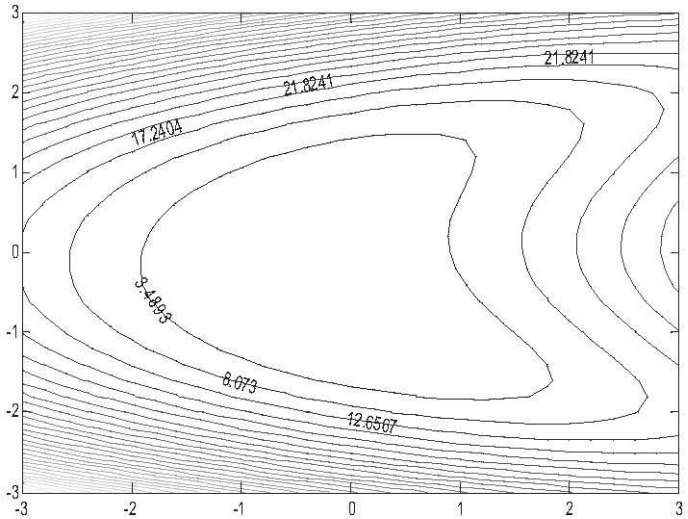

Fig. 2.2 Graphical representation of the objective function (2.1) by contours.

The minimum point fulfills the condition that the first order partial derivatives of function in Eq. 2.1 are null:

$$\begin{cases} \dfrac{\partial F(x_1, x_2)}{\partial x_1} = 2 + 2x_1 + 2(x_1 - x_2^2) = 0\,; \\ \dfrac{\partial F(x_1, x_2)}{\partial x_2} = 1 - 2x_2 - 4x_2(x_1 - x_2^2) = 0\,; \end{cases} \tag{2.2}$$

By solving the system in Eq. 2.2 the optimum solution obtained is:

$$\begin{cases} x_1^* = -0.185; \\ x_2^* = -0.793; \end{cases} \tag{2.3}$$

In this example, the optimal solution of the optimization problem is easily determined both graphically and analytically. In terms of graphical representation it is easy to estimate the position of the minimum point if we represent the function in Eq. 2.1 by constant value curves (see Fig. 2.2). In practical cases, however, when the objective function is strongly nonlinear and the number of decision variables is large, neither of these alternatives is particularly useful, but we can still successfully apply a numerical method.

In the next example we consider the same objective (see Eq. 2.1), accompanied by the inequality-type constraint, denoted $g(x_1, x_2)$, that has the expression:

$$g(x_1,x_2) = 2x_1 - 1000x_2 \leq 0\,; \tag{2.4}$$

This new example is an *optimization problem with constraints*. In this situation, it is necessary to calculate the values of the decision variables for which the objective function (2.1) takes a minimum and in the same time the constraint (2.4) imposed on the problem is verified. In Fig. 2.3, both the objective function (2.1) and the constraint (2.4) are represented in a tri-orthogonal axis system.

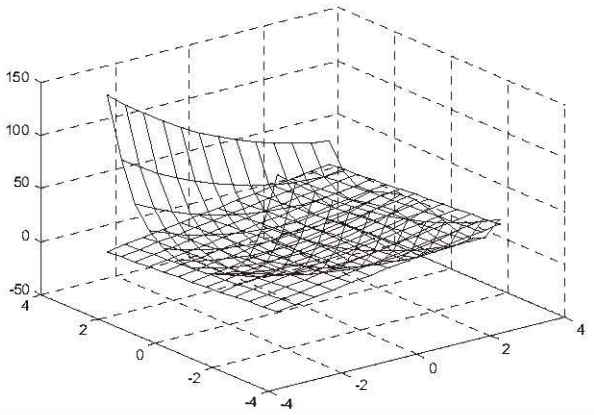

Fig. 2.3 The 3D image of the objective function (2.1), along with the constraint (2.4).

Code 2.4 illustrates the Matlab source code needed to draw the surface $f(x_1,x_2)$ from Eq. 2.1, as well as the constraint $g(x_1,x_2)$ from Eq. 2.4.

```
%            fig2_3graphic.m
% we define the ranges of the variables x,y
[x,y] = meshgrid(-3:0.5:3,-3:0.5:3);
z1 = f(x,y);
z2 = g(x,y);
mesh(x,y,z1,'LineWidth',1.3, 'FaceColor', 'none',...
  'EdgeColor', 'black');
hold on;
mesh(x,y,z2, 'LineWidth',0.5, 'FaceColor', ...
  'none', 'EdgeColor', 'black');
hold off;
```

Code 2.4 *fig2_3graphic.m*.

In Fig. 2.4, even if the same functions are represented by contours from Fig. 2.4, it is not easy to gauge the change of the optimal point location with restrictions. This point is close to the origin of the x_1–x_2 coordinate system, and solving the problem numerically leads to the result:

$$\begin{cases} x_1^* = -0.4999; \\ x_2^* = -0.0005; \end{cases} \quad (2.5$$

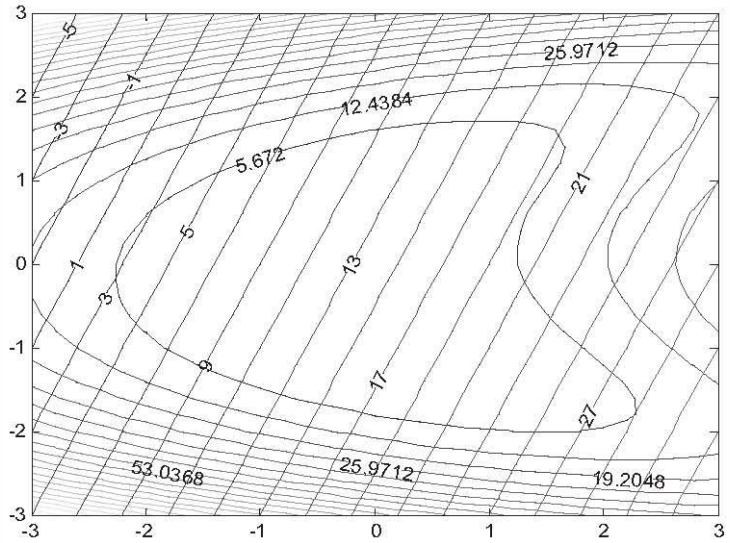

Fig. 2.4 The contours of both objective function (2.1) and constraint (2.4).

The source code needed to draw the contours of both the function from Eq. 2.1 and its constraint from Eq. 2.4 can be seen in Code 2.5.

```
%              fig2_4graphic.m
%
% Plot two surfaces on the same graph
[x,y] = meshgrid(-3:0.2:3,-3:0.2:3);
z1 = f(x,y);
z2 = g(x,y);
%
[c,h] = contour(x,y,z1,20);
hold on;
[d,e] = contour(x,y,z2,20);
hold off;
% manually labeling of contours
clabel(c,h,'manual');
clabel(d,e,'manual');
%
```

Code 2.5 *fig2_4graphic.m*.

As in the case of Code 2.3, when you execute this program the reticle will appear, so that each desired value of the contours that belong to $f(x_1,x_2)$ can be displayed in the desired position. After finishing the marking of the function $f(x_1,x_2)$ contours, press the *Enter* key and continue in the same way with the contours that belong to the constraint $g(x_1,x_2)$. At the end of this operation, press the *Enter* key again.
In the optimal point with constraints, the objective function (2.1) will have the value $f^{min} = 7.5$, while, at the same point, the constraint (2.4) is still satisfied, that is $g(x_1, x_2) = -0.000546 < 0$.

2.3 The general mathematical model

After having seen the two examples above, we can formulate the general mathematical model of the optimization problem in the following form:

$$\begin{aligned} &Minimize\ F(X) \\ &\text{Subject to:} \\ &g_j(X) \le 0; \qquad j = 1,2,\dots,m \\ &h_k(X) = 0; \qquad k = 1,2,\dots,l \\ &x_i^{min} \le x_i \le x_i^{max}; \quad i = 1,2,\dots,n \end{aligned} \tag{2.6}$$

$$where\ X = \begin{Bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{Bmatrix};$$

Within this model, the constraints can also be linear or nonlinear functions, depending on the decision variables. At the same time, these inequalities can be expressed through explicit or implicit functions and can be evaluated both analytically and numerically according to the procedures we have. It is often important to know whether these functions and their first-order derivatives are continuous. If the equality constraints $h_k(X)$ are given in explicit form, then they can be used to reduce the number of decision variables considered. However, this alternative can not always be applied because equality restrictions, even given in explicit form, may have complicated non-linear expressions. The inequalities $x_i^{min} \le x_i \le x_i^{max}$, $i = 1,2,\ldots,n$, are called explicit constraints. These constraints are included in the global set of constraints, but it is preferable to treat them separately as they define the domain of existence for the decision variables. The way the general mathematical model (Eq. 2.6) of the optimization problem was defined is not unique, and other equivalent formulations can be found in the literature.

2.4 The iterative computation

The greater part of numerical optimization problems require an innitial starting point, called X^0, defined by some values for the decision variables. Going on from this start point, the computation of the optimum point is completed iteratively. One of the most common equations used by iterative algorithms is:

$$X^q = X^{q-1} + \alpha^* \cdot S^{q-1}; \tag{2.7}$$

where:

- q is an iterative index;
- S – the search vector that operate in the space defined by the constraints set of the mathematical model;
- α^* - a scalar called the leap factor. It defines the jump from the point X^{q-1} to the point X^q, towards direction S.

Let us consider a mathematical model of an optimization problem with constraints. For an easier display of the situation, let suppose that the

vector X depends on two variables (see Fig. 2.5). Also, let us consider $x_1^0 = -8$ and $x_2^0 = -28$ as the coordinates of a starting point X^0. Let us admit that we know the minimization direction indicating by the vector S, whose coordinates are:

$$S = \begin{Bmatrix} -2.00 \\ -0.77 \end{Bmatrix};$$

If the leap factor is $\alpha = 2$, then we can calculate the coordinates of the point X^1 as:

$$X^1 = X^0 + \alpha \cdot S = \begin{Bmatrix} -8 \\ -28 \end{Bmatrix} + 2 \cdot \begin{Bmatrix} -2.00 \\ -0.77 \end{Bmatrix} = \begin{Bmatrix} -8 \\ -28 \end{Bmatrix} + \begin{Bmatrix} -4 \\ -1.54 \end{Bmatrix} = \begin{Bmatrix} -12.0 \\ -29.6 \end{Bmatrix};$$

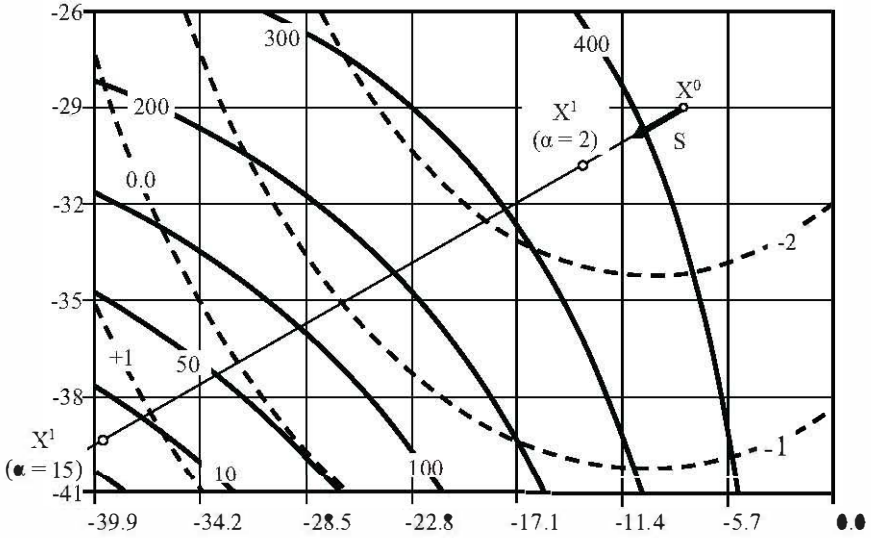

Fig. 2.5 The iterative calculation of the optimum solution.

At this moment, examining Fig. 2.5, we may estimate that $F(X^1) \approx 360$ and $g(X^1) \approx -2.8$, which is a better position relative to $F(X^0) \approx 425$. Because of the leap factor value $\alpha = 2$, we arriv at a point X^1 where $F(X^1)$ has a lower value than in X^0 while the constraint $g(X^1)$ is still respected. This fact suggests to increase the leap factor α. Now, let us consider a leap factor $\alpha = 15$. If we repeat the previous calculations, we get:

$$X^1 = X^0 + \alpha \cdot S = \begin{Bmatrix} -8 \\ -28 \end{Bmatrix} + 15 \cdot \begin{Bmatrix} -2.00 \\ -0.77 \end{Bmatrix} = \begin{Bmatrix} -8 \\ -28 \end{Bmatrix} + \begin{Bmatrix} -30.00 \\ -11.55 \end{Bmatrix}$$
$$= \begin{Bmatrix} -38.00 \\ -39.55 \end{Bmatrix};$$

This time, it can be appreciated that in the new point X^1, the objective function has the value $F(X1) \approx 8$ that is much better than before, but $g(X1) \approx +1.6$, which means that the constraint is not respected. So, the leap factor value chosen was too large, and must be decreased. In the iterative Eq. 2.7, for the leap factor a notation such as α^* was used. This indicates an optimal value for the leap factor, which is a value that leads to the decreasing of the objective function in the direction of vector S, while the constraints are still respected.

From a practical point of view, all estimates made above must be done automatically on a computer, and using different values of α, we can apply a numerical interpolation method to determine that α^* value, for which the objective function $F(X)$ takes the lowest value in direction of S and at the same time all additional conditions imposed by the restrictions are respected. Through this search in different directions, we transform the problem of optimization dependent on n variables into a problem dependent on a single variable, namely α. Consequently, the name of this optimization procedure is *one-dimensional search*. Once the point X^1 is reached, the minimization direction at this point must be determined and then a reduction of the value of the objective function along the new minimization direction should be attempted.

From this example, it can be easily concluded that nonlinear optimization algorithms based on the Eq. 2.7 can be separated in two parts. The first part concerns the determination of the search direction S. The second part concerns the determination of the optimal α^*, that is the leap factor which defines the movement size in the search direction S. As will be seen, each of these components must be integrated into a numerical optimization program, and plays a fundamental role in the efficiency of a numerical optimization algorithm.

2.5 The existence and uniqueness of the optimal solution

From a practical point of view, it is especially important to know if the optimal point is a relative or a global one. We must keep in mind that the optimization problems encountered in practical situations are nonlinear, nonconvex and show multiple optimal local points. So, it is recommended

to restart the optimization procedure several times, from different starting points, and if the result of optimization is the same, then we can be convinced that we have found the optimal point. Sometimes, however, it is possible to determine the mathematical conditions necessary to prove the existence of the optimal point and at the same time their sufficiency to prove that the optimal point is a global optimum and not just a relative one.

2.5.1 The existence and uniqueness of the optimal solution in the absence of constraints

For the beginning, we intend to determine the existence and uniqueness of the minimum point of the objective function $F(X)$, when no additional conditions are imposed on the optimization problem. Fermat's theorem tells us that if a point X is an extreme point for the function $F(X)$, then the gradient of the function must be zero at that point, that is:

$$\nabla F(X) = 0; \tag{2.8}$$

where:

$$\nabla F(X) = \begin{Bmatrix} \frac{\partial F(X)}{\partial x_1} \\ \frac{\partial F(X)}{\partial x_2} \\ \vdots \\ \frac{\partial F(X)}{\partial x_n} \end{Bmatrix}; \tag{2.9}$$

As we know, the reciprocal of this theorem is not true and therefore condition (Eq. 2.8) is necessary but not enough to make sure that the the point X is indeed the minimum of the function $F(X)$. This is simple to demonstrate if we refer to Fig. 2.6, which is the graphical variation of a function dependent on a single variable.

In this case, the gradient of the function $F(x)$ is actually its first-order derivative in relation to the variable x. As can easily be seen in Fig. 2.6, the gradient of the function $F(x)$ is zero in each of the three points x_1, x_2 or x_3 respectively. At points where the first-order derivative is zero, the tangent to the graph of $F(x)$ is horizontal and in Fig. 2.6, the tangent to the function graph is horizontal in all three points A, B and C respectively.

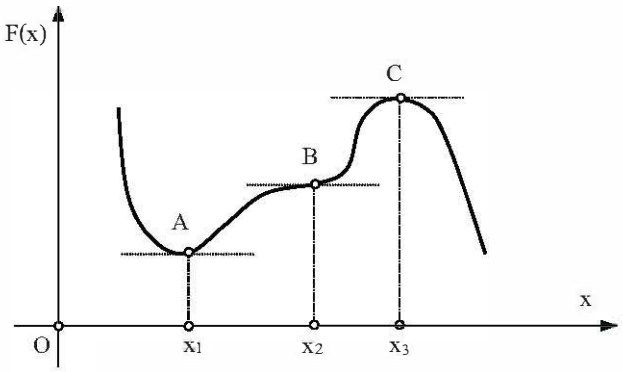

Fig. 2.6 Relative optima of a function that depends on a single

Point x_1 is a minimum point, point x_3 is a maximum point, while point x_2 is neither a minimum point nor a maximum point. However, the first-order derivative of the function *F(x)* is zero at each of the three points. Therefore, the condition only that the gradient of the function *F(X)* being zero at a point does not immediately mean that that point is an extreme one for the function.

From mathematical analysis we know that to admit a minimum point, the second-order derivative of a function must be positive, and for the function in Fig. 2.6, this condition is only satisfied by the point x_1. In the general case, for functions dependent on n variables, this is equivalent to the condition that the *Hessian* matrix is positively defined. Through *Hessian* matrix, the matrix of the second order partial derivatives of *F(X)* is understood, meaning:

$$H(X) = \begin{Bmatrix} \frac{\partial^2 F(X)}{\partial x_1^2} & \frac{\partial^2 F(X)}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 F(X)}{\partial x_1 \partial x_n} \\ \frac{\partial^2 F(X)}{\partial x_2 \partial x_1} & \frac{\partial^2 F(X)}{\partial x_2^2} & \cdots & \frac{\partial^2 F(X)}{\partial x_2 \partial x_n} \\ \cdots & \cdots & \cdots & \cdots \\ \frac{\partial^2 F(X)}{\partial x_n \partial x_1} & \frac{\partial^2 F(X)}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 F(X)}{\partial x_n^2} \end{Bmatrix}; \qquad (2.10)$$

Definition:

A function F(X) is convex (concave) if and only if for any pair of distinct u and v points in the domain of F(X) and for $0 < \theta < 1$,

$$\theta \cdot f(u) + (1-\theta) \cdot f(v) \left\{ \begin{matrix} \leq \\ \geq \end{matrix} \right\} f(\theta \cdot u + (1-\theta) \cdot v); \qquad (2.11)$$

Note: If F(X) is a linear function, then it is convex and concave at the same time.

Theorem 1:

If $F_1(X)$ and $F_2(X)$ are both convex (concave), then $F_1(X) + F_2(X)$ is also convex (concave).

Theorem 2:

A continuous and double differentiable function F(X) is convex if and only if $|H_i(X)| \geq 0$, $i = 1, 2, ..., n$, where $H_i(X)$ denotes the diagonal minors of the hessian matrix (2.10).

Theorem 3:

A continuous, double differentiable function F(X) is concave if and only if $|H_1(X)| \leq 0$, $|H_2(X)| \geq 0$, $|H_3(X)| \leq 0$, ..., $(-1)^n \mid H_n(X)| \geq 0$.

Example:

Determine the character of the function:

$$F(x_1, x_2) = -(x_1 - 1)^2 - (x_2 - 1)^2; \qquad (2.12)$$

If we represent the graph of this function, we will get the one in Fig. 2.7.

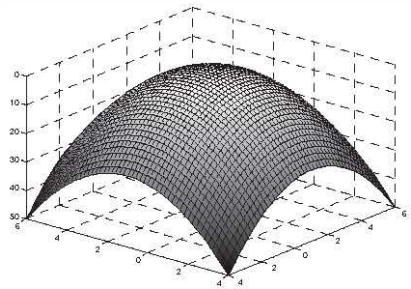

Fig. 2.7 The graph of the function 2.12.

To find the character of the function in Eq. 2.12, its gradient must be calculated:

$$\nabla F(X) = \begin{pmatrix} \dfrac{\partial F}{\partial x_1} \\ \dfrac{\partial F}{\partial x_1} \end{pmatrix} = \begin{pmatrix} -2(x_1 - 1) \\ -2(x_2 - 1) \end{pmatrix}; \qquad (2.13)$$

The *Hessian* matrix is:

$$H(X) = \begin{bmatrix} \frac{\partial^2 F}{\partial x_1^2} & \frac{\partial^2 F}{\partial x_1 \partial x_2} \\ \frac{\partial^2 F}{\partial x_2 \partial x_1} & \frac{\partial^2 F}{\partial x_2^2} \end{bmatrix} = \begin{bmatrix} -2 & 0 \\ 0 & -2 \end{bmatrix}; \quad (2.14)$$

The evaluation of the *Hessian* matrix minors shows us that:

$|H_1(X)| = -2 < 0$;
$|H2(X)| = +4 > 0$;

Therefore, function in Eq. 2.12 is concave (it admits a maximum point), which can also be seen in Fig. 2.7.

2.5.2 Existence and uniqueness of the optimal solution in the presence of constraints

For the beginning, let us consider an optimization problem that has an inequality constraint. To examine this situation we will refer to Fig.2.8.

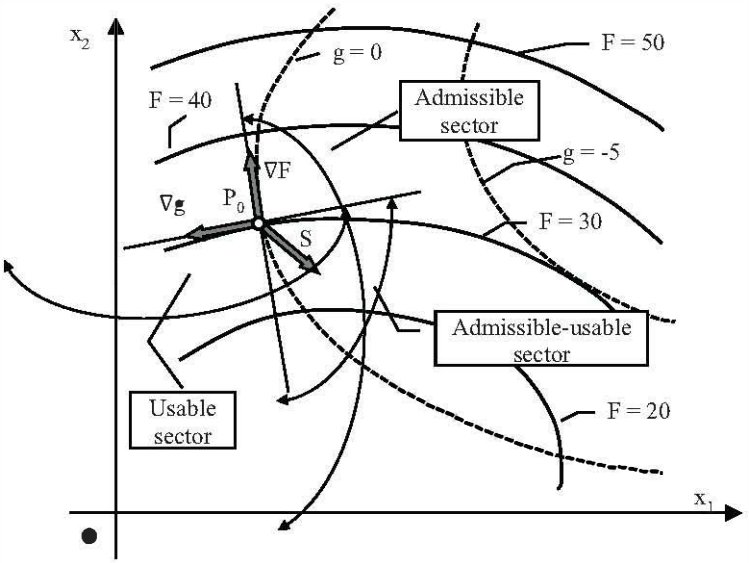

Fig. 2.8 The search direction of the optimal solution.

Let us start from the point P_0. In order to find a new point with a lower value of the objective function, a search direction must be calculated in a way to avoid overcomes the constraint. Any direction S in which the objective function is minimized will be called *usable direction*. Clearly, the tangent to the contour of the objective function in this point will limit all possibly usable directions. Any direction S lateral to this line, which will reduce the value of the objective function, is defined as *usable direction*, and this portion of the space will be called *usable sector*. Here we should note that the scalar product of any S-direction vector that we take in the usable sector with the objective function gradient will be negative or zero, meaning:

$$S \cdot \nabla F(X) \leq 0; \tag{2.15}$$

This means that the angle between the two vectors has to be inside the interval of [90°, 270°] because the scalar product of two vectors is defined as the product of the absolute magnitude multiplied by the cosine of the angle between the two vectors, and the cosine of an angle is negative for values of the angle within the above mentioned range. Analogously, a hyperplane tangent to a constraint surface at point P_0 will limit the *admissible sector*, where the direction vector S is defined as admissible if a small movement in that direction does not exceed the limit imposed by the constraint. In this case, the scalar product between the direction vector S and the constraint gradient is negative or null, meaning:

$$S \cdot \nabla g(X) \leq 0; \tag{2.16}$$

Therefore, the angle between the vectors S and $\nabla g(X)$ must be also in the range of [90°, 270°]. In conclusion, in order for the direction vector S to lead toward the optimal point, it must at the same time be an acceptable and usable direction, so any direction vector S, located in the *admissible-usable sector* satisfies this criterion. If the direction vector S is almost tangent to the hyperplane that limits the *admissible sector*, a small move in that direction will result in the exceeding of the constraint, but will quickly lead to a decrease in the value of the objective function. On the other hand, if the direction vector S is chosen almost tangent to the contour line of the constraint, moving in that direction will always respect the condition imposed by that constraint, but the value of the objective function will not decrease greatly. Moreover if the objective function is nonlinear, there may be a possibility that the value of the objective function may begins to increase. Generally, optimization problems may have more active

restrictions, where a restriction is considered active if its value is less than or equal to zero. In this case, the direction vector S that fulfills all the constraints must be found. The direction vector S must simultaneously meet both requirements, meaning to be located in the admissible-usable sector. This is mathematically expressed as follows:

$$\begin{aligned} &S \cdot \nabla F(X) \leq 0; \\ &S \cdot \nabla g_j(X) \leq 0; \quad \forall j \, for \, which \, g_j(X) \leq 0. \end{aligned} \tag{2.17}$$

Let us consider now the point P_1 in Fig. 2.9, which is the optimal (minimum) point for this example. At this point, the gradient of the objective function and the gradient of the constraint have the same direction, but different orientations. For this, the vector S will at the same time be tangent to the contour line of the objective function $F(X)$ and the contour line of the constraint, so it will form an angle of 90° with both gradients. It means that:

$$\nabla F(X) + \sum_{j=1}^{m} \lambda_j \cdot \nabla g_j(X) + \sum_{k=1}^{l} \lambda_{m+k} \cdot \nabla h_k(X) = 0; \tag{2.18}$$

where $\lambda_j \leq 0$, and λ_{m+k} is without sign restriction.

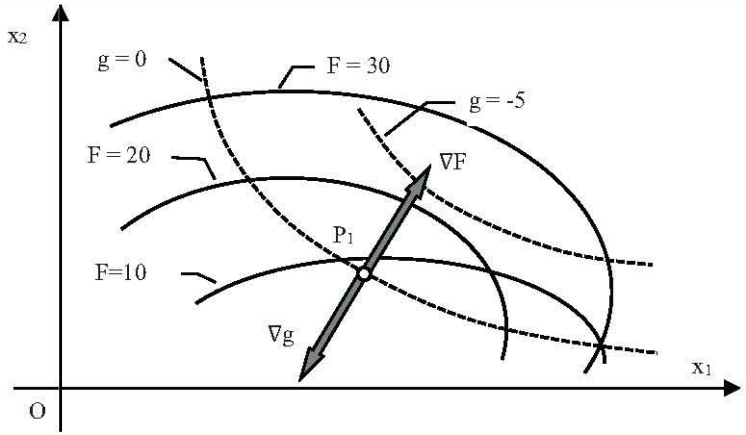

Fig. 2.9 A limit case: ∇g and ∇F are opposite.

If the optimization problem is without constraints, then the Eq. 2.18 is reduced to the condition in Eq. 2.8 to zeroing the objective function

gradient. Eq. 2.18 is one of the Kuhn-Tucker conditions that defines the existence of an optimum point, and this condition must be fulfilled. If vector X defines an optimum point, in addition to condition in Eq. 2.18, the following two conditions must be fulfilled:

$$X \text{ must be situated in the admissible sector;} \tag{2.19}$$

$$\lambda_j \cdot g_j(X) = 0, \quad j = 1,2,\dots,m, \quad \lambda_j \geq 0. \tag{2.20}$$

Condition (2.19) expresses the need for point X to satisfy all the constraints. Also, equation (2.20) imposes the condition that if one of the constraints $g_j(x)$ is not exactly satisfied (i.e. $g_j(x) \leq 0$), then, accordingly, Lagrange's multiplier λ_j must be null.

2.6 Conclusions

Understanding the conditions of existence and uniqueness of a solution for optimization problems is a very important aspect in solving them. Mathematical modeling of processes often leads to highly non-linear functions, functions that are neither unimodal nor convex. The vast majority of the theoretical work in this field (Bonnans,2006), (Nesterov, 2004), (Nesterov, 2018), deals extensively with the general formulation of the optimization problems and the conditions under which these problems can be solved. The trouble is that optimization issues cannot be resolved as we would like, or most often can not be solved with the method we chose. For this reason, a user should understand that for solving such problems, one should be well informed regarding the basic theoretical notions they are going to use. Several specialized books are accompanied by examples and exercises (Kochenderfer, 2019) that deserve attention. Some optimization problems may not admit a solution. Detailed knowledge of such notions can spare us unnecessary effort.

3.

OPTIMIZATION TECHNIQUES FOR ONE VARIABLE UNCONSTRAINED FUNCTIONS

3.1 Introduction

If the objective function whose extreme point must be determined depends only on one variable, in the absence of additional restrictions, we have to solve the equation:

$$\frac{df(x)}{dx} = 0; \tag{3.1}$$

As we have already seen, this is the necessary condition for the existence of an extreme point, be it a minimum or a maximum. To solve Eq. 3.1, besides the fact that the function *f(x)* must be differentiable, the expression of the equation must be as simple as possible, so that we can determine its solution. In the following, some numerical optimization methods for determination of the optimal solution will be presented, even when the above conditions are not satisfied.

3.2 Finding the boundaries of the interval containing the optimal solution

In all numerical procedures presented in this chapter, we suppose that the boundaries of the interval that contain optimal solution are known in advance. If these are not yet known, then we need a method that finds them as simply as possible. We assume that the function $y = f(x)$ is unimodal[1] and the extreme point is situated in the positive range of the Ox axis, as can be seen Fig. 3.1. For this reason, we can set the left bound x_a of the interval containing the minimum point of the function $y = f(x)$ in the origin of the coordinate axis system. Let us assume that $x_a = \tau$, where $\tau =$

[1] The unimodal function has a single extreme point in the searching interval.

0,381966[2]. Let us calculate the values of $f(x)$ in $x_a = 0$ and $x_b = \tau$. If $f(x_a) < f(x_b)$ then, given that the function is unimodal and the minimum point is in the positive range of the Ox axis, we conclude that x_b is indeed the right bound of the interval containing this point and the search process is considered finished. If $f(x_a) > f(x_b)$, then we take point x_b as x_1 and we recalculate x_b:

$$x_b = (1 + \tau) \cdot x_1 + \tau \cdot x_a; \tag{3.2}$$

We recalculate the value of the function in the new point x_b. If $f(x_1) > f(x_b)$ then we will take x_1 as x_a, point x_b as x_1 and recalculate point x_b with Eq. 3.2 (see step 3 in Fig. 3.2). The numerical procedure continues in the same way, with the recalculation of points x_a, x_1 and x_b, until $f(x_1) < f(x_b)$.

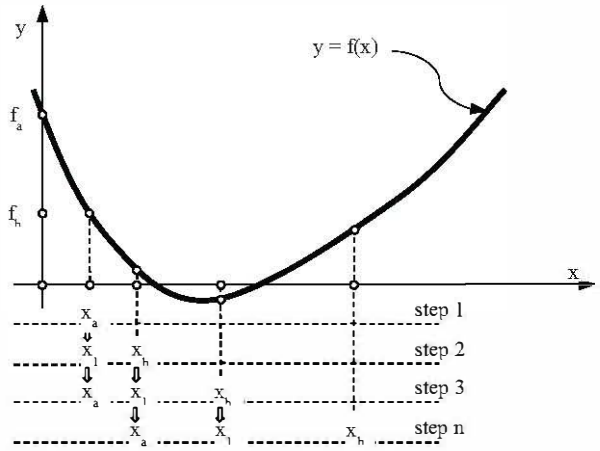

Fig. 3.1 Steps to find the interval containing the minimum point of the function $y = f(x)$.

```
1  %
2  %                    BOUNDARIES
3  % This program calculates the boundaries of the
4  % interval that includes the minimum point of
5  % an unimodal function, whose minimum point is
6  % located in the positive domain of Ox axis
7  %
8  % open a file to save results
```

[2] This reduction rate will be detailed later, in the paragraph about the golden section optimization method.

```
fp = fopen('results.txt','w');
fprintf(fp,'                    BOUNDARIES\n')'
fprintf(fp,'This program calculates the boundaries of
the\n');
fprintf(fp,'interval that includes the minimum point of
for\n');
fprintf(fp,'a unimodal function, whose minimum point
is\n');
fprintf(fp,'located in the positive domain of Ox
axis\n\n');
% initial left bound
xa = 0.0;
tau = 0.381966;
xb = tau;
fa = f(xa);
fb = f(xb);
fprintf(fp,'xa=%f   xb=%f   fa=%f   fb=%f\n',...
    xa,  xb,  fa,  fb);
if fa < fb
    left_bound = xa;
    right_bound = xb;
end
while fa > fb
    x1 = xb;
    f1 = fb;
    xb = (1+tau)*x1+tau*xa;
    fb = f(xb);
    fprintf(fp,'xa=%f   xb=%f   fa=%f   fb=%f\n',...
        xa,  xb,  fa,  fb);
    if f1 >= fb
        xa = x1;
        fa = f1;
    elseif f1 < fb
        left_bound = xa;
        right_bound = xb;
    end
end
% print to file the final results
fprintf(fp,'\n  Final result:\n');
fprintf(fp,'leftBound=%f    rightBound=%f\n',...
    left_bound,right_bound);
% close the file
fclose(fp);
%
```

Code 3.1 *boundaries.m.*

The program *boundaries.m* in Code 3.1 calculates the left and right bounds of the interval that contains the minimum point of a function. It makes a call to a function *f.m* in Code 3.2.

```
function [val] = f(x)
% the evaluation of function f in a point x
% f(x) represents the objective function
% f(x) = x*x - 4*x + 3;
val = x*x - 4*x + 3;
end
```

Code 3.2 *f.m*.

The objective function in Code 3.2 is a parabola of equation:

$$f(x) = x^2 - 4x + 3; \tag{3.3}$$

The function in Eq. 3.3 has a minimum point whose coordinates are [2,-1]. In this case, the program will start the search process taking as left bound the origin of the coordinate system and, step by step, will include within its boundaries this minimum point.

Table 3.1 History of iterations to find boundaries.

step	left bound x_a	right bound x_b	$f_a = f(x_a)$	$f_b = f(x_b)$
1	0.000000	0.381966	3.000000	1.618034
2	0.000000	0.527864	3.000000	1.167184
3	0.381966	0.875388	1.618034	0.264752
4	0.527864	1.411383	1.167184	-0.653530
5	0.875388	2.284852	0.264752	-0.918860
6	1.411383	3.696687	-0.653530	1.878748

After the execution of the program, the value of x_a = 1.411383 was determined as the left bound for the minimum point of the function in Eq. 3.3, and for the right bound x_b = 3.696687.

3.3 The grid method

Let us consider an unimodal function $y = f(x)$, and let us also admit as known the bounds a and b of the interval containing the minimum point of this function. We choose a number of n equal subdivisions of the interval $[a, b]$. The size δ of a subdivision (see Fig. 3.2) is:

$$\delta = (b - a)/n\,; \tag{3.4}$$

We calculate the value of the function both at the ends of the interval $[a, b]$ and for each value of $(a+i\cdot\delta$, where $i = 1,2,\ldots,n)$. From the latter, we choose that subdivision bound for which the value of the function is lowest and designate it A_1. We then choose the value A_1 for x_{min} and the search interval at the next step will have the minimum and the maximum limits to the left and right of point A_1 respectively: the left bound = (x_{min} - δ) and the right bound = (x_{min} + δ), as we can see in Fig. 3.2. At this stage, we can consider the first iteration finished. For the second iteration, the new search interval, of length 2δ (see Fig. 3.2), will be divided into other n subintervals. We recalculate the new δ, then evaluate the value of the function within each subinterval, after which we choose the point A_2, where the function to be minimized takes the lowest value. We recalculate the new bounds of the search interval and so on.

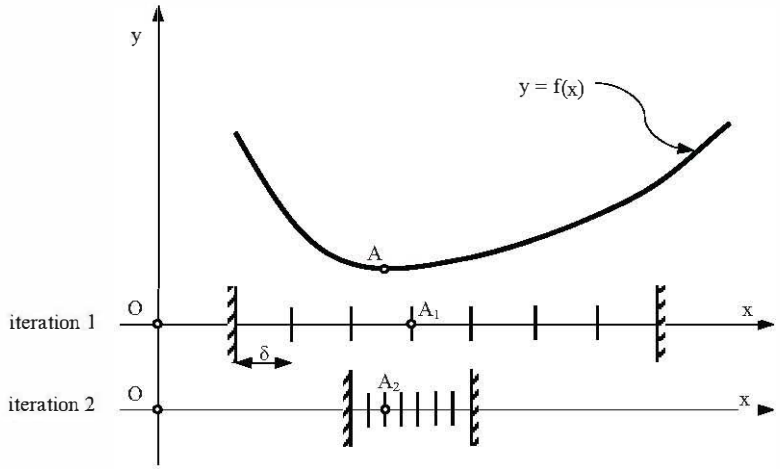

Fig. 3.2 The search process of the grid method.

During each iteration we calculate the absolute value of the length of the search interval and compare it with a precision factor ε, chosen in advance. The search process of the grid[3] method stops when the absolute value of the length of the search interval becomes less or equal to the precision

[3] The name of the method stems from the grid as a series of bars dividing an interval or domain.

factor ε. The program in Code 3.3 uses the objective function defined by Eq. 3.3 and listed in Code 3.2.

```
%
%                  Grid optimization
%
% define the boundaries of the search interval
% that were found with 'boundaries.m'
xInf = 1.41;
xSup = 3.69;
% define the convergence criterion
eps = 0.001;
% define the number of grids
grids = 20;
% calculate the value of the increment
increment = (xSup-xInf)/grids;
% assign the minimum value of the objective function
fMin = f(xInf);
precedent_fMin = fMin;
% open a file
fp = fopen('results.txt','w');
fprintf(fp,'      Grid Optimization Method\n\n');
% delta_l absolute length of the searching interval
% initial value for delta_l
delta_l = abs(xSup - xInf);
xMin = xInf;
% print initial values
fprintf(fp,'  Input data:\n');
fprintf(fp,'eps = %f\n',eps);
fprintf(fp,'grids = %d\n',grids);
fprintf(fp,'xInf=%f  xSup=%f  xMin=%f  fMin=%f\n',...
    xInf,  xSup,  xMin,  fMin);
fprintf(fp,'\n');
fprintf(fp,'  Iteration:\n');
%
while delta_l>eps
    x = xInf;
    for i=0:grids
        func = f(x);
        if func<fMin
            xMin = x;
            fMin = func;
        end
        x = x+increment;
    end
```

```
43        fprintf(fp,'xInf=%f xSup=%f xMin=%f fMin=%f\n',...
44            xInf,  xSup,  xMin,  fMin);
45        actual_fMin = fMin;
46        delta_1 = abs(xSup - xInf);
47        xInf = xMin-increment;
48        xSup = xMin+increment;
49        increment = (xSup-xInf)/grids;
50        precedent_fMin = actual_fMin;
51    end
52    % save the final results into file
53    fprintf(fp,'\nFinal result:\n');
54    fprintf(fp,'xInf=%f  xSup=%f  xMin=%f  fMin=%f\n',...
55        xInf,  xSup,  xMin,  fMin);
56    fMin = precedent_fMin;
57    % close the file
58    fclose(fp);
59    %
```

Code 3.3 *gridOptimization.m*.

We used as boundaries of the search interval the results from running the *boundaries.m* program (see Table 3.1). A convergence factor $\varepsilon = 10^{-3}$ was chosen, and the number of subintervals at each iteration was set to 20. Table 3.2 shows step-by-step the performance of the *gridOptimization.m* program. It is easy to notice how the length of the search interval for each iteration has been reduced. Also the *fMin* value associated with each iteration, along with the *xMin* coordinate corresponding to that value, found for the minimum point, can be noted.

Table 3.2 The results of the search process.

xInf	xSup	xMin	fMin
1.410000	3.690000	1.980000	-0.999600
1.866000	2.094000	2.002800	-0.999992
1.991400	2.014200	2.000520	-1.000000
1.999380	2.001660	1.999950	-1.000000
1.999836	2.000064	1.999996	-1.000000

The equation that defines the objective function at line 5 in Code 3.2 can be replaced by required objective function. Also, in Code 3.3 the boundaries of the searching interval (lines 6 and 7) and the convergence criterion (line 9) that defines the desired result accuracy can be changed. The speed of calculations can be influenced by the number of grids chosen at line 11, due to the increasing of the number of objective function evaluations. The same effect of increasing the accuracy can be achieved by

choosing a smaller value for the convergence factor ε. In any case, the principle of the method is very simple and the design of the source code does not involve any difficulties.

3.4 The golden section method

As in the case of the preceding paragraph, we assume that the limits of the interval $[a, b]$ containing the minimum of the function are known in advance. We also admit that the objective function is unimodal and is located in the positive domain of the Ox axis. We choose two points x_1 and x_2 inside the interval $[a, b]$ as in Fig. 3.3.

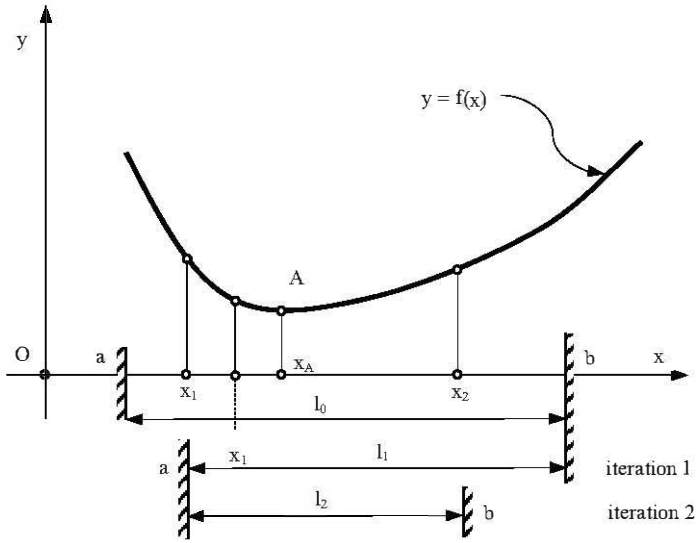

Fig. 3.3 Narrowing the interval containing the minimun point by the golden section method

We calculate the value of the objective function for both points x_1 and x_2. If $f(x_1) > f(x_2)$, given that the function admits a single point in the positive domain of the Ox axis, the function $f(x)$ can pass through the points $(x_1, f(x_1))$ and $(x_2, f(x_2))$ in only two ways, as can be seen in Fig. 3.4. If x^* is the minimum point we are looking for, we may notice that in the first possible situation (see Fig. 3.4.a), the point x_2 is to the right of x^*, while in the second situation (see Fig. 3.4 b), x_2 is to the left of x^*. Therefore,

nothing can be said with certainty about the position of x_2. However, in both cases the point x_1 is located to the left of point x^*. In this way, we can move the left bound of the searching interval containing the minimum point from the old position into point x_1 (see Fig. 3.3).

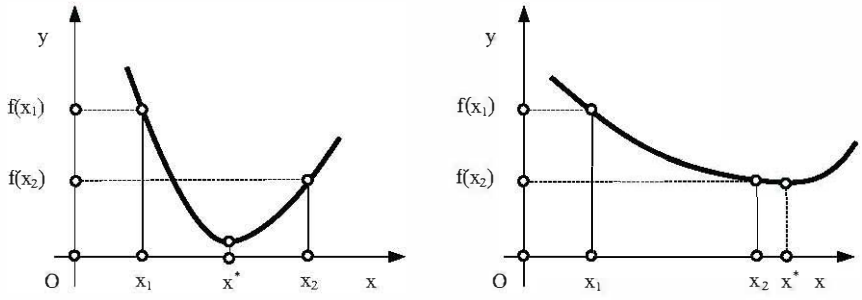

Fig. 3.4 Choosing the new lower bound of the $[a, b]$ interval.

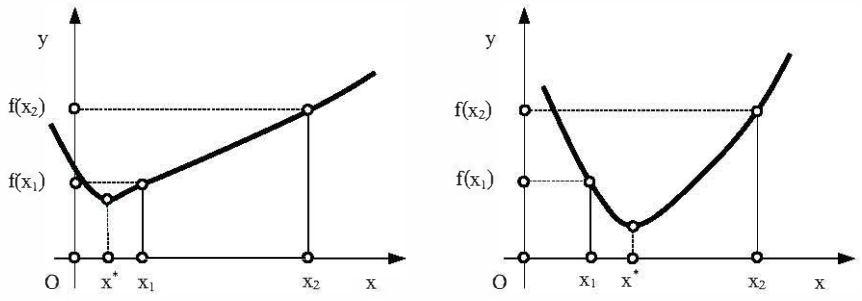

Fig. 3.5 Choosing the new upper bound of the $[a, b]$ interval.

After this maneuver, the length of the search interval was reduced from the initial length value l_0 to length l_1, the moment at which the first iteration can be considered finished. In the new interval $[a, b]$ of length l_1, we consider two more internal points x_1 and x_2, for which we will calculate the value of the objective function. If we get $f(x_1) < f(x_2)$, we will analyze how our function can pass through the points $(x_1, f(x_1))$ and $(x_2, f(x_2))$, respecting the conditions initially imposed. To do this, let us consider the

two possible cases (see Fig. 3.5).

It can be noticed that in the first possible situation (see Fig. 3.5.a), the point x_1 is situated to the right of x^*, while in the second situation (see Fig. 3.5.b), x_1 is found to the left of the point x^*. Therefore, nothing can be said with certainty about the position of x_1. Instead, in both cases point x_2 is to the right of point x^*. In this way, we can move the right bound of the searching interval $[a, b]$ containing the minimum point, from the old position, to point x_2 (see Fig. 3.3). Thereby, the length of the search interval was reduced from the length l_1 to length l_2, the moment at which the second iteration can be considered finished. By proceeding in a similar way, iteration by iteration, will be reduced the length of the interval $[a, b]$ while at the same time keeping the minimum point within its range. The optimization procedure is completed when the length of the interval $[a, b]$ becomes less than a convergence factor ε, initially chosen. Let l_k the length of the $[a_k, b_k]$ interval after iteration k (see Fig. 3.6).

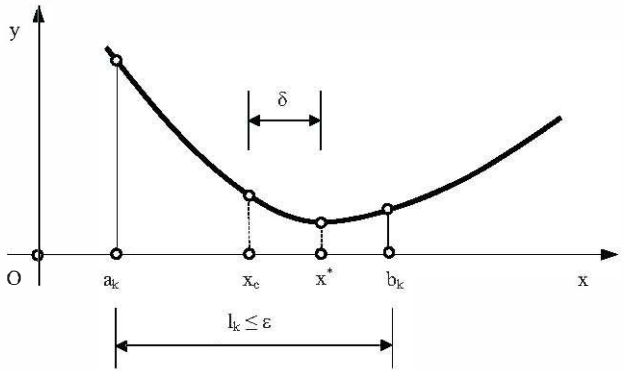

Fig. 3.6 The searching interval after iteration k.

If the length of l_k is less than ε, then we can choose as abscissa of the minimum point the center x_c of the interval $[a_k, b_k]$. Thus:

$$x_c = \frac{a_k + b_k}{2}; \tag{3.5}$$

If the real x^* abscissa of the minimum point is not exactly at the center of this range, which would be only a random one, then the calculated abscissa x_c is located in one half of the $[a_k, b_k]$ interval. Because $l_k \leq \varepsilon$, then the calculation error δ will be smaller even than $\varepsilon/2$. Code 3.4 displays the

source code of the golden section method for finding the extreme of an unimodal objective function.

We will use the parabola defined by Eq. 3.3 as the objective function, considering as bounds of the search interval the results obtained with *boundaries.m*. Running the *goldenSection.m* program will lead us to the results in Table 3.3.

```
%
%  Finding the minimum point of a unimodal function
%      that depends on a single variable, using the
%                  Golden Section method
%
% the convergence criterion
eps = 0.001;
% define the boundaries of the search interval
a = 1.41;
b = 3.69;
% the interval reduction ratio
tau = 0.381966;
% the initial length of the searching interval
intervalLength = b-a;
% open a file to save the results
fp = fopen('results.txt','w');
% print input data to file
fprintf(fp,'      Golden Section Method\n\n');
fprintf(fp,'Input data:\n');
fprintf(fp,'convergence criterion eps = %f\n',eps);
fprintf(fp,'a = %f       b = %f\n',a,b);
fprintf(fp,'tau = %f\n\n',tau);
fprintf(fp,'Intermediary results:\n');
% start the iterative search
i = 1;
while (intervalLength > eps)
    % calculate the inner points of the interval (a, b)
    x1 = (1-tau)*a+tau*b;
    x2 = tau*a+(1-tau)*b;
    f1 = f(x1);
    f2 = f(x2);
    if(f1<f2)
        b = x2;
    else
        a = x1;
    end
    % print the intermediary results
    fprintf(fp,'iteration no. %d:  a = %f  b = %f\n',i,a,b);
```

```
39      intervalLength = b-a;
40      i = i + 1;
41  end
42  % approximate the solution in the middle of the interval
43  xOptim = (a + b)/2;
44  % calculate the optimal value of the objective function
45  fMinim = f(xOptim);
46  % save the final results to file
47  fprintf(fp,'\nFinal results:\n');
48  fprintf(fp,'xOptim = %f\n',xOptim);
49  fprintf(fp,'fMinim = %f\n',fMinim);
51  % close the file
51  fclose (fp);
52  %
```

Code 3.4 *goldenSection.m*.

Table 3.3 Iteration history of *goldenSection.m* search.

iteration	a	b
1	1.410000	2.819118
2	1.410000	2.280883
3	1.742648	2.280883
4	1.742648	2.075295
5	1.869708	2.075295
6	1.948235	2.075295
7	1.948235	2.026762
8	1.978230	2.026762
9	1.978230	2.008225
10	1.989687	2.008225
11	1.996768	2.008225
12	1.996768	2.003848
13	1.996768	2.001144
14	1.998439	2.001144
15	1.999472	2.001144
16	1.999472	2.000505
17	1.999867	2.000505

For the objective function, the Code 3.2 *f.m* was used. The final result is $fMin$ = -1.000 for $xMin$ = 2.000186.

3.5 The Fibonacci method

In the case of the Fibonacci method we start looking for the minimum point assuming that the bounds a_1 and b_1 of the interval containing the

minimum point are known. Then, the initial length of the search interval is:

$$l = b_1 - a_1; \tag{3.6}$$

If F_n is the Fibonacci number that satisfies the property:

$$F_n = F_{n-1} + F_{n-2}, \qquad with\ F_0 = F_1 = 1; \tag{3.7}$$

then we can build the well-known Fibonacci sequence, in which each term is equal to the sum of the two preceding ones. We divide the search interval into F_n equal-size segments of length $(b_1 - a_1)/F_n$.

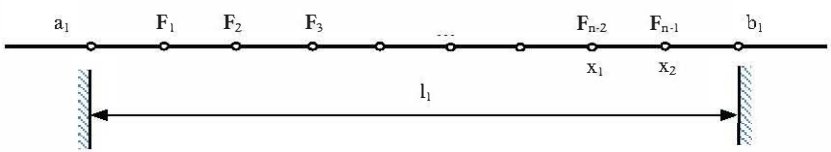

Fig.3.7 Initial search interval, with F_n subdivisions.

The x_i coordinates of the F_i (i = *1, 2, ..., n-1*) boundaries of the segments can be calculated by equation:

$$x_i = a_1 + \frac{i}{F_n}(b_1 - a_1); \qquad i = 1,2,\dots,(n-1). \tag{3.8}$$

Thus, at the first iteration we will take as x_1 and x_2 the points:

$$x_1^1 = a_1 + \frac{F_{n-2}}{F_n}(b_1 - a_1); \tag{3.9}$$

$$x_2^1 = a_1 + \frac{F_{n-1}}{F_n}(b_1 - a_1); \tag{3.10}$$

We can calculate the value of function $f(x)$ at the points x_1 and x_2, according to the Eq. 3.9 and Eq. 3.10, and then refine the search interval thus:

- If $f(x_1^1) \le f(x_2^1)$ we keep the left bound in a_1 and move the right bound from b_1 to x_2^1. This is similar to the procedure we used for the golden section method (see Fig. 3.4 and Fig. 3.5). By this narrowing of the search interval, the new interval, whose bounds

we denote a_2 and b_2, will have the length l_2:

$$l_2 = b_2 - a_2 = a_1 + \frac{F_{n-1}}{F_n}(b_1 - a_1) - a_1 = \frac{F_{n-1}}{F_n}(b_1 - a_1); \tag{3.11}$$

- If, on the contrary, $f(x_1^1) \geq f(x_2^1)$, then we keep the right bound of the search interval in b_1 and move the left bound from a_1 to x_1^1. In this situation, the length of the search interval becomes:

$$l_2 = b_2 - a_2 = a_1 + \frac{F_{n-2}}{F_n}(b_1 - a_1) - a_1 = \frac{F_{n-2}}{F_n}(b_1 - a_1); \tag{3.12}$$

With refinement of the search interval from l_1 to l_2, the first iteration is considered completed and n will be decremented. The analogous process of narrowing the search interval is repeated (n-1) times. The search interval is narrowed according to the relationship:

$$b_{i+1} - a_{i+1} = x_2^i - a_i = b_i - x_1^i = \frac{F_{n-i}}{F_{n-i+1}}(b_i - a_i); \tag{3.13}$$

For this reason, after (n-1) iterations, the length of the search interval that contains the minimum point can be expressed as:

$$l_n = b_n - a_n = \frac{1}{F_n}(b_1 - a_1); \tag{3.14}$$

Because number F_n in the Fibonacci sequence must satisfy the condition:

$$F_n \geq \frac{b_1 - a_1}{\varepsilon}; \tag{3.15}$$

where $(b_n - a_n) \leq \varepsilon$, after (n-1) iterations, the length l_n of the search interval becomes smaller than the chosen precision factor. For this reason, choosing the minimum point at the center of the search interval $[a_n, b_n]$ gives a solution that meets the chosen precision. Code 3.5 shows the *fibonacci.m* program.

```
%
%                  Fibonacci method
%  Finding the minimum point for an unimodal function
%      that depends on a single variable
% define the boundaries of the search interval
%
a = 1.41;
b = 3.69;
% define the convergence criterion 'eps'
eps = 1e-3;
% calculate the number 'n' of array elements
k = (b-a)/eps;
% build the Fibonacci array fib(n)
f1 = 1;
f2 = 1;
s = f1 + f2;
n = 2;
%
while (s<=k)
        s = f1 + f2;
        f1 = f2;
        f2 = s;
        n = n + 1;
end
%
% open a file to save results
fp = fopen('results.txt','w');
%
fprintf(fp,'     Fibonacci method\n\n');
fprintf(fp,' Input data:\n');
fprintf(fp,'a=%f   b=%f   n=%d   eps=%f\n',...
    a, b, n, eps);
fprintf(fp,'\n');
fib = zeros(1,n);
%
fib(1) = 1;
fib(2) = 1;
for i = 3:1:n
    fib(i) = fib(i-2)+fib(i-1);
end
%
x1 = a + (fib(n-2)/fib(n))*(b-a);
x2 = a + (fib(n-1)/fib(n))*(b-a);
initialLength = b-a;
%
```

```
% iteration index
k = 0;
%
% print intermediary results to file
fprintf(fp,' Intermediary results\n');
%
while(n > 2)
    f1 = f(x1);
    f2 = f(x2);
    if(f1<f2)
        b = x2;
        x2 = x1;
        x1 = a + (fib(n-2)/fib(n))*(b-a);
        n = n-1;
    elseif (f1>f2)
        a = x1;
        x1 = x2;
        x2 = a + (fib(n-1)/fib(n))*(b-a);
        n = n - 1;
    end
    fprintf(fp,'iteration no.%2d   a=%f  b=%f \n',...
        (k+1), a, b);
    initialLength = b-a;
    k = k + 1;
end
%
xMin = (a+b)/2;
yMin = f(xMin);
%
% print the final results to file
fprintf(fp,'\n');
fprintf(fp,' Final results:\nxMin=%f   fMin=%f\n',...
    xMin,  yMin);
% close the file
fclose (fp);
%
```

Code 3.5 *fibonacci.m*.

Let us consider the same optimization problem as for the golden section optimization method in the previous paragraph, with the same input parameters. It means that the search interval $[a,b]$ is set to [1.41,3.69], while the convergence criterion ε = 1e-3. Table 3.4 shows how the initial search interval evolved during the reduction process.

Table 3.4 Iteration history of *fibonacci.m* search.

iteration	a	b
0	1.410000	3.690000
1	1.410000	2.819118
2	1.410000	2.280882
3	1.742648	2.280882
4	1.742648	2.075295
5	1.869708	2.075295
6	1.948235	2.075295
7	1.948235	2.026761
8	1.978228	2.026761
9	1.978228	2.008220
10	1.989680	2.008220
11	1.996769	2.008220
12	1.996769	2.003858
13	1.996769	2.001131
14	1.998405	2.001131
15	1.999495	2.001131
16	1.999495	2.000586

From Table 3.4 one can note that the reduction rate of the interval which includes the solution is basically the same as the reduction ratio achieved through the golden section method.

3.6 Quadratic approximation

Sometimes, in order to make things easier, one can use instead of a nonlinear objective function, an approximation of it with a second degree parabola in the form of:

$$p(x) = ax^2 + bx + c; \tag{3.16}$$

To make this approximation possible, it is necessary to know the values of f_1 and f_2 of the objective function in two points x_1 and x_2, as well as the value f_1' of the first order derivative of the objective function at the point x_1. Thus, we will have the system:

$$f_1 = ax_1^2 + bx_1 + c; \tag{3.17}$$

$$f_2 = ax_2^2 + bx_2 + c; \tag{3.18}$$

$$f_1' = 2ax_1 + b; \tag{3.19}$$

Solving the system of Eq. 3.17 to 3.19, will lead to the following equations necessary to calculate the coefficients *a, b, c* of the parabola in Eq. 3.16:

$$a = \frac{f_1' \cdot (x_1 + x_2) - (f_1 - f_2)}{(x_1 + x_2)^2}; \tag{3.20}$$

$$b = f_1' - 2ax_1; \tag{3.21}$$

$$c = f_2 - ax_2^2 - bx_2; \tag{3.22}$$

To use the program *quadraticOpt.m* in Code 3.6, the calls for two functions *f.m* in Code 3.7 and *fprim.m* in Code 3.8 need to be performed.

```
%
%     The approximation of a nonlinear function
%         with a second degree parabola
%
% the program makes two calls: one to the function
% f(x) and a second one to its first derivative.
% it is assumed that two points and the values of
% the function f(x) in these points are known.
%
% open a text file to store the results
fp = fopen('results.txt','w');
fprintf(fp,' The approximation of a nonlinear
function\n');
fprintf(fp,'  with a second degree parabola\n\n');
%
fprintf(fp,'the function to be approximated is:\n');
fprintf(fp,'     f(x) = x+5*sin(x);\n');
fprintf(fp,'\n Input data:\n');
fprintf(fp,'x1 = %6.3f\nx2 = %6.3f\n',x1,x2);
x1 = -3.5;
x2 = 0.5;
f1 = f(x1);
f2 = f(x2);
fprintf(fp,'f(x1) = %6.3f\nf(x2) = %6.3f\n',f1,f2);
% f1prim is the first derivative of f in x1
f1prim = fprim(x1);
% we calculate the coefficients a, b, c
% of the second degree parabola
%
```

```
%       f(x) = a*x^2 + b*x + c;
%
fprintf(fp,'\nthe parabola that approximate f(x):\n');
fprintf(fp,'   f(x) = a*x^2 + b*x + c;\n\n');
a = (f1prim*(x1+x2)-(f1-f2))/((x1 + x2)^2);
b = f1prim-2*a*x1;
c = f2 - a*x2^2 - b*x2;
fprintf(fp,'a = %6.3f\nb = %6.3f\nc = %6.3f\n',...
    a,b,c);
% store the values of the original function
% in a vector called 'fnc'
fnc = zeros(1,40);
i = 1;
for x = x1:0.1:x2
    fnc(1,i) = f(x);
    i = i + 1;
end
% plot the fnc function
plot(fnc,'LineWidth',2);
% store the values of the second degree
% parabola in the vector fnc2
fnc2 = zeros(1,40);
i = 1;
for x = x1:0.1:x2
    fnc2(1,i) = a*x^2 + b*x + c;
    i = i + 1;
end
% plot the approximation function 'fncApp'
hold on
plot(fnc2,'color','black','LineWidth',2);
% coordinate x of the minimum point
xMin = -b/(2*a);
% the minimum point
delta = b^2-4*a*c;
yMin = -delta/(4*a);
fprintf(fp,'\n the minimum of the parabola is:\n');
fprintf(fp,'xMin = %6.3f\n',xMin);
fprintf(fp,'fMin = %6.3f\n',yMin);
% close the file
fclose(fp);
%
```

Code 3.6 *quadraticOpt.m*.

```
function [val] = f(x)
% the evaluation of function f in a point x
% f(x) represents the objective function
val = x+5*sin(x);
return;
end
%
```

Code 3.7 *f.m*.

```
function [val] = fprim(x)
% the evaluation of function fprim in a point x
% fprim represents the first derivative of f(x)
val = 1+5*cos(x);
return;
end
%
```

Code 3.8 *fprim.m*.

Let us consider the expression of objective function listed in Code 3.7. Running the *quadraticOpt.m* program, the following values for the parabola coefficients are obtained:

$$a = 1.743 \qquad b = 8.521 \qquad c = -1.799$$

Solving the Eq. 3.16 with the coefficients above leads to the minimum *fMin* = -12.211612, at point *xMin* = -2.443900.

3.7 Cubic approximation

If a better precision than that obtained by approximating the objective function by a second degree parabola is needed, a nearer approximation can be applied. By virtue of this, the non-linear objective function *f(x)* can be approximated by a third degree polynomial of the form:

$$p(x) = ax^3 + bx^2 + cx + d; \tag{3.23}$$

For this, it is necessary to know the values f_1, f_2 and f_3 of the objective function *f(x)* at the points x_1, x_2, x_3, as well as the value of the derivative f_1' of *f(x)* at x_1. For determining the coefficients *a*, *b*, *c*, *d* of the polynomial (see Eq. 3.23), we have to solve the system of four equations with four unknowns:

$$f_1 = ax_1^3 + bx_1^2 + cx_1 + d; \tag{3.24}$$

$$f_2 = ax_2^3 + bx_2^2 + cx_2 + d; \tag{3.25}$$

$$f_3 = ax_3^3 + bx_3^2 + cx_3 + d; \tag{3.26}$$

$$f_1' = 3ax_1^2 + 2bx_1 + c; \tag{3.27}$$

Solving system made by Eq. 3.24 to 3.27 leads us to the following computational relationships:

$$a = \frac{f_3 - f_1}{(x_3 - x_2)(x_3 - x_1)^2} - \frac{f_2 - f_1}{(x_3 - x_2)(x_2 - x_1)^2} + \frac{f_1'}{(x_2 - x_1)(x_3 - x_1)}; \tag{3.28}$$

$$b = \frac{f_2 - f_1}{(x_2 - x_1)^2} - \frac{f_1'}{x_2 - x_1} - a(2x_1 + x_2); \tag{3.29}$$

$$c = f_1' - 3ax_1^2 - 2bx_1; \tag{3.30}$$

$$d = f_1 - ax_1^3 - bx_1^2 - cx_1; \tag{3.31}$$

To determine the optimal point, we must determine the real solutions of the first-order derivative of the polynomial Eq. 3.23 given by a second degree equation (see Eq. 3.16). If the equation discriminant is positive, then the second degree equation admits two real roots, and from among these, we will pick up the actual root within the search interval of the extreme point. If the discriminant of the equation is null, then we will have a real double root. The character of this point (minimum or maximum), is decided by the sign of the second order derivative at this point.

Let us consider again the same objective function listed in Code 3.7. It is a nonlinear function that will be approximated by a third degree polynomial of the form of Eq. 3.23. To do this, the *cubicOpt.m* program in Code 3.9 is used.

```
1  %
2  %              Cubic Optimization
3  % the approximation of a non-linear function
4  % with a third degree polynomial and the
5  % determination of the optimal point
```

```
%
% open a text file to save results
fp = fopen('results.txt','w');
fprintf(fp,'              Cubic Optimization\n');
fprintf(fp,'the approximation of a non-linear
function\n');
fprintf(fp,'with a third degree polynomial and
the\n');
fprintf(fp,'determination of the optimal point\n\n');
fprintf(fp,'   Input data:\n');
% there are three known points and the values
% of f(x) at these points
x1 = -3.5;
x2 = 0.5;
x3 = -2.0;
fprintf(fp,'x1 = %6.3f\nx2 = %6.3f\nx3 = %6.3f\n',...
    x1,x2,x3);
f1 = f(x1);
f2 = f(x2);
f3 = f(x3);
fprintf(fp,'f1 = %6.3f\nf2 = %6.3f\nf3 = %6.3f\n',...
    f1,f2,f3);
% f1prim is the derivative of f in x1
f1prim = fprim(x1);
% store the values of the original function in
% a vector called 'fnc'
fnc = zeros(1,40);
i = 1;
for x = x1:0.1:x2
    fnc(1,i) = f(x);
    i = i + 1;
end
% calculate the coefficients of
% the third degree polynomial
a = (f3-f1)/((x3-x1)^2*(x3-x2))-(f2-f1)/((x2-...
    x1)^2*(x3-x2))+f1prim/((x2-x1)*(x3-x1));
b = (f2-f1)/(x2-x1)^2-f1prim/(x2-x1)-a*(2*x1+x2);
c = f1prim-3*a*x1^2-2*b*x1;
d = f1-a*x1^3-b*x1^2-c*x1;
% form the third degree polynomial
fnc3 = zeros(1,40);
i = 1;
for x = x1:0.1:x2
    fnc3(1,i) = a*x^3 + b*x^2 + c*x + d;
    i = i + 1;
```

```
end
% calculating the discriminant of derivative of fnc3
discrim = 4*b^2-12*a*c;
switch discrim
    case 0
        x1prim = -b/(3*a);
        x2prim = x1prim;
    otherwise
        x1prim = (-2*b-sqrt(discrim))/(6*a);
        x2prim = (-2*b+sqrt(discrim))/(6*a);
end
%
if (x1<x1prim&&x1prim<x2)
    xExtrem = x1prim;
end
if (x1<x2prim&&x2prim<x2)
    xExtrem = x2prim;
end
% we analyze the sign of the second order
% derivative of polynomial fnc3:
%    fnc3sec = 6*a*x+2*b
if (6*a*xExtrem+2*b) > 0
    optim = ('xExtrem is the minimum point');
else
    optim = ('xExtrem is the maximum point');
end
% the abscissa of the optimum point
xMin = xExtrem;
% the optimal value of the function in xOptim
fMin = f(xMin);
%save results to file
fprintf(fp,'xMin = %9.6f\nfMin = %9.6f\n',...
    xMin,fMin);
% draw the original function
plot(fnc,'LineWidth',2);
% plot the third degree polynomial
hold on;
plot(fnc3,'color','red','LineWidth',2);
hold off;
% close the file
fclose(fp);
%
```

Code 3.9 *cubicOpt.m*.

To run the program *cubicOpt.m* in Code 3.9, the first derivative of the objective function listed in Code 3.8 must also be used. Table 3.5 contain the results of the optimum point that belongs to the objective function defined in Code 3.7, solved with *gridOptimization.m*, *goldenSection.m*, *fibonacci.m*, *quadraticOpt.m* and *cubicOpt.m*.

Table 3.5 The optimum sulution obtained by different methods.

method	xMin	fMin
gridOptimization.m	-1.772150	-6.671134
goldenSection.m	-1.772174	-6.671134
fibonacci.m	-1.772246	-6.671134
quadraticOpt.m	-2.443900	-12.211612
cubicOpt.m	-1.433025	-6.385647

In the case of using one of the three optimization methods grid, golden section or fibonacci, the results are accurate within the limits of the convergence factor ε = 1e-3. It can be noted that in the case of quadratic optimization, the accuracy is quite low, but, if the degree of the polynomial approximation is increased to three, the result is considerably optimized.

3.8 The minimum of a single variable constrained function

In different applications, the mathematical model of the optimization problem includes some constraints. While the objective function depends on a single variable, these constraints must be of inequality type only. The existence of a single equality type constraint only leads to trivializing the optimization problem. Taking into account this observation, the mathematical model of the optimization problem can be written as follows:

$$Minimize\ f(x); \tag{3.32a}$$

$$Subject\ to:$$

$$g_j(x) \leq 0; \qquad j = 1,2,\ldots,m. \tag{3.33b}$$

In this case, the minimum point of the objective function defined by the above model must satisfy all the inequality constraints. For this reason, we will reformulate the objective function in the following way:

$$\emptyset(x, r_p) = f(x) + \delta \cdot r_p \cdot \sum_{j=1}^{m} g_j^2(x); \tag{3.34}$$

Here $O(x,r_p)$ is called pseudo-objective function, δ is a scalar that takes the value zero if all constraints (see Eq. 3.32b) are satisfied and one if at least one of the constraints set is not satisfied, and r_p is also a scalar called penalty parameter. This parameter takes positive values that are increased at each iteration. The expression $\delta \cdot r_p \cdot \sum_{j=1}^{m} g_j^2(x)$ is called penalty function and takes a positive value if we take into account that under the sum operator we always have a positive quantity. Also, we have to remark that the graph of the pseudo-objective function (see Eq. 3.33) will overlap with the graph of the objective function inside the admissible region, defined by the problem constraints. Outside the admissible region the graph of the pseudo-objective function will lie above the graph of the objective function.

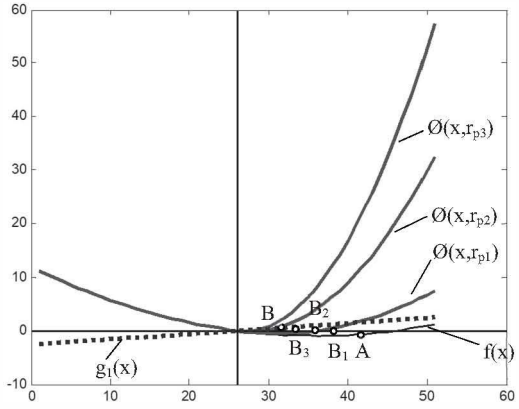

Fig. 3.8 The effect of the increase of the penalty parameter, on the position of the optimal solution with constraints.

As long the penalty parameter grows, the optimal solution of the pseudo-objective function (considered as a function without constraints), moves toward the true optimal solution of the objective function with constraints (see Fig. 3.8). Point A from Fig. 3.8 is the optimum without constraints, while point B is the optimum with constraints. In this case we took into consideration just a single inequality constraint $g_1(x) \leq x\text{-}1$. The frontier of the admissible region is stretched toward the left side of point $x = 1$ (that

is, the point where the graph of $g_1(x)$ crosses the Ox axis). Any x value located in the interval $(-\infty, 1]$, respects the constraint. The optimum with constraints is the result of the intersection between the frontier of the admissible region and the graph of the objective function $f(x)$, expressed by Eq. 3.34. If we take a penalty parameter $r_{p1} = 1$, we get the graph of the function $O_1(x, r_{p1})$. If we now apply an optimization method without constraints to the pseudo-objective function $O_1(x, r_{p1})$, taken as a function without constraints, we will get as solution the point B_1. Increasing the value of the penalty parameter, the new pseudo-objective function $O_2(x, r_{p2})$ will have as solution the point B_2, nearer to B than B_1. Now, if we continue to increase the penalty parameter we will get optimal solutions B_i nearer the real optimum with constraints. It can be noted that:

$$\lim_{r_p \to \infty} B_i = B; \tag{3.35}$$

However one cannot reach infinite values of the penalty parameter since there are limitations to the magnitude of representation of numbers in the internal memory of the computer. This is why one should establish in advance the precision factor ε as a convergence criterion, and when the optimum achieve this precision, the computer program stops.

The program that solves this type of problems is listed in Code 3.10. As example the objective function from Eq. 3.3 is taken, and as inequality constraint:

$$g_1(x) = x - 1 \leq 0; \tag{3.36}$$

```
%
%              Constrained Optimization
%     single variable constrained optimization
%
% Input data:
% the convergence criterion
eps = 0.001;
% convergence criterion used between iterations
leftBound = -1.5;
rightBound = 3.7;
a = leftBound;
b = rightBound;
% interval reduction ratio
tau = 0.381966;
% the penalty factor increment
```

```
increment = 5;
rp = 0;
% open a file to save the results
fp = fopen('results.txt','w');
%
fprintf(fp,'      Golden Section Method\n\n');
fprintf(fp,'Input data:\n');
fprintf(fp,'convergence criterion eps = %f\n',eps);
fprintf(fp,'leftBound = %f       rightBound =
%f\n',a,b);
fprintf(fp,'reduction factor     tau = %f\n\n',tau);
%
[xOptim, fMinim] = goldenSectionFunc(a,b,eps,tau,rp);
%
fprintf(fp,'\nOptimum without constraints:\n');
fprintf(fp,'xOptim = %f\n',xOptim);
fprintf(fp,'fMinim = %f\n',fMinim);
%
fMinimPrecedent = fMinim;
rp = rp + increment;
a = leftBound;
b = rightBound;
%
[xOptim, fMinim] = goldenSectionFunc(a,b,eps,tau,rp);
%
fprintf(fp,'\nOptimum with constraints:\n');
fprintf(fp,'rp = %f\n',rp);
fprintf(fp,'xOptim = %f\n',xOptim);
%
deltaF = fMinim - fMinimPrecedent;
fprintf(fp,'fMin = %f       ',fMinim);
fprintf(fp,'fMinPrecedent = %f  deltaF = %f\n\n',...
    fMinimPrecedent, deltaF);
%
while deltaF>eps
    rp = rp + increment;
    a = leftBound;
    b = rightBound;
    fMinimPrecedent = fMinim;
    %
    [xOptim,fMinim] = ...
        goldenSectionFunc(a,b,eps,tau,rp);
    %
    fprintf(fp,'rp = %f\n',rp);
    fprintf(fp,'xOptim = %f\n',xOptim);
```

```
61        deltaF = fMinim - fMinimPrecedent;
62        fprintf(fp,'fMinim = %f     ',fMinim);
63        fprintf(fp,'fMinPrec = %f   ',fMinimPrecedent);
64        fprintf(fp,'deltaF = %f\n\n',deltaF);
65    end
66    % close the file "results.txt"
67    fclose (fp);
68    %
```

Code 3.10 *singleVarConstrainedOpt.m.*

In Code 3.10, at line 27 a call to a function that returns the minimum point of the pseudo-objective function is made. This function is built on the frame of the golden section optimization method (see Code 3.11) and requires as actual parameters the boundaries of the search interval $[a,b]$, the convergence parameter ε, the reduction rate of the search interval τ and the penalty parameter r_p.

```
1    function [xOptim,fMinim]=...
2          goldenSectionFunc(a,b,eps,tau,rp)
3    % This is a function built on the frame of
4    % the Golden Section Optimization method
5    %
6    % this function needs as actual parameters:
7    % 1) left limit of the search interval 'a'
8    % 2) right limit of the search interval 'b'
9    % 3) the convergence parameter 'eps'
10   % 4) the reduction rate 'tau'
11   % 5) the penalty parameter 'rp'
12   %
13   intervalLength = b-a;
14   while (intervalLength > eps)
15       x1 = (1-tau)*a+tau*b;
16       x2 = tau*a+(1-tau)*b;
17      if g(x1)<=0
18           f1 = f(x1);
19       else
20           f1 = f(x1)+rp*g(x1)*g(x1);
21       end
22       if g(x1)<=0
23           f2 = f(x2);
24       else
25           f2 = f(x2)+rp*g(x2)*g(x2);
26       end
27       if(f1<f2)
28           b = x2;
```

```
29        else
30            a = x1;
31        end
32        intervalLength = b-a;
33    end
34    xOptim = (a + b)/2;
35    fMinim = f(xOptim);
36    return;
37    %
```

Code 3.11 *goldenSectionFunc.m*.

The function in Code 3.11 always returns the actual value of the minimum point for a specific value of the penalty parameter. This value is compared with the previous minimum point and if the absolute difference between these two points *deltaF* satisfy the convergence criterion, then the program stops. To run the program in Code 3.10, beside *goldenSectionFunc.m* in Code 3.11, it is necessary to have the functions *f.m* from Code 3.2, and *g1.m* in Code 3.12.

```
1    function [val] = g1(x)
2    % the evaluation of function g1 in a point
3    %
4    val = x-1;
5    return;
6    end
7    %
```

Code 3.12 *g1.m*.

Table 3.6 shows the iteration history for solving the mathematical model that has as objective function from Eq. 3.3, with an inequality constraint in Eq. 3.36. The search interval was set to [-1.5,3.7] and the convergence criterion to ε = 1e-3. The minimum point without constraint is *xMin* = 2.0 and *fMin* = -1.0. This value of *fMin* without constraint is considered as *fMinPrecedent*.

Table 3.6 History iterations for constrained optimization.

rp	xMin	fMin	fMinPrecedent	deltaF
5	1.166609	-0.305460	-1.000000	0.694540
10	1.090919	-0.173572	-0.305460	0.131888
15	1.062433	-0.120968	-0.173572	0.052604
20	1.047740	-0.093201	-0.120968	0.027767
25	1.038315	-0.075163	-0.093201	0.018038
30	1.032147	-0.063260	-0.075163	0.011902
35	1.027778	-0.054785	-0.063260	0.008475

40	1.024522	-0.048443	-0.054785	0.006342
45	1.021610	-0.042753	-0.048443	0.005690
50	1.019810	-0.039228	-0.042753	0.003525
55	1.017798	-0.035279	-0.039228	0.003949
60	1.016341	-0.032416	-0.035279	0.002863
65	1.014885	-0.029549	-0.032416	0.002867
70	1.013985	-0.027775	-0.029549	0.001774
75	1.013085	-0.026000	-0.027775	0.001776
80	1.012185	-0.024222	-0.026000	0.001777
85	1.011629	-0.023123	-0.024222	0.001099
90	1.011073	-0.022023	-0.023123	0.001100
95	1.010173	-0.020243	-0.022023	0.001781
100	1.009829	-0.019562	-0.020243	0.000681

To draw the graph in Fig. 3.8, Code 3.13 may be used. For the objective function in Code 3.2 and inequality constraint in Code 3.12, the boundaries at lines 5 and 6 are defined. The initial value of the penalty parameter was established as $r_p = 1$ at line 23, and the increment for r_p is set to 4 (see line10).

```
1   %
2   %     Mathematical model graphical display
3   %
4   % define the boundaries
5   leftBound = -1.5;
6   rightBound = 3.5;
7   % increment for graphical display
8   increment = 0.1;
9   % increment for penalty function
10  incrementRp = 4;
11  % number of function values
12  n = floor((rightBound - leftBound)/increment);
13  func = zeros(1,n);
14  j = 1;
15  for i = leftBound:increment:rightBound
16      func(1,j) = f(i);
17      j = j+1;
18  end
19  % plot the original function
20  plot (func, 'color','black','linewidth',2);
21  hold on;
22  % the penalty factor rp
23  rp = 1;
24  for m = 1:3
25      pseudoFunc = zeros(1,n);
```

```
26      j = 1;
27      for i = leftBound:increment:rightBound
28          if g1(i)<=0
29              pseudoFunc(1,j) = f(i);
30          else
31              pseudoFunc(1,j) = f(i) + rp*(g1(i))^2;
32          end
33          j = j+1;
34      end
35      % plot pseudoFunc
36      plot (pseudoFunc, 'color', 'red');
37      % increase  the penalty factor rp
38      rp = rp + incrementRp;
39  end
40  % draw Ox axis
41  line([0 60],[0 0], 'color','black');
42  % draw Oy axis
43  line([26 26],[-10 60], 'color','black');
44  % draw the constraint 'g(x)' graphic
45  n = floor((rightBound - leftBound)/increment);
46  constraint_g = zeros(1,n);
47  j = 1;
48  for i = leftBound:increment:rightBound
49      constraint_g(1,j) = g1(i);
50      j = j+1;
51  end
52  % plot the original function
53  plot (constraint_g);
54  hold off;
55  %
```

Code 3.13 *constrainedFunctionGraph.m*.

3.9 Conclusions

The optimization methods presented in this chapter refer to functions dependent on a single variable, without constraints. At first glance, it seems a narrow field, considering that most of the practical problems refer to functions dependent on several variables, with constraints. Moreover, many of the methods start from some initial assumptions, which seem to further narrow their scope of applicability. However, many of the initial assumptions do not restrict the applicability of the methods. What makes these methods important is that they are often part of multi-dimensional optimization methods with restrictions. By various mathematical

transformations, the determination of an optimal point of a multi-variable dependent function with restrictions can be reduced to a recursive unidimensional, unrestricted optimization sequence. For this reason, unidimensional optimization methods without constraints, represent basically the fundamental element of solving the vast majority of optimization problems.

4.

OPTIMIZATION TECHNIQUES FOR N VARIABLES UNCONSTRAINED FUNCTIONS

4.1 Introduction

In many situations, the question is of determining the minimum of a function that depends on several variables, without having additional constraints. In this case, the optimization problem is simply written as:

$$\begin{aligned} &Minimize\ F(X);\\ &where\ X = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}. \end{aligned} \tag{4.1}$$

From a theoretical point of view, in order to determine the solution of the optimization problem given by Eq. 4.1, we have to solve the equation:

$$\nabla F(X) = 0; \tag{4.2}$$

where the gradient of the objective function is represented by the vector of the first order partial derivatives of *F(X)*. From a practical point of view, it is often very difficult to solve the system (4.2) directly, because the gradient of *F(x)* may have strong non-linear components. For this reason, a numerical optimization procedure is most often recommended. For this, a start point is chosen randomly and then, in a recursive manner, a numerical procedure is applied. This procedure will determine the desired solution, within the limits of a pre-determined precision.

According to the numerical technique applied, optimization methods can be classified into three groups: methods of the orders zero, one and two. To determine the optimal solution, by the zero order methods, only the values of the function at certain points, chosen according to a specific strategy, are calculated and analyzed. By first order methods, the

information given by the gradient of the objective function is used to determine the minimization direction. It is known that the gradient of a function is a vector indicating the direction of maximum increase of the objective function. Therefore, to find the minimum point, it will go in the opposite direction to that indicated by the gradient vector. To determine the optimal solution, the second-order optimization methods also use the information given by the *Hessian* matrix (the matrix whose elements are represented by the second order partial derivatives of the objective function).

4.2 The random search method

Although it is considered to be one of the most simple methods of multidimensional optimization, both as a working principle and as a difficulty in algorithmic implementation, the random search method is rather a method of blind search. It can be used when computer time is not the most important criterion. To implement the method we need a pseudo-random number generator. Fortunately, such pseudo-random number generators with uniform distribution are implemented on most compilers. In order to limit the search procedure to a confined space, we can impose on the decision variables certain limits of the form:

$$a_i \leq x_i \leq b_i; \qquad i = 1,2, \ldots, n. \tag{4.3}$$

The Matlab routine function *rand* generates a subunit random number at each call and let us call this number r. While the search must be done in the interval (a_i,b_i) for each variable x_i, we want the random number to be generated within this range. For this reason, the following translation must be made:

$$x_i = a_i + r \cdot (b_i - a_i); \tag{4.4}$$

Basically, we randomly generate a set of values for each variable and calculate the value of the objective function at that point. We initialize the minimum value function *fMin* with a high value, and each time we find a lower value than the current value of *fMin* for the objective function, we update both the value of *fMin* with the new value and the coordinates of the point where this value was found. This search method is very general and many of the randomly generated points are not useful. However, initiating a local search procedure in the neighbourhood of an actual value

of *fMin* is not recommended as long as there is a risk of reaching a local optimum point.

```
%       Random search method
%
fp = fopen( 'randomSearch.txt', 'w' );
fprintf(fp, '\n      Random search method:\n\n');
% the number of decision variables
nVar = 2;
% the number of random generated points
n = 1000000;
% the convergence factor
epsilon = 1e-3;
% pre-allocation of vectors a and b
for i = 1:nVar
    a = zeros(1,nVar);
    b = zeros(1,nVar);
end
% set-up of the search limits
for i=1:nVar
    a(i)=-10.0;
    b(i)=10.0;
end
fMin = 1e6;     % initialize fMin
fPrecedent = fMin;
% global search procedure
for i=1:n
    % random generation: variable x1
    x1 = rand;
    x1 = a(1)+x1*(b(1)-a(1));
    % random generation: variable x2
    x2 = rand;
    x2 = a(2)+x2*(b(2)-a(2));
    % calculate the objective function
    func = f(x1,x2);
    if (func<fMin)
        fMin = func;
        x1Min = x1;
        x2Min = x2;
        % printing intermediate results
        fprintf(fp,'i=%d  x1=%f  x2=%f  fmin=%f\n',...
           i, x1, x2, fMin);
        if abs(fMin - fPrecedent)<=epsilon
            break;
        else
```

```
43                fPrecedent = fMin;
44            end
45        end
46    end
47    % printing final result
48    fprintf(fp, '\n      FINAL RESULT:\n\n');
49    fprintf(fp, 'x1Min=%f\n x2Min=%f\n\n fMin=%f\n\n',...
50       x1Min, x2Min, fMin);
51    fclose(fp);
52    %
```

Code 4.1 *randomSearch.m.*

Both the total number n of randomly generated points, the number of decision variables *nVar* and the search domain limits (a_i, b_i) must be entered by the user. The expression of the objective function must be entered within the function *f.m*. The source code for this function can be seen in Code 4.2.

```
1    % the evaluation of function f in a point
2    function [val] = f(x,y)
3    % insert your objective function here:
4    val = 2*x+y+(x.^2-y.^2)+(x-y.^2).^2;
5    end
6    %
```

Code 4.2 *f.m.*

The search efficiency depends on the number n of randomly generated points within the search domain (a_i, b_i).

Table 4.1 Iteration history of the *Random search.*

Iteration	x1	x2	fMin
1	0.910496	0.655198	3.107471
6	-0.636397	-0.093225	-0.553570
378	-0.491392	-0.950567	-0.649522
1439	0.039340	-1.136716	-0.779146
1623	-0.124084	-0.911013	-1.063558
6292	-0.163118	-0.754482	-1.087001
17162	-0.157280	-0.803144	-1.094289
136094	-0.203102	-0.797031	-1.094395

Table 4.1 shows the way the minimization procedure evolves within the search interval. One can be notice that although the total number of random points was chosen at $n = 10^6$, the program performed only 136094

iterations, until the precision ε = 0.001, initially imposed, was reached. However, search efficiency is low and this appears more evident with the increase in the number of decision variables and, respectively, with the increasing complexity of the objective function evaluations. The last line in Table 4.1 shows that the optimal value of the objective function is *fMin* = -1.0943995. The point where the objective function takes this optimal value has the coordinates x_1 = -0.203102 and x_2 = -0.797031.

If the optimization problem has three or more decision variables, then at line 31 in *randomSearch.m* from Code 4.1, one or more new blocks of instructions must be inserted, as:

```
% random generation: variable x3
x3 = rand;
x3 = a(3)+x3*(b(3)-a(3));
```

Then, the call to the objective function – now at line 32, must be updated, as:

```
func = f(x1,x2,x3);
```

The *if* conditional block from lines 33 to 45 must be also updated, while the number of decision variables is increased. A new line of the form below must be added now after line 36:

```
x3Min = x3;
```

The *printf* instruction, now at line 38, should be updated as:

```
fprintf(fp,'i=%d x1=%f x2=%f x3=%f fmin=%f\n', i,x1,x2,x3,
fMin);
```

The expression of the objective function within the *f.m* function (see Code 4.2) should not be forgotten.

4.3 The random path method

To find the solution to the minimization problem, the random path method uses an iterative relationship of the form:

$$X^{i+1} = X^i + \alpha \cdot S^i \tag{4.5}$$

where i is an iterative index, X is the vector of the decision variables, α is a scalar size, called leap factor in the S direction, and S is the vector of the minimization direction. The search procedure starts from a randomly chosen point. Whatever this start point is, we have to reach the same solution. The coordinates s_i of the minimization direction vector S, where i is the number of decision variables, are randomly chosen using the *rand* function. From X^0, the coordinates of the next point X^1 are calculated with Eq. 4.5. We choose a small initial value for the leap factor, and if at point X^1 so calculated with Eq. 4.5 the current value of the objective function is lower than the previous value (the one at the starting point), the leap factor is incremented. This procedure is repeated as long as the leap factor is increasing and, the current value of the objective function is still less than its previous value. In other words, it goes until the minimum value in that direction is reached.

From point X^1, a new random direction S^1 is chosen and, if this direction leads to smaller values of the objective function than its value in point X^1 (now for us X^1 is the previous point), then the leap factor is iteratively incremented until the objective function reaches the minimum value in this direction. The point where the function has a minimum value in the S^1 direction is X^2. The program calculates the absolute difference of the values of the objective function at these points and, if it becomes smaller than a convergence factor initially chosen, the program stops. However, the program stops only if the convergence condition is maintained a prescribed number of times $nMax$, to make sure that there no local optimum point is reached.

```
%       The random path method
%
n = 0;
nMax = 5;
% the coordinates of the start point X0
x1zero = 3.0;
x2zero = 3.5;
% the convergence factor
epsilon = 1e-5;
% innitial value of the leap factor
alfa0 = 0.001;
% the range for S
a = -10.0;
b = 10.0;
% the function value at the start point
F0 = f(x1zero,x2zero);
Fprecedent = F0;
```

```
18    f0 = F0;
19    % open the file "randomPath.txt"
20    fp = fopen( 'randomPath.txt', 'w' );
21    %
22    % generate a random direction S
23    s1 = rand;
24    s2 = rand;
25    alfa = alfa0;
26    increment = alfa0;
27    % the coordinates of X1
28    x1one = x1zero + alfa*s1;
29    x2one = x2zero + alfa*s2;
30    % the objective function value in X1
31    F1 = f(x1one,x2one);
32    Factual = F1;
33    % initialize the counter i
34    i = 1;
35    % variable 'go' remains 'true' as long as
36    % the convergence criteria are not fulfilled
37    %
38    go = true;
39    while go
40        while (Factual>=Fprecedent)
41            % generate a random direction S
42            s1 = rand;  s1 = a + s1*(b-a);
43            s2 = rand;  s2 = a + s2*(b-a);
44            % the coordinates of X1
45            x1one = x1zero + alfa*s1;
46            x2one = x2zero + alfa*s2;
47            % the objective function value in X1
48            F1 = f(x1one,x2one);
49            Factual = F1;
50        end
51        fprintf(fp,'i= %d x1 = %f x2= %f fmin= %f\n',...
52           i,x1one,x2one,F1);
53        i = i + 1;
54        f1 = F1;
55        while (Factual<Fprecedent)
56            Fprecedent = Factual;
57            alfa = alfa + increment;
58            % the coordinates of X1
59            x1one = x1zero + alfa*s1;
60            x2one = x2zero + alfa*s2;
61            % the objective function value in X1
62            F1 = f(x1one,x2one);
```

```
63        end
64        deltaF = abs(F1-Fprecedent);
65        F0 = Factual;
66        x1zero = x1one;
67        x2zero = x2one;
68        alfa = alfa0;
69        if(abs(f0-f1)<=epsilon)
70            n = n + 1;
71        end
72        f0 = f1;
73        % test nMax times the convergence is fulfilled
74        if(n==nMax)
75            go = false;
76            break;
77        end
78    end
79    fprintf(fp, '\n      FINAL RESULT:\n\n');
80    fprintf(fp, 'x1=%f x2=%f  fMin=%f\n',...
81       x1one, x2one, f0);
82    fclose(fp);
83    %
```

Code 4.3 *randomPath.m*.

Running this program for the same mathematical model, as in the case of the random search method, we will reach the convergence condition in less than 1000 iterations, compared to the 136094 iterations of the previous case. (see Table 4.1).

Table 4.2 Iteration history of the Random path search.

Iteration	x1	x2	fMin
100	2.877877	2.925782	40.692337
200	2.737068	2.220548	15.067995
300	2.337830	1.718921	9.285854
400	1.628150	1.505413	5.553512
500	0.969801	1.264528	2.941544
600	0.311277	0.988930	1.174890
700	-0.154467	0.545346	0.167056
800	-0.478664	-0.164447	-0.663959
900	-0.259214	-0.727738	-1.086345

After 916 iterations, the program reached the convergence and the final result is $fMin$ = -1.095275. The optimum point has the coordinates: x_1= -0.183059 and x_2 = -0.793613.

4.4 The relaxation method

Although it is a method of multidimensional optimization, *the relaxation method* determines the optimal solution on the principle of one-dimensional search methods. Each iteration is solved in a number of steps equal to the number of decision variables. It begins from a randomly chosen starting point X^0. The components x_2^0, x_3^0, ..., x_n^0 of the starting point are fixed and the first component x_1^0 of the starting point is considered as variable. Along the axis corresponding to this variable, the search is one-dimensional. At the minimum point determined along this direction (see Fig. 4.1), the value x_1^0 determined in the previous step and the values of the components x_3^0, x_4^0, ...,x_n^0, are fixed constantly and the component x_2^0 is considered to be the actual variable. The minimum value of the objective function along this direction will be determined, then it will be considered as a variable x_3^0, while the other components of X^0 will be fixed as constants and so on.

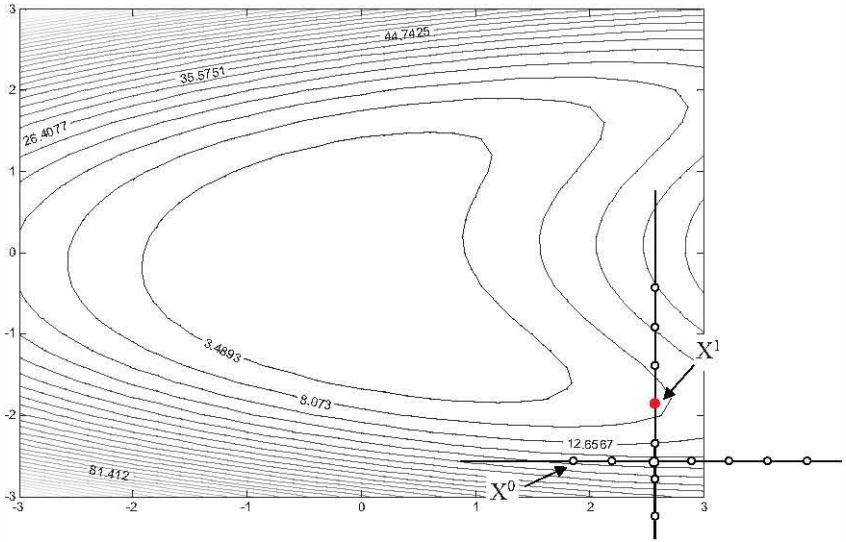

Fig. 4.1 The search principle of the relaxation method.

If the objective function is unimodal, starting from the starting point X^0, the variation of the objective function along that direction is similar to that in Fig. 4.2.

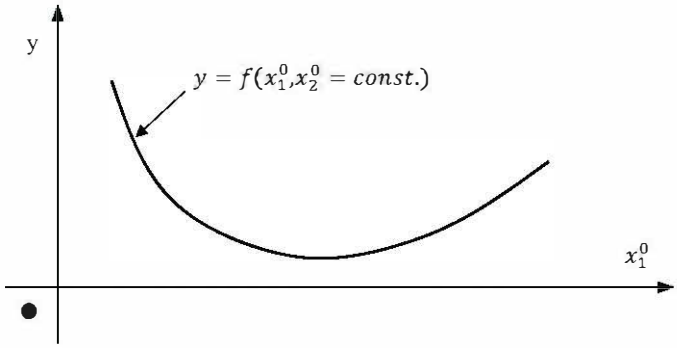

Fig. 4.2 Variation of the objective function along the direction given by x_1^0.

Taking this into account, we can apply along this line a unidimensional optimization method without constraints (in the case illustrated in Fig. 4.2, x_1^0 is the variable) and determine the minimum point, keeping all the variables as constants, except the one corresponding to that direction. Once we determine the minimum of the objective function in the direction of the last variable, an iteration is considered to be completed. Finally, the convergence test is performed. The *mrelax.m* shows the source code of the relaxation method.

```
%
%       The Relaxation Method (MULTI-GRID)
%
j = 1;
% the start point
x1zero = 3.0;
x2zero = 3.5;
%
% the search interval
a1 = -5.0;
b1 = 5.0;
a2 = -4.0;
b2 = 4.0;
%
% number of subintervals for local search
n = 250.0;
delta=1.0e3;
epsilon = 1.0e-6;
%
% open the file"relaxationMethod.txt"
```

```
fp = fopen( 'mrelax.txt', 'w' );
fprintf(fp, '\n\n The Relaxation Method (MULTI-
GRID)\n');
fprintf(fp,
'====================================\n\n');
%
ls1 = a1;
ld1 = b1;
ls2 = a2;
ld2 = b2;
%
fprintf(fp,'The limits of the search interval:\n\n');
fprintf(fp,'ls1=%f  ld1=%f\n ls2=%f  ld2=%f\n\n', ...
   ls1, ld1, ls2, ld2);
% the objective function value in the start point
fprintf(fp, '  The coordinates of the start point');
fprintf(fp, '  \n\n');
fprintf(fp, '   x1zero=%f   x2zero=%f\n\n', ...
   x1zero, x2zero);
f_precedent = f(x1zero,x2zero);
fprintf(fp, '   F(X0)=%f\n\n', f_precedent);
fMin = f_precedent;
% start the search
x2 = x2zero;
%
    while(delta>epsilon)
        % Block I: for variable "x1"
        x1 = ls1;
        increment = abs(ld1-ls1)/n;
        %
        while(x1<=ld1)
            func = f(x1,x2);
            if(func<fMin)
                fMin = func;
                x1Min = x1;
            end
            x1 = x1 + increment;
        end
        %
        % Block II: for variable "x2"
        x1 = x1Min;
        x2 = ls2;
        increment = abs(ls2-ld2)/n;
        %
        while(x2<=ld2)
```

```
66              func = f(x1,x2);
67              if(func<fMin)
68                  fMin = func;
69                  x2Min = x2;
70              end
71              x2 = x2 + increment;
72          end
73          %
74          f_actual = fMin;
75          fprintf(fp, 'i=%d  x1=%f  x2=%f fMin=%f\n',...
76            j, x1Min, x2Min, f_actual);
77          % Check the convergence condition
78          delta = abs(f_actual-f_precedent);
79          j = j + 1;
80          f_precedent = f_actual;
81          x2 = x2Min;
82      end
83  %
84  fprintf(fp, '\n      FINAL RESULT:\n\n');
85  fprintf(fp, '   x1Min=%f\n   x2Min=%f\n\n);
86  fprintf(fp, '   fMin=%f\n\n', x1Min, x2Min, fMin);
87  fclose(fp);
88  %
```

Code 4.4 *mrelax.m*.

The relaxation method in *mrelax.m* is implemented for a two decision variables unconstrained objective function, given by Eq. 2.1. The search interval was fixed to [-5.0,5.0] for x_1 and to [-4.0,4.0] for x_2. Along each direction corresponding to a decision variable, a local search based on the *Grid method* is made. For the *Grid method*, the number of subintervals was set to 250. The variable *delta*, initially set to a high value, is used to check the absolute difference of the objective function between two consecutive iterations. The convergence factor *epsilon* is set to1.0e-6. The start point X^0 has the coordinates $x_1^0 = 3.0$ and $x_2^0 = 3.5$. The computational results are saved in the *mrelax.txt* file. After 8 iterations, the program reached the convergence. Table 4.3 shows the history of these iterations.

Table 4.3 The iteration history of *mrelax.m*.

Iteration	x1	x2	fMin
1	4.960000	-2.368000	26.965334
2	2.320000	-1.728000	5.751951
3	1.000000	-1.312000	0.486993

(Table 4.3 continue)			
4	0.360000	-1.056000	-0.751306
5	0.040000	-0.896000	-1.035328
6	-0.080000	-0.832000	-1.081494
7	-0.160000	-0.800000	-1.094400
8	-0.200000	-0.800000	-1.094400

The last line in Table 4.3 represents the final result of the computation, after convergence has been reached. Consequently, the optimal solution to this problem is *fMin* = -1.0944, and corresponds to a point whose coordinates are x_1 = -0.2 and x_2 = -0.8.

4.5 The gradient method

One of the best-known optimization methods, but at the same time the one with the lowest performance among first-order methods, is the gradient method. As an optimization method of the first order it uses the first-order derivative of the objective function to determine the direction vector for minimization. Despite its low performance, this method is very important because it can be the starting point for other, more sophisticated first-order methods. As with other optimization methods, it starts from a randomly chosen point X^0, within the search space. The minimization vector S has the same direction as the gradient, but the orientation is opposite (see Fig. 4.3). Therefore:

$$S^i = -\nabla F(X^i); \tag{4.6}$$

Vector S^i is used in Eq. 4.5 in order to perform the one-dimensional search procedure. The method has a slow convergence, which can be explained by the fact that starting from the second iteration, no information regarding the previous iterations is used in determining the minimization direction. But this principle from the gradient method is used as the initial direction in most of the performing methods (Vanderplaats, 1999).
The following program has been built for an objective function dependent on two variables, but with a minimal effort from a user, it is possible to switch function of more variables. The program begins by setting up the starting point coordinates at lines 7 and 8. Then, the convergence factor *epsilon* is equal to 1.0e-6. It follows an increment used for calculating partial derivatives. The user needs to choose the initial leap factor and the increment used for increasing the leap factor.

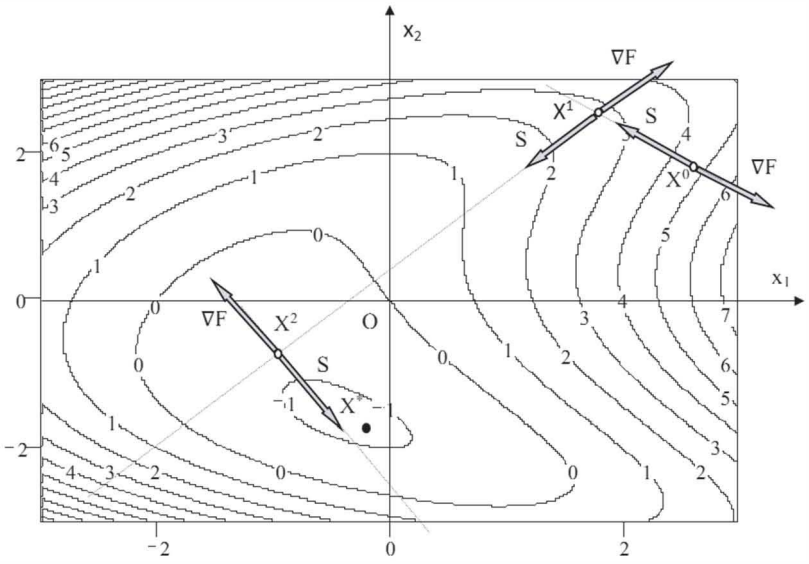

Fig. 4.3 The geometrical interpretation of the gradient method search

Users needs to take care that all these initial settings are specific for each optimization problem and should therefore be adjusted accordingly. Code .5 illustrates a Matlab implementation of this optimization method for the case of the same objective function seen in Code 4.2.

```
%
%       The Gradient Method
%
fp = fopen( 'results.txt', 'w' );
% x1zero, x2zero - the coordinates of the start point;
% add x2zero ... depending on the number of variables
x1zero = 3.0;
x2zero = 3.5;
% x1one, x2one - the coordinates of the current point;
% add x3one ... depending on the number of variables
% s1,s2-the coordinates of the minimization vector
direction
% add s3 ..., depending on the number of variables
%
% epsilon - the factor of convergence
epsilon = 1.0e-6;
```

```
% increment for the calculation of the partial
derivatives
delta = 0.00001;
% the initial leap factor
alfa0 = 0.001;
% the increment of alfa
incAlfa = 0.001;
%
fprintf(fp,'     The Gradient Method\n\nthe start
point:\n');
fprintf(fp,'x1zero = %f\nx2zero = %f\n\n',
x1zero,x2zero);
%
f_precedent = f(x1zero,x2zero);
% the minimization direction in X0
x1 = x1zero;    x2 = x2zero;
fa = f(x1,x2);
x1 = x1 + delta;
fb = f(x1,x2);
difference = fa-fb;
s1 = -difference/(-delta);
%
x2 = x2 + delta;
fb = f(x1,x2);
difference = fa-fb;
s2 = -difference/(-delta);
%
% X1 calculation
alfa = alfa0;
x1unu = x1zero + alfa*s1;   x2unu = x2zero + alfa*s2;
f_actual = f(x1unu,x2unu);
while(f_actual<f_precedent)
    f_precedent = f_actual;
    alfa = alfa + incAlfa;
    x1unu = x1zero + alfa*s1; x2unu = x2zero + alfa*s2;
    f_actual = f(x1unu,x2unu);
end
%
fMin = f_actual;
x1Min = x1unu;  x2Min = x2unu;
deltaF = abs(f_precedent-f_actual);
iteration = 1;
%
while(deltaF>=epsilon)
    func = f_actual;
```

```
62        x1p = x1Min;
63        x2p = x2Min;
64        x1 = x1p;
65        x2 = x2p;
66        fa = f(x1,x2);
67        x1 = x1 + delta;
68        fb = f(x1,x2);
69        difference = fa-fb;
70        s1 = -difference/(-delta);
71        x2 = x2 + delta;
72        fb = f(x1,x2);
73        difference = fa-fb;
74        s2 = -difference/(-delta);
75        %
76        % X1 calculation
77        alfa = alfa0;
78        x1one = x1p + alfa*s1;  x2one = x2p + alfa*s2;
79        f_actual = f(x1unu,x2unu);
80          while(f_actual<f_precedent)
81              f_precedent = f_actual;
82              alfa = alfa + incAlfa;
83              x1one = x1p + alfa*s1; x2one = x2p + alfa*s2;
84              f_actual = f(x1one,x2one);
85          end
86          %
87        fMin = f_actual;
88        x1Min = x1one;  x2Min = x2one;
89        deltaF = abs(func-f_actual);
90        fprintf(fp, ...
91          'i = %d   s(%f,%f)  x1=%f  x2=%f  fMin=%f\n',...
92          iteration, s1, s2, x1Min, x2Min, f_actual);
93        iteration = iteration + 1;
94    end
95    %
96    fprintf(fp, '\n      FINAL RESULT:\n\n');
97    fprintf(fp, ' x1=%f\n   x2=%f\n\n   fMin=%f\n\n',...
98        x1Min, x2Min, fMin);
99    fclose(fp);
100   %
```

Code 4.5 *gradientOpt.m*.

The coordinates of vector S that is opposite to the gradient vector are calculated by finite differences. Those who choose to, can also calculate the derivatives using the matlab function *diff*. After running the program,

the results that were obtained and saved in *results.txt* file, can be seen in Table 4.4.

Table 4.4 The iteration history for the gradient method.

Iteration	s1	s2	x1	x2	fMin
1	-8.114028	-6.229671	3.149386	1.798762	14.788042
11	0.145777	0.078997	-0.259290	-0.744251	-1.088216
21	0.142554	0.072315	-0.258715	-0.743951	-1.088277
31	0.135048	0.056971	-0.257332	-0.743314	-1.088421
41	0.016920	-0.047359	-0.199986	-0.788704	-1.094986
51	0.004904	0.005441	-0.186641	-0.792488	-1.095272

After executing 51 iterations, the value found for the minimum of the objective function is *fMin* = -1.095272, and the coordinates of this point are x_1 = -0.186641 and x_2 = -0.792488.

For this optimization problem, we note that *the relaxation method* needed only 8 iterations to find the optimal value of the objective function, while the gradient method performed 51 iterations for the same thing. And we should not forget that the relaxation method is a zero-order method, and Code 4.4 implementation uses *the Grid method* for local search, which is also a zero-order method.

4.6 The conjugate gradient method

The conjugate direction or *conjugate gradient method* only requires a simple modification of the gradient method, with a remarkable increase in the convergence rate. It is as simple to program as the gradient method. The conjugate direction is calculated as follows:

$$S^i = -\nabla F(X^i) + \beta_i \cdot S^{i-1}; \tag{4.7}$$

and the β_i scalar is given by the equation:

$$\beta_i = \frac{\left|\nabla F(X^i)\right|^2}{\left|\nabla F(X^{i-1})\right|^2}; \tag{4.8}$$

where i is an iterative index.The following implementation of the conjugate gradient method is made on the structure of the *gradientOpt.m* from Code 4.5.

```
%
%       The Conjugated Gradient Method
%
fp = fopen( 'conjugatedGrad.txt', 'w' );
% the coordinates of the start point
x1zero = 3.0;
x2zero = 3.5;
fprintf(fp,'     The Conjugated Gradient Method\n\n);
fprintf(fp,'the start point:\n');
fprintf(fp,'x1zero = %f\nx2zero = %f\n\n',...
   x1zero,x2zero);
% the convergence criterion
epsilon = 1.0e-6;
% the partial derivatives increment
delta = 0.00001;
% deltaF = 1e3;
% the initial leap factor
alfa0 = 0.001;
% the increment of alfa
incAlfa = 0.001;
%
f_precedent = f(x1zero,x2zero);
% fMin = f_precedent;
% the minimization direction in X0
x1 = x1zero;
x2 = x2zero;
%
fa = f(x1,x2);
x1 = x1 + delta;
fb = f(x1,x2);
diff = fa-fb;
s1 = -diff/(-delta);
%
x2 = x2 + delta;
fb = f(x1,x2);
diff = fa-fb;
s2 = -diff/(-delta);
% the calculation of X1
alfa = alfa0;
x1unu = x1zero + alfa*s1;
x2unu = x2zero + alfa*s2;
f_actual = f(x1one,x2one);
%
while(f_actual<f_precedent)
    f_precedent = f_actual;
```

```
46        alfa = alfa + incAlfa;
47        x1unu = x1zero + alfa*s1;
48        x2unu = x2zero + alfa*s2;
49        f_actual = f(x1one,x2one);
50    end
51    fMin = f_actual;
52    x1Min = x1one;
53    x2Min = x2one;
54    deltaF = abs(f_precedent-f_actual);
55    % the direction in start point == precedent direction
56    s1p = s1;
57    s2p = s2;
58    %
59    i = 0;
60    while(deltaF>=epsilon)
61        func = f_actual;
62        x1p = x1Min;
63        x2p = x2Min;
64        % the calculation of beta
65        beta = ...
66    ((sqrt(s1*s1+s2*s2))*(sqrt(s1*s1+s2*s2)))...
67    /((sqrt(s1p*s1p+...
68            s2p*s2p))*(sqrt(s1p*s1p+s2p*s2p)));
69        % the minimization direction in current point
70        x1 = x1p;
71        x2 = x2p;
72        fa = f(x1,x2);
73        x1 = x1 + delta;
74        fb = f(x1,x2);
75        diff = fa-fb;
76        s1 = -diff/(-delta);
77        x2 = x2 + delta;
78        fb = f(x1,x2);
79        diff = fa-fb;
80        s2 = -diff/(-delta);
81        % the calculation of X1
82        alfa = alfa0;
83        x1one = x1p + alfa*(s1+beta*s1p);
84        x2one = x2p + alfa*(s2+beta*s2p);
85        f_actual = f(x1one,x2one);
86            while(f_actual<f_precedent)
87                f_precedent = f_actual;
88                alfa = alfa + incAlfa;
89                x1unu = x1p + alfa*(s1+beta*s1p);
90                x2unu = x2p + alfa*(s2+beta*s2p);
```

```
91                f_actual = f(x1one,x2one);
92            end
93        fMin = f_actual;
94        x1Min = x1one;
95        x2Min = x2one;
96        deltaF = abs(func-f_actual);
97        s1p = s1;
98        s2p = s2;
99        i = i + 1;
100       fprintf(fp,'i=%d ', i);
101       fprintf(fp,'s1=%f s2=%f x1=%f x2=%f fMin=%f\n',...
102          s1, s2, x1Min, x2Min, f_actual);
103   end
104   %
105   fprintf(fp, '\n       FINAL RESULT:\n\n');
106   fprintf(fp, ' x1=%f\n   x2=%f\n\n   fMin=%f\n\n',...
107       x1Min, x2Min, fMin);
108   fclose(fp);
109   %
```

Code 4.6 *conjugatedGradientOpt.m.*

Executing the *conjugatedGradientOpt.m.* program for the same objective function given in Code 4.2, will get the convergence in 34 iterations (see Table 4.5), compared to the gradient method which, under the same initial conditions and starting from the same starting point, reached convergence in 51 iterations.

Table 4.5 Iteration history of *conjugatedGradientOpt.m.*

Iteration	s1	s2	x1	x2	fMin
1	-8.114028	-6.229671	3.159886	1.685762	15.249801
5	-8.932347	-4.235545	3.089115	1.649129	14.786891
10	0.034073	0.050207	-0.196726	-0.792981	-1.095026
15	0.041352	0.069197	-0.196304	-0.792268	-1.095066
20	0.037641	0.060021	-0.195909	-0.791623	-1.095096
25	0.034354	0.051950	-0.195550	-0.791064	-1.095120
30	0.031441	0.044849	-0.195221	-0.790581	-1.095138
34	0.013736	0.002158	-0.192996	-0.787570	-1.095185

The optimal solution found is $fMin$ = -1.095185, and the coordinates of the minimum point are x_1 = -0.192996 and x_2 = -0.787570. Note that regardless of the user-selected start point, the solution obtained must be the same. However, the position of the start point influences the number of iterations performed by the optimization program.

4.7 About convergence criteria

An important aspect of any optimization algorithm is to detect the moment when we have approached the optimal solution with the accuracy of ε, in other words the achieving of convergence. As a rule, several criteria for convergence verification must be used.

4.7.1 The absolute difference of the objective function values

One convergence criterion is the absolute difference of the objective function values. While the search process progresses in the direction of the optimal point, the absolute difference between the objective function values between two successive iterations becomes smaller (see Fig. 4.4). If we note the absolute difference of the values of the objective function with dF_{ad}, then at iteration i:

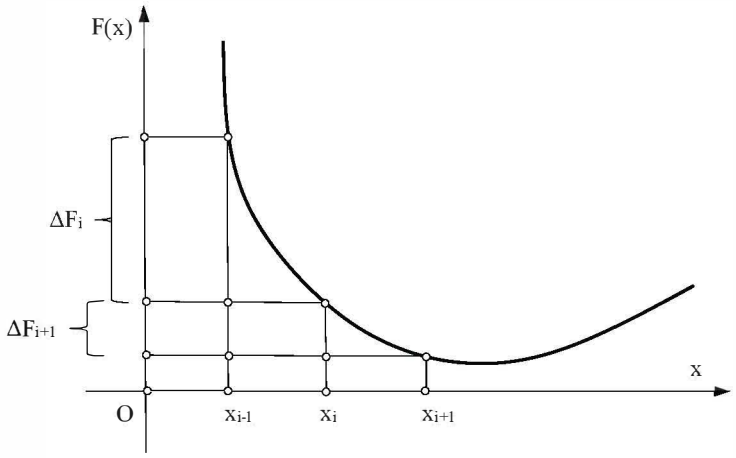

Fig. 4.4 The absolute difference of the objective function values between two successive iterations.

$$dF_{ad} = \left|F(X^i) - F(X^{i-1})\right|; \tag{4.9}$$

The convergence condition is met when $dF_{ad} \leq \varepsilon_{ad}$, where ε_{ad} is a very small quantity (equivalent to the initial imposed accuracy).

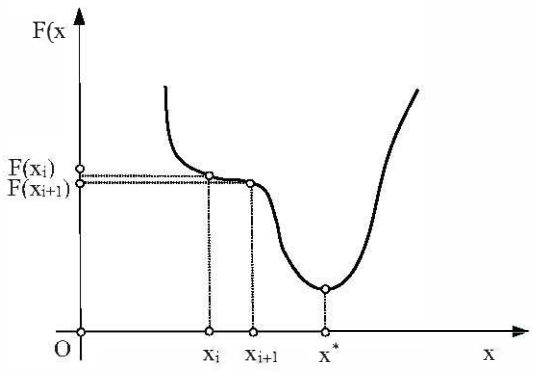

Fig. 4.5 Possible variation of the objective function slope

Sometimes, it may happen that on some portions the objective function has a slow slope and then the slope rises again (see Fig. 4.5). In such cases there is the risk that once the convergence condition is satisfied, we deduce that the point sought for is x_{i+1} and not x^*. Therefore, the convergence condition must be met several times in succession to ensure that we have determined the true optimum. Also, it may happen that in the optimal point area, the slope of the objective function is very small on large portions so that the search process risks stopping too early. To avoid this, we can assign a very low value to the convergence factor. However, this is not recommended because a very small value of the convergence factor, even if it increases the precision of the calculation, leads to an increase in the number of iterations and hence implicitly in the computer working time.

4.7.2 The relative difference of the objective function values

In order to avoid working with very small values for ε_d, the relative difference of the objective function values between two successive iterations can be used as a convergence factor. Thus:

$$dF_{rel} = \frac{dF_{ad}}{max\{F(X^i), 10^{-12}\}}; \tag{4.10}$$

In Eq. 4.10, as denominator we use the maximum value between the value of the objective function at the iteration i and a very small quantity (for example 10^{-12}) respectively, in order to avoid division by zero. As before, we will consider the convergence condition satisfied if $dF_{rel} \leq \varepsilon_{rel}$, where ε_{rel} is a small quantity, initially imposed. This condition must also be fulfilled several times in succession, for the same reasons as mentioned above.

4.7.3 The gradient equal to zero

The value of the gradient as a convergence criterion is used for the first or second order optimization methods. From a geometrical point of view, the derivative of the first order of the objective function at a certain point is represented by the tangent to the function graph at that point. Thus, at the optimal point (minimum or maximum), the tangent to the objective function graph is horizontal. Therefore, it is sufficient to check when the value of the first-order derivative (or the value of the objective function gradient) approaches zero with a certain accuracy. If we impose an ε_{tan} equal to a very low value, the condition to be checked is:

$$\nabla F(X^i) \leq \varepsilon_{tan}; \tag{4.11}$$

4.7.4 The maximum number of iterations

A criterion often used in convergence algorithms is the maximum number of iterations. Although very important, this criterion does not provide information about the current position in the search process, relative to the position of the optimal point. It is useful when the algorithm chosen to determine the optimal solution has too slow a convergence or the optimization program contains a programming error, which leads to repeating some calculations indefinitely.

4.8 Conclusions

The multivariable unconstrained methods represent a step ahead in solving optimization problems, according to their general mathematical model (see Eq. 2.6). This chapter presented *the random search* method as an example of the zero-order optimization method. The random search principle is simple to implement on the computer and also easy to

generalize to the desired number of decision variables It uses a recursive search relationship, similar to that used by first-order methods, but is still a zero-order method. In the same context of similarly with the first-order methods searching principle, still a zero-order method, *the relaxation method* is included, which has a search technique very similar to that of the gradient method. The chapter also includes two first-order methods, *the gradient method* and *the conjugate gradient method*. While the range of the first and second-order optimization methods is abounding, the book by Michalewicz & Fogel (2004) can be a good starting point.

5.

OPTIMIZATION TECHNIQUES FOR N VARIABLES CONSTRAINED FUNCTIONS

5.1 Introduction

In this chapter we will define the methods through which we will be able to use the minimization algorithms studied in the previous chapters to solve general optimization problems. The model of the optimization problem includes this time inequality and equality constraints. Optimization problems in which both the objective function and its constraints may have nonlinear form, closest to reality. That's why the numerical techniques that solve this kind of optimization problems are of great importance.

5.2 The random search method with constraints

We have already seen how the random search method works, as a zero-order method without constraints, within the previous chapter. The search principle is random, which means that every point in the search space has equal chances[1] to be analyzed. It is obvious that the precision of the search process increases with the number of points to be analyzed and with the working time as well. From this point of view, the random search method is not the most recommended, but it has the advantage of simplicity and ease of computer implementation. Our next task is to show how the source code of this method can be converted to solve an optimization problem with constraints.

The source code *randomSearch.m* is designed to determine the optimum of an objective function dependent on two decision variables, but it has been shown how with simple addition of source code lines, this

[1] The points generated by the *rand()* function are uniformly distributed in the range [0,1], making equal the chances of exploration across the workspace.

source code can be adapted for any number of decision variables. The framework of the optimization problem broadly remains the same, which means that we will limit the search to a space initially defined by explicit constraints applied to decision variables, of the form:

$$a_i \leq x_i \leq b_i; \qquad i = 1,2,\ldots,n. \tag{5.1}$$

Now, we will make a simple modification of the program *randomSearch.m*, (see Code 4.1) and we will transform the search method from one without constraints into one with constraints.

```
%       Random search method with constraints
%
fp = fopen( 'constrRanSearch.txt', 'w' );
fprintf(fp,'\nRandom search method with
constraints:\n\n');
% the number of decision variables
nVar = 2;
% the number of random generated points
n = 100000;
% the convergence factor
epsilon = 1e-3;
% pre-allocation of vectors a and b
for i = 1:nVar
    a = zeros(1,nVar);
    b = zeros(1,nVar);
end
% set-up of the search limits
for i=1:nVar
    a(i)=-3.5;
    b(i)=3.5;
end
fMin = 1e6;     % initialize fMin
fPrecedent = fMin;
% global search procedure
for i=1:n
    % random generation: variable x1
    x1 = rand;
    x1 = a(1)+x1*(b(1)-a(1));
    % random generation: variable x2
    x2 = rand;
    x2 = a(2)+x2*(b(2)-a(2));
    % calculate the objective function
    func = f(x1,x2);
```

```
34        % here we force the program to retain
35        % only those points
36        % which respect the constraint
37        if (func<fMin && g1(x1,x2)<=0)
38            fMin = func;
39            x1Min = x1;
40            x2Min = x2;
41            % printing intermediate results
42            fprintf(fp,'i=%d x1=%f x2=%f  fmin=%f\n',...
43                i, x1, x2, fMin);
44            if abs(fMin - fPrecedent)<=epsilon
45                break;
46            else
47                fPrecedent = fMin;
48            end
49        end
50    end
51    % printing final result
52    fprintf(fp, '\n      FINAL RESULT:\n\n');
53    fprintf(fp, 'x1Min=%f\n  x2Min=%f\n\n  fMin=%f\n',...
54        x1Min, x2Min, fMin);
55    fprintf(fp,'   g(x1Min,x2Min) = %f\n',...
56        g1(x1Min,x2Min));
57    fclose(fp);
58    %
```

Code 5.1 *constrRandomSearch.m*.

The major difference between the optimization methods without and with constraints is that the solution to the problem must satisfy the constraints. Please take note of line 35 in Code 5.1. It takes into account that there is an inequality constraint g_1 that must be respected. So, the points that are examined are only those that respect the condition:

```
if (func<fMin && g1(x1,x2)<=0)
```

If the optimization problem has more than one inequality constraint, then the line 35 should be:

```
if (func<fMin && g1(x1,x2)<=0 && g2(x1,x2)<=0 && ...)
```

We plan to run the *constrRandomSearch.m* source code, taking as objective function the one from *f.m* (see Code 4.2), and as constraint the inequality:

$$g_1(x_1, x_2) = 8 \cdot x_1 - 6 \cdot x_2 + 21 \leq 0; \tag{5.2}$$

The search interval [-3.5,3.5] was taken for each decision variable, while the number of points was chosen as n = 1e5. Fig. 5.1 shows the objective function and constraint graphics in 3D and contours. To draw the graphs from Fig. 5.1, the source code from Code 5.2 can be used.

```
%
%   Graphical display of the objective function
%            and inequality constraint
%
% the graphical display range setup
[x1,x2] = meshgrid(-3:0.1:3,-3:0.1:3);
%
% the objective function
f = 2*x1+x2+(x1.^2-x2.^2)+(x1-x2.^2).^2;
%
% the constraint
g = 8*x1-6*x2+21;
%
% the 3D display of f(x1,x2) and g(x1,x2)
subplot(2,1,1);
surf(x1,x2,f);
hold on;
%
surf(x1,x2,g);
hold off;
title('The 3D display of f(x1,x2) and g(x1,x2)');
xlabel('-3 < x1 < 3');
ylabel('-3 < x1 < 3');
% the contours of f(x1,x2) and g(x1,x2)
subplot(2,1,2);
[c] = contour(x1,x2,f,'-','color','k');
clabel(c);
hold on;
[c] = contour(x1,x2,g,':','color','k');
clabel(c);
hold off;
title('The contours of f(x1,x2) and g(x1,x2)');
xlabel('-3 < x1 < 3');
ylabel('-3 < x1 < 3');
%
```

Code 5.2 *objectivePlusConstraintGraphic.m*.

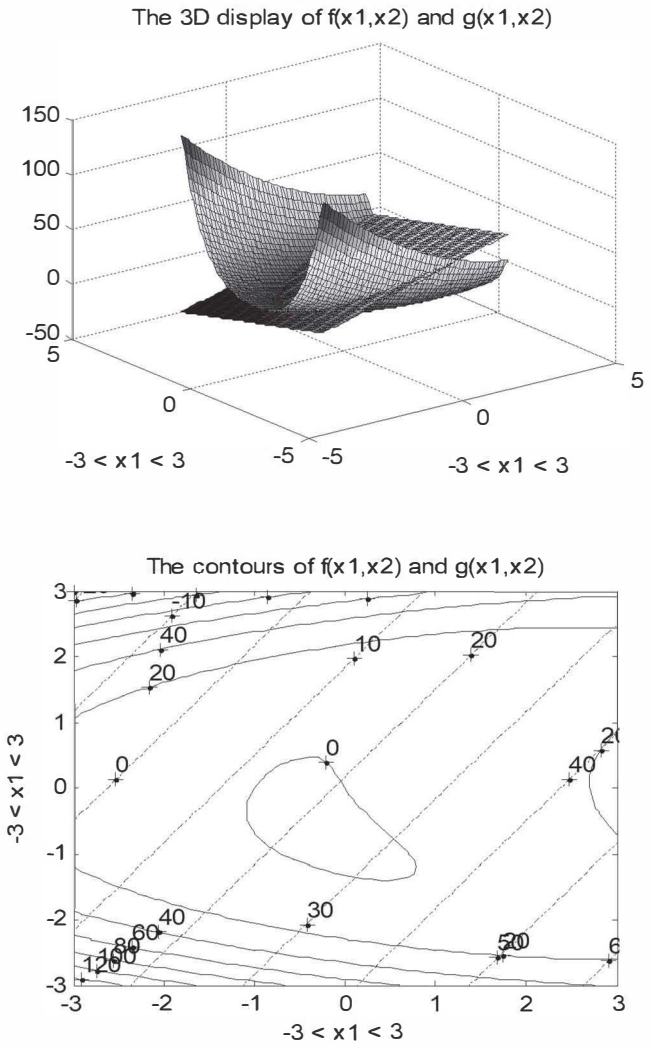

Fig. 5.1 The objective function and constraint graphs, represented in 3D and by contours, respectively.

Table 5.1 shows the progress of the search until the convergence factor ε = 1e-3 was reached.

Table 5.1 The search history of *constrRandomSearch.m*.

iteration	x_1	x_2	fMin
2	-3.005747	-0.251720	12.127374
26	-2.627158	0.404364	9.676324
415	-2.754995	-0.029760	9.644239
905	-2.412978	0.421908	7.953604
1274	-2.433734	0.298457	7.629550
2330	-2.151900	0.634046	7.081384
24143	-2.234099	0.524540	7.068694
71512	-2.199666	0.571777	7.067724

From Table 5.1 it can be noted that the optimum point f^{min} = 7.067724 has the coordinates x_1 = -2.199666 and x_2 = 0.571777. In this point, the value of the constraint is g_1 = -0.027985, which means that the constraint is satisfied almost at the limit.

5.3 The exterior penalty function method

Many methods for solving optimization problems with constraints use the principles to solve optimization problems without constraints, but the original problem is reformulated. So, as we have already shown at the end of chapter 3, classical methods of *SUMT* (*Sequential Unconstrained Minimization Techniques*) assume the construction of an objective pseudo-function, such as:

$$\emptyset(X, r_p) = f(X) + \delta \cdot R_p \cdot P(X); \tag{5.3}$$

where $f(X)$ is the original objective function, δ is a scalar that takes the value zero if all the restrictions are respected and one if at least one restriction is not respected, R_p is also a scalar, called penalty factor, and $P(X)$ is called polynom or function of penalty. The expression of the polynomial $P(X)$ is:

$$P(X) = \sum_{j=1}^{m} \{max[0, g_j(X)]\}^2 + \sum_{k=1}^{l} [h_k(X)]^2; \tag{5.4}$$

It should be noted that $P(X)$ has always a positive value which is added to $f(X)$ outside the admissible domain, where at least one of the constraints is violated. The value of the penalty factor R_p is kept constant over an iteration, but varies from one iteration to another, depending on the type of penalty method. The basic concepts are easy to understand and to use, without having to enter into theoretical details. That is why the purpose of this chapter is to explain and then develop optimization programs that can be used in various research applications, so that discussions will be limited to the general aspects of these methods, as well as the geometric interpretation of these aspects.

The source code of the exterior penalty function method is developed on the frame of the gradient method (see Code 4.5). Code 5.3 shows the implementation of this method.

```
%
%        The exterior penalty function method
%    based on a gradient method implementation
%
%    Input data:
x1zero = -3.0;      % x1 coordinate of the start point
x2zero = 3.0;       % x2 coordinate of the start point
epsilon = 1e-3;     % convergence factor
alfa0 = 0.001;      % leap factor - initial value
incAlfa = 0.001;    % leap factor increment
Rp = 1.0;           % penalty factor
incRp = 0.5;        % penalty factor increment
increment for partial derivatives calculus
delta = 0.00001;
% initialize the absolute difference of f
deltaF = 1e3;
% initialize the absolute difference of Fp
deltaFp = 1e3;
%
% open a file
fp = fopen('penaltyExt.txt','w');
% write input data to file
fprintf(fp,'The exterior penalty function method\n');
fprintf(fp,'based on a gradient method');
fprintf(fp,implementation\n\n');
fprintf(fp,'the start point X0(x1zero,x2zero):\n');
fprintf(fp,'x1zero = %6.3f x2zero = %6.3f\n',...
    x1zero,x2zero);
fprintf(fp,'convergence factor: epsilon = %f\n',...
    epsilon);
```

```
31  fprintf(fp,'initial leap factor alfa0 = %f\n',...
32      alfa0);
33  fprintf(fp,'leap factor increment incAlfa = %f\n',...
34      incAlfa);
35  fprintf(fp,'the penalty factor Rp = %f\n',Rp);
36  %
37  fPrecedent = f(x1zero,x2zero);
38  fMin = fPrecedent;
39  fp1 = fPrecedent;
40  while (deltaFp>=epsilon)
41      % direction of minimization
42      x1 = x1zero;
43      x2 = x2zero;
44      if(g1(x1,x2)>=0)
45          fa = pseudo_f(x1,x2,Rp);
46      else
47          fa = f(x1,x2);
48      end
49      %
50      x1 = x1 + delta;
51      if (g1(x1,x2)>=0)
52          fb = pseudo_f(x1,x2,Rp);
53      else
54          fb = f(x1,x2);
55      end
56      %
57      diff = fa - fb;
58      s1 = -diff / (-delta);
59      x2 = x2 + delta;
60      if (g1(x1,x2)>=0)
61          fb = pseudo_f(x1,x2,Rp);
62      else
63          fb = f(x1,x2);
64      end
65      %
66      diff = fa - fb;
67      s2 = -diff / (-delta);
68      % x1 calculation
69      alfa = alfa0;
70      x1one = x1zero + alfa*s1;
71      x2one = x2zero + alfa*s2;
72      if (g1(x1one,x2one)>=0)
73          fActual = pseudo_f(x1one,x2one,Rp);
74      else
75          fActual = f(x1one,x2one);
```

```
76      end
77      %
78      while(fActual<fPrecedent)
79          fPrecedent = fActual;
80          alfa = alfa + incAlfa;
81          x1unu = x1zero + alfa*s1;
82          x2unu = x2zero + alfa*s2;
83          if(g1(x1one,x2one)>=0)
84              fActual = pseudo_f(x1one,x2one,Rp);
85          else
86              fActual = f(x1one,x2one);
87          end
88      end
89      %
90      fMin = fActual;
91      x1Min = x1one;
92      x2Min = x2one;
93      deltaF = abs(fPrecedent - fActual);
94      %
95      while(deltaF>=epsilon)
96          %
97          func = fActual;
98          % alfa = alfa0;
99          x1p = x1Min;
100         x2p = x2Min;
101         %
102         % direction of minimization in actual point
103         x1 = x1p;
104         x2 = x2p;
105         if(g1(x1,x2)>=0)
106             fa = pseudo_f(x1,x2,Rp);
107         else
108             fa = f(x1,x2);
109         end
110         %
111         x1 = x1 + delta;
112         if (g1(x1,x2)>=0)
113             fb = pseudo_f(x1,x2,Rp);
114         else
115             fb = f(x1,x2);
116         end
117         %
118         diff = fa - fb;
119         s1 = -diff / (-delta);
120         x2 = x2 + delta;
```

```
121          %
122          if (g1(x1,x2)>=0)
123              fb = pseudo_f(x1,x2,Rp);
124          else
125              fb = f(x1,x2);
126          end
127          diff = fa - fb;
128          s2 = -diff / (-delta);
129          % x1 calculation
130          alfa = alfa0;
131          x1one = x1p + alfa*s1;
132          x2one = x2p + alfa*s2;
133          if (g1(x1one,x2one)>=0)
134              fActual = pseudo_f(x1one,x2one,Rp);
135          else
136              fActual = f(x1one,x2one);
137          end
138          %
139          while(fActual<fPrecedent)
140              fPrecedent = fActual;
141              alfa = alfa + incAlfa;
142              x1unu = x1p + alfa*s1;
143              x2unu = x2p + alfa*s2;
144              if(g1(x1one,x2one)>=0)
145                  fActual = pseudo_f(x1one,x2one,Rp);
146              else
147                  fActual = f(x1one,x2one);
148              end
149          end
150          fMin = fActual;
151          x1Min = x1one;
152          x2Min = x2one;
153          deltaF = abs(func - fActual);
154      end
155      %
156      fp2 = fMin;
157      deltaFp = abs(fp1 - fp2);
158      fp1 = fp2;
159      Rp = Rp + incRp;
160  end
161  % save final result
162  fprintf(fp,'\n   Final result:\n');
163  fprintf(fp,'x1Min = %f     x2Min = %f \n    fMin =
164  %f\n', x1Min,x2Min,f(x1Min,x2Min));
165  fprintf(fp,'constraint g1 = %f\n',g1(x1Min,x2Min));
```

```
% close the file 'penaltyExt.txt'
fclose(fp);
%
```

Code 5.3 *penaltyExt.m*.

While the exterior penalty function is a first-order optimization method, it uses first-order derivatives to calculate the minimization direction. That is because it is known that the gradient of a function is a vector which always points in the direction of the maximum value increase of the function. If the location is the start point, then in this point all the components of the gradient of the objective function must be calculated. So, depending on the position of the start point, if this location is in the admissible region, then the objective function value is found by calling the function in Code 5.4. If the point location is outside the admissible region, then the function value is calculated by calling the pseudo-objective function in Code 5.5. After calculating of the objective function in the start point, the same objective function is evaluated in a second point situated in the neighborhood of the start point. This neighborhood is defined by the parameter *delta*, defined among the input parameters.

```
% the evaluation of function f in a point
function [val] = f(x1,x2)
val = 2*x1+x2+(x1.^2-x2.^2)+(x1-x2.^2).^2;
return;
end
%
```

Code 5.4 *f.m*.

With the value of the function calculated at two very close points, the derivative of the function can be approximated by dividing the difference between these two values of the function by the *delta* parameter. As a rule, the minimization direction is always taken opposite to that of the gradient direction. The value of the pseudo-objective function in Code 5.5 is calculated by calling the objective function (see Code 5.4) and the constraint (see Code 5.6).

```
% the evaluation of pseudo-function in a point
function [val] = pseudo_f(x1,x2,Rp)
val = f(x1,x2)+Rp*g(x1,x2)*g(x1,x2);
return;
end
%
```

Code 5.5 *pseudo_f.m*.

```
% the evaluation of restriction g in a point
function [val] = g(x1,x2)
val = 8*x1-6*x2+21;
return;
end
%
```

Code 5.6 *g1.m*.

Once a minimization direction is calculated, the next step is to find the *leap factor* value along that direction. The minimum of the objective function found in this direction becomes *fActual*. There is an *fPrecedent* (initial settled or found at the precedent iteration) and during the current iteration the difference *deltaFp* between these two values is calculated. While the parameter *deltaFp* is higher than the convergence criterion ε, then the penalty parameter Rp is increased and the searching process is continued until the condition $deltaFp < \varepsilon$ is attained.

If we execute the program with the objective function defined in Code 5.4 and subjected to the inequality constraint $g_1(x_1,x_2) \leq 0$, we achieve the minimum value of the objective function as $f^{min} = 7.036444$, while the coordinates of this optimum are $[x_1,x_2] = [-2.125745, 0.657550]$. The value of the constraint in the optimum point is $g_1 = 0.048738$, which tells us that the calculation of the optimum point is done from outside the admissible region.

By running the program *penaltyExtGraphic.m* in Code 5.7, the contours of the pseudo-objective function as well as of the constraint can be displayed, for several values of the penalty parameter Rp.

```
%
%            The exterior penalty method
%   Graphical display of the penalty parameter Rp
%        influence on the objective function
%
% setup the display range
xstart = -10;
% computational resolution
increment = 0.5;
n = abs(2*xstart)/increment;
% initial values of variables
x1zero = xstart;
x2zero = xstart;
% matrix dimension: nxn elements
y1 = zeros(n,n);
y2 = zeros(n,n);
```

```
y3 = zeros(n,n);
y4 = zeros(n,n);
yg = zeros(n,n);
% four different penalty parameter values
Rp = [0.1,1.5,5.0,19.0];
% fill the matrices
x1 = x1zero;
for i = 1:n
    x2 = x2zero;
    for j = 1:n
        if g(x1,x2)>=0
            y1(i,j) = pseudo_f(x1,x2,Rp(1));
            y2(i,j) = pseudo_f(x1,x2,Rp(2));
            y3(i,j) = pseudo_f(x1,x2,Rp(3));
            y4(i,j) = pseudo_f(x1,x2,Rp(4));
        elseif g(x1,x2)<0
            y1(i,j) = f(x1,x2);
            y2(i,j) = f(x1,x2);
            y3(i,j) = f(x1,x2);
            y4(i,j) = f(x1,x2);
        end
        yg(i,j) = g(x1,x2);
        x2 = x2 + increment;
    end
    x1 = x1 + increment;
end
% plot the contours for Rp1
subplot(2,2,1);
contour(y1,12,'color','k');
hold on;
[c] = contour(yg,'color','k');
clabel(c);
% mark the optimum solution
x1Min = -1.786502;
x2Min = 0.524388;
%
scatter(x1Min+20, x2Min+20, 'markerfacecolor','r');
hold off;
title('f(x1,x2) and g(x1,x2); case: Rp = 0.1');
xlabel('-10 < x1 < 10');
ylabel('-10 < x2 < 10');
% plot the contours for Rp2
subplot(2,2,2);
contour(y2,20,'color','k');
hold on;
```

```
[c] = contour(yg,'color','k');
clabel(c);
% mark the optimum solution
x1Min = -2.163331;
x2Min = 0.565068;
%
scatter(x1Min+20, x2Min+20, 'markerfacecolor','r');
hold off;
title('f(x1,x2) and g(x1,x2); case: Rp = 1.5');
xlabel('-10 < x1 < 10');
ylabel('-10 < x2 < 10');
% plot the contours for Rp3
subplot(2,2,3);
contour(y3,20,'color','k');
hold on;
[c] = contour(yg,'color','k');
clabel(c);
% mark the optimum solution
x1Min = -2.186094;
x2Min = 0.569595;
%
scatter(x1Min+20, x2Min+20, 'markerfacecolor','r');
hold off;
title('f(x1,x2) and g(x1,x2); case: Rp = 5.0');
xlabel('-10 < x1 < 10');
ylabel('-10 < x2 < 10');
% plot the contours for Rp4
subplot(2,2,4);
contour(y4,20,'color','k');
hold on;
[c] = contour(yg,'color','k');
clabel(c);
% mark the optimum solution
x1Min = -2.197439;
x2Min = 0.566585;
%
scatter(x1Min+20, x2Min+20, 'markerfacecolor','r');
hold off;
title('f(x1,x2) and g(x1,x2); case: Rp = 19.0');
xlabel('-10 < x1 < 10');
ylabel('-10 < x2 < 10');
%
```

Code 5.7 *penaltyExtGraphic.m*.

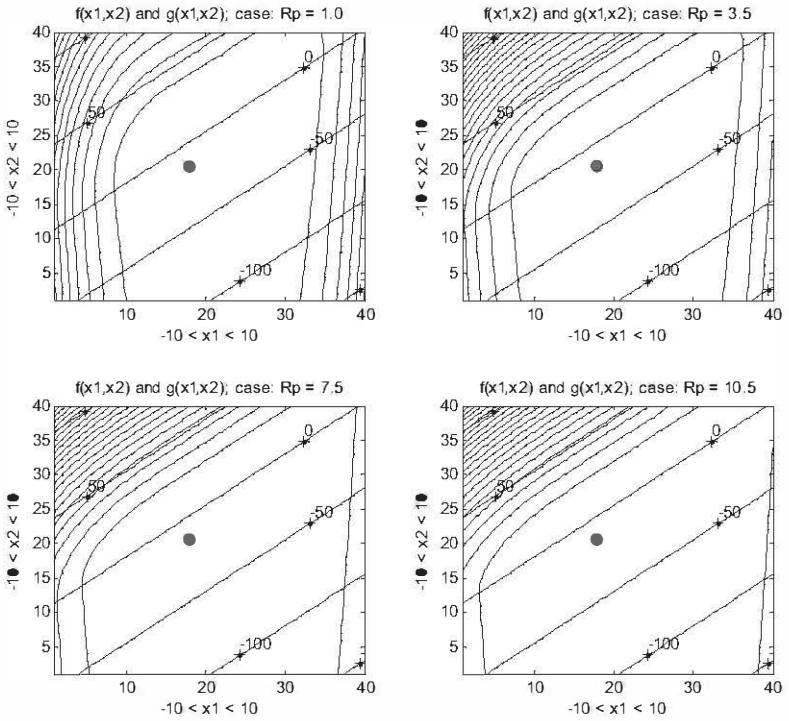

Fig. 5.2 The contours of the pseudo-objective function and of the constraint, for different values of the penalty parameter Rp.

Table 5.2 shows the change in the position of the point of minimum when the values of the penalty parameters increase.

Table 5.2 History of iterations at the increase of Rp.

Rp	x1Min	x2Min	fMin	g(x1,x2)
1.0	-2.1412	0.5689	6.6229	0.4572
1.5	-2.0926	0.6538	6.7703	0.3369
2.0	-2.1023	0.6546	6.8466	0.2536
2.5	-2.1079	0.6556	6.8931	0.2034
3.0	-2.1121	0.6556	6.9238	0.1697
3.5	-2.1152	0.6555	6.9458	0.1455
4.0	-2.1166	0.6566	6.9632	0.1275
4.5	-2.1184	0.6565	6.9761	0.1134
5.0	-2.1198	0.6565	6.9865	0.1021

5.5	-2.1210	0.6565	6.9950	0.0928
6.0	-2.1220	0.6565	7.0020	0.0851
6.5	-2.1228	0.6565	7.0080	0.0786
7.0	-2.1235	0.6565	7.0132	0.0730
7.5	-2.1241	0.6565	7.0176	0.0681
8.0	-2.1238	0.6576	7.0224	0.0640
8.5	-2.1243	0.6576	7.0259	0.0602
9.0	-2.1247	0.6576	7.0290	0.0569
9.5	-2.1251	0.6576	7.0317	0.0539
10.0	-2.1254	0.6576	7.0342	0.0512
10.5	-2.1257	0.6576	7.0364	0.0487

From Table 5.2 one can note that with the increase in the penalty factor, there is a slight increase in the value of *fMin*. This is due to the fact that the minimum point moves from the minimum value without constraints to the minimum value with constraints. At the same time, it can be noted that the value of the constraint in the considered point approaches the boundary $g(x_1,x_2) = 0$, but from the outside of the admissible domain of the solutions. From a practical point of view, if the optimization program fulfills the convergence condition, then the solution found lies within the required accuracy limits. However, certain precautions should be taken into account. If we look at the expression of the penalty polynom, we can see that the expressions of the restrictions are squared, which means we are working with large numbers. Under these circumstances there is a risk of overcoming the order of numbers representation in the internal memory of the computer. This alternative is not at all convenient because intermediate solutions are outside the admissible domain and practically do not fulfill the constraints of the optimization problem.

5.4 The interior penalty function method

The problem of finding the solution to the optimization problem with constraints starting from within the admissible domain is solved by a reformulation of the problem in a different manner than that from the method of exterior penalty function. In the case of the interior penalty method, the penalty polynom can be expressed as:

$$P(X) = R'_p \cdot \sum_{j=1}^{m} \left[-\frac{1}{g_j(X)} \right] + R_p \cdot \sum_{k=1}^{l} [h_k(X)]^2; \tag{5.5}$$

where the meaning of the penalty parameter R_p is similar to the exterior

penalty function method. However, there is a second penalty parameter, called R_p', which takes high positive values at first, then decreases with each iteration. According to Eq. 5.5, the expression of the objective pseudo-function becomes:

$$\emptyset(X, R_p, R_p') = F(X) + R_p' \cdot \sum_{j=1}^{m} \left[-\frac{1}{g_j(X)} \right] + R_p \cdot \sum_{k=1}^{l} [h_k(X)]^2; \tag{5.6}$$

It can be noted that the last term of Eq. 5.6 is similar to the one used for the exterior penalty function method. Thus, the R_p penalty parameter will take small values at the beginning of the optimization process, and its value will increase iteration by iteration. Another alternative for expressing the penalty function is the following:

$$P(X) = R_p' \cdot \sum_{j=1}^{m} \{-\log [-g_j(X)]\} + R_p \cdot \sum_{k=1}^{l} [h_k(X)]^2; \tag{5.7}$$

with the observation that $g_j(X)$ must be necessarily negative. This means that the point X has to be permanently in the admissible domain of solutions, where the inequality $g_j(X) \leq 0$ is satisfied. The alternative offered by Eq. 5.7 is recommended as being slightly better conditioned numerically (Vanderplaats, 1998).

In the following the source code of the program for optimization by interior penalty function method is presented. This program is built on the frame of the optimization program based on the random search method (see Code 4.1 *randomSearch.m*.).

```
%
%      The interior penalty function method
% based on the random search method implementation
%
fp = fopen( 'penaltyInt.txt', 'w' );
fprintf(fp,'\nRandom search method with');
fprintf(fp, 'constraints:\n\n');
% penalty parameter
Rp = 10.0;
incRp = 0.2;
```

```
% the number of decision variables
nVar = 2;
% the number of random generated points
n = 100000;
% the convergence factor
epsilon = 1e-3;
%
% pre-allocation of vectors a and b
for i = 1:nVar
    a = zeros(1,nVar);
    b = zeros(1,nVar);
end
%
% set-up of the search limits
for i=1:nVar
    a(i)=-3.5;
    b(i)=3.5;
end
%
% initialize fMin
fMin = 1e6;
fPrecedent = fMin;
% global search procedure
for i=1:n
    % random generation: variable x1
    x1 = rand;
    x1 = a(1)+x1*(b(1)-a(1));
    % random generation: variable x2
    x2 = rand;
    x2 = a(2)+x2*(b(2)-a(2));
    % calculate the objective function
    if g1(x1,x2)<=0
        func = f(x1,x2);
    else
        func = pseudo_f(x1,x2,Rp);
    end
    %
    if (func<fMin && g1(x1,x2)<=0)
        fMin = func;
        x1Min = x1;
        x2Min = x2;
        % printing intermediate results
        fprintf(fp,'i=%d x1=%f x2=%f\n',...
```

```
54              i, x1, x2);
55          %
56          fprintf(fp,'fmin=%f g1 = %f\n',...
57              fMin, g1(x1Min,x2Min));
58          if abs(fMin - fPrecedent)<=epsilon
59              break;
60          else
61              fPrecedent = fMin;
62          end
63          %
64          Rp = Rp - incRp;
65      end
66  end
67  %
68  % printing final result
69  fprintf(fp, '\n      FINAL RESULT:\n\n');
70  fprintf(fp, 'x1Min=%f\n  x2Min=%f\n\n   fMin=%f\n',...
71      x1Min, x2Min, fMin);
72  fprintf(fp,'g(x1Min,x2Min) = %f\n',...
73      g1(x1Min,x2Min));
74  % close the file
75  fclose(fp);
76  %
```

Code 5.8 *penaltyInt.m*.

To run this program one needs Code 5.4 *f.m*, Code 5.6 *g1.m* and one of the two variants of the pseudo-objective function (Code 5.9 or Code 5.10).

```
1   % the evaluation of function f in a point
2   function [val] = pseudo_f(x,y,Rp)
3   %
4   val = f(x,y)+Rp*(-1/g1(x,y));
5   return
6   end
7   %
```

Code 5.9 *pseudo_f.m*.

```
1   % the evaluation of function f in a point
2   function [val] = pseudo_f(x,y,Rp)
3   %
4   val = f(x,y)+Rp*(-log(-g1(x,y)));
5   return
6   end
7   %
```

Code 5.10 The second variant of *pseudo_f.m*.

If we use the version offered by *pseudo_f.m* in Code 5.9 and run *penaltyInt.m*, we get the results showed in Table 5.3. Table 5.4 contain the results for the same optimization problem, but the implementation of the *pseudo_f.m* used is that in Code 5.10.

Table 5.3 Iterations history for the pseudo-objective function in Code 5.9.

iteration	x1	x2	fMin	g1(x1,x2)
1	-0.628403	3.136207	101.937902	-2.844465
11	-2.643854	-0.013284	8.679693	-0.071127
481	-2.503071	0.341481	8.346820	-1.073456
2319	-2.354760	0.425143	7.508563	-0.388936
2386	-2.164296	0.681805	7.484980	-0.405194
2480	-2.108921	0.694252	7.154775	-0.036885
13792	-2.225674	0.548360	7.132492	-0.095546
33590	-2.198824	0.571200	7.058205	-0.017790

Table 5.4 Iterations history for the pseudo-objective function in Code 5.10

iteration	x1	x2	fMin	g1(x1,x2)
2	-3.141791	1.409967	29.324052	-12.594131
20	-3.441762	0.962525	24.07958	-12.309245
38	-2.543993	0.123555	8.042011	-0.093274
574	-2.157431	0.664810	7.319384	-0.248310
16814	-2.294466	0.444771	7.134090	-0.024357
22193	-2.192428	0.589404	7.114608	-0.075853
90410	-2.126673	0.665947	7.097573	-0.009071

On close examination of these results, it cam be easily noted that there is a small difference between the coordinates of the minimum point. In the first case the minimum point has the coordinates (-2.198824, 0.571200), while in the second case the coordinates are (-2.126673, 0.665947). A small difference remains between the values of the minimum point, which is *fMin* = 7.058205 in the first case and *fMin* = 7.097573 in the second case.

What is more important is the fact that the approximation of the optimal solution is done inside the admissible domain. This make any transitional solution accepted in case the program stops for some reason, for instance the exceeding the representation order of numbers in the internal memory of the computer.

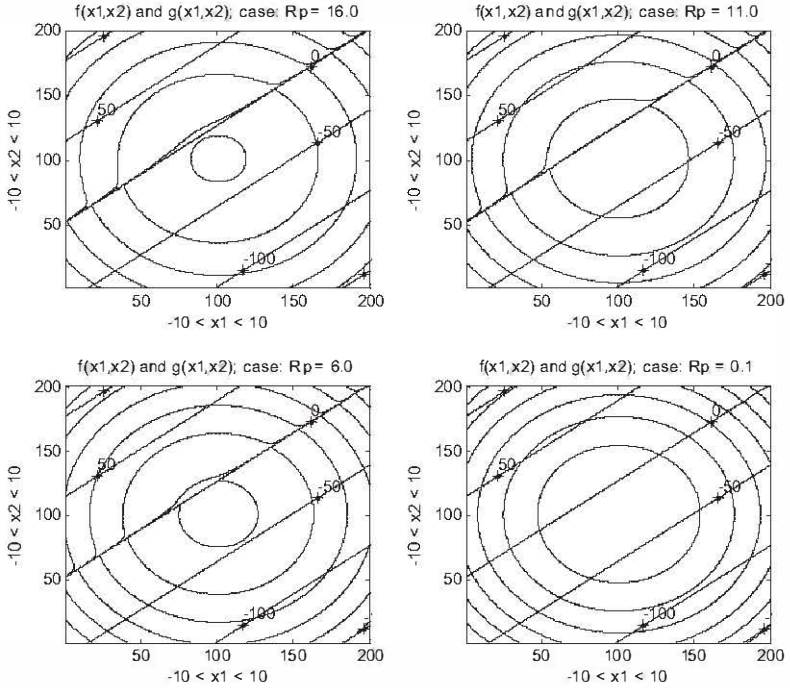

Fig. 5.3 Graphical display of the pseudo-objective function from Code 5.9, for different values of the penalty parameter.

The inner contour of the pseudo-objective function in Fig. 5.3 is moving toward the limit of the inequality constraint $g_1 = 0$, while the penalty parameter decreases iteration by iteration.

```
%            The interior penalty method
%   Graphical display of the penalty parameter Rp
%        influence on the objective function
% matrix dimension: 200x200 elements
n = 200;
y1 = zeros(n,n); y2 = zeros(n,n);
y3 = zeros(n,n); y4 = zeros(n,n);
yg = zeros(n,n);
increment = 0.1;
% four different penalty parameter values:
Rp = [16.0,11.0,6.0,0.1];
% initial value for x1
```

```
x1 = -10.0;
for i = 1:n
    % initial value for x2
    x2 = -10.0;
    for j = 1:n
        if g(x1,x2)>=0
            y1(i,j) = pseudo_f(x1,x2,Rp(1));
            y2(i,j) = pseudo_f(x1,x2,Rp(2));
            y3(i,j) = pseudo_f(x1,x2,Rp(3));
            y4(i,j) = pseudo_f(x1,x2,Rp(4));
        elseif g(x1,x2)<0
            y1(i,j) = f(x1,x2); y2(i,j) = f(x1,x2);
            y3(i,j) = f(x1,x2); y4(i,j) = f(x1,x2);
        end
        % calculate the constraint value
        yg(i,j) = g(x1,x2);
        % incrementation of x2
        x2 = x2 + increment;
    end
    % incrementation of x1
    x1 = x1 + increment;
end
% plot the contours for Rp1
subplot(2,2,1);
contour(y1,6,'color','k');
hold on;
[c] = contour(yg,'color','k');
clabel(c);
hold off;
title('f(x1,x2) and g(x1,x2); case: Rp = 16.0');
xlabel('-10 < x1 < 10'); ylabel('-10 < x2 < 10');
% plot the contours for Rp2
subplot(2,2,2);
contour(y2,6,'color','k');
hold on;
[c] = contour(yg,'color','k');
clabel(c);
hold off;
title('f(x1,x2) and g(x1,x2); case: Rp = 11.0');
xlabel('-10 < x1 < 10'); ylabel('-10 < x2 < 10');
% plot the contours for Rp3
subplot(2,2,3);
contour(y3,6,'color','k');
hold on;
[c] = contour(yg,'color','k');
```

```
58    clabel(c);
59    hold off;
60    title('f(x1,x2) and g(x1,x2); case: Rp = 6.0');
61    xlabel('-10 < x1 < 10'); ylabel('-10 < x2 < 10');
62    % plot the contours for Rp4
63    subplot(2,2,4);
64    contour(y4,6,'color','k');
65    hold on;
66    [c] = contour(yg,'color','k');
67    clabel(c);
68    hold off;
69    title('f(x1,x2) and g(x1,x2); case: Rp = 0.1');
70    xlabel('-10 < x1 < 10');
71    ylabel('-10 < x2 < 10');
72    %
```

Code 5.11 *penaltyIntGraph.m*.

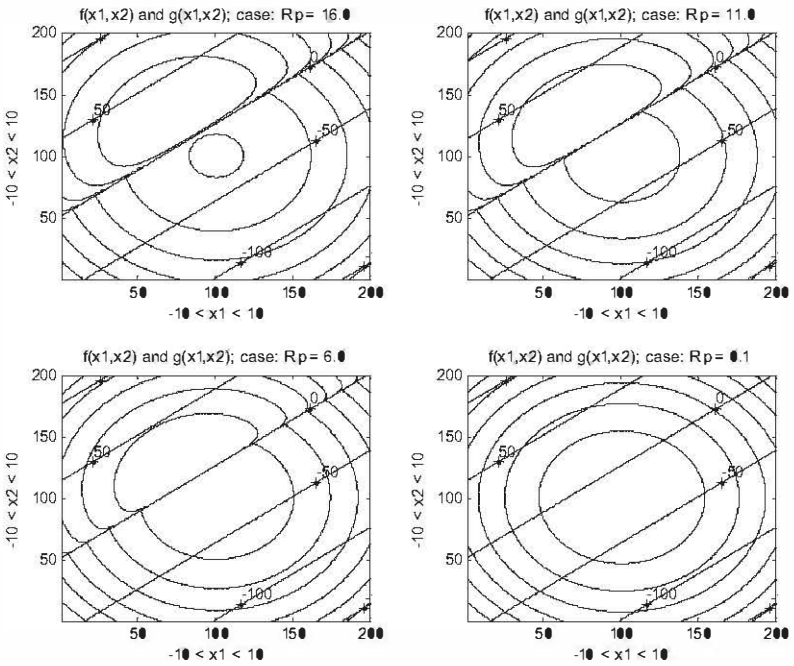

Fig. 5.4 Graphic of the pseudo-objective function from Code 5.10, for different values of the penalty parameter.

To draw the graphs from Fig. 5.3 and Fig. 5.4, the Code 5.11 can be used. It works together with Code 5.4, Code 5.6, and for the pseudo-objective function can be used one of the Code 5.9 or Code 5.10.

5.5 Conclusions

In this chapter, three optimization methods have been presented. All these three methods are able to solve a multivariable constrained optimization problem. The first of these, the random search method with constraints, is a continuation of the unconstrained random search method, presented in the previous chapter. The chapter continues with the methods of exterior and interior penalty function respectively. These methods require the selection of a starting point. But regardless of the chosen starting point, the search method must find the problem solution. If the optimization problem has local optimal points then things get complicated and other methods should be used. It is recommended, however, to restart the search procedure from several different starting points, in order to be sure about the correctness of the result.

6.

GLOBAL OPTIMIZATION

6.1 Introduction

Despite the diversity of the optimization problems, these are mostly solved in quite similar manners. At the same time, even though a great number of algorithms exists, practically, each one is applicable only in a very narrow field. As optimization problems are generally difficult to solve, having multiple local optima, there is still hope to find an approximate solution. Optimization literature offers us two "reasonable" methods for determining *Probably Approximately Correct* (PAC) solutions: the *Monte Carlo* method as a general search principle and the *multistart method* for local search, respectively.

The *Monte Carlo* method is based on the following principle: if the best option is needed, it should be tried "at random" many times and then the best option found between those attempts chosen. If there are enough different attempts, the best option found will almost certainly be an optimal global value. This method is valid both mathematically and intuitively. The advantages of the method are both its simplicity and its universality. But it has the disadvantage of being too slow.

In contrast to the *Monte Carlo* method, which is a good *global search* method, the *multistart* principle is based on local search. Here any optimization method presented in previous chapters can be used, depending of course, on the specificity of the problem. In principle, it starts from a starting point x_0 and a search is made between the possible values of its neighborhood. The point at which the best value is determined is denoted x_1, and is followed by a search in its neighborhood. Analogously, x_2, x_3, ... are determined until the method reaches an x_n point where it seems (depending on the required precision) that the best value was found. Then x_n can be considered a local optimal point. It cannot assume that x_n is the general or absolute optimum of the optimization problem, but it is the optimal point in relation to its immediate neighborhood. In mathematics, it is easy to define the term neighborhood. The principle of the *multistart* method requires rerunning the local search

procedure from several different starting points and choosing the best result of these attempts.

Sometimes a single starting point is enough. For some particular problems, it is sufficient to choose a single starting point and to perform the local search from that starting point, the absolute optimum is definitely found. Such a problem is called convex. Unfortunately, the vast majority of optimization problems in economics, physics, engineering etc. are non-convex. The only important thing in convex optimization problems is the speed of convergence. But convexity is quite a special property. In general, local search methods do not perform well unless they are applied according to the *multistart* principle. It should be noted that the *multistart* principle is effective in cases where optimization problems do not have many optimal local points.

6.2 The Monte Carlo method

Methods for which the search principle is based on random numbers are generally called the *Monte Carlo* methods. At present, random numbers are often replaced by computer-generated pseudo-random numbers by randomization procedures. The *Monte Carlo* method has been successfully applied to solving linear equation systems, to calculating the inverse matrix, to evaluating multiple integrals, to solving the Dirichlet problem, to solving functional equations of a variety of types and so on. It has also been used in the field of nuclear physics.

The principle of the *Monte Carlo* method will be illustrated in the case of a simple integral calculation problem, although the *Monte Carlo* method is more efficient for calculating multiple integrals. For this, the following integral will be considered:

$$y = \int_0^1 \sqrt{1 - x^2}\, dx; \tag{6.1}$$

The function under the integral represent a circle arc of 90°, that can be inscribed in a square, whose edge size is the unit (see Fig. 6.1).

In Matlab, every time the function *rand()* is called, a random number r_i, $0 \le r_i \le 1$, is generated. The numbers r_i are called *pseudo-random numbers*[1] and are uniformly distributed. If the *rand()* function is called $2N$ times, the

[1] While these numbers are generated based on a mathematical method, they cannot be considered purely random, therefore the designation *pseudo-random*.

result will be a set of N pairs that might be regarded as N points with the coordinates (x_i, y_i) in a plane xOy. Because these $2N$ numbers are uniformly distributed in the interval [0,1], then the N points are uniformly distributed across the area of the square.

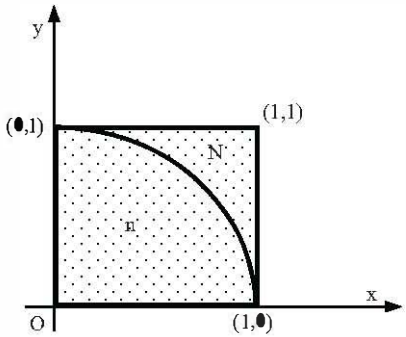

Fig. 6.1 The circle arc area inscribed within a unit area square.

Using a specific method it is interesting to count how many of the N points cover the area under the circle arc in Fig. 6.1. If this information is known and the result of this count is n, it is possible to make the following statement:

$$\frac{circle\ arc\ area}{square\ area} = \frac{n}{N}; \tag{6.2}$$

This proportion becomes valid as the number N of uniformly distributed points on the square area generated and in the mean time becomes higher. From Eq. 6.2, the circle arc area becomes:

$$\begin{aligned} circle\ arc\ area &= \int_0^1 \sqrt{1-x^2}\,dx \\ &= square\ area \cdot \frac{n}{N}; \end{aligned} \tag{6.3}$$

But the square area is equal to unit, while its edge size is a unit as well. So, in this simple case, the Eq. 6.3 becomes:

$$i = \int_0^1 \sqrt{1-x^2}\,dx = \frac{n}{N}; \tag{6.4}$$

While the value of this integral signifies the area of a quarter of a circle that has a radius equal to unit, multiplying the value of this integral by 4 generates an approximation of the number Pi.

Code 6.1 shows a short Matlab program that calculates the integral i, defined by Eq. 6.4.

```
%
%     Integration by Monte Carlo method
%
% open a text file to save results
fp = fopen('results.txt','w');
fprintf(fp,'  Integration by Monte Carlo method\n\n');
% perform calculation a predefined number of steps
nSteps = 100;
% memory allocation
intVal = zeros(1,nSteps);
i = 1;  %initial value of the index for nSteps
% the number of points generated inside square area
N = 100000;
for k = 1:nSteps
    % n is the number of points (x,y) that are
    % generated under the circle arc
    n = 0;
    for j = 1:N
        x = rand();
        y = rand();
        if y <= sqrt(1-x^2)
            n = n + 1;
        end
    end
    intVal(1,i) =n/N;
    i = i + 1;
end
% the mean value of intVal
theMean = mean(intVal);
% the standard deviation of the intVal
theStd = std(intVal);
% print results to file
fprintf(fp,'the number of points generated in the');
fprintf(fp,'square area\n   N = %f\n',N);
fprintf(fp,'the number of points under the circle');
fprintf(fp,'arc\n   n = %f\n',n);
fprintf(fp,'\nthe mean value of the ');
fprintf(fp,'integral = %f\n',theMean);
fprintf(fp,'standard deviation  %f\n',theStd);
```

```
40  % close the text file
41  fclose(fp);
42  %
```

Code 6.1 *MonteCarloIntegration.m*.

Calculation of the integral is done in a number of steps assigned at line 7 in the source code of the program. The total number N of points randomly generated inside the unit square area is also set at line 12. Lines 18 and 19 generate the pair of points (x,y) and lines 20, 21 and 22 count the number n of points that are placed inside the circle arc area. The program stores all integral values calculated *nSteps* times in a string called *intVal*. For these values the mean and standard deviation are evaluated. Table 6.1 shows this information for two cases, corresponding to $N = 1000$ and 5000 points, respectively.

Table 6.1 The results of Monte Carlo integration.

N	nSteps	Mean	Standard deviation	Pi approximation
1000	500	0.7851	0.0123	3.1404
5000	500	0.7854	0.0055	3.1416

Please note that if this code is rerun, it might get a slightly different result, depending on the random numbers generated each time the *rand()* function is called. Also it is worth mentioning that the standard accuracy of *Monte Carlo* integration is at about 1e-3, and increases as the number of N points randomly generated increases. The running time also increases with N. But the power of *Monte Carlo* integration becomes significant in the case of multiple integrals, a subject not detailed in this book.

6.3 Global optimization algorithm

In the following, a global optimization algorithm applicable to solving nonconvex problems is proposed. It is effective even if a problem has multiple local optima. Let us imagine that our objective function has the equation:

$$f(x_1, x_2) = -0.02 \cdot sin(x_1 + 4x_2) - 0.2 \cdot cos(2x_1 + 3x_2) - 0.2 \cdot sin(2x_1 - x_2) + 0.4 \cdot cos(x_1 - 2x_2); \quad (6.5)$$

While the objective function in Eq. 6.5 depends on two variables x_1 and x_2, its graphical representation is a surface (see Fig. 6.2). From Fig. 6.2 it is

evident that this surface has many peaks and valleys, interpretable as many local minimum (or maximum) points, depending on the problem scope. Usually, a numerical optimization procedure risks ending in a local optimum point instead of an absolute[2] minimum point.

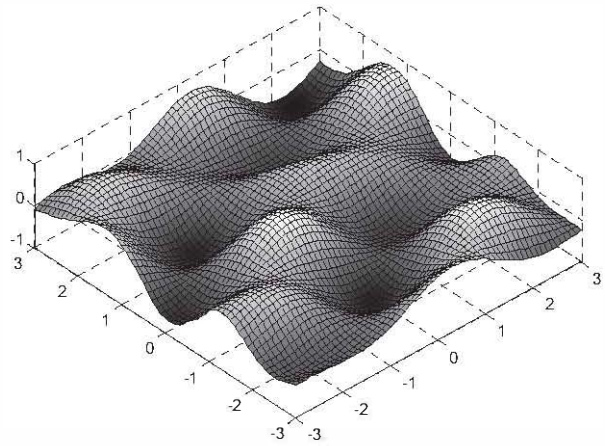

Fig. 6.2 The surface depicted by Eq. 6.5.

Even if the objective function depends on two variables only, simply viewing its graphical representation doesn't help too much in understanding where the location of the solution of this problem is. The Code 6.2 draws the surface in Fig. 6.2.

```
1   %           functionGraphPlot.m
2   %
3   % This program draws the mesh of a multimodal
4   % function that depends on two variables
5   %
6   % set drawing limits on x1 axis
7   a1 = -3;
8   b1 = 3;
9   % set drawing limits on x1 axis
10  a2 = -3;
11  b2 = 3;
12  % set the increment on x1 axis
```

[2] By absolute minimum point one must understand the minimum value of the objective function relative to the search area.

```
13  increment1 = 0.1;
14  % set the increment on x2 axis
15  increment2 = 0.1;
16  % steps on x1 axis
17  n1 = (b1 - a1)/increment1;
18  % steps on x2 axis
19  n2 = (b2 - a2)/increment2;
20  % memory preallocation of fGraph matrix
21  fGraph = zeros(n1,n2);
22  % calculation of fGraph values
23  x1 = a1;
24  for i1 = 1:n1
25      x2 = a2;
26      for i2 = 1:n2
27          fGraph(i1,i2) = f(x1,x2);
28          x2 = x2 + increment2;
29      end
30      x1 = x1 + increment1;
31  end
32  % drawing of fGraph
33  mesh(fGraph);
34  %
```

Code 6.2 *functionGraphPlot.m*.

At lines 7,8 and 10,11 the drawing limits are specified. Then the increment for each axis is set as desired. These increment values enable the calculation of points number n_1 and n_2 on each axis. The entire set of points that are calculated on the surface is $n_1 x n_2$. The surface values on these points are stored in a matrix called *fGraph*, whose dimensions are ($n_1 x n_2$). This matrix is initialized at line 21. It follow two nested ***for*** cycles, one for each of the objective function variables, where the objective function *f.m* (see Code 6.3) is called, and the values calculated in each point are stored in *fGraph* matrix. In the end, the *mesh* function is called at line 33 in order to draw the surface.

```
1  % This is an objective function, dependent on two
2  % variables x1 and x2. It must be adjusted according
3  % to the mathematical model of the problem
4  function [val] = f(x1,x2)
5  % the evaluation of function f in a point (x1,x2)
6  val = -0.02*sin(x1+4*x2)-0.2*cos(2*x1+3*x2)-
7  0.2*sin(2*x1-x2)+0.4*cos(x1-2*x2);
8  return;
9  end
```

Code 6.3 *f.m*.

The design of the general optimization method follows two main steps. The first step relies on a global search based on *Monte Carlo* principles. The second step involve a principle based on a local search method. The program of the global optimization method follows in Code 6.4.

```
%
%            GLOBAL OPTIMIZATION METHOD
%
% set the number of random numbers generated
% for global search
n = 10000;
% set the grid numbers for local search
gridNumber = 250;
% set the interval size divisor for local search
isd = 20;
% the initial search limits on each axis
a1 = -3;
b1 = 3;
a2 = -3;
b2 = 3;
% set the absolute value of difference between
% actual and precedent value of the objective
function
delta = 1.0e3;
% set the convergence criterion value
epsilon = 1.0e-4;
% set the mimimum initial value of f
minF = 1.e20;
% open a text file to save results
fp = fopen('results.txt','w');
fprintf(fp,'      GLOBAL OPTIMIZATION METHOD\n\n');
fprintf(fp,'the size of random numbers  n = %d\n',n);
fprintf(fp,'grid numbers for local search: %d\n',...
    gridNumber);
fprintf(fp,'interval size divisor for local');
fprintf(fp,'search: %d\n',isd);
fprintf(fp,'initial search limits on each axis\n');
fprintf(fp,'a1 = %5.2f      b1 = %5.2f\n',a1,b1);
fprintf(fp,'a2 = %5.2f      b2 = %5.2f\n',a2,b2);
fprintf(fp,'the absolute difference between');
fprintf(fp,' function values:\n');
fprintf(fp,'        delta = %f\n',delta);
fprintf(fp,'the convergence criterion value:\n');
fprintf(fp,'        epsilon = %f\n',epsilon);
%
```

```
%  Global search (by a Monte Carlo method)
%
for i = 1:n
    x1 = a1 + (b1-a1)*rand;
    x2 = a2 + (b2-a2)*rand;
    % define here x3 = a3 + (b3-a3)*rand;
    % and so on ...
    func = f(x1,x2);   % adjust f(x1,x2,x3,...)
    if func<minF
        minF = func;
        x1_min = x1;
        x2_min = x2;   % append x3_min = x3; ...
    end
end
precF = minF;
%
%    Local search (by a Multi-Grid method)
%
% define the limits of intervals for local search
%
% left limit; x1 axis
%
ls1 = x1_min - abs(b1-a1)/isd;
if ls1<a1
    ls1 = a1;
end
%
% right limit; x1 axis
ld1 = x1_min + abs(b1-a1)/isd;
if ld1>b1
    ld1 = b1;
end
% left limit; x2 axis
ls2 = x2_min - abs(b2-a2)/isd;
if ls2<a2
    ls2 = a2;
end
%
% right limit; x2 axis
ld2 = x2_min + abs(b2-a2)/isd;
if ld2>b2
    ld2 = b2;
end
%
while delta>epsilon
```

```
86      % The block for x1 variable (keep x2=constant)
87      x2 = x2_min;
88      x1 = ls1;
89      increment = abs(ld1-ls1)/gridNumber;
90      while x1 <= ld1
91          func = f(x1,x2);
92          if func<minF
93              minF = func;
94              x1_min = x1;
95          end
96          x1 = x1 + increment;
97      end
98      %
99      ls1 = x1_min - increment;
100     if ls1<a1
101         ls1 = a1;
102     end
103     %
104     ld1 = x1_min + increment;
105     if ld1>b1
106         ld1 = b1;
107     end
108     %
109     % The block for x2 variable (keep x1=constant)
110     x1 = x1_min;
111     increment = abs(ls2-ld2)/gridNumber;
112     %
113     while x2<=ld2
114         func = f(x1,x2);
115         if func<minF
116             minF = func;
117             x2_min = x2;
118         end
119         x2 = x2 + increment;
120     end
121     %
122     ls2 = x2_min - increment;
123     if ls2<a2
124         ls2 = a2;
125     end
126     %
127     ld2 = x2_min + increment;
128     if ld2>b2
129         ld2 = b2;
130     end
```

```
131       %
132       actF = minF;
133       % check the convergence criterion
134       delta = abs(actF-precF);
135       precF = actF;
136   end
137   % draw the surface graph
138   %
139   x1 = -3:0.1:3;
140   x2 = -3:0.1:3;
141   % create the meshgrid
142   [x1,x2] = meshgrid(x1,x2);
143   %
144   % calculate the value of the F function in
145   % each point of the grid
146   F = f(x1,x2);
147   contour(x1,x2,F,15);
148   %
149   % mark the optimum solution
150   hold on;
151   scatter(x1_min, x2_min, 'markerfacecolor','r');
152   hold off;
153   %
154   fprintf(fp,'\n\n the minimum value: Fmin =');
155   fprintf(fp,'%f\n',minF);
156   fprintf(fp,'the coordinates of the optimal');
157   fprintf(fp,'point:\n');
158   fprintf(fp,' x1 = %f     x2 = %f\n',x1_min,x2_min);
159   % close the text file
160   fclose(fp);
161   %
```

Code 6.4 *globalOptimization.m*.

Initially the global search procedure is designed based on *Monte Carlo* principles, which means that a number n of pairs of points (x_i, y_i) are randomly generated inside the search area, defined by the search limits (lines 12 to 15). From these points, the point (x,y) that has the minimum value is thought to have more chances to be placed near the true optimum point. This statement is more accurate as the number n of points that are randomly generated inside the search area increases. The search for this point with value *minF* and coordinates *(x1_min,x2_min)* is done by lines 41 to 54 in the Code 6.4.

Starting from this point, a local search procedure is designed. This procedure is based on the Grid method (see the program *gridOptimization.m*

on Code 3.3 from chapter 3). First thing to do is to define a neighborhood of the starting point on each axis (lines 57 to 83). This neighborhood should be set for each axis separately. Then along each axis a search of the minimum point in that direction is made successively. In this particular case, the local search along the x_1 axis starts from the left bound of the neighborhood, that is ls_1, while x_2 is kept constant. Once the minimum point along x_1 axis is found, it is kept constant, while the local search is performed along the x_2 axis. When the search along all axes is finished, it means that one iteration is over. The value of the objective function at this point is compared with the value of the similar point at the previous iteration. If the difference between these two points denoted *delta* is less than a precision factor called *epsilon*, initially set, then the search stops, or continues otherwise. This part of local search is integrated in the source code between lines 85 to 136.

If an objective function has more than two variables – let us say three variables, than the source code must be adjusted. The source code of the objective function must be changed regarding the calling of the function. At line 4 in Code 6.3 one should write:

```
function [val] = f(x1,x2,x3);
```

The global search based on *Monte Carlo* principles must be also adjusted from two to three variables. After the line 45 one must insert, for the third variable, the line:

```
x3 = a3 + (b3-a3)*rand;
```

Then, the call of the objective function at line 48 must be adapted for three variables, as:

```
func = f(x1,x2,x3);
```

When the condition from the ***if*** block is fulfilled (lines 49 to 53 in Code 6.4), then after line 52 one should append:

```
x3_min = x3;
```

so that the third coordinate of the *minF* point is saved. Following after line 83, is the insertion of neighborhood assignment for the third variable x_3:

```
% left limit on x3 axis
ls3 = x3_min - abs(b3-a3)/isd;
```

```
if ls3<a3
    ls3 = a3;
end
% right limit on x3 axis
ld3 = x3_min + abs(b3-a3)/isd;
if ld3>b3
    ld3 = b3;
end
```

Concerning the local search along the first variable axis x_1, the values of x_2 and x_3 must be kept constant, so after lines 87 and 110, the line should be inserted:

```
x3 = x3_min;
```

Then, a new block for local search along x_3 axis should be inserted after line 130 in the Code 6.4 in the following manner:

```
% The block for variable x3 (keep x1,x2=constant)
    X1 = x1_min;
    x2 = x2_min;
    x3 = ls3;
    increment = abs(ld3-ls3)/gridNumber;
    while x3 <= ld3
        func = f(x1,x2,x3);
        if func<minF
            minF = func;
            x3_min = x3;
        end
        x3 = x3 + increment;
    end
    ls3 = x3_min - increment;
    if ls3<a3
        ls3 = a3;
    end
    ld3 = x3_min + increment;
    if ld3>b3
        ld3 = b3;
    end
```

After the line 137, the Code 6.4 continues with the objective function graph, represented as contours. The program ends by marking the minimum point with a red circle (see Fig. 6.3).

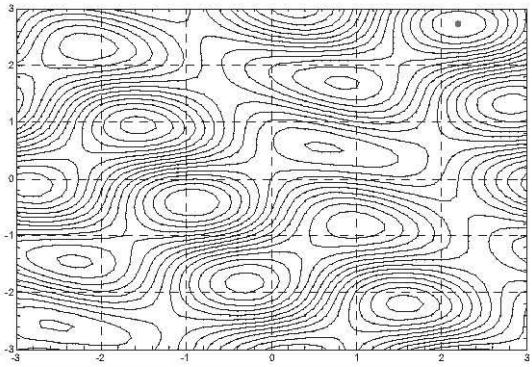

Fig. 6.3 The contours of the objective function defined by Eq. 6.5; the optimal point is indicated by the red circle in the upper-right part of the search area.

Executing the program *globalOptimization.m* for the objective function defined by the Eq. 6.5, will get the minimum value F_{min} = -0.806656, and the coordinates of the minimum point are: x_1 = 2.207522, x_2 = 2.732910.

6.4 Conclusions

In this chapter, a method has been designed to search for the optimal solution of a problem that presents multiple local optimum points. The search was performed in two steps. The first step consists in a general search procedure of *Monte Carlo* type. The second step is represented by a local *grid* search procedure. The example considered in this chapter is that of a problem dependent on two variables. This example was chosen to graphically illustrate the performance of the algorithm. The generalization of the program for a large number of variables is very simple. Also, having experience from the previous chapter, it is easy to add supplementary conditions (i.e. constraints) within the search steps, to make the algorithm capable of solving optimization problems with constraints.

7.

MULTICRITERIA OPTIMIZATION

7.1 Introduction

In practical optimization problems, it is often necessary to satisfy several criteria simultaneously. Under such conditions, a solution that is of maximum reliability, is the most technologically feasible solution with minimal material consumption, small size, minimum cost, maximum benefit etc. It is possible to find cases in which the same solution is optimal from all points of view, but this is a very rare case, for certain theoretical situations only and usually have no practical correspondent in real situations. Most of the time, we are confronted with the need to reconcile qualities that are contrary to each other from the point of view of optimization criteria and hesitate in finding the best or the worst of the possible compromises. If we want the choice to be made by the computer, then it is necessary to draw up an algorithm that contains certain rules of appreciation, allowing the choice of the optimal solution with discernment.

7.2 Some mathematical foundations of multi-criteria optimization

Let us consider the vector function, denoted $F(X)$, of k components, each of the components being in turn a function dependent on the vector X of the decision variables:

$$F(X) = [f_1(X), f_2(x), \dots, f_k(X)]^T; \tag{7.1}$$

We also consider a set of inequality and equality constraints, as follows:

$$g_j(X) \leq 0; \qquad j = 1,2, \dots, m; \tag{7.2}$$

$$h_j(X) = 0; \qquad j = m+1, m+2, \dots, m+p. \tag{7.3}$$

$$X = [x_1, x_2, \dots, x_n]^T \tag{7.4}$$

X is a vector from the space R^n, the indices m, n, p, k belonging to the set N of natural numbers. The functions $g_j(X)$ and $h_j(X)$ are defined for all finite values of the variables $x_1, x_2, ..., x_n$. They delimit the space $\Omega_X \in R^n$ of admissible *solutions* and each point X of the *admissible domain*, defines a possible solution (see Fig. 7.1). The vector $F(X)$, denoted by Eq. 7.9, is the relation that associates the vectors of R^n to each element X in the *admissible domain*. The k components of the vector $F(X)$ represent the various mathematical functions that model the optimization criteria.

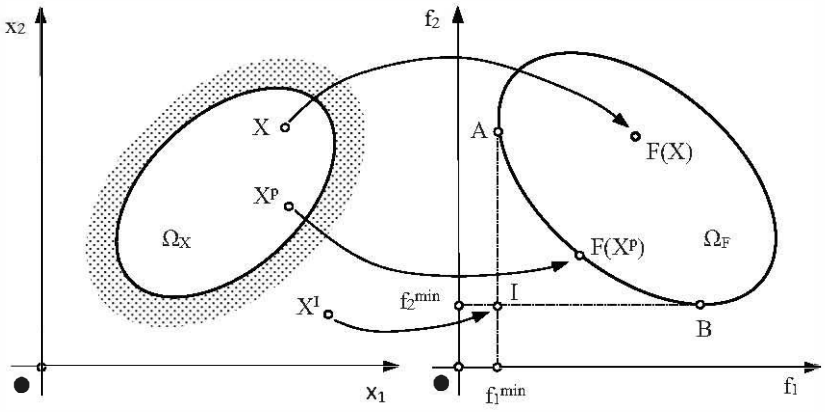

Fig. 7.1 The correspondence between the set of X points Ω_X in the admissible domain of solutions and the set of images of these points by means of the vector F, Ω_F.

But to optimize in this case does not simply mean to find a solution that minimizes (or maximizes) objective functions, as in the case of a monocriterial optimization problem. This is about determining the best solution in terms of the set of all criteria, and for this, the most important is defining the principle of optimality. To simplify the idea, we will consider that all objective functions should be minimized. There will be nothing to lose from the generality of information making this hypothesis, because we can easily transform any maximization problem into a minimization problem by the relationship:

$$max\{f(X)\} = min\{-f(X)\}; \tag{7.5}$$

Generally, we cannot get the ideal solution X^{ideal} that minimizes simultaneously all the components of the objective function vector. The optimal solution will be at most one of compromise. This solution can be reached in two phases.

In the first phase, the calculation of the non-superior solutions called the *Pareto-optimal* set must be performed. The *Pareto-optimal* set consists of those solutions for which no objective function can be improved without simultaneously degrading at least one other objective function. Fig. 7.1 shows an optimization problem with an objective function vector F that has two components $f_1(X)$ and $f_2(X)$, where the decision variable vector X depends on two variables x_1 and x_2. For any point $X(x_1,x_2)$ of the admissible domain of solutions Ω_X, there is a single point $F(X)$ belonging to the set Ω_F, whose coordinates are $f_1(X)$ and $f_2(X)$. Therefore, the set Ω_F represents the image of the admissible domain of solutions Ω_X, of the multicriteria optimization problem, by means of the objective function vector $F(X)$. The AB frontier of the set Ω_F represents the *Pareto-optimal* set. Points A and B on this frontier satisfy the condition $f_1^{min} = A$ and $f_2^{min} = B$ respectively. The point labeled I, which is outside the range Ω_F corresponds to the point X^{ideal}, which simultaneously optimizes both the components f_1 and f_2 of the objective function vector. From the *Pareto* set of solutions, in the second phase of solving, the most favorable solution to the situation considered is selected, if this solution obeys some specific rules that are imposed to the optimization problem.

7.3 The method of global criterion

Let us consider a multicriterial optimization problem in which the objective function vector has k components $f_i(X)$, $(i = 1,2,..., k)$ and the additional conditions are given by m inequality constraints $g_j(X) \leq 0$, $(j = 1, 2, ..., m)$. The first phase in solving this problem lies in determining the ideal solution, that is, the vector $f_i(X^{ideal})$ satisfying the conditions of minimum existence, for each of the objective functions taken independently of each other. In other words, we solve the problem of monocriterial optimization, for each of the vector components of the objective function separately. In this case, the global criterion is expressed as follows:

$$F^{(p)} = \left[\sum_{i=1}^{k} \left| f_i(X) - f_i(X^{ideal}) \right|^p \right]^{1/p}; \qquad (7.6)$$

Therefore the vector X needs to be found, whose norm, defined by Eq. 7.6, is minimal and at the same time all the restrictions are respected. Common norms used are:

$$p = 1 \qquad F^{(1)} = \sum_{i=1}^{k} \left| f_i(X) - f_i(X^{ideal}) \right|; \tag{7.7}$$

$$p = 2 \qquad F^{(2)} = \left[\sum_{i=1}^{k} \left| f_i(X) - f_i(X^{ideal}) \right|^2 \right]^{1/2}; \tag{7.8}$$

$$p = \infty \qquad F^{(\infty)} = \max_{i = 1,2,\ldots,k} \left| f_i(X) - f_i(X^{ideal}) \right|; \tag{7.9}$$

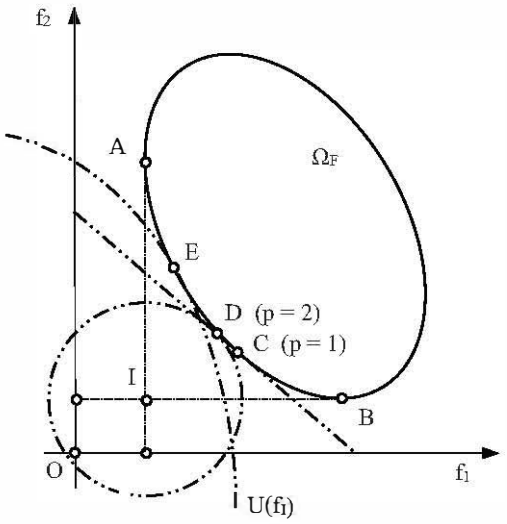

Fig. 7.2 Graphic interpretation of the global criterion method.

If the objective function vector has only two components, according to Fig. 7.2, the use of the $p = 1$ case denotes minimizing the sum of the distances between the values f_i of the objective function and their ideal values $f_i(X^{ideal})$. Graphically, the solution to the problem is represented by

point C in Fig. 7.2, given by the right parallel to the second bisector of the coordinate axes, tangent to the *Pareto-optimal* set.

The use of $p = 2$ is employed to determine the smallest distance between the ideal point I and the Pareto-optimal set. The solution is represented by point D in Fig. 7.2, where the minimum radius circle with the center in I is tangent to the *Pareto-optimal* set.

Finally, the case $p = \infty$ is reduced to minimizing the maximum distance with respect to the ideal point I (see point A in Fig. 7.2).

In conclusion, the optimal solution will largely depend on the choice of exponent p from the Eq. 7.6. No formula can be given regarding the choice of p, its value depending in each particular case studied.

Taking into account that in practice, rarely, all components of the objective function vector are of the same dimensional nature, or of the same order of magnitude, we will define the global criterion in the form:

$$F^{(p)} = \left[\sum_{i=1}^{k} \left|\frac{f_i(X) - f(X^{ideal})_i}{f_i(X^{ideal})}\right|^p\right]^{1/p}; \tag{7.10}$$

Eq. 7.10 has the advantage of equalizing the way the optimization procedure works, so that each criterion is treated equal to one another.

Example

Let us consider the case of an optimization problem that needs minimization and has two objective functions f_1 and f_2. Each objective function depends on two decision variables x_1 and x_2. There are no constraints to this problem. The mathematical model of this problem can be written as follows:

Minimize

$$F(f_1, f_2)^T; \tag{7.11}$$

where:

$$f_1(x_1, x_2) = (x_1 - 0.9)^2 + (x_2 - 1.3)^2 \tag{7.12}$$

$$f_2(x_1, x_2) = (x_1 + 1)^2 + \frac{(x_2 + 0.5)^2}{8}; \tag{7.13}$$

While the function described by Eq. 7.12 and Eq. 7.13 depends on two decision variables x_1 and x_2, their graphs are surfaces. We will consider the search space defined by intervals:

$$-3 \leq x_1 \leq 3 \tag{7.14}$$

$$-3 \leq x_2 \leq 3 \tag{7.15}$$

While there are no additional constraints imposed to this problem, the search space will be a square centered in the origin of the cartesian system. To solve this optimization problem, the relaxation method was used (see Code 4.4 *mrelax.m*). In this source code, the lines 6 to 13 were updated as follows:

```
6   x1zero = 1.0;
7   x2zero = 1.0;
8   %
9   % the searching interval
10  a1 = -3.0;
11  b1 = 3.0;
12  a2 = -3.0;
13  b2 = 3.0;
```

Code 7.1 Update for *mrelax.m*.

To define the norm of order p, as a Matlab function, the following source code should be used:

```
1   % the evaluation of function f in a point
2   function [val] = f(x1,x2)
3   % the objective function vector depends
4   % on two functions;
5   % each function depends on two variables x1 and x2;
6   %
7   %       f1(x1,x2) = (x1-0.9).^2+(x2-1.3).^2;
8   %       f2(x1,x2) = (x1+1).^2+(x2+0.5).^2/8;
9   %
10  % insert the order 'p' of the objective function:
11  p = 1;
12  val = (((x1-0.9).^2+(x2-1.3).^2)^p + ((x1+1).^2+...
13      (x2+0.5).^2/8)^p)^(1/p);
14  end
15  %
```

Code 7.2 The objective function *f.m*.

Before solving the multicriterial optimization problem, there the optimal solution for each criterion must be found solved individually. The same method of relaxation can be used, with the same update from Code 7.1. This time, at line 12 and 13, as objective function f_1, one should use the expresion from line 7 and for f_2 the expression from line 8. After solving the optimization problems individually, the following results were determined:

Table 7.1 The results of monocriterial optimization.

x_1	x_2	f_1	f_2
0.888000	1.296000	0.000160	3.967746
-1.008000	-0.504000	6.894880	0.000066

The first line in Table 7.1 shows the optimal value of the first criterion f_1 and the coordinates x_1 and x_2 where this optimal value is reached. At the point where f_1 takes the optimal value, the value of the second criterion is $f_2 = 3.967746$. The second line in Table 7.1 shows the optimal value of the second criterion f_2 and the coordinates of this optimal value. In the point where f_2 is optimal, the value of the first criterion is $f_1 = 6.894880$.

For solving the multicriterial optimization problem, in the Code 4.4 *mrelax.m*, the call to the norm function $F^{(p)}$ must be addressed to the expression in the Code 7.2. To analyze different possible solution from Pareto-optimal set, the user must play with different values of the norm p. In Table 7.2 optimal solutions of the multicriterial optimization problem, for different values of norm p are presented.

Table 7.2 Optimal solutions of the multicriterial optimization problem for different values of the norm p.

p	x_1	x_2	f_1	f_2	$F^{(p)}$
1	-0.048000	1.104000	0.937120	1.227906	2.165026
2	-0.096000	1.080000	1.040416	1.129266	1.535483
3	-0.096000	1.080000	1.040416	1.129266	1.369102
4	-0.096000	1.056000	1.051552	1.119858	1.293041
5	-0.096000	1.056000	1.051552	1.119858	1.249607
6	-0.096000	1.056000	1.051552	1.119858	1.221662
7	-0.096000	1.032000	1.063840	1.110594	1.202044
8	-0.096000	1.032000	1.063840	1.110594	1.187528
9	-0.096000	1.032000	1.063840	1.110594	1.176418

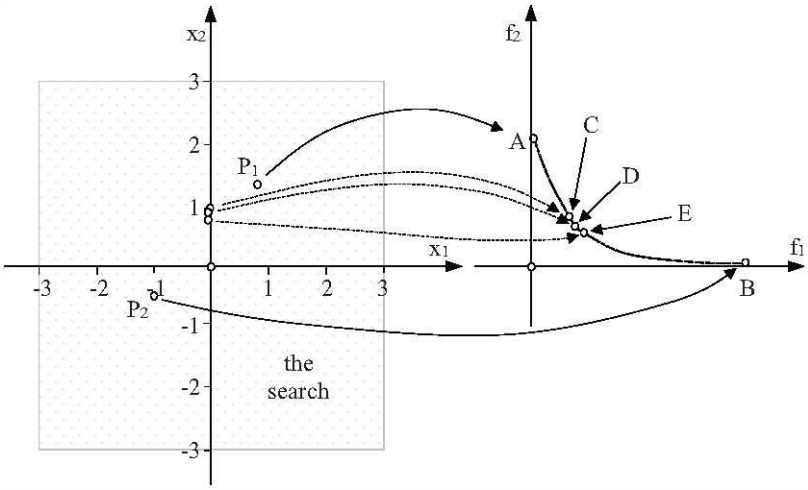

Fig. 7.3 Graphic illustration of the multicriteria optimization problem.

The point P_1 from the search space in Fig. 7.3 has the coordinates (0.888,1.296) and these coordinates belong to the optimal value of the first criterion f_1. The point P_2 from the same search space has the coordinates (-1.008,-0.504) and these coordinates belong to the optimal value of the second criterion f_2.

The point A of coordinates (f_1,f_2) = (1.6e-4,3.967746) is an end point of the *Pareto-optimal* set. Its first coordinate f_1 = 1.6e-4 represents the optimal value for the first criterion taken individually. The point B represent the other end point of the Pareto-optimal set. Its second coordinate is the optimal value f_2 = 6.6e-5 for the second criterion taken individually. The points C, D and E belong to the *Pareto-optimal* set and represent optimal solutions of the objective function vector $F^{(p)}(f_1,f_2)$, obtained for norm p = 1, 2 and 4, respectively (see Table 7.2).

The *Pareto-optimal* set is the frontier of the Ω_F domain. All points belonging to this set are potential solutions to the multicriteria optimization problem. Which of these points is the best solution? It depends on the specificity of each problem. In addition, we should not forget that the final stage of an optimization process involves testing the numerical solution. It should also be noted that the dotted part of the *Pareto-optimal* border in Fig. 7.3 is only hinted at. The only points determined with certainty are those marked A, B, C, D and E.

For the same example, we will now use another approach to solving the multicriteria optimization problem. This new approach will involve the

use of the relative norm function, according to Eq. 7.10. In addition, we will try to use a certain degree β_i of importance for each criterion. For this purpose, the Code 7.2 of the function *f.m* should be modified as you can see in Code 7.3.

```
1   %
2   % the evaluation of function f in a point
3   function [val] = f(x1,x2)
4   %
5   %  the objective function vector depends on
6   %  two functions:
7   %
8   %       f1(x1,x2) = (x1-0.9).^2+(x2-1.3).^2;
9   %       f2(x1,x2) = ((x+1).^2+(y+0.5).^2/8);
10  %
11  f1min = 0.000160;
12  f2min = 0.000066;
13  %
14  % the importance factors:
15  beta1 = 0.1;
16  beta2 = 0.9;
17  %
18  val = beta1*((x1-0.9).^2 + ...
19      (x2-1.3).^2 - f1min)/f1min +...
20      beta2*((x1+1).^2 +...
21      (x2+0.5).^2/8 - f2min)/f2min;
22  end
23  %
```

Code 7.3 Update of *f.m* that take into account some degrees of importance.

Lines 11 and 12 in Code 7.3 initialize the optimal values for both criteria f_1 and f_2, called f_1^{min} and f_2^{min}. Lines 15 and 16 initialize the values of weights β_1 and β_2. The sum of these weights must be equal to one unit.
The expression of the objective function vector F (see lines of code 18 to 21) is defined as the relative norm of order $p = 1$, as of Eq. 7.10, in which the weighting coefficients β_i were introduced. If we solve the problem using the relaxation method again, for the different values of β_i, the results in Table 7.3 are obtained.

Table 7.3 Variation of the optimal solution for different β_i values.

β_1	β_2	x_1	x_2	f_1	f_2	$F^{(1)}$
0,1	0,9	-0,912000	-0,024000	5,036320	0,036066	3638,509091
0,2	0,8	-0,816000	0,312000	3,920800	0,116274	6309,381818
0,3	0,7	-0,720000	0,552000	3,183904	0,216738	8267,556364
0,4	0,6	-0,600000	0,744000	2,559136	0,353442	9609,949091
0,5	0,5	-0,456000	0,888000	2,008480	0,536754	10341,818182
0,6	0,4	-0,264000	0,888000	1,524640	0,782514	10404,911515
0,7	0,3	-0,072000	1,104000	0,983200	1,182786	9676,800000
0,8	0,2	0,192000	1,176000	0,516640	1,771986	7951,854545
0,9	0,1	0,504000	1,248000	0,159520	2,643954	4902,290909

The use of the relative norm, as already outlined, is of great importance in the conditions in which the criteria have different dimensional units or different orders of magnitude. If the order of magnitude differs significantly, then an absolute difference of the form $(f_i(X)-f_i(X^{ideal}))$ will force the optimization procedure to minimize that greater value difference, neglecting the smaller size differences. If we divide the difference $(f_i(X) - f_i(X^{ideal}))$ to $f_i(X^{ideal})$, then the orders of magnitude will be equal, which will force the optimization procedure to act equally on each criterion. In addition, weighting coefficients β_i allow us to incline the balance in favor of one or the other of the criteria.

The coefficient β_1 is related to the criterion f_1 while β_2 is related to criterion f_2. If we begin the analysis with $\beta_1 = 0.1$ and $\beta_2 = 0.9$, it means that the importance of the criterion f_1 is very small compared to the criterion f_2. It can be noticed that the solution obtained is located, closer to the optimal position of criterion f_2, obtained by monocriterial optimization (see line 2 of Table 7.1 and line 1 of Table 7.3).

If instead we assign greater importance to criterion f_1 than to the second criterion, then the solution obtained is located, as a position, closer to the optimal position of criterion f_1 obtained by monocriterial optimization (see line 1 of Table 7.1 and the last line of Table 7.3). For values of the weighting coefficients β_i, located between these two extreme values, the solutions will occupy intermediate positions between the two obtained initially. Fig. 7.4 and Fig. 7.5 show this analysis for the values in Table 7.3.

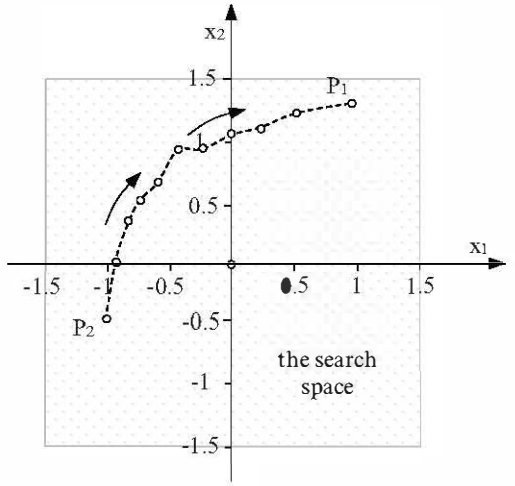

Fig. 7.4 The weights' influence on the location of the optimum.

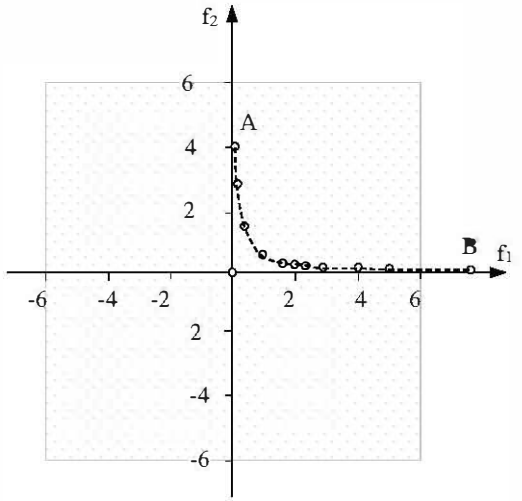

Fig. 7.5 The weights' influence on *Pareto-optimal* set.

As in Fig. 7.3, the points A and B represent the images of the points P_1 and P_2 through the objective function relative to norm $F^{(p)}$, where $p = 1$.

7.4 The *Pareto-optimal* set

If the domain Ω_F is known, it is posssible to separate the points that belong to the *Pareto-optimal* set. Let us consider a multicriterial optimization problem which depends on two criteria, f_1 and f_2. From the definition of the *Pareto-optimal* set, it is known that the points that belong to this set cannot improve one criterion without degrading at least another criterion.

```
%
%          Find the Pareto-optimal set for
%      a group of randomly generated points
%
% n is the number of points whose coordinates (x, y)
% are randomly generated
n = 10;
% initialize the strings x and y
x = zeros(1,n);
y = zeros(1,n);
% allocate for the elements of the strings x and y,
% subunit values, randomly generated
for i = 1:1:n
    x(i) = rand;
    y(i) = rand;
end
% The vector s(n) has all elements equal to one.
% The value 1 is reserved to points that belong to
% Pareto-optimal set
s = zeros(1,n);
for i=1:1:n
    s(i) = 1;
end
% looking for point A
xA = x(1);
yA = y(1);
rangA = 0;
for i = 2:1:n
    if x(i) < xA
        xA = x(i);
        yA = y(i);
        rangA = i;
    elseif (x(i) == xA && y(i) < yA)
        xA = x(i);
        yA = y(i);
```

```
36          rangA = i;
37      end
38  end
39  % looking for point B
40  xB = x(1);
41  yB = y(1);
42  rangB = 0;
43  for i = 2:1:n
44      if y(i) < yB
45          xB = x(i);
46          yB = y(i);
47          rangB = i;
48      elseif (y(i) == yB && x(i) < xB)
49          xB = x(i);
50          yB = y(i);
51          rangB = i;
52      end
53  end
54  mPO = 0;  % the number of points in the Pareto-optimal
55  set,
56            % including A and B
57  for i = 1:1:n
58      for j = 1:1:n
59          if x(i)>x(j) && y(i)>y(j)
60              s(i) = 0;
61          end
62      end
63  end
64  for i=1:1:n
65      if s(i)==1
66          mPO = mPO + 1;
67      end
68  end
69  xPO = zeros(1,mPO);
70  yPO = zeros(1,mPO);
71  j = 1;
72  for i = 1:1:n
73      if s(i) == 1
74          xPO(j) = x(i);
75          yPO(j) = y(i);
76          j = j + 1;
77      end
78  end
79  % the graphic of the total set of points
80  % mean time, mark the extreme points of the set
```

```
81  subplot(1,2,1);
82  scatter(x,y);
83  subplot(1,2,2);
84  scatter(xPO,yPO);
85  hold on;
86  scatter(xA,yA,'markerfacecolor','y',
87  'markeredgecolor','r');
88  scatter(xB,yB,'markerfacecolor','g',
89  'markeredgecolor','r');
90  hold off;
91  %
```

Code 7.4 *pareto.m*.

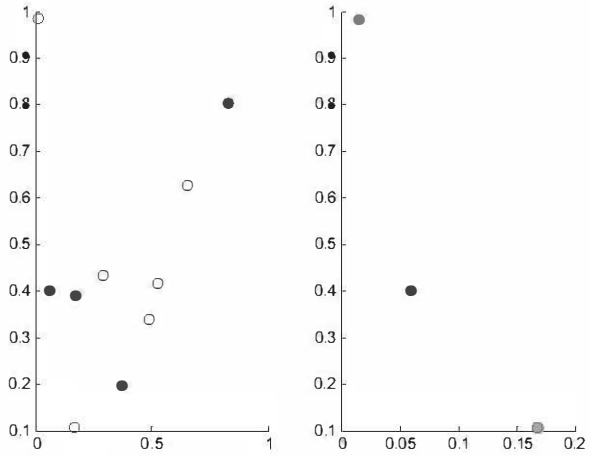

Fig. 7.6 A set of randomly generated points and their corresponding *Pareto-optimal* set.

On the left side of the Fig. 7.6 one can notice a set of ten points that were randomly generated (see lines 13 to 16 from Code 7.4). On Code 7.4, this set of points can be adjusted at line 7. The points representing the limits of the *Pareto-optimal* set and that were noted A and B are determined from lines 24 to 38, and from lines 39 to 53, respectively. Each point generated by this program is represented in Fig. 7.6 by a small empty circle. The limits of the *Pareto-optimal* set are represented by filled circles. Each time the program is executed, another set of points is generated and another *Pareto-optimal* set is located.

If in Code 7.4, lines 8 to 16 are replaced by lines 1 to 14 from Code 7.5, then instead of a randomly generated set of points, n points will be generated on a circle of radius r. The solution found by the program *pareto.m* is in Fig. 7.7.

```
1     % Generating points on the circumference of a circle
2     alpha_i = 2*pi/n;
3     alpha = 0;
4     r = 10;
5     % Initializing vectors x and y
6     x = zeros(1,n);
7     y = zeros(1,n);
8     % Alocating randomly subunit values to vectors x and y
9     for i = 1:1:n
10        x(i) = r*cos(alpha);
11        y(i) = r*sin(alpha);
12        alpha = alpha + alpha_i;
13    end
14    %
```

Code 7.5 Update for *pareto.m*.

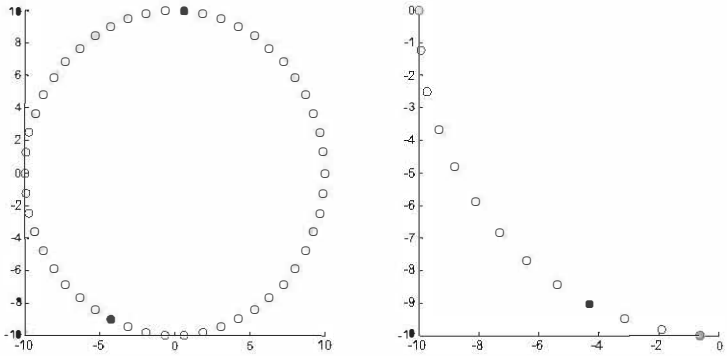

Fig. 7.7 A set of points on a circle and its corresponding *Pareto-optimal* set.

7.5 Conclusions

This chapter dealt with problems that have several optimization criteria. An issue here is understanding the *Pareto-optimal* set concept. Multicriteria optimization takes place in a few steps. As we have learned,

the problem requires separately solving each criterion. Then the mathematical model has to be designed with all criteria taken into account. The literature is rich in papers and books dealing with this subject (Osyczka,1992). In case of the global criterion method discussed here, care must be taken regarding the criterion norm value, which is case specific, and then the testing of the numerical results should be performed.

8.

TRAVELING SALESMAN PROBLEM

8.1 Introduction

Let us consider n points $(p_1, p_2, ..., p_n)$ in a plane and a matrix called d, of size $(n \times n)$, whose elements are distances between the points p_i and p_j from that plane. A path is defined as a closed route that visits each point, a single time, and returns to the starting point. We are interested in finding the path of minimum length with these properties (i.e. the hamiltonian path). If the points are identified by the coordinates (x_i,y_i) and the distances between them respect the conditions $d(p_i,p_j) \neq 0$ for any $i \neq j$ and $d(p_i,p_i) = 0$, the problem is considered symmetrical if $d(p_i,p_j) = d(p_j,p_i)$. Alternatively, if $d(p_i,p_j) \neq d(p_j,p_i)$ the problem is considered asymmetrical. For such a situation, let us imagine that all n points represent cities, and the city p_i is located in a valey, while the city p_j is located somewhere high on a mountain. Rating the distances in liters of fuel consumed, getting from p_i to p_j, will entail a higher fuel consumption than getting from p_j to p_i. Therefore, the major difference between symmetric and asymmetric problems is the matrix of distances. However the main principles on which the optimization algorithms are constructed remain the same for both types of problems.

For the symmetrical problem with n points, any permutation represents a possible solution (a path), and there are $n!$ permutations. If we take a careful look at the pemutation $(p_1, p_2, p_3, p_4, ..., p_{n-2}, p_{n-1}, p_n, p_1)$, we notice that $(n-1)$ different permutations exist representing the same path:

p_1	p_2	p_3	p_4	...	p_{n-2}	p_{n-1}	p_n	p_1	is similar to
p_2	p_3	p_4	p_5	...	p_{n-1}	p_n	p_1	p_2	or to
p_3	p_4	p_5	p_6	...	p_n	p_1	p_2	p_3	and so on.

Also, the path:

p_1	p_2	p_3	p_4	...	p_{n-2}	p_{n-1}	p_n	p_1	is similar to
p_1	p_{n-1}	p_{n-2}	p_{n-3}	...	p_4	p_3	p_2	p_1	read backwards

These last two permutations represent the same path traced in opposite directions, their lengths being the same. This is why, in the case of symetrical problems, the total number of different paths N_T is:

$$N_T = \frac{(n-1)!}{2}; \tag{8.1}$$

With increasing n, N_T increases so much (see Table 8.1), that an exhaustive exploration becomes impossible, despite the high performances of computers.

Table 8.1 The evolution of N_T for different numbers of cities.

N	N_T
8	2.520
9	20.160
10	181.440
11	1.814.400
12	19.958.400
13	239.500.800
14	3.113.510.400
15	43.589.145.600

The time needed to solve the problem depends on the number of points, the type of algorithm (its degree of complexity) and the speed of the computer. Many years of research in this field have led to all the more impressive solutions. Thus, concerning the solutions of the TSP, the performance over the last 40 years increased from n = 318 (Crowder & Padberg,1980) to n = 2392 (Padberg & Rinaldi, 1987, 1991) or n = 7397 (Applegate & Cook, 1993) and so on.

There are two alternatives of solving *Traveling Salesman Problem* (*TSP*). They involve either designing an algorithm, usually a greedy one, to compute a valid path step-by-step or an algorithm that starts from a valid solution and, based on a local search, improves on that particular solution. Algorithms satisfying the above principles are called *heuristic algorithms*. The *heuristic algorithms* that are able to build, step-by-step, a valid solution are called *constructive heuristics*, while those from the second category are called *improvement heuristics* or *successive augmentation heuristics*. Within *constructive heuristic* algorithms, the most well known are *NN* or *Nearest Neighbor* (Rosenkrantz et al,1977), *Greedy* (Bentley, 1992), *Clarke-Wright* (Frieze, 1979) *Christofides*

(Lawler et al, 1985) etc. The group of *successive augmentation heuristics* includes methods such as *2-opt, 3-opt* (Croes, 1986; Knox, 1994), *Simulated Annealing* with different alternatives (Kirkpatrik et al, 1983; Metropolis et al, 1953), *genetic algorithms* (Valenzuela & Jones, 1994), *neural networks* with alternatives (Angeniol et al, 1988; Peterson, 1990; Peterson & Söderberg, 1989; Simmen, 1991; Xu & Tsai, 1991).
These algorithms usually employ the following general routines:

a) random generation of a solution S, a putative valid path;
b) the attempt to find an improved solution S_i, by applying some transformation to solution S;
c) if an improved solution was found (i.e. *length of* S_i < *length of* S), then S_i replace S and step b is reiterated;
d) if an improved solution cannot be found, then S becomes a local optimum. The procedure repeats step b until the time limit initially selected is reached, or the solution satisfies the quality criterion also initially selected.

8.2 Conventional methods to solve TSP

An important application[1] of *TSP* is the design of the minimum length tool path to drill *Printed Circuit Boards* (*PCB*). Starting from this application, in several industrial companies, in the absence of a computer program that designs a good solution for the tool path, the problem is simply solved minimalistically, for example by sorting holes' coordinates horizontally or vertically.

8.2.1 Sorting horizontally

In this instance, point coordinates are sorted by position, starting with points from the *lower-left* corner of the *PCB*. The program that builds the path based on these considerations is *convH.m*. The coordinates *(x,y)* of points are found in a text file that can be seen in Fig. 8.1. In the first column of the list above the point index can be found, while the second and third columns correspond to the x and y coordinates. To sort horizontally the coordinates *(x,y)* of all n points in the list, we have to

[1] Although drilling *PCB*s represents an important application of the *Traveling Salesman Problem*, other several applications exist such as computer wiring, X-Ray crystallography, vehicle routing, mask plotting in *PCB* production, the order-picking problem in warehouses and so on.

verify if in the sequence $p_1, p_2, p_3, \ldots, p_n$, any two consecutive points p_i, p_{i+1} satisfy the conditions:

$$y_{p(i)} > y_{p(i+1)}; \tag{8.2}$$

and

$$y_{p(i)} = y_{p(i+1)} \quad \textit{and} \quad x_{p(i)} \geq x_{p(i+1)}; \tag{8.3}$$

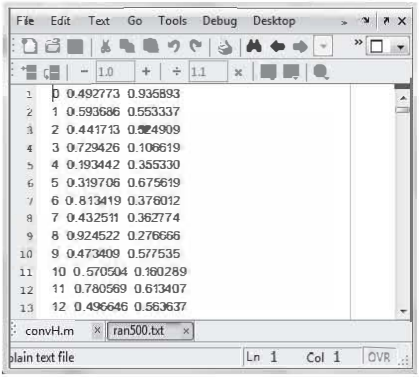

Fig. 8.1 The input data format.

If one of the conditions expressed by Eq. 8.2 and Eq. 8.3 is not satisfied, a change of position between point p_i and p_{i+1} occurs. A counter keeps track if in the data string a position change between two succesive elements occurred, and in this case, it allows to continue the data string sorting operation commands. The sorting is considered complete when the counter reaches zero. Code 8.1 shows the *convH.m* program that creates this sorted list of n points in the plane, according to the criteria mentioned by Eq. 8.2 and Eq. 8.3.

```
%                  CONVENTIONAL METHOD
%            (sorting the data horizontally)
%
data = load('ran50.tsp');
% count the file size
[n,~,~] = size(data);
x = zeros(n+1,1);
y = zeros(n+1,1);
for i=1:1:n
```

```
    x(i) = data(i,2);
    y(i) = data(i,3);
end
% open a file to save results
fp = fopen('resultsH.txt','w');
fprintf(fp,'         CONVENTIONAL METHOD\n');
fprintf(fp,'   (sorting the data horizontally)\n\n');
% order of points vector
order = zeros(1,n+1);
for i=1:n
    order(1,i) = i;
end
% Begin from left-lower corner
count = 1;   % to enter the "while" block
while (count ~=0)
    count = 0;
    for i = 1:n
        if(y(i)>y(i+1))
            count = count + 1;
            sx = x(i);
            sy = y(i);
            so = order(1,i);
            x(i) = x(i+1);
            y(i) = y(i+1);
            order(1,i) = order(1,i+1);
            x(i+1) = sx;
            y(i+1) = sy;
            order(1,i+1) = so;
        end
         if(y(i)==y(i+1) && x(i)>=x(i+1))
            count = count + 1;
            sx = x(i);
            sy = y(i);
            so = order(1,i);
            x(i) = x(i+1);
            y(i) = y(i+1);
            order(1,i) = order(1,i+1);
            x(i+1) = sx;
            y(i+1) = sy;
            order(1,i+1) = so;
         end
    end
end
% last point in order is the same with the first one
order(1,1) = order(1,n+1);
```

```
% calculate the path length
pathLength = 0;
for i = 1:n
    dist = sqrt((x(i+1)-x(i))^2+(y(i+1)-y(i))^2);
    pathLength = pathLength + dist;
end
% now add to the distance the segment
% from end point to the start point
dist = sqrt((x(n+1)-x(1))^2+(y(n+1)-y(1))^2);
pathLength = pathLength + dist;
fprintf(fp,'the path length = %f\n',pathLength);
fprintf(fp,'\n The order of points:\n');
fprintf(fp,'(read it line by line)\n\n');
for i = 1:n
    fprintf(fp,'%d  ',order(1,i));
    if ceil(i/10) == floor(i/10)
        fprintf(fp,'\n');
    end
end
fprintf(fp,'%d\n',order(1,n+1));
%
u = zeros(n+1,1);
v = zeros(n+1,1);
for i=1:1:n
    u(i) = x(i);
    v(i) = y(i);
end
u(n+1) = u(1);
v(n+1) = v(1);
% display the path
line(u,v);
% close the file
fclose(fp);
%
```

Code 8.1 *convH.m*.

After all points are sorted according to the *lower-left* criterion, the last point is joined with the first one in order to close the circuit. Fig. 8.2 shows how this method works for a test problem with 500 randomly generated points within the interval [0,1]. To run the *convH.m* program from Code 8.1 for another list of points, that list should be created acording to the example in Fig. 8.1. At line 4, the name of the *ASCII* text file that contains the coordinates (x_i,y_i) of n points in a plane must replace the actual file name. The *convH.m* program saves, the *order* vector of the

points in the *results.txt* file which is needed when a *CNC* program for a machine-tool must be designed. The vectors *u(i)* and *v(i)*, with *i = 1, 2, ..., (n+1)*, contain the (x_i, y_i) coordinates of all *n* data points, and the point *(n+1)* has the same coordinates with the first point, where the path starts. Lines 68 to 74 print the order of point to *results.txt* file in a series of lines with 10 points per each line. The method excels in principle and programming simplicity, but the result is not quite astonishing.

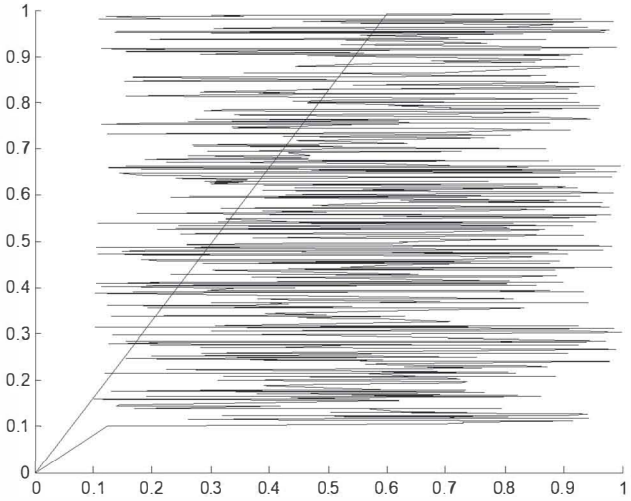

Fig. 8.2 The path generated by *convH.m* for the *ran500* problem.

8.2.2 Sorting vertically

In this method, we are applying the same principle as in the previous case, but this time, the points are sorted in the vertical direction. We start from the same lower-left corner and we build the path point-by-point and at the end of the sorting process we connect the final point with the initial one. For a vertical path through all points, we must meet the following conditions:

$$x_{p(i)} > x_{p(i-1)}; \tag{8.4}$$

and

$$x_{p(i)} = x_{p(i+1)} \quad and \quad y_{p(i)} \geq y_{p(i+1)}; \tag{5}$$

Code 8.2 illustrates the program *convV.m* that solves this type of sorting. The program saves the vector that contains the index order of sorted points in the *results.txt* file. Because this order is the goal of the search, it can be handled in the required style. For instance, the coordinates (x_i, y_i) in sorted order can be used to generate the *CNC* code for a drilling machine-tool directly.

```
%                  CONVENTIONAL METHOD
%              (sorting the data vertically)
%
data = load('ran50.tsp');
[n,~,~] = size(data);
x = zeros(n+1,1);
y = zeros(n+1,1);
for i=1:1:n
    x(i) = data(i,2);
    y(i) = data(i,3);
end
% open a file to save results
fp = fopen('resultsV.txt','w');
fprintf(fp,'         CONVENTIONAL METHOD\n');
fprintf(fp,'    (sorting the data vertically)\n\n');
% order of points vector
order = zeros(1,n+1);
for i=1:n
    order(1,i) = i;
end
% Begin from left-lower corner
count = 1;   % to enter the "while" block
while (count ~=0)
    count = 0;
    for i = 1:n
        if(x(i)>x(i+1))
            count = count + 1;
            sx = x(i);
            sy = y(i);
            so = order(1,i);
            x(i) = x(i+1);
            y(i) = y(i+1);
            order(1,i) = order(1,i+1);
            x(i+1) = sx;
            y(i+1) = sy;
```

```
            order(1,i+1) = so;
        end
        %
        if(x(i)==x(i+1) && y(i)>=y(i+1))
            count = count + 1;
            sx = x(i);
            sy = y(i);
            so = order(1,i);
            x(i) = x(i+1);
            y(i) = y(i+1);
            order(1,i) = order(1,i+1);
            x(i+1) = sx;
            y(i+1) = sy;
            order(1,i+1) = so;
        end
    end
end
% last point in order is the same as the first one
order(1,1) = order(1,n+1);
% path length calculation
pathLength = 0;
for i = 1:n
    dist = sqrt((x(i+1)-x(i))^2+(y(i+1)-y(i))^2);
    pathLength = pathLength + dist;
end
% now add the distance from end point
% to the start point
dist = sqrt((x(n+1)-x(1))^2+(y(n+1)-y(1))^2);
pathLength = pathLength + dist;
%
fprintf(fp,'the path length = %f\n',pathLength);
fprintf(fp,'\n The order of points:\n');
fprintf(fp,'(read it line by line)\n\n');
for i = 1:n
    fprintf(fp,'%d  ',order(1,i));
    if ceil(i/10) == floor(i/10)
        fprintf(fp,'\n');
    end
end
fprintf(fp,'%d\n',order(1,n+1));
%
u = zeros(n+1,1);
v = zeros(n+1,1);
for i=1:1:n
    u(i) = x(i);
```

```
81        v(i) = y(i);
82    end
83    u(n+1) = u(1);
84    v(n+1) = v(1);
85    % display the path
86    line(u,v);
87    % close the file
88    fclose(fp);
89    %
```

Code 8.2 *convV.m*.

Fig. 8.3 shows the path generated by *convV.m*, for a problem with 500 points that were randomly generated from the interval [0,1].

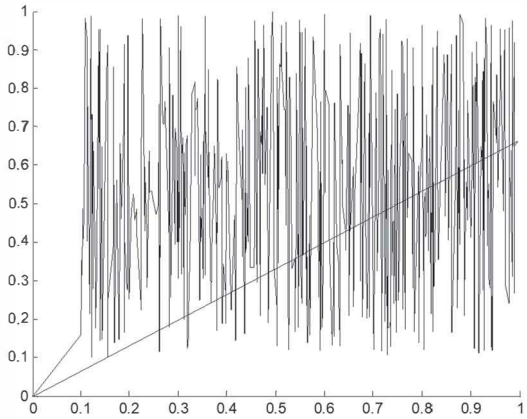

Fig. 8. 3 The path generated by *convV.m* for *ran500* problem.

8.3 Nearest Neighbor

The main idea of the *Nearest Neighbor* (*NN*) method is as simple as those of the two methods previously described. We chose a starting point randomly, then proceed to the nearest point not yet visited. In doing this, the objective is to build a so called *matrix of distances*, denoted ***d***, with $n \times n$ elements. This matrix is symmetrical across its main diagonal. This case corresponds to the symmetric problem where the distance from point p_i to point p_{i+1} is equal to the distance from point p_{i+1} to point p_i. The

succession of points on the path emerging in this way obviously depends on the starting point.

As one can notice from Fig. 8.4, the last step of the path building represents the connection between the first point and the last one in the list, a segment that usually intersects other segments of the path. Removing segments intersections leads always to a shorter path length. Therefore, the solution based on the *Nearest Neighbor* principle (see Fig. 8.75.a) having segment intersections is longer than the path from Fig. 8.5.b with no such intersections. This problem of intersections removing will be addressed later, since their removal improves the solution because each removed intersection means reduction in path length. It should be noted that the succession of the points along the path depends on the starting point. If the path starts from a different point, the optimal path might be a different one.

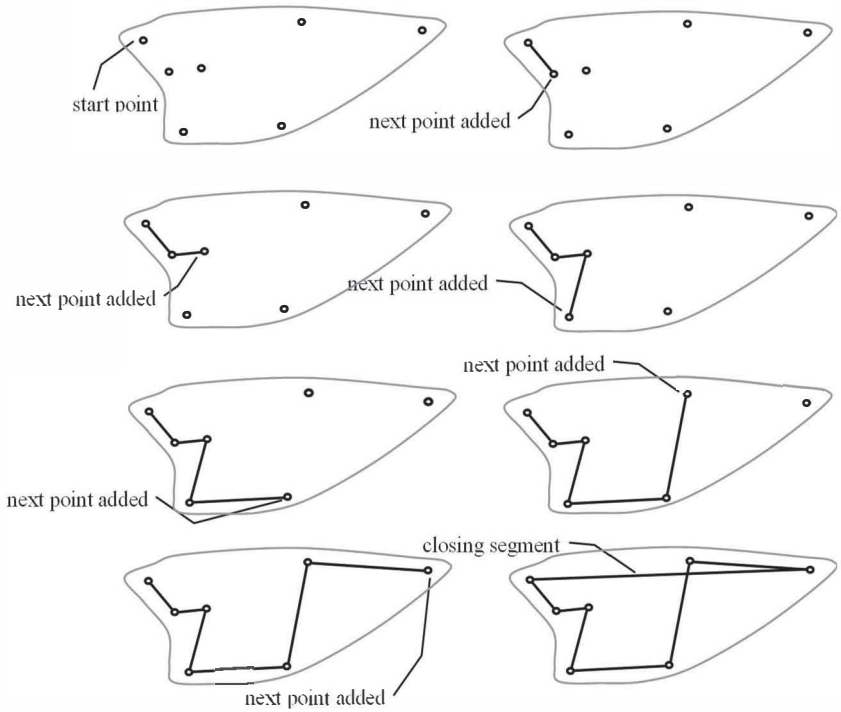

Fig. 8.4 Path building, based on the *NN* principle.

In the following, the source code for a *NN.m* program applying the *Nearest Neighbor* method is shown. Running this program, a path that traverses n points in a plane can be calculated.

```
%       TRAVELING SALESMAN PROBLEM
%              Nearest Neighbor (NN)
%
% read input data
data = load('ran10.txt');
% open a file to save results
fp = fopen('results.txt','w');
fprintf(fp,' TRAVELING SALESMAN PROBLEM\n');
fprintf(fp,'          Nearest Neighbor (NN)\n\n');
% count the file size
[n,~,~] = size(data);
u = zeros(n);
v = zeros(n);
for i=1:1:n
    u(i) = data(i,2);
    v(i) = data(i,3);
end
% build matrix 'd' of distances between cities
% allocate memory for matrix
d = zeros(n,n);
% the main diagonal of matrix d is zero
for i=1:1:n-1
    for j=i+1:1:n-1
        if i~=j
            d(i,j) = sqrt((u(i)-u(j))*(u(i)-
u(j))+(v(i)-v(j))*(v(i)-v(j)));
            d(j,i) = d(i,j);
        end
    end
end
% calculate the distances from last city to the others
for i=1:1:n
    d(i,n) = sqrt((u(i)-u(n))*(u(i)-u(n))+...
        (v(i)-v(n))*(v(i)-v(n)));
    d(n,i) = d(i,n);
end
% define a structure called 'city'.
% the order of cities in the structure is as the input
% data have been read.
city = struct ([]);
for i=1:1:n
```

```
42      city(i).index = i;
43      city(i).x = u(i);
44      city(i).y = v(i);
45      city(i).mark = 0;
46  end
47  % the cities already visited have the "city(i).mark =
48  1"
49  order  = zeros(1,n);
50  % Choose the first city for start
51  order(1) = 1;
52  city(1).mark = 1;
53  % initialize dmin with a high value
54  dMin = 1.e10;
55  % build the order to go through cities
56  % the current index is ic = 1
57  % (it means that we are in city no.1)
58  ic = 1;
59  for k = 2:n
60      for i = 1:n
61          if ic~=i
62              if city(i).mark==0
63                  if d(ic,i)<dMin
64                      dMin = d(ic,i);
65                      ic = i;
66                  end
67              end
68          end
69      end
70      order(k) = ic;
71      city(ic).mark = 1;
72      dMin = 1e10;
73  end
74  % calculation of the path length
75  pathLength = 0.0;
76  for i=1:1:n-1
77      pathLength = pathLength + d(order(i),order(i+1));
78  end
79  % close the path with the segment d(n,1)
80  pathLength = pathLength + d(order(n),order(1));
81  % draw the path
82  u = zeros(n+1);
83  v = zeros(n+1);
84  for i=1:1:n
85      u(i) = city(order(i)).x;
86      v(i) = city(order(i)).y;
```

```
 87  end
 88  u(n+1) = u(1);
 89  v(n+1) = v(1);
 90  line(u,v);
 91  % the complete path
 92  orderNN = zeros(1,n+1);
 93  for i = 1:n
 94      orderNN(1,i) = order(i);
 95  end
 96  orderNN(1,n+1) = order(1);
 97  fprintf(fp,'\n The order of points:\n');
 98  fprintf(fp,'(read it line by line)\n\n');
 99  for i = 1:n
100      fprintf(fp,'%d  ',orderNN(1,i));
101      if ceil(i/10) == floor(i/10)
102          fprintf(fp,'\n');
103      end
104  end
105  fprintf(fp,'%d\n',orderNN(1,n+1));
106  % save the path length to file
107  fprintf(fp,'the path length = %f\n',pathLength);
108  % close the file
109  fclose(fp);
110  %
```

Code 8.3 *NN.m*.

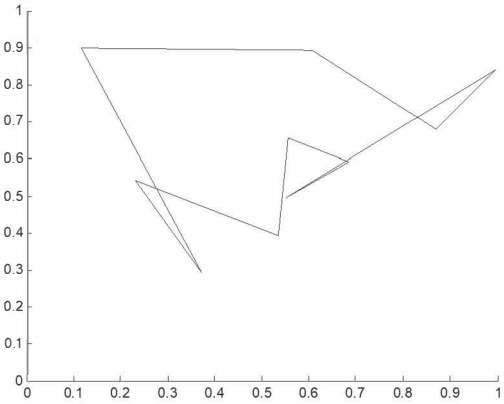

Fig. 8.5 Ten point path done by *NN.m*.

As input data we use the Cartesian coordinates of ten points randomly generated in the interval [0,1], from the file *ran10.txt*. From Fig. 8.5 we can notice that the path yields three segment intersections. According to the previous statement, the removal of segment intersections must lead to a shorter path length. However, it should not be forgotten that the removal of an intersection may lead to new segment intersection.

Fig. 8.6 shows the ten point path after removal of the segment intersections. The entire procedure for removing segments is detailed in the section 8.4.

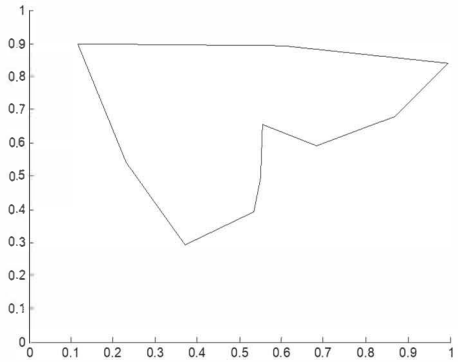

Fig. 8.6 The path from Fig. 8.5, after applying the *2Opt* procedure.

8.4 Determining the intersection of two segments

To verify if two segments intersect each other, it suffices to check whether one of these segments crosses the other one. As a rule, if one segment $\overline{p_1 p_2}$ crosses a straight line it follows that points p_1 and p_2 are on different sides of that line. A particular scenario arises when both points are crossed by that line.

Therefore, two straight line segments intersect each other when one of the following conditions is met:

a) each segment crosses the line of the other segment;

b) one end of a segment is collinear with the other segment.

Determining the intersection of segments is based on the cross product of two vectors $\overline{p_1}$ and $\overline{p_2}$. This is represented geometrically by the signed area of the parallelogram in Fig. 8.7, given by points (0,0), p_1, p_2, $p_1+p_2 = (x_1+x_2, y_1+y_2)$.

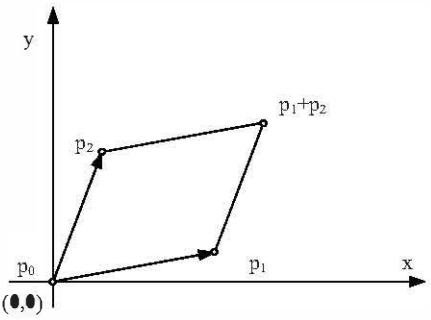

Fig. 8.7 The signed area of the parallelogram defined by the cross product of vectors $\overline{p_1}$ and $\overline{p_2}$.

This area is equivalent to the value of the determinant of the matrix of vectors $\overline{p_1}$ and $\overline{p_2}$:

$$p_1 \times p_2 = \begin{vmatrix} x_1 & x_2 \\ y_1 & y_2 \end{vmatrix} = x_1 y_2 - x_2 y_1 = -p_2 \times p_1; \tag{8.6}$$

If $p_1 \times p_2$ is positive then the rotation from $\overline{p_2}$ to $\overline{p_1}$ is clockwise. If $p_1 \times p_2$ is negative, then the rotation from $\overline{p_2}$ to $\overline{p_1}$ is counterclockwise. If $p_1 \times p_2 = 0$, then p_1 and p_2 are collinear. To check the orientation of the vectors $\overline{p_0 p_1}$ and $\overline{p_0 p_2}$ to one another (point p_0 is by convention the origin of the Cartesian system; otherwise it can be simply translated into the origin) we calculate the cross product:

$$\begin{aligned}(p_1 - p_0) \times (p_2 - p_0) = (x_1 - x_0) \times (y_2 - y_0) - \\ (x_2 - x_0) \times (y_1 - y_0);\end{aligned} \tag{8.7}$$

If the result is positive, the rotation from $\overline{p_0 p_2}$ to vector $\overline{p_0 p_1}$ is clockwise. If the result is negative, then the rotation from $\overline{p_0 p_2}$ to vector $\overline{p_0 p_1}$ is counterclockwise. For more details regarding this procedure consult (Cormen, 2003).

8.5 Removing segment intersections

If a path contains segment intersections, there is always an alternative path, without segment intersections, of shorter length. This statement is

based on the triangle rule. The rule states that in each triangle, the sum of two edges is greater than the length of the third edge.

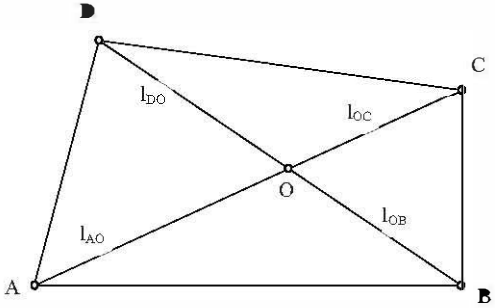

Fig. 8.8 Diagonals in a quadrilateral *ABCD*.

Let us consider an ordinary quadrilateral *ABCD* (see Fig. 8.12). Point O is the intersection of its diagonals *AC* and *BD*. If we apply the above rule in triangles *AOD* and *BOC*, we get:

$$l_{AO} + l_{DO} > l_{AD}; \tag{8.8}$$

$$l_{OC} + l_{OB} > l_{BC}; \tag{8.9}$$

Adding the inequalities above, term by term and we get:

$$(l_{AO} + l_{OC}) + (l_{BO} + l_{OD}) > l_{AD} + l_{BC}; \tag{8.10}$$

Eq. 8.10 is equivalent to:

$$l_{AC} + l_{BD} > l_{AD} + l_{BC}; \tag{8.11}$$

If we apply the same rule to triangles *AOB* and *COD* we get:

$$l_{AC} + l_{BD} > l_{AB} + l_{CD}; \tag{8.12}$$

In other words, in any ordinary quadilateral (convex or concave), the sum of diagonals is always greater than the sum of two opposite edges. Since two intersecting segments can be seen as diagonals in a quadrilateral, removal of the intersection will lead to a new path of shorter length. This is the basic idea of the *twopt.m* procedure.

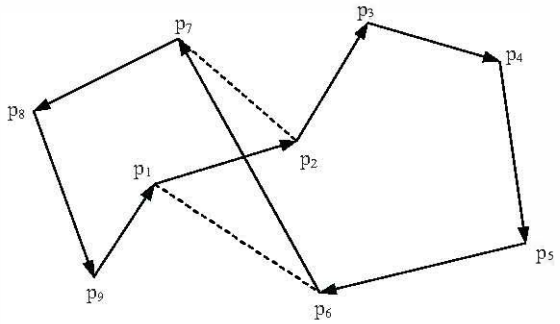

Fig. 8.9 A path with intersecting segments.

In Fig. 8.9, segment $\overline{p_1p_2}$ intersects segment $\overline{p_6p_7}$. If we replace these segments with $\overline{p_1p_6}$ and $\overline{p_2p_7}$, respectively, the new path will have a shorter length than the previous path. The initial path was:

$$p1, p2, p3, p4, p5, p6, p7, p8, p9, p1 \tag{8.13}$$

After applying *twoOpt.m* procedure, the new path is:

$$p1, p6, p5, p4, p3, p2, p7, p8, p9, p1 \tag{8.14}$$

It should be noted that several changes took place in the initial order. On the one hand, path segments $\overline{p_1p_2}$ and $\overline{p_6p_7}$ were replaced by segments $\overline{p_1p_6}$ and $\overline{p_2p_7}$. At the same time, the reading order of points p_3, p_4, p_5 was reversed. The program that eliminates segment intersections, denoted *removeIntersections.m*, calls six other functions. These functions are *segmentsIntersection.m*, *twoOpt1.m*, *translation.m*, *direction.m*, *onSegment.m* and *findLegth.m*.

Within the *removeIntersections.m* program, a set of n = 50 points, uniformly distributed in the interval $[a,b]$ = [1,100], are randomly generated (in this case, the range [1,100] was arbitrarily chosen, but margins a and b may take any other values, as desired). The initial path version is given by the random order in which the n points were generated. The length l_1 of the path is then calculated. To detect whether a path intersection exists, the *removeIntersections.m* program checks the position of segment $\overline{p_1p_2}$ relative to segments $\overline{p_3p_4}$, $\overline{p_4p_5}$, ..., $\overline{p_{n-1}p_n}$. If an intersection is found, then the *twoOpt1.m* function removes it and the program *removeIntersections.m* resumes testing for path intersections. To make sure the verification is complete, after the first segment has been

checked against all other segments in the list, a translation of points is performed, so that p_2 becomes the new p_1, point p_3 becomes the new p_2 and so on. After the translation of points, the detection of segment intersections continues with the same segment $\overline{p_1p_2}$ relative to all other segments. In this way, this new segment $\overline{p_1p_2}$ is in fact segment $\overline{p_2p_3}$ from the previous list, before translation. The *removeIntersections.m* program performs something on the order of n^2 tests for path intersections. In fact, in (Johnson & McGeoch, 1997) it has been shown that in order to eliminate all segment intersections, we must call function *twoOpt1.m* n^3 times, where n is the number of points in the list.

```
%               INTERSECTIONS REMOVAL
% This program generates a set of n points randomly
% and then finds a path that visits each point once.
% It removes the segment intersections in order to
% make the path as short as possible. To optimize
% the path, only the procedure 2Opt is used.
%
n = 32;
% the 'n' points are generated inside the interval
[a,b].
a = 1;
b = 100;
x = a + (b-a).*rand(1,n);
y = a + (b-a).*rand(1,n);
% plot the path in the order the points were generated
%
subplot(1,2,1);
plot(x,y);
% l1 is the path length that has segments intersected
l1 = findLength(x,y,n);
count = 0;
%
for v = 1:n
    for j = 1:n
        for i = 3:n-1
            r = ...
              segmentsIntersection(x(1),y(1),x(2),...
              y(2),x(i),y(i),x(i+1),y(i+1));
            if r==1
                count = count+1;
                [x,y] = twoOpt(x,y,i);
            end
            % the translation of points
```

```
            [x,y] = translation(x,y,n);
        end
    end
end
%
% l2 is the path length whose segments intersections
% were removed
l2 = findLength(x,y,n);
% plot the path without segments intersections
subplot(1,2,2);
% connect the last and the first points
x(n+1) = x(1);
y(n+1) = y(1);
plot(x,y);
%
```

Code 8.4 *removeIntersections.m*.

The function *twoOpt1.m* eliminates the intersection between two segments according to the principle described in Fig. 8.9. In addition, this function reverses the reading order of points between the intersected segments.

```
%         twoOpt1.m
% This function removes segment intersections
% It takes as input arguments two vectors x and y
% with point coordinates and the index i of the
% segment that generates the intersection
function [x,y] = twoOpt(x,y,i)
%
m = i-1;
j = floor(m/2);
%
for k = 1:j
    xTemp = x(2+k-1);
    yTemp = y(2+k-1);
    x(2+k-1) = x(2+m-k);
    y(2+k-1) = y(2+m-k);
    x(2+m-k) = xTemp;
    y(2+m-k) = yTemp;
end
%
return;
end
%
```

Code 8.5 *twoOpt1.m*.

```
%              direction.m
% This function calculates the direction of a
% segment according to other segments.
% The function needs as actual parameters
% the coordinates (x,y) of segments end points.
function [d] = direction(xi,yi,xj,yj,xk,yk)
    d = (xk-xi)*(yj-yi) - (xj-xi)*(yk-yi);
return;
end
%
```

Code 8.6 *direction.m*.

```
%         segmentsIntersection.m
% This function detects the intersection of two
segments
% The function needs the coordinates (x,y) of segments
% end points as actual parameters
% It makes calls to the function 'direction.m'.
%
function[inter] = segmentsIntersection(x1,y1,...
    x2,y2,x3,y3,x4,y4)
%
d1 = direction(x3,y3,x4,y4,x1,y1);
d2 = direction(x3,y3,x4,y4,x2,y2);
d3 = direction(x1,y1,x2,y2,x3,y3);
d4 = direction(x1,y1,x2,y2,x4,y4);
%
if (((d1>0 && d2<0) || (d1<0 && d2>0)) && ...
        ((d3>0 && d4<0) || (d3<0 && d4>0)))
    inter = 1;
        elseif (d1==0 && onSegment(x3,y3,x4,...
                y4,x1,y1))
            inter = 1;
        elseif (d2==0 && onSegment(x3,y3,x4,...
                y4,x2,y2))
            inter = 1;
        elseif (d3==0 && onSegment(x1,y1,x2,...
                y2,x3,y3))
            inter = 1;
        elseif (d4==0 && onSegment(x1,y1,x2,...
                y2,x4,y4))
            inter = 1;
        else
            inter = 0;
end
```

```
return
end
%
```

Code 8.7 *segmentsIntersection.m*.

```
%               direction.m
% This function calculates the direction of a
% segment according to on other segment.
% The function needs the coordinates (x,y) of
% segments end points as actual parameters.
function [d] = direction(xi,yi,xj,yj,xk,yk)
    d = (xk-xi)*(yj-yi) - (xj-xi)*(yk-yi);
return;
end
%
```

Code 8.8 *onSegment.m*.

```
%                translation.m
% This function makes a translation of points,
% so that point1 becomes point2, point2
% becomes point3 and so on.
function [x,y] = translation(x,y,n)
%
xTemp = x(1);
yTemp = y(1);
for k = 2:n
    x(k-1) = x(k);
    y(k-1) = y(k);
end
x(n) = xTemp;
y(n) = yTemp;
%
return;
end
%
```

Code 8.9 *translation.m*.

```
%              findLength.m
% this function calculates the path length
function [length] = findLength(x,y,n)
length = 0;
for i = 1:(n-1)
    length = length + sqrt((x(i)-x(i+1))^2+(y(i)-
y(i+1))^2);
end
```

```
9     return;
10    end
11    %
```

Code 8.10 *findLength.m*.

Based on the Cartesian coordinates of all n points in the file, the *findLength.m* function calculates the total length of the path. This function is called twice, first before the removal of segment intersections and then after applying the procedure *twoOpt*. These two path variants, with and without segment intersections, are displayed in Fig. 8.10, in the case of the file with 50 randomly generated points.

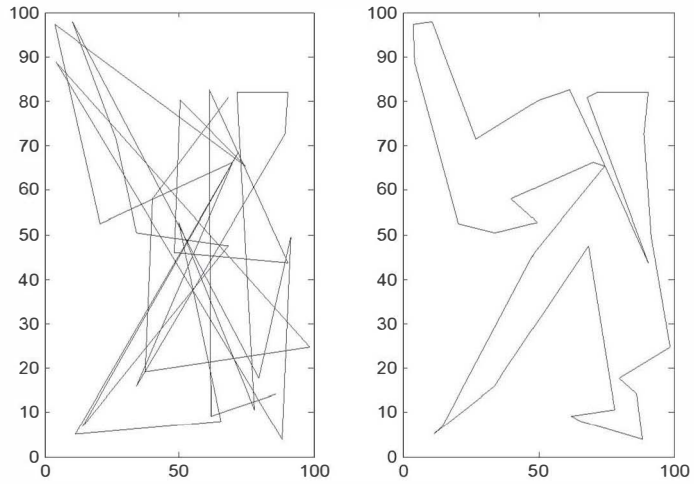

Fig. 8.10 The path with intersected segments and after removing intersections.

8.6 Design of an insertion-type method

Relative to the above mentioned methods, an *insertion-type* method has much better performance, taking into account the way the path is built. The method most often starts with the convex hull of the set of n points, after which all interior points are added one by one to the path. However, the design of the path can also be started from three inner points that will form a triangle. The rest of the points to be added to the path can be found both inside and outside this triangle.

When a point is inserted into the path, its position is chosen so that the path length increase is minimal.

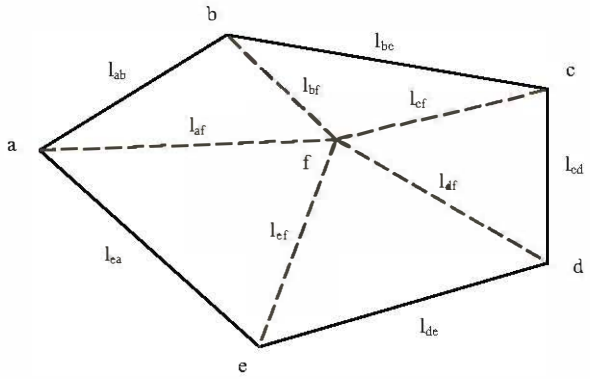

Fig. 8.11 Determining the place to insert point f.

To find out the proper insertion position of the inner point f (see Fig. 8.11), relative to the preliminary path (a,b,c,d,e,a), we have to examine the following sums:

$$\begin{aligned} l_1 &= l_{af} + l_{fb} - l_{ab}; \\ l_2 &= l_{fb} + l_{fc} - l_{bc}; \\ l_3 &= l_{fc} + l_{fd} - l_{cd}; \\ l_4 &= l_{fd} + l_{fe} - l_{de}; \\ l_5 &= l_{fe} + l_{fa} - l_{ea}; \end{aligned} \tag{8.15}$$

If, after evaluating the above sums, we obtain for example:

$$l_1 = min\{l_i |\ i = 1,2,3,4,5\}; \tag{8.16}$$

then the best inserting position of the point in the path sequence is between points a and b. Therefore, the complete path from Fig. 8.11 will be *(a f b c d e a)*. Of course, it is not mandatory to start building the path with the convex hull of points. As already mentioned, the method might start with any three points, consisting a triangle. There may be a series of points inside and outside of that triangle. The insertion of any point located inside or outside the starting triangle is based on the above mentioned principle (i.e. the minimum increase in the path length). Here we have to make the important remark that the order of points on the path depends directly on

the insertion order. As a consequence, the total path length depends also on the insertion order of points. The path building speed is determined by the complexity of the algorithmic calculations. If we start with a preliminary path of three points, the insertion of the fourth point into the path requires three sets of calculations. Based on these calculations we can decide the proper insertion position of the fourth point in the path. Next, four sets of calculations will be needed to insert the fifth point in the path and so on. As a consequence, the total number N_T of calculations is:

$$N_T = 3 + 4 + \cdots + (n-1) = \frac{n(n+1)}{2} - (n+3) = \theta(n^2); \tag{8.17}$$

Thus, the algorithm complexity is $\theta(n^2)$, so that an increase in the number of points entails a quadratic increase in the number of calculations N_T.

Code 8.11 illustrates the *tsp.m* program that builds a path based on an insertion type algorithm. The program starts by building a data structure that contains points' coordinates and an index assigned to each point. This index has the value zero as long as the point is not yet inserted in the path, and takes the value one once it is included in the path. The program generates a random permutation of n points and sorts the points' coordinates according to this permutation. The first three points in this list make up the starting triangle, which is basically the partial path. Relative to the partial path, the program takes the fourth point from the random sequence and inserts it in the partial path according to the principle described above. Then the fifth point in the random sequence is inserted in the partial path of four points and so on. After completing the path, the function *two_opt1.m* is called to remove any of the segment intersections. A counter will take into account the number of detected and removed segment intersections from the path. The program displays the path both before and after the removal of segment intersections.

We must repeat the remark that the path built by this algorithm depends on the order of insertion of points. This fact can be simply shown by rerunning the program, on another randomly generated permutation. Both the length and appearance of the new path will confirm that this is the case. Therefore, this algorithm could be improved by including it in a loop with a specified number of steps. At each step, a different random permutation is generated and a different path with a different length emerges. From the entire set of solutions, it is enough to save the variant with minimum length. Practically, in this way, we combine the principle of the *insertion-type* method with the *Monte Carlo random search*.

The probability to find a good path solution depends on the number of iterations of the program. With increasing number of iterations, the running time increases quadratically. A balance must therefore be struck between the quality of the solution and the duration of its search.
Because the program generates a good solution for small-scale problems, a solution to solve large-scale problems is partitioning. The set of n points is divided into two or more partitions, each partition is solved individually, and then it must be made the junction of the solutions.

```
%
%       TRAVELING SALESMAN PROBLEM
%
% open input data file
data = load('ran500.txt');
%
% open a file to save results
fp = fopen('results.txt','w');
%
fprintf(fp,'       TRAVELING SALESMAN PROBLEM\n\n');
% count file size
[n,~,~] = size(data);
fprintf(fp,'the number of points is n = %d\n',n);
u = zeros(n);
v = zeros(n);
for i=1:1:n
    u(i) = data(i,2);
    v(i) = data(i,3);
end
%
% initialize the matrix of distances between points
d = zeros(n,n);
% build the matrix of distances between points
% main diagonal of matrix 'd' is zero
%
for i=1:1:n-1
    for j=i+1:1:n-1
        if i~=j
            d(i,j) = sqrt((u(i)-u(j))*(u(i)-
u(j))+(v(i)-v(j))*(v(i)-v(j)));
            d(j,i) = d(i,j);
        end
    end
end
%
```

```
% the distances from the last point in the list
for i=1:1:n
    d(i,n) = sqrt((u(i)-u(n))*(u(i)-u(n))+(v(i)-
v(n))*(v(i)-v(n)));
    d(n,i) = d(i,n);
end
%
% define the structure of type point in the order
% as data were read
%
point = struct ([]);
for i=1:1:n
    point(i).index = i;
    point(i).x = u(i);
    point(i).y = v(i);
end
%
% 'swappoint' is used to change the points
% position in the list
swappoint = struct('index',0,'visit',0,'x',0,'y',0);
% generate a random permutation
p = randperm(n);
% pointPermOrder contains the points in the
% permutation order
pointPermOrder = struct([]);
for i=1:1:n
    pointPermOrder(i).index = p(i);
    pointPermOrder(i).x = point(p(i)).x;
    pointPermOrder(i).y = point(p(i)).y;
end
%
% build a preliminary order called 'basicOrder'
% this order is based on insertion principle.
% the ordering starts with first three points from
% 'pointPermOrder'. It continues by adding the rest
% of points from p(n) permutation.
%
basicOrder = struct ([]);
for i=1:3
    basicOrder(i).index = pointPermOrder(i).index;
    basicOrder(i).x = pointPermOrder(i).x;
    basicOrder(i).y = pointPermOrder(i).y;
end
% initialize the dMin by a high value
dMin = 1e5;
```

```
% start the rest of (n-3) points in the
% list 'basicOrder'
%
k = 4;
for i=4:n
    % check distances from point j to
    % the rest of (i-1) points
    % first three point are already in the list
    % called 'basicOrder'
    for j=1:i-2
        if i==k
            dist = ...
d(pointPermOrder(k).index,basicOrder(j).index)+...
d(pointPermOrder(k).index,basicOrder(j+1).index)-...
d(basicOrder(j).index,basicOrder(j+1).index);
        end
        %
        if i<k
            dist = ...
d(pointPermOrder(i).index,basicOrder(j).index)+...
d(pointPermOrder(i).index,basicOrder(j+1).index)-...
d(basicOrder(j).index,basicOrder(j+1).index);
        end
        %
        if dist<=dMin
            dMin = dist;
            indexInsert = j;
        end
        %
    end
    %
    % now for point (i-1) also
    dist = ...
d(pointPermOrder(i).index,basicOrder(i-1).index)+...
d(pointPermOrder(i).index,basicOrder(1).index)-...
d(basicOrder(i-1).index,basicOrder(1).index);
    %
    if dist<=dMin
        dMin = dist;
        indexInsert = i-1;
    end
    %
    % insert point i in 'basicOrder'
    if indexInsert<i-1
       for m=i:-1:indexInsert+2
```

```
126             basicOrder(m) = basicOrder(m-1);
127         end
128         %
129         basicOrder(indexInsert+1) = pointPermOrder(i);
130     end
131     %
132     if indexInsert == i-1
133         basicOrder(indexInsert+1) = pointPermOrder(i);
134     end
135     %
136     dMin = 1e5;
137     k = k + 1;
138 end
139 %
140 % pathLength calculation
141 pathLength = 0.0;
142 for i=1:1:n-1
143     pathLength = pathLength + ...
144         d(basicOrder(i).index,basicOrder(i+1).index);
145 end
146 %
147 % close the path with segment d(n,1)
148 pathLength = pathLength + ...
149     d(basicOrder(n).index,basicOrder(1).index);
150 %
151 % plot the path before 2Opt
152 subplot(1,2,1);
153 u = zeros(n+1);
154 v = zeros(n+1);
155 for i=1:1:n
156     u(i) = basicOrder(i).x;
157     v(i) = basicOrder(i).y;
158 end
159 u(n+1) = u(1);
160 v(n+1) = v(1);
161 line(u,v);
162 %
163 % vectors allocation
164 x = zeros(1,n);   y = zeros(1,n);   index =
165 zeros(1,n);
166 for i = 1:n
167     x(1,i) = basicOrder(i).x;
168     y(1,i) = basicOrder(i).y;
169     index(1,i) = basicOrder(i).index;
170 end
```

```
171  count = 0;
172  for j = 1:n
173      for i = 3:n-1
174          r = ...
175  segmentsIntersection(x(1,1),y(1,1),x(1,2),...
176              y(1,2),x(1,i),y(1,i),x(1,i+1),y(1,i+1));
177          %
178          if r==1
179              count = count+1;
180              % applying 2Opt procedure
181              [x,y,index] = twoOpt(x,y,index,i,n);
182          end
183          %
184          % applying the translation of points
185          [x,y] = translation(x,y,n);
186      end
187  end
188  %
189  % structure reallocation after 2Opt procedure
190  for i = 1:n
191      basicOrder(i).x = x(1,i);
192      basicOrder(i).y = y(1,i);
193      basicOrder(i).index = index(1,i);
194  end
195  %
196  % print to file the initial order of points
197  fprintf(fp,'\n the order of points:\n');
198  for i = 1:n
199      fprintf(fp,'%d  ',basicOrder(i).index);
200  end
201  %
202  % plot the optimal path
203  subplot(1,2,2);
204  for i=1:n
205      u(i) = basicOrder(i).x;
206      v(i) = basicOrder(i).y;
207  end
208  %
209  u(n+1) = u(1);
210  v(n+1) = v(1);
211  line(u,v);
212  % length path calculation
213  pathLength = 0.0;
214  for i=1:n-1
215      pathLength = pathLength + ...
```

```
216          d(basicOrder(i).index,...
217          basicOrder(i+1).index);
218   end
219   %
220   % close the path with the segment d(n,1)
221   pathLength = pathLength + d(basicOrder(n).index,...
222       basicOrder(1).index);
223   fprintf(fp,'\n the optimal path length: l = %f\n',
224   pathLength);
225   % close the file
226   fclose(fp);
227   %
```

Code 8.11 *tsp.m*.

The *tsp.m* program from Code 8.11 makes several calls to the functions *direction.m* (see Code 8.6), *segmentsIntersection.m* (see Code 8.7), *onSegment.m* (see Code 8.8), *translation.m* (see Code 8.9). These functions were used by the program *removeIntersections.m* from Code 8.4. Besides these functions, the program from Code 8.4 calls also the function *twoOpt1.m* (see Code 8.5). Instead of *twoOpt1.m* function, the *tsp.m* program calls the function *twoOpt2.m*.

```
1   %                    twoOpt2.m
2   % This function removes segment intersections.
3   % The function checks the first segment between
4   % points 1 and 2, against segments (3,4), (4,5), ...
5   % Then it makes the translation and continues.
6   % It takes as input arguments the coordinates (x,y)
7   % of points, their index taken from data structure,
8   % the contour i where an intersection is detected,
9   % and the number n of points.
10  %
11  function [x,y,index] = twoOpt2(x,y,index,i,n)
12  m = i-2+1;
13  if m==0
14      xTemp = x(2);
15      yTemp = y(2);
16      iTemp = index(2);
17      x(2) = x(i);
18      y(2) = y(i);
19      index(2) = index(i);
20      x(i) = xTemp;
21      y(i) = yTemp;
22      index(i) = iTemp;
```

```
23   end
24   if m~=0
25       j = floor(m/2);
26       for k = 1:j
27           xTemp = x(2+k-1);
28           yTemp = y(2+k-1);
29           iTemp = index(2+k-1);
30           x(2+k-1) = x(2+m-k);
31           y(2+k-1) = y(2+m-k);
32           index(2+k-1) = index(2+m-k);
33           x(2+m-k) = xTemp;
34           y(2+m-k) = yTemp;
35           index(2+k-1) = iTemp;
36       end
37   end
38   % translate the vector of n points
39   xTemp = x(1);
40   yTemp = y(1);
41   iTemp = index(1);
42   for k = 2:n
43       x(k-1) = x(k);
44       y(k-1) = y(k);
45       index(k-1) = index(k);
46   end
47   x(n) = xTemp;
48   y(n) = yTemp;
49   index(n) = iTemp;
50   return;
51   end
52   %
```

Code 8.12 *twoOpt2.m*.

To use the program *tsp.m* one's own input data needs to be loaded at line 5 in the Code 8.11 from Fig. 8.1. All input data must be submitted as a text file, according to the requirements already described in Fig. 8.1. The program *tsp.m* assesses then the size of input data, in line 12. The program also assigns memory to the matrix of distances between points and counts all these distances at lines 26 to 41. The program then builds a preliminary order, based on a random permutation (line 57) and arranges all points in the list according to this permutation. The first three points in the permutation make up the *basicOrder* (see lines 74 to 78). Then the user has to assign a variable *dMin* with an inplausibly high value initially chosen at line 80 (i.e. *dMin* = 1e5), to take place as the initial minimum value of the path length. Then, the remaining *(n-3)* points are added one by

one, according to the order in the random permutation (lines 81 to 149). Lines 152 to 161 make the plot of the path before the segments intersections removal. Lines 171 to 194 detects the segments intersections and call the function *twoOpt.m* to remove them. Lines 202 to 211 plot the path after the segments intersections removal.

Fig. 8.12 displays a solution to a problem of 280 points in a printed circuit board, found by the program *tsp.m*, before and after the removal of segment intersections.

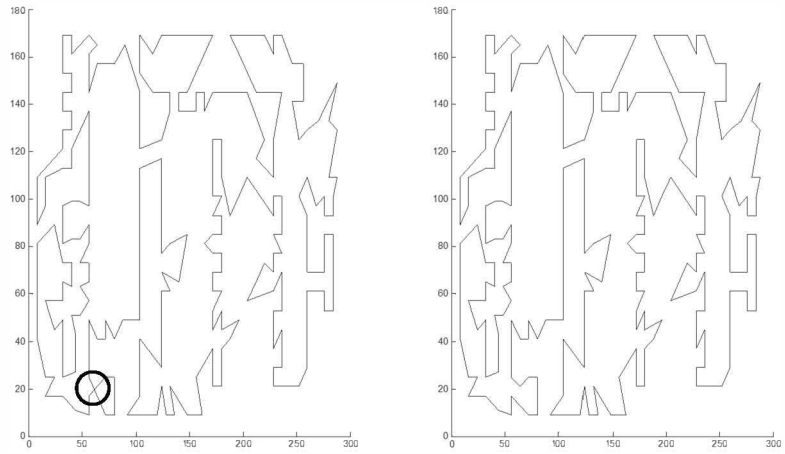

Fig. 8.12 The solution of *tsp.m* program for a 280-point problem, before and after segments intersections removal.

The order that coresponds to the path from Fig. 8.12 can be found in the *basicOrder* structure from inside the Matlab workspace. The variable *pathLength* represents the total length of the path.

8.7 Conclusions

The importance of Traveling Salesman Problem solving methods is revealed by the multitude of their applications (Applegate, 2006), (Gutin & Punnen, 2007), (Cook, 2012). Among the most well known practical applications, the drilling of printed circuit boards, the path designing of electric cable, the genome mapping or DNA sequencing, planning of touristic routes etc. are worth mentioning.

In this chapter, two conventional methods of constructing a path that goes through n points in a plane are described. These sorting horizontally

and sorting vertically techniques have been used in industrial practice for drilling printed circuit boards, now being replaced by more elaborate methods. Then, the Nearest Neighbor method was presented due to its simplicity and easiness to implement on the computer. It was also shown how a path variant can be improved by removing, if they exist, the intersected path segments. At the end of the chapter, a program of insertion type is designed. By modifying the local search procedure, the researcher can simply improve the program performance.

9.

OPTIMAL NESTING

9.1 Introduction

The optimization of material consumption can be achieved by an appropriate orientation of the cutting profile on the material strip. This profile's orientation is not so easy to find, given that both it's the width and the pitch distance between two successive positions change with its rotation. It is well known that material consumption is a critical parameter, depending on the relationship between the area of the profile to be cut and the area of raw material. Choosing the orientation that leads to a minimum step between two consecutive cuts does not necessarily mean optimal material consumption, and manually assessing the material consumption when both width and pitch are variable is a very difficult task.

Attempts to solve this problem date back to 1976 when Adamovitz & Albano proposed including two-dimensional stamping profiles into minimal rectangles. The orientation of 2D profiles in minimal rectangles is an idea also found in (Nee, 1984) and (Martin & Stephenson, 1988). However, it should be noted that this method does not lead to optimal material consumption. Let us consider that we have a profile in the form of a parallelogram. This profile, when is not included in a rectangle, can lead to zero-loss as in Fig. 9.1a), whereas if we include it in a rectangle, the losses can become considerable (see Fig. 9.1b)).

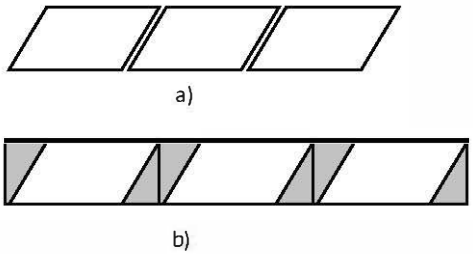

Fig. 9.1 The orientation of a parallelogram on the material strip.

Therefore, the idea of including the profile in the minimum rectangle may become an impediment in determining the optimal orientation. The insertion of the profiles to be cut into geometric shapes other than the rectangle were also examined. In Martin & Stephenson (1988), the profiles were included in mutually matching geometric shapes that were then repeated across the length of the strip. In Qu & Sanders (1987), cut profiles were included in convex polygons. The same was done in Dori & Bassat (1984) and Karoupi & Loftus (1991) respectively. However, just as the profiles are included in the minimal rectangle, the use of different 2D geometric shapes leads to the same, far from optimal, results.

There have also been attempts to apply heuristic algorithms, such as the Metropolis algorithm, to simulate recrystallization (Jain et al, 1992; Theodocrates & Grimsley, 1995). Genetic algorithms (Ismail, 1996) have also been designed, especially in cases where a number of different profiles have to be placed on a material strip. In contrast, when it comes to a single profile, these algorithms require a very long computation time (Nye, 2000).

Another method of solving the problem can be given by incremental rotation algorithms (Chow, 1979; Nee, 1984; Prasad & Somasundaram, 1992; Lin & Hsu, 1996). An increment of profile rotation is selected and then for each rotation the minimum material width and pitch are calculated. After a complete rotation, the option offering the best material consumption coefficient is considered the optimal one. As a rule, the rotation increment is fixed to one or two degrees. However, if the real optimum is located between two incremental values, then it will surely be missed, which will cause small inefficiencies in the optimization process. A more precise algorithm is given in Joshi & Sudit (1994). In this chapter it is shown that optimal positioning of a profile on a strip of finite length will converge towards the optimum solution of placing the same profile on an infinite length strip when the finite length strip is relatively long, as is the case with the cutting operation.

In order to prevent the overlapping of adjacent profiles, the concept of obstacle-space from robotics is used. The obstacle space is represented by a polygon obtained by the Minkovski sum of two polygons. Thus, a reference point of the second polygon follows the contour of the first polygon, the new polygon thus obtained being the sum of Minkowski. Practically, the polygon obtained with the Minkowski sum represents the minimum space in which the two polygons touch each other, but do not overlap. With the Minkowski sum it is demonstrated (Joshi & Sudit, 1994) that when the width of the material strip is greater than the width of the profile, whatever its orientation, the optimal one corresponds to the case

where the pitch difference between successive profiles is minimal. Also, there is an optimal orientation of the profile even if the width of the material strip is predetermined. And here the situation is more complex, because if it concerns the predetermined width of the strip, the cost of the material strip per kilogram also depends on the bandwidth. And then the problem of determining that orientation of the profile on the material strip can be solved so that the material consumption is optimal and the total costs are minimal.

9.2 The Minkowski sum

Let us consider two convex or concave polygons A and B. The Minkowski sum is defined by:

$$A \oplus B = \{a + b |\ a \in A, b \in B\}; \tag{9.1}$$

The importance of this sum lies in its simplifying the overlapping verification procedure for two polygons and, replacing it with verifying the appurtenance of a point to a polygon. This will simplify the calculations necessary to determine the width of the profile for a given orientation and the pitch spacing between the successive profiles. For two polygons A and B, the Minkowski sum $A \oplus B$ is invariant relative to the mutual position of the two. If we have to position a single profile, the Minkowski sum is $A \oplus (-A)$, where $(-A)$ represents the symmetrical profile of A relative to the origin of the coordinate system. Without taking into account the origin of the coordinate system used to define polygon vertices, the sum $A \oplus (-A)$ is by default centered. However, if we are dealing with two polygons A and B, they overlap if the $A \oplus B$ sum contains the origin of the coordinate system. To avoid the overlapping, the two profiles must be translated relative to each other. This translation will implicitly lead to the translation of the Minkowski sum $A \oplus B$, so that when this sum no longer contains the origin of the coordinate system, there is no overlap between profiles A and B. This property leads to the observation that the distance from the origin of the coordinate system to the profile perimeter given by the Minkowski sum $A \oplus B$ in any direction is the pitch distance between profiles in that direction. Thus, for a given profile, the determination of the pitch distance is equivalent to determining the distance from the origin to the Minkowski sum perimeter in that given direction. Similarly, the minimum width of the material for a given profile orientation is precisely the maximum distance from the origin of the

coordinate system to the Minkowski sum perimeter in a direction perpendicular to the pitch direction.

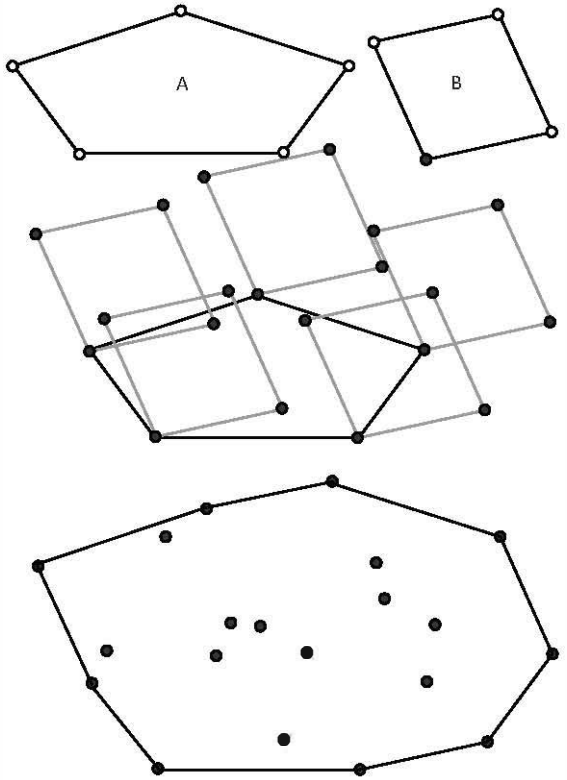

Fig. 9.2 Designing the Minkowski sum in the case of two convex polygons.

9.3 The Minkowski sum for convex polygons

In the case of two convex polygons A and B, the Minkowski sum construction is reduced to determining the convex hull of the points resulting when a reference point on polygon B moves along polygon A, as shown in Fig. 9.2. The construction of the convex hull starts from the left-lower point of the points set. The rule states that each point added to the

convex hull should form a segment that is left-oriented relative to the direction of the previous segment.

For this, let us consider two vectors $\overrightarrow{OA}$ and $\overrightarrow{OB}$ (see Fig. 9.3). To verify that the vector $\overrightarrow{OB}$ is left-hand oriented (anti-clockwise) versus vector $\overrightarrow{OA}$, we need to determine the sign of the vector product $\overrightarrow{OB}$x$\overrightarrow{OA}$. To calculate this product we use the relationship:

$$\overrightarrow{OB} \times \overrightarrow{OA} = (x_B - x_O)(y_A - y_O) - (x_A - x_O)(y_B - y_O); \tag{9.2}$$

If the sign of this product is positive, then $\overrightarrow{OB}$ it is oriented clockwise relative to vector $\overrightarrow{OA}$.

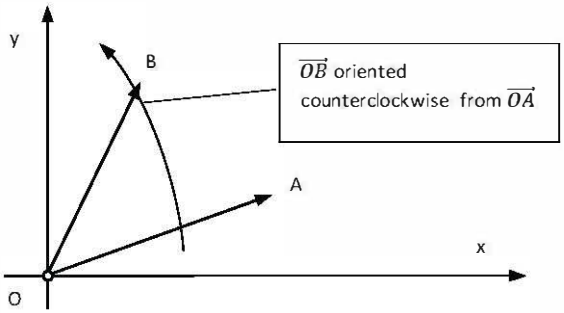

Fig. 9.3 Oriented segments.

We can build the Minkowski sum without referring to the convex hull of a set of points, because when the convex polygon has many segments, this method becomes laborious.

We read all the segments on the contour of each polygon counterclockwise. Each segment will have, by construction, a starting point, an endpoint, an angle of inclination to the horizontal axis and a length. All segments are placed with the starting point in an origin O, so that we keep the initial orientation angle for each segment (see Fig. 9.4). Next, the segments are placed in the ascending order of the angles made by each one with the horizontal axis, as in Fig. 9.5. If there are segments of polygons A and B with the same orientation angle, then we can start at that step by placing any of them, the configuration of the Minkowski sum remaining invariant. The shape of the Minkowski sum is also invariant to the planar orientation of the polygons.

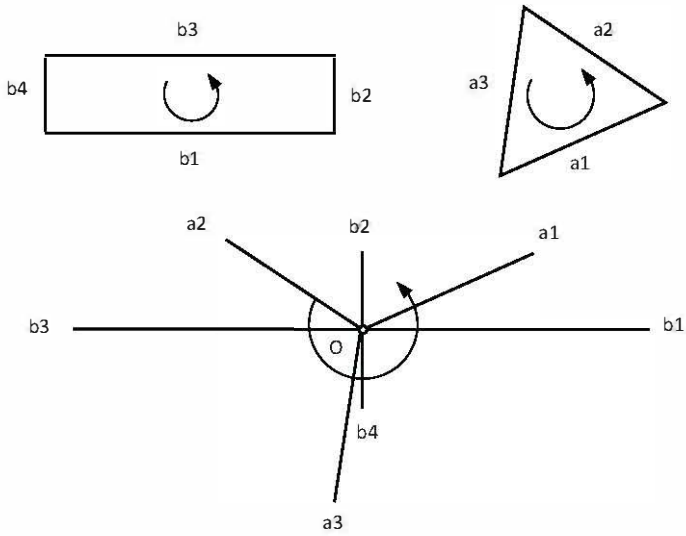

Fig. 9.4 Polygon segments aranged in the ascending order of the angle formed by each segment with the horizontal axis.

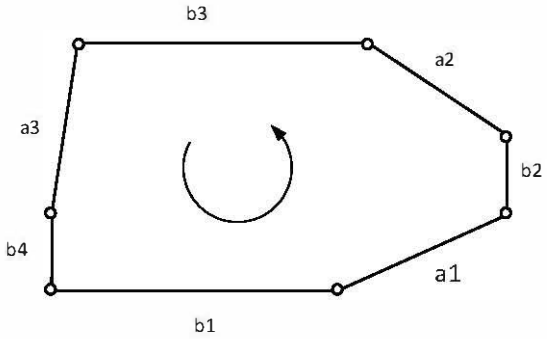

Fig. 9.5 Minkowski sum for polygons A and B.

In the case of several polygons, the Minkowski sum can be made by taking two polygons and to the Minkowski sum polygon thus generated, the third polygon can be added in a similar manner and so on. In this situation, the shape of the Minkowski sum is also invariant to the order of addition polygons. The program *minkowskiConvex.m*, written in Matlab, is based on these concepts.

```
%              minkowskiConvex.m
%
% This program calculates the coordinates of the
% Minkowski sum polygon vertexes. It calls two
% functions: arrange.m and drawPolygon.m
%
% set the number of polygons A and B segments:
nA = 4;      nB = 3;
% define a data structure for a polygon
A = struct ([]);
A(1).start.x = 0; A(1).start.y = 0;
A(1).final.x = 10; A(1).final.y = 1;
A(2).start.x = 10; A(2).start.y = 1;
A(2).final.x = 9; A(2).final.y = 11;
A(3).start.x = 9; A(3).start.y = 11;
A(3).final.x = -1; A(3).final.y = 10;
A(4).start.x = -1; A(4).start.y = 10;
A(4).final.x = 0; A(4).final.y = 0;
%
B = struct ([]);
B(1).start.x = 0; B(1).start.y = 0;
B(1).final.x = 8; B(1).final.y = 4;
B(2).start.x = 8; B(2).start.y = 4;
B(2).final.x = 3; B(2).final.y = 9;
B(3).start.x = 3; B(3).start.y = 9;
B(3).final.x = 0; B(3).final.y = 0;
%
for i = 1:1:nA
    A(i).length = sqrt((A(i).final.x-...
    A(i).start.x)^2+(A(i).final.y-A(i).start.y)^2);
end
%
for i = 1:1:nB
    B(i).length = sqrt((B(i).final.x-
B(i).start.x)^2+...
        (B(i).final.y-B(i).start.y)^2);
end
%
% calculate the slope of polygon A segments
for i = 1:1:nA
    A(i).angle = atan(abs(A(i).final.y-...
    A(i).start.y)/abs(A(i).final.x-A(i).start.x));
    %
    if(A(i).start.x>=A(i).final.x && A(i).start.y<=...
            A(i).final.y)
```

```
46          A(i).angle = pi - A(i).angle;
47      elseif(A(i).start.x>=A(i).final.x && ...
48                A(i).start.y>=A(i).final.y)
49          A(i).angle = A(i).angle + pi;
50      elseif (A(i).start.x<=A(i).final.x && ...
51                A(i).start.y>=A(i).final.y)
52          A(i).angle = 2*pi - A(i).angle;
53      end
54      A(i).angle = A(i).angle*180/pi;
55  end
56  %
57  % calculate the slope of polygon B segments
58  for i = 1:1:nB
59      B(i).angle = atan(abs(B(i).final.y-...
60              B(i).start.y)/abs(B(i).final.x-...
61              B(i).start.x));
62      %
63      if(B(i).start.x>=B(i).final.x && B(i).start.y<=...
64              B(i).final.y)
65          B(i).angle = pi - B(i).angle;
66      elseif(B(i).start.x>=B(i).final.x && ...
67              B(i).start.y>=B(i).final.y)
68          B(i).angle = B(i).angle + pi;
69      elseif (B(i).start.x<=B(i).final.x && ...
70              B(i).start.y>=B(i).final.y)
71          B(i).angle = 2*pi - B(i).angle;
72      end
73      B(i).angle = B(i).angle*180/pi;
74  end
75  %
76  % the size of Minkowski sum polygon
77  nM = nA + nB;
78  M = A;
79  for i = 1:1:nB
80      M(i+nA) = B(i);
81  end
82  %
83  % sort the elements of M, 'M(i).angle' is a sort key
84  M = arrange(M,nM);
85  % align all segments with start point in the origin
86  for i = 1:1:nM
87      M(i).final.x = M(i).final.x - M(i).start.x;
88      M(i).final.y = M(i).final.y - M(i).start.y;
89      M(i).start.x = 0;
90      M(i).start.y = 0;
```

```
91   end
92   %
93   % recalculate the vertex coordinates of
94   % Minkowski sum polygon
95   for i = 2:1:nM
96       M(i).start.x = M(i-1).final.x;
97       M(i).start.y = M(i-1).final.y;
98       %
99       if i == nM
100          M(i).final.x = M(1).start.x;
101          M(i).final.y = M(1).start.y;
102      else
103          M(i).final.x = M(i).final.x + M(i-1).final.x;
104          M(i).final.y = M(i).final.y + M(i-1).final.y;
105      end
106  end
107  %
108  % draw the polygons A and B
109  % the color of polygon A is blue
110  subplot(1,2,1);
111  drawPolygon(A,nA,2,'b');
112  hold on;
113  % the color of polygon B is red
114  drawPolygon(B,nB,2,'r');
115  hold off;
116  % draw Minkowski sum polygon
117  % the color of polygon M is black
118  subplot(1,2,2);
119  drawPolygon(M,nM,2,'k');
120  %
```

Code 9.1 *minkowskiConvex.m*.

In the program from Code 9.1, the coordinates of the two polygons A and B are explicitly defined within a data structure. This data structure contains the coordinates of the respective start and final points of each segment (lines 10 to 26), the length of the segment (lines 28 to 37) and its angle with the horizontal axis (lines 40 to 74). The *minkowskiConvex.m* program for convex polygons also uses two functions, *arrange.m* and *drawPolygon.m*. The *arrange.m* function arranges the segments that compose polygons A and B in an ascending order according to values the angle that each segment makes with the horizontal axis, while the function *drawPolygon.m* draws the two polygons as well as the Minkowski sum polygon.

```
%                    arrange.m
% the function 'arrange' sort the elements of the
% class in increasing order according to the angle
% that each segment makes with horizontal axis
%
function [M] = arrange(M,nM)
%
for i=1:nM
    for j=1:nM
        if(M(i).angle<M(j).angle)
            temp=M(i);
            M(i)=M(j);
            M(j)=temp;
        end
    end
end
%
```

Code 9.2 *arrange.m*.

```
%       drawPolygon.m
% This is a function that draws a polygon
% it takes as actual arguments the data
% in structure X, the number of polygon
% segments nX, the line width 'w', and the
% line colour 'c'.
%
function [] = drawPolygon (X,nX,w,c)
u = zeros(nX+1);
v = zeros(nX+1);
%
for i=1:1:nX
    u(i) = X(i).start.x;
    v(i) = X(i).start.y;
end
%
u(nX+1) = u(1);
v(nX+1) = v(1);
line(u,v,'LineWidth',w,'color',c);
end
%
```

Code 9.3 *drawPolygon.m*.

In Fig. 9.6.a the polygons A (the parallelogram) and B (the triangle) as well as the shape of the Minkowski sum polygon (Fig. 9.6.b) are illustrated.

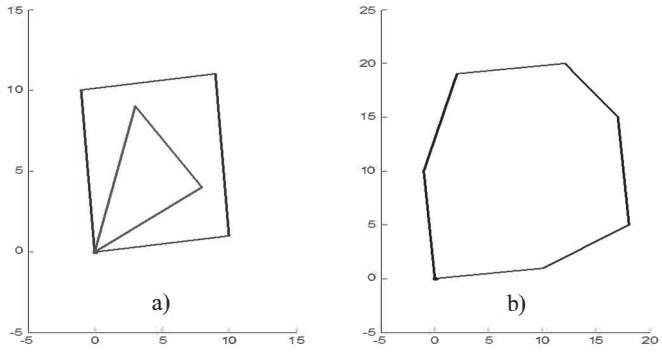

Fig. 9.6 a) Polygons A, B, and b) the Minkowski sum polygon $[A \oplus B]$.

If both polygons A and B are convex then we can draw the convex hull of a cloud of points using the *DelaunayTri(x,y)* and *convexHull(dt)* functions in Matlab. The following program generates randomly a set of n points, for which the convex hull of this set of points is drawn. Now let us imagine that for each position of the polygon B rotating around the polygon A, we mark by points the corresponding positions of each vertex of B. The points resulting from the rotation of polygon B around the polygon A form a cloud of points, and the convex hull of this cloud is exactly the Minkowski sum polygon of $[A \oplus B]$.

```
1   %                theConvexHull.m
2   % This program calculates the convex hull of
3   % a set of points randomly generated. The user
4   % might replace lines 9, 11 and 12 with a
5   % command to load their own set of points in
6   % order to generate the convex hull.
7   %
8   % define the number of points in set
9   pointsNumber = 50;
10  % generate the set of points
11  x = rand(pointsNumber,1);
12  y = rand(pointsNumber,1);
13  % determining the convex hull
14  dt = DelaunayTri(x,y);
15  k = convexHull(dt);
16  [nCH,~] = size(k);
17  xCH = zeros(nCH,1);
```

```
yCH = zeros(nCH,1);
for i = 1:nCH
    xCH(i) = x(k(i));
    yCH(i) = y(k(i));
end
% graphical representation of convex hull
plot(xCH,yCH); hold on;
% the graphical representation of the set of points
plot(dt.X(:,1),dt.X(:,2), '.', 'markersize',10);
%
```

Code 9.4 *theConvexHull.m*.

In Fig. 9.7, the convex hull is determined via *theConvexHull.m* program for a set of 50 points randomly generated. You have to take care with this program because every time it will be run, a different set of points will be randomly generated.

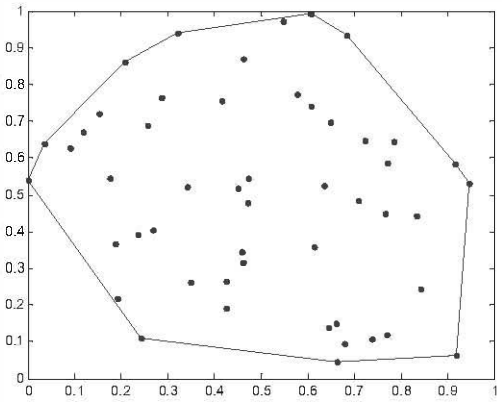

Fig. 9.7 The convex hull of a set of points.

9.4 The Minkowski sum for concave polygons

Just as in the previous case, if we work with concave polygons, the segments on the contour of each polygon are also read in the same counterclockwise direction. Also, for each segment on the contour of the polygon there is acorresponds a starting point, an end point, one length and a slope angle made by the segment with the horizontal axis,

respectively. The Minkowski sum polygon is built by mean of the *Ghosh* diagram (Burke, 2007) seen in Fig. 9.8.

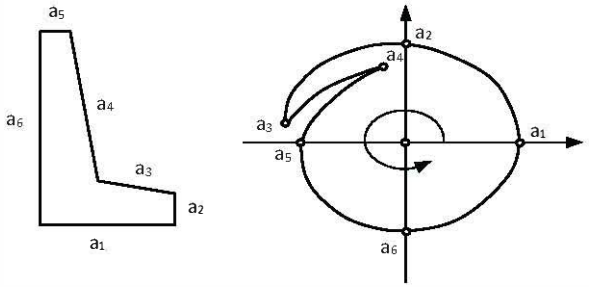

Fig. 9.8 The *Ghosh* diagram for a concave polygon.

If we want to place on the material strip optimally just a single concave profile as in Fig. 9.9 then we need a second polygon symmetrical to the first polygon relative to the origin of the cartesian coordinate system.

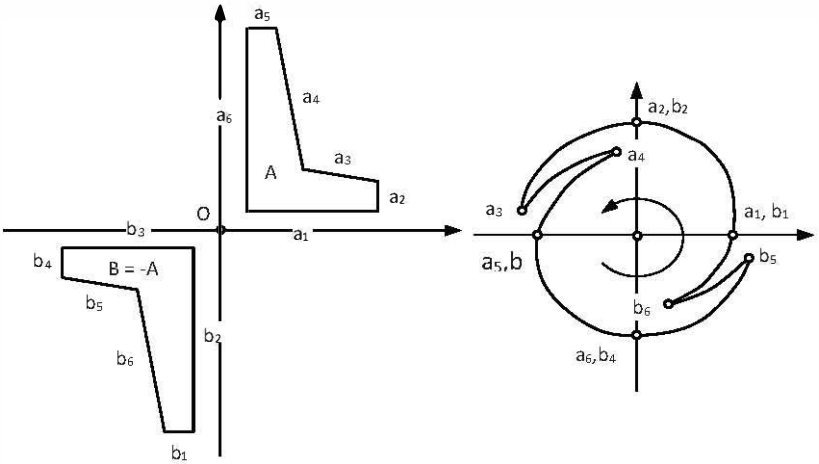

Fig. 9.9 *Ghosh* diagram for concave polygons A and $B = -A$.

To build the Minkowski sum, we put together the segments of polygons A and B in the order of the *Ghosh* diagram. It is important that the order of reading the segments remains the one given by the counterclockwise reading, with the observation that the reading sense is reversed every time the contour of the polygon encounters a concavity direction.

A less complex method of building the Minkowski sum polygon involves splitting the concave polygons into convex polygons. To this end let us assume that we have a concave polygon A and we want to determine its optimal position on the material strip. In this situation, we need the polygon -A, symmetric to the original polygon in relation to the origin of the coordinate axis system. If the concave polygon A is decomposed into k convex polygons, the same decomposition also applies to the polygon -A. Thus, polygon A can be written as follows:

$$A = \bigcup_{i=1}^{k} A_i; \tag{9.3}$$

where A_i are convex polygons. Then:

$$-A = \bigcup_{i=1}^{k} \{-A_i\}; \tag{9.4}$$

Next, the Minkowski sum polygon for convex polygon pairs must be determined:

$$[A_i \oplus -A_1], [A_i \oplus -A_2], \ldots, [A_i \oplus -A_k], \quad i = 1,2,\ldots,k \tag{9.5}$$

It is of note that we have k^2 such pairs and therefore as many polygons of Minkowski sums, which can be determined simply, using the programs *minkowskiConvex.m* or *theConvexHull.m* in section 9.3.
Finally, the Minkowski sum polygon for the concave polygon A will be determined as the union of the Minkowski sums found for the pairs of convex polygons.

```
%           minkowskiConcave.m
%
% For the determination of the sum, the initial
% concave polygon A was decomposed in two
% convexe polygons. the Minkowski sum for
% polygon pairs [A1+(-A1)], [A1+(-A2)],
% [A2+(-A1)], [A2+-(A2)] was determined.Finally
% the union of the Minkowski sums was made.
%
% read the coordinates (x,y) of the peaks of
% the polygons A1 and A2
%
```

```
% polygon A1:
A1 = load('polygonA1.txt');
% find the vertexes number of polygon A1
[n1,~] = size(A1);
% polygon A2:
A2 = load('polygonA2.txt');
% find the vertexes number of polygon A2
[n2,~] = size(A2);
% calculate the Minkowski sum
[mx,my] = minkowskiSum(A1,n1,A2,n2);
% draw Minkowski sum polygon
line(mx,my,'Color','k','LineWidth',2);
%
```

Code 9.5 *minkowskiConcave.m*.

The program *minkowskiConcave.m* loads the coordinates of convex polygons A_1 and A_2, whose union constitutes the concave polygon A. In the Code 9.5 there is a call to the function *minkowskiSum.m* at line 22. This function calculates the polygon of Minkowski sum and returns the control to the program *minkowskiConcave.m*, that, in the end, draws the Minkowski sum polygon.

```
%                  minkowskiSum.m
% This function calculates the Minkowski sum in
% the case of a concave polygon. To do this, you
% should decompose the initial concave polygon
% in several convex polygons. This function is
% built for a concave polygon that is decomposed
% into two convex polygons A1 and A2. The
% coordinates of these polygons are in:
% polygonA1.txt and polygonA2.txt files.
%
function [bx,by] =  minkowskiSum(A1,n1,A2,n2)
xA1 = zeros(1,n1);
yA1 = zeros(1,n1);
for i=1:n1
    xA1(1,i) = A1(i,1);
    yA1(1,i) = A1(i,2);
end
xA2 = zeros(1,n2);
yA2 = zeros(1,n2);
for i=1:n2
    xA2(1,i) = A2(i,1);
    yA2(1,i) = A2(i,2);
end
```

```
% Coordinates x,y of polygon -A1
xmA1 = -xA1;
ymA1 = -yA1;
% Coordinates x,y of polygon -A2
xmA2 = -xA2;
ymA2 = -yA2;
% addition of A1 to -A1
[~,n1] = size(xA1); [~,n2] = size(xA2);
% the size of Minkowski sum is n1xn2
% S1: A1 with -A1; S2: A1 with -A2;
% S3: A2 with -A1; S4: A2 with -A2
xS1 = zeros(1,n1*n2);   yS1 = zeros(1,n1*n2);
xS2 = zeros(1,n1*n2);   yS2 = zeros(1,n1*n2);
xS3 = zeros(1,n1*n2);   yS3 = zeros(1,n1*n2);
xS4 = zeros(1,n1*n2);   yS4 = zeros(1,n1*n2);
k = 1;
for i = 1:n1
    for j = 1:n1
        xS1(1,k) = xA1(1,i) + xmA1(1,j);
        yS1(1,k) = yA1(1,i) + ymA1(1,j);
        k = k+1;
    end
end
k = 1;
for i = 1:n1
    for j = 1:n2
        xS2(1,k) = xA1(1,i) + xmA2(1,j);
        yS2(1,k) = yA1(1,i) + ymA2(1,j);
        k = k+1;
    end
end
k = 1;
for i = 1:n2
    for j = 1:n1
        xS3(1,k) = xA2(1,i) + xmA1(1,j);
        yS3(1,k) = yA2(1,i) + ymA1(1,j);
        k = k+1;
    end
end
k = 1;
for i = 1:n2
    for j = 1:n2
        xS4(1,k) = xA2(1,i) + xmA2(1,j);
        yS4(1,k) = yA2(1,i) + ymA2(1,j);
        k = k+1;
```

```
    end
end
% make the transpose of 8 vectors
xS1 = transpose(xS1);
yS1 = transpose(yS1);
xS2 = transpose(xS2);
yS2 = transpose(yS2);
xS3 = transpose(xS3);
yS3 = transpose(yS3);
xS4 = transpose(xS4);
yS4 = transpose(yS4);
[i1,~] = size(xS1);
[i2,~] = size(xS2);
[i3,~] =size(xS3);
[i4,~] = size(xS4);
data1 = zeros(i1,2);
data2 = zeros(i2,2);
data3 = zeros(i3,2);
data4 = zeros(i4,2);
%
for i = 1:i1
    data1(i,1) = xS1(i,1);
    data1(i,2) = yS1(i,1);
end
for i = 1:i2
    data2(i,1) = xS2(i,1);
    data2(i,2) = yS2(i,1);
end
for i = 1:i3
    data3(i,1) = xS3(i,1);
    data3(i,2) = yS3(i,1);
end
for i = 1:i4
    data4(i,1) = xS4(i,1);
    data4(i,2) = yS4(i,1);
end
% the removal of multiple points
[M1,ind1] = unique(data1,'rows','first');
[~,ind1] = sort(ind1);
M1 = M1(ind1,:);
[M2,ind2] = unique(data2,'rows','first');
[~,ind2] = sort(ind2);
M2 = M2(ind2,:);
[M3,ind3] = unique(data3,'rows','first');
[~,ind3] = sort(ind3);
```

```
M3 = M3(ind3,:);
[M4,ind4] = unique(data4,'rows','first');
[~,ind4] = sort(ind4);
M4 = M4(ind4,:);
[i1,~] = size(ind1);
[i2,~] = size(ind2);
[i3,~] = size(ind3);
[i4,~] = size(ind4);
for i = 1:i1
    xS1(i) = M1(i,1);
    yS1(i) = M1(i,2);
end
for i = 1:i2
    xS2(i) = M2(i,1);
    yS2(i) = M2(i,2);
end
for i = 1:i3
    xS3(i) = M3(i,1);
    yS3(i) = M3(i,2);
end
for i = 1:i4
    xS4(i) = M4(i,1);
    yS4(i) = M4(i,2);
end
% build the convex hull
dt1 = DelaunayTri(xS1,yS1);
k1 = convexHull(dt1);
dt2 = DelaunayTri(xS2,yS2);
k2 = convexHull(dt2);
dt3 = DelaunayTri(xS3,yS3);
k3 = convexHull(dt3);
dt4 = DelaunayTri(xS4,yS4);
k4 = convexHull(dt4);
[m1,~] = size(k1);
[m2,~] = size(k2);
[m3,~] = size(k3);
[m4,~] = size(k4);
% convex hull ch1:
ch1x = zeros(m1,1);
ch1y = zeros(m1,1);
for i = 1:m1
    ch1x(i,1) = xS1(k1(i),1);
    ch1y(i,1) = yS1(k1(i),1);
end
% convex hull ch2:
```

```
ch2x = zeros(m2,1);
ch2y = zeros(m2,1);
for i = 1:m2
    ch2x(i,1) = xS2(k2(i),1);
    ch2y(i,1) = yS2(k2(i),1);
end
% convex hull ch3:
ch3x = zeros(m3,1);
ch3y = zeros(m3,1);
for i = 1:m3
    ch3x(i,1) = xS3(k3(i),1);
    ch3y(i,1) = yS3(k3(i),1);
end
% convex hull ch4:
ch4x = zeros(m4,1);
ch4y = zeros(m4,1);
for i = 1:m4
    ch4x(i,1) = xS4(k4(i),1);
    ch4y(i,1) = yS4(k4(i),1);
end
% the union of convex hulls 1 and 2
[a1x,a1y] = polybool('union',ch1x,ch1y,ch2x,ch2y);
% arrange the coordinates of the sum
% in counterclockwise direction
poly2ccw(a1x,a1y);
% the union of convex hulls 3 and 4
[a2x,a2y] = polybool('union',ch3x,ch3y,ch4x,ch4y);
% arrange the coordinates of the sum
% in counterclockwise direction
poly2ccw(a2x,a2y);
% rearrangement of  the coordinates of the sum
% in clockwise direction
[ax,ay] = polybool('union',a1x,a1y,a2x,a2y);
[bx,by] = poly2ccw(ax,ay);
return;
end
%
```

Code 9.6 *minkowskiSum.m*.

For example, let us consider the concave polygon from Fig. 9.10a, which was manually decomposed into two convex polygons A_1 and A_2 (see Fig. 9.10b). By executing the *minkowskiConcave.m* program for the polygon A decomposed into polygons A_1 and A_2, the Minkowski sum of polygon in Fig. 9.11 is obtained.

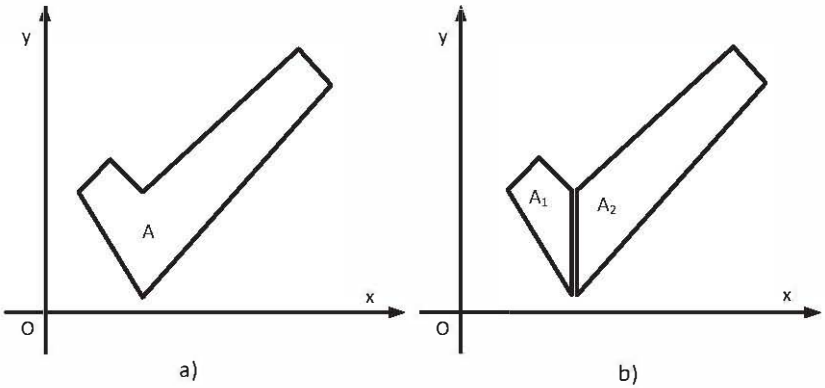

Fig. 9.10 The concave polygon A, manually decomposed into two convex polygons A_1 and A_2.

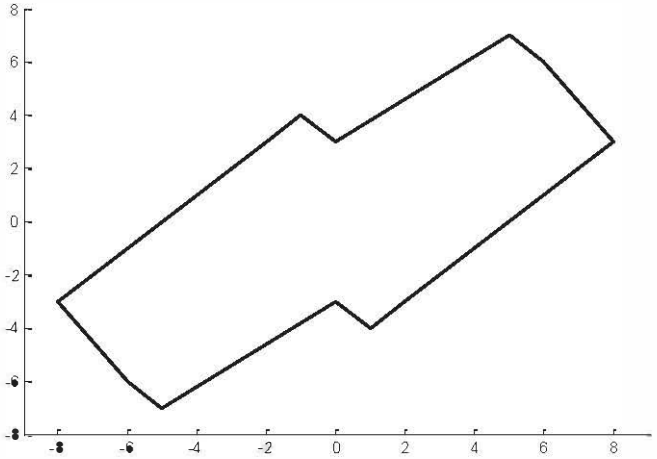

Fig. 9.11 The Minkowski sum polygon, corresponding to the concave polygon A.

9.5 Optimal orientation of a polygon

Once the Minkowski sum polygon is determined, the optimal orientation of the concave profile on the material strip follows. In Fig. 9.12 the Minkowski sum polygon for polygons A and $B = -A$ is drawn. From the origin of the coordinate system, a direction of material strip must be choosen. This direction makes the angle α with the horizontal axis. The

distance measured from the line that define this direction to the furthest point on the Minkowski sum polygon from this line is the minimum width w of the material strip. The line that define the material strip direction crosses the polygon of the Minkowski sum. The distance measured from the origin of the coordinate system to this point is the *pitch*. The concave profile can be positioned on the material strip repeatedly with this *pitch* distance. The material strip area corresponding to the polygonal profile is equal to the product of the width of the material strip and the *pitch* distance. Knowing the coordinates (x, y) of the polygon vertexes, we can calculate its area with the relationship:

$$A_{polygon} = \frac{1}{2}\sum_{i}(x_i y_{i+1} - x_{i+1} y_i); \tag{9.6}$$

The ratio between the profile area and the material strip area, signifies the material consumption. The closer the material consumption coefficient is to unit, the more efficient is the use of material. An increment is chosen for the angle with which the line that represents the direction of the material strip wil be rotated.

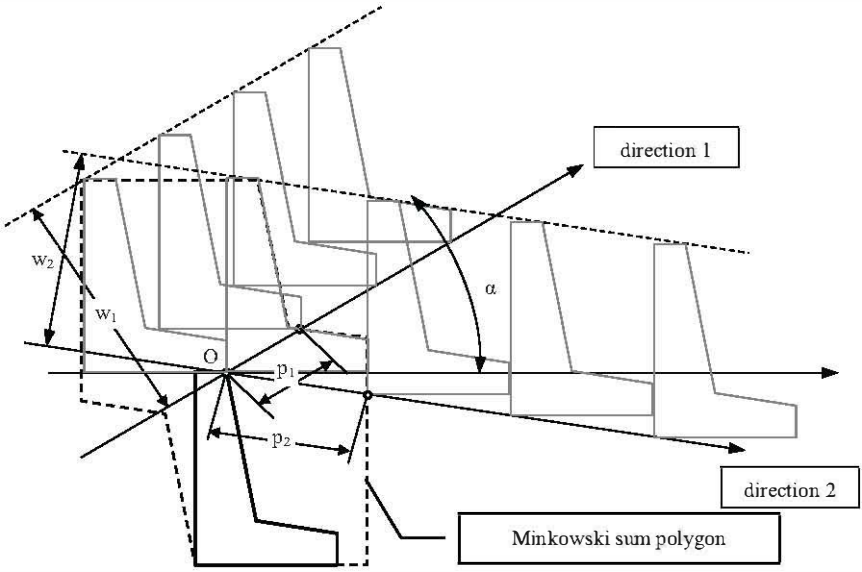

Fig. 9.12 The profile orientation towards the direction of the material strip.

For each incremental value of this angle, the material consumption coefficient is calculated. The inclination angle α that corresponds to a minimum value of the material consumption coefficient is selected. The smaller the increment α is, the more accurate the result will be. If the arrangement of more distinct profiles A_1, A_2, ..., A_n is necessary, then the material consumption coefficient depends on the position of each profile and their order of arrangement of them. In this way, any permutation of the initial order may become a solution to the problem. From this point of view, the optimal nesting problem is a combinatorial optimization problem.

Although this process gives us the possibility to choose the profile layout on the material strip so that the material consumption is minimal, there are ways to improve the results. Thus, for the "L" type profile in Fig. 9.12, the positioning on the material strip is determined according to Fig. 9.13.

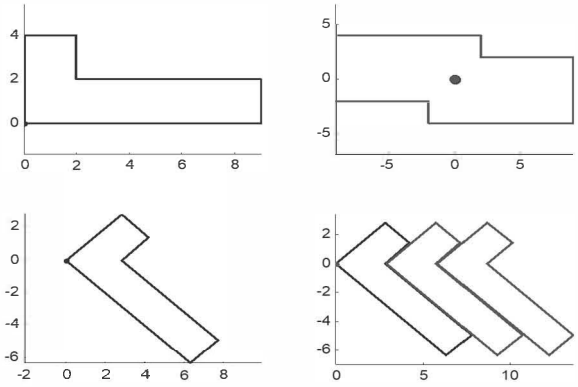

Fig. 9.13 The layout of the "L" type profile on the material strip.

If the symmetrical polygon of the "L" profile in relation to the origin of the Cartesian coordinate system (let us call it A_2), performs an orbital motion around the initial profile (called A_1), we can determine the position of A_2 relative to A_1 so that the convex hull of both profiles has a minimal area and at the same time $A_1 \cap A_2 = O$ (see Fig. 9.14, case f).

It can be noted that positioning on the material strip of profiles A_1 and A_2, according to their location in Fig. 9.14f, produces zero waste.

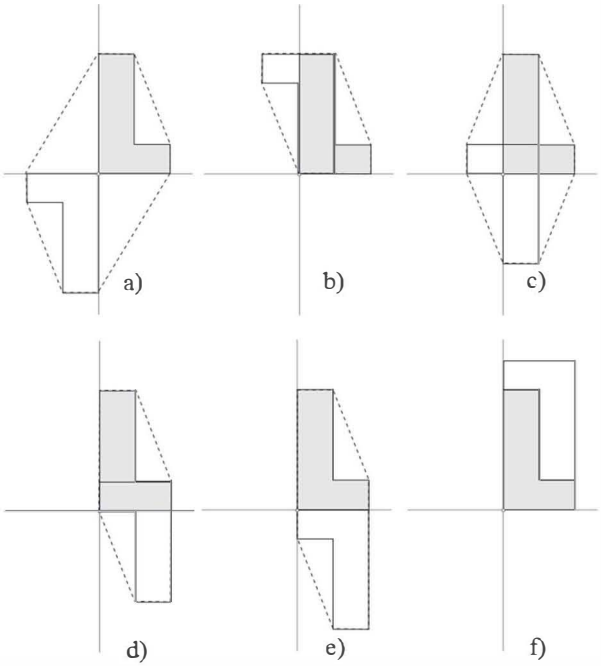

Fig. 9.14 Orbital movement of A_2 around A_1.

```
%                    shapeLmatch1.m
% this program find the proper position of an L-shape
% profile that matches its own symmetrical profile
% that both profiles cover a minimal area
%
% open a file to save results
fp = fopen('results.txt','w');
% define the coordinates of polygon A
xA = [2 8 7 2 1 0 2];
yA = [0 6 7 3 4 3 0];
%
axis equal;
hold on;
% define the polygon B as -A
xB = -xA;
yB = -yA;
% get the number of vertices of polygon A
[~,dimA] = size(xA);
```

```
19  dimB = dimA;
20  % H is the set of points as result of orbit motion
21  % of polygon B around polygon A, vertex by vertex
22  dimH = 2*dimA;
23  xH = zeros(dimH,1);
24  yH = zeros(dimH,1);
25  for i = 1:dimA
26      xH(i) = xA(i);
27      yH(i) = yA(i);
28  end
29  % area calculation of polygon A
30  areaA = polyarea(xA,yA);
31  % initialize the value of ratio between twice the
32  % area of polygon A and the area of the convex hull
33  optimReport = 0;
34  % start with point B(ij)
35  for ib = 1:dimB
36      for ia = 1:dimA
37          distx = xA(ia)-xB(ib);
38          disty = yA(ia)-yB(ib);
39          for i = 1:dimB
40              xBprim = xB + distx;
41              yBprim = yB + disty;
42          end
43          for i=1:dimA
44              xH(dimA+i) = xBprim(i);
45              yH(dimA+i) = yBprim(i);
46          end
47          %
48          [xNew,yNew] = eraseMultiplePoints...
49              (transpose(xH),transpose(yH));
50          %
51          dt = DelaunayTri(xNew,yNew);
52          k = convexHull(dt);
53          % get the number of vertices nCH of the
54          %    convex hull
55          [nCH,~] = size(k);
56          %
57          xCH = zeros(nCH,1);
58          yCH = zeros(nCH,1);
59          %
60          for i = 1:nCH
61              xCH(i) = xNew(k(i));
62              yCH(i) = yNew(k(i));
63          end
```

```
64          % area calculation of convex hull
65          areaCH = polyarea(xCH,yCH);
66          % erase the strings xCH and yCH
67          clear xCH;
68          clear yCH;
69          % find the intersection of polygons A si B
70          [ix,iy] = polybool('intersection',xA, yA, ...
71              xBprim, yBprim);
72          % area calculation of polygon I = A
73  intersected
74          %      with Bprim (B translated)
75          areaInt = polyarea(ix,iy);
76          if areaInt == 0
77              report = (2*areaA)/areaCH;
78          elseif areaInt ~= 0
79              report = 0;
80          end
81  % to see the orbiting motion, uncomment
82  % the following five lines of code
83  %         h1 = plot(xA,yA,'Color', 'b');
84  %         h2 = plot(xBprim,yBprim,'Color','r');
85  %         pause(0.01);
86  %         delete(h1);
87  %         delete(h2);
88          fprintf(fp,'B_vertex = %d', ib);
89          fprintf(fp,'   intArea=%f', areaInt);
90          fprintf(fp,'  areaCH=%f',areaCH);
91          fprintf(fp,'  2*areaA=%f',2*areaA);
92          fprintf(fp,'  report = %f\n',report);
93          if report > optimReport
94              optimReport  = report;
95              xBoptim = xBprim;
96              yBoptim = yBprim;
97              fprintf(fp,'report=%f\n',report);
98          end
99      end
100 end
101 % draw both polygons
102 line(xA, yA, 'Color','r','LineWidth',2);
103 plot(xBoptim, yBoptim, 'Color', 'b','LineWidth',2);
104 hold off;
105 % close the 'results.txt' file
106 fclose(fp);
    %
```

Code 9.7 *shapeLmatch1.m*.

The source code *shapeLmatch1.m* uses a function that remove multiple points leaving instead of multiple identical points just a single one. The source code of this function is listed in Code 9.7.
The program *shapeLmatch1.m* defines the coordinates (*x,y*) of the vertices of the polygon *A* in the form of two strings *xA* and *yA* (lines 9 and 10). Then it defines poligon *B* as *-A*. Two strings *xH* and *yH* are also defined (lines 23, 24). These strings contain the coordinates *x* and *y* of the vertices of polygons *A* and *B*, during the orbital movement of polygon *B* around the polygon *A*. The source code lines 35 to 99 search for that position of polygon *B* around polygon *A*, corresponding to a minimum area of convex hull of both polygons, while polygon *B* does not intersect polygon *A*. This program calls the function *eraseMultiplePoints.m* (see Code 9.8), before the construction of the convex hull. Lines 82 to 86 are defined as comments. If the character % in front of each of these lines is deleted, one can note the movement of polygon *-A* around the polygon *A* at the cost of running time. However, in order to avoid a lower speed of the program, the activation of these instructions is left available to the user.

```
1   %            eraseMultiplePoints.m
2   %  This function erase multiple points from a set;
3   %  the function receives as inputs two strings x and y
4   %  and returns two other strings x_new and y_new,
5   %  with the multiple points already removed
6   %
7   function [x_new,y_new] = eraseMultiplePoints(x,y)
8   [~,n] = size(x);
9   mark = zeros(1,n);
10  k = 0;
11  for i = 1:n-1
12       k = k + 1;
13      for j = (i+1):n
14          if x(i)==x(j) && y(i) == y(j)
15              mark(k) = 1;
16          end
17      end
18  end
19  %
20  k = 0;
21  for i = 1:n
22      if mark(i) ~= 0
23          k = k+1;
24      end
```

```
25  end
26  x_new = zeros(1,n-k);
27  y_new = zeros(1,n-k);
28  j = 1;
29  for i = 1:n
30      if mark(i) ==0
31          x_new(1,j) = x(i);
32          y_new(1,j) = y(i);
33          j = j + 1;
34      end
35  end
36  % we convert line vectors into column vectors
37  x_no = transpose(x_new);
38  y_no = transpose(y_new);
39  x_new = x_no;
40  y_new = y_no;
41  return;
42  %
```

Code 9.8 *eraseMultiplePoints.m*.

In Fig. 9.15 we can see the result of the program *shapeLmatch1.m* for the polygon A defined by (x,y) coordinates of its vertices at lines 9 and 10. If providethis result as input data to a program that attempts to find the optimal position of the profile on a strip of material, it is evident that the waste will be zero, if we do not take into account the material left over due to some technical consideration. In case we want to match together two different polygons A and B, the program does not differ too much. Instead of defining polygon B as the symmetric of polygon A in relation to the origin of the coordinate system, its cordinates (x,y) should be loaded or defined as for polygon A. Else, the principle of work remains unchanged. Code 9.9 shows the source code of this program, called *shapeLmatch2.m*.

```
1   %
2   %                       shapeLmatch2.m
3   % this program find the proper position of two
4   different
5   % profiles to match one another, so that both profiles
6   % will cover the minimum area
7   %
8   % open a file to save results
9   fp = fopen('results.txt','w');
10  % define the coordinates of polygon A
11  xA = [3 6 6 4 4 1 1 2 2 5 5 1 0 0 3];
12  yA = [0 0 2 2 4 4 5 5 6 6 8 8 7 3 0];
```

```
%
axis equal;
hold on;
% define the coordinates of polygon B
xB = [4 6 5 2.5 2.5 1 1 4 4];
yB = [2 2 6 6 5 5 4 4 2];
% get the number of vertices of polygons A and B
[~,dimA] = size(xA);
[~,dimB] = size(xB);
% H is the set of points as result of orbital motion
% of polygon B around polygon A, vertex by vertex
dimH = dimA+dimB;
xH = zeros(dimH,1);
yH = zeros(dimH,1);
for i = 1:dimA
    xH(i) = xA(i);
    yH(i) = yA(i);
end
% area calculation of polygon A
areaA = polyarea(xA,yA);
% area calculation of polygon B
areaB = polyarea(xB,yB);
% initialize the value of ratio between twice area
% of polygon A and area of the convex hull
optimRatio = 0;
% start with point B(ij)
for ib = 1:dimB
    for ia = 1:dimA
        distx = xA(ia)-xB(ib);
        disty = yA(ia)-yB(ib);
        for i = 1:dimB
            xBprim = xB + distx;
            yBprim = yB + disty;
        end
        for i=1:dimB
            xH(dimA+i) = xBprim(i);
            yH(dimA+i) = yBprim(i);
        end
        %
        [x_new,y_new]=eraseMultiplePoints...
            (transpose(xH),transpose(yH));
        %
        dt = DelaunayTri(x_new,y_new);
        k = convexHull(dt);
        % get the number of vertices nCH of
```

```
58          % the convex hull
59          [nCH,~] = size(k);
60          for i = 1:nCH
61              xCH(i) = x_new(k(i));
62              yCH(i) = y_new(k(i));
63          end
64          % area calculation of convex hull
65          areaCH = polyarea(xCH,yCH);
66          % find the intersection of polygons A si B
67          [ix,iy] = polybool('intersection',xA, yA, ...
68              xBprim, yBprim);
69          % area calculation of polygon I = A
70          % intersected with Bprim (B translated)
71          areaInt = polyarea(ix,iy);
72          if areaInt == 0
73              ratio = (areaA+areaB)/areaCH;
74          elseif areaInt ~= 0
75              ratio = 0;
76          end
77  %
78  % to see the orbital motion, remove % sign before
79  % the following five lines of code
80  %         h1 = plot(xA,yA,'Color', 'b');
81  %         h2 = plot(xBprim,yBprim,'Color','r');
82  %         pause(0.1);
83  %         delete(h1);
84  %         delete(h2);
85          fprintf(fp,'B_vertex = %d    ', ib);
86          fprintf(fp,'intArea=%f   ', areaInt);
87          fprintf(fp,'areaCH=%f   ', areaCH);
88          fprintf(fp,'area(A+B)=%f   ', (areaA+areaB));
89          fprintf(fp,'report = %f\n', ratio);
90          %
91          if ratio > optimRatio
92              optimRatio  = ratio;
93              xBoptim = xBprim;
94              yBoptim = yBprim;
95              fprintf(fp,'report=%f\n',ratio);
96          end
97      end
98  end
99  % draw both polygons
100 line(xA, yA, 'Color','r');
101 plot(xBoptim, yBoptim, 'Color', 'b');
102 hold off;
```

```
103  % close the 'results.txt' file
104  fclose(fp);
105  %
```

Code 9.9 shapeLmatch2.m.

To enter the coordinates x and y of the polygons vertexes, a simple and direct method was preferred. More attention was devoted to present the principle of the method, than to the way of reading data. Anyway, the user can easily decide for another method to load the data.

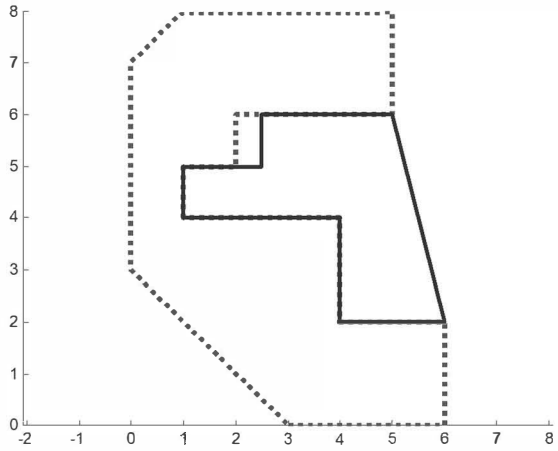

Fig. 9.15 The matching between two polygons A and B.

9.6 Nesting 2D design

The nesting 2D program accepts as input data the coordinates (x,y) of the vertices of a convex polygon, marked A. Next, a data structure, called a in which each element of the structure is related to a segment of polygon A, is defined. Since the coordinates of polygon A vertices are read counterclockwise, each segment of the polygon has a starting point (*x_start*, *y_start*), a final point (*x_final*, *y_final*), an angle of inclination *alpha*, a *direction* and a *marker*. The segment orientation can be either *right* or *left* in relation to the direction of the previous segment. The

marker has a value of one if the segment to which it corresponds has already been included in the Minkowski sum and zero otherwise. The Minkowski sum is based on the principles described in paragraph 9.2. Although the function that builds the Minkowski sum polygon is designed to work with convex polygons, it is also able to resolve polygons that contain concavities, if they are not complicated[1]. However, if the Minkowski sum polygon is not correctly drawn, then the polygon A with concavities must be decomposed into several convex polygons (*minkowskiConcave.m* will be used).

Next, the center of the Minkowski sum polygon is moved into the origin of the Cartesian system of coordinates. The angle between a horizontal line and the line that connects the origin of the coordinates system with each of the Minkowski sum polygon vertices, is calculated. As shown in (Nye, 2001), along the segments that border the Minkowski sum polygon, the variation of the material consumption coefficient is linear. For this reason, it is enough to examine what happens when the mobile line crosses the vertices of the Minkowski sum polygon. The Minkowski sum polygon is rotated clockwise, so that the mobile line (seen in Fig. 9.12) crosses one turn at each vertex of the polygon. Then the minimum width of material strip required as well as the minimum pitch distance are calculated. The material consumption coefficient is determined as the ratio between the area of polygon A and the material strip area. It saves that position of polygon A that corresponds to the maximum consumption coefficient value. All this analysis is made within the program between lines 163 to 192.

```
1   %                          nesting2D.m
2   % This program determines the optimal orientation
3   % of a polygonal profile on a material strip.
4   % The program is able to determine both the width
5   % of the material strip and the pitch distance.
6   % A limited width of material strip can be set.
7   %
8   % read the coordinates of the polygon
9   p = load('polygonA.txt');
10  % define data structure of the polygon
11  polygon = ...
12  struct('x_start',0,'y_start',0,'x_final',0,...
13  'y_final',0,'alpha',0,'direction','left','marker',0);
14  a = polygon;
```

[1] More cavities or holes.

```
15  [n,~] = size(p);
16  % in case of strip width imposed just activate line 19
17  % by erasing % sign before maxStripWidth variable
18  % instead of 7.5 insert your desired value
19  %
20  % maxStripWidth = 7.5;
21  %
22  % define the coordinates of the polygon (denoted A)
23  for i = 1:n
24      a(i).x_start = p(i,1);
25      a(i).y_start = p(i,2);
26  end
27  for i = 1:(n-1)
28      a(i).x_final = p(i+1,1);
29      a(i).y_final = p(i+1,2);
30  end
31  a(n).x_final = p(1,1);
32  a(n).y_final = p(1,2);
33  % we assume that the polygon A is convex;
34  % so: direction = 'left'
35  for i=1:n
36      a(i).direction = 'left';
37      a(i).marker = 0;
38      % marker = 0, means that the segment is not
39      % included in Minkowski sum polygon.
40  end
41  % the segment orientation
42  for i = 2:n
43      crossProduct = (a(i-1).x_final-a(i-1).x_start)*...
44          (a(i).y_final-a(i-1).y_start)-...
45          (a(i).x_final-a(i-1).x_start)*(a(i-1).y_final-
46  ...
47          a(i-1).y_start);
48      if (crossProduct<0)
49          a(i).direction = 'right';
50      end
51  end
52  % get all the segments of polygon A with starting
53  point
54  % in origin (new polygon 'pol' -> for alpha
55  calculation)
56  pol = polygon;
57  for i = 1:n
58      pol(i).x_start = 0;
59      pol(i).y_start = 0;
```

```
60  end
61  for i = 1:n
62      pol(i).x_final = a(i).x_final - a(i).x_start;
63      pol(i).y_final = a(i).y_final - a(i).y_start;
64  end
65  % find the slope of polygon A segments
66  a = polygonSegmentSlope(a,pol,n);
67  % define the coordinates of polygon B = -A
68  b = polygon;
69  for i = 1:n
70      b(i).x_start = -a(i).x_start;
71      b(i).y_start = -a(i).y_start;
72  end
73  for i = 1:(n-1)
74      b(i).x_final = -a(i).x_final;
75      b(i).y_final = -a(i).y_final;
76  end
77  b(n).x_final = -a(n).x_final;
78  b(n).y_final = -a(n).y_final;
79  % while A is convex => B is convex;
80  % so: direction = 'left'
81  for i=1:n
82      b(i).direction = 'left';
83      b(i).marker = 0;
84      % marker = 0, means that the segment is not
85      % included in Minkowski sum polygon.
86  end
87  % the segments orientation
88  for i = 2:n
89      crossProduct = (b(i-1).x_final-b(i-1).x_start)*...
90          (b(i).y_final-b(i-1).y_start)-...
91          (b(i).x_final-b(i-1).x_start)*(b(i-1).y_final-
92  ...
93          b(i-1).y_start);
94      if (crossProduct<0)
95          b(i).direction = 'right';
96      end
97  end
98  % get all the segments of polygon B with
99  % the start point in origin
100 for i = 1:n
101     pol(i).x_start = 0;
102     pol(i).y_start = 0;
103 end
104 for i = 1:n
```

```
105      pol(i).x_final = b(i).x_final - b(i).x_start;
106      pol(i).y_final = b(i).y_final - b(i).y_start;
107  end
108  % find the slope of polygon B segments
109  b = polygonSegmentSlope(b,pol,n);
110  % 1. find the lower-left vertex of polygon B
111  indexLow = 1;
112  for i=2:n
113      if (b(i).y_start<=b(indexLow).y_start)
114          indexLow = i;
115      end
116  end
117  index = indexLow;
118  for i=1:n
119      if (b(i).x_start<=b(index).x_start &&
120  b(i).y_start...
121              == b(indexLow).y_start)
122          index = i;
123      end
124  end
125  % 2. reorder the list of polygon segments with vertex
126  % v_index; there is no change to the order of the
127  % segments between them
128  for j = 1:index-1;
129      c = b(1);
130      for i = 1:n-1
131          b(i) = b(i+1);
132      end
133      b(n) = c;
134  end
135  % the Minkowski sum polygon has the size 2*n
136  m = polygon;
137  % the Minkowski sum
138  [a,m] = minkowskiSumConvex(a,b,m,n);
139  % the Minkowski sum polygon is called "mink"
140  mink = polygon;
141  for i=1:2*n
142      mink(i) = m(i);
143  end
144  % move the center of Minkowski sum polygon
145  % in origin
146  xCenter = 0.0;
147  yCenter = 0.0;
148  % calculate the angle of each vertex
149  % of Minkowski sum polygon
```

```
150  angle = zeros(1,2*n);
151  for i=1:2*n
152      [angle(1,i)] = angleCalculation(
153  mink(i).x_start,...
154          mink(i).y_start);
155  end
156  % Calculate the area of the initial polygon
157  polygonArea = 0;
158  for i = 1:n-1
159      polygonArea = polygonArea+a(i).x_start*...
160          a(i+1).y_start-a(i+1).x_start*a(i).y_start;
161  end
162  polygonArea = polygonArea + a(n).x_start*...
163      a(1).y_start-a(1).x_start*a(n).y_start;
164  polygonArea = polygonArea/2;
165  stripWidth = 0;
166  % initialize efficiency coefficient with a very small
167  value
168  efficiency = 0.0;
169  optimumStripWidth = 0;
170  optimumPitch = 0;
171  % rotate the polygon clockwise with angle angle(i)
172  for i=1:2*n
173      for j = 1:2*n
174          xPrim = mink(j).x_start*cos(angle(1,i))+...
175              mink(j).y_start*sin(angle(1,i));
176          yPrim = -mink(j).x_start*sin(angle(1,i))+...
177              mink(j).y_start*cos(angle(1,i));
178          mink(j).x_start = xPrim;
179          mink(j).y_start = yPrim;
180          %
181          if (mink(j).y_start>stripWidth)
182              stripWidth = mink(j).y_start;
183          end
184      end
185      %
186      pieceBridge = abs(mink(i).x_start);
187      stripArea = stripWidth*pieceBridge;
188      efficiencyCoefficient = polygonArea/stripArea;
189      % if a maximum strip width is allowed, then set
190      % maxStripWidth at line 19
191      % Then set line 183 and 184 active,
192      % instead of line 185.
193      %if(efficiencyCoefficient>efficiency && ...
194      % stripWidth<=maxStripWidth)
```

```
195         if(efficiencyCoefficient>efficiency)
196             efficiency = efficiencyCoefficient;
197             optimumStripWidth = stripWidth;
198             optimumPitch = pieceBridge;
199             angleOptim = angle(1,i);
200         end
201         mink = m;
202     end
203     %
204     subplot(2,2,1);
205     axis equal;
206     xa = zeros(n+1);
207     ya = zeros(n+1);
208     for i = 1:n
209         xa(i) = a(i).x_start;
210         ya(i) = a(i).y_start;
211     end
212     xa(n+1) = xa(1);
213     ya(n+1) = ya(1);
214     line(xa,ya,'Color','b','LineWidth',2);
215     subplot(2,2,2);
216     axis equal;
217     % draw the polygon of Minkowski sum
218     u = zeros(2*n+1,1);
219     v = zeros(2*n+1,1);
220     for i=1:2*n
221         u(i) = m(i).x_start;
222         v(i) = m(i).y_start;
223     end
224     u(2*n+1) = u(1);
225     v(2*n+1) = v(1);
226     line(u,v,'Color','r','LineWidth',2);
227     % mark the center of the polygon of Minkowski sum
228     hold on;
229     scatter(xCenter, yCenter, 'markerfacecolor','r');
230     hold off;
231     %
232     % rotate the polygon with optimum angle
233     xRotated = zeros(n+1);
234     yRotated = zeros(n+1);
235     for j = 1:n
236         xRotated(j) = a(j).x_start*cos(angleOptim)+...
237             a(j).y_start*sin(angleOptim);
238         yRotated(j) = -a(j).x_start*sin(angleOptim)+...
239             a(j).y_start*cos(angleOptim);
```

```
240  end
241  xRotated(n+1) = xRotated(1);
242  yRotated(n+1) = yRotated(1);
243  subplot(2,2,3);
244  axis equal;
245  line(xRotated,yRotated,'Color','b','LineWidth',2);
246  subplot(2,2,4);
247  axis equal;
248  % draw several (let us say 5) pieces
249  % in stamping position
250  %
251  % step value between stamped pieces
252  pieceBridge = 0.5;
253  optimumPitch = optimumPitch+pieceBridge;
254  line(xRotated,yRotated,'Color','b','LineWidth',2);
255  xMultiple = zeros(n+1);
256  yMultiple = zeros(n+1);
257  % 4 = the number of drawn pieces on the strip - 1.
258  for k = 1:4
259      for j = 1:n
260          xMultiple(j) = xRotated(j)+k*optimumPitch;
261          yMultiple(j) = yRotated(j);
262      end
263      xMultiple(n+1) = xMultiple(1);
264      yMultiple(n+1) = yMultiple(1);
265      %
266      hold on;
267      line(xMultiple,yMultiple,'Color','r',...
268      'LineWidth',2);
269  end
270  hold off;
271  %
272  % A series of variables are deleted from workspace,
273  % in order to make the result more clear
274  clear m b c stripArea polygonArea;
275  clear efficiencyCoefficient crossProduct i index;
276  clear p pol polygon indexLow j k stripWidth n;
277  clear pieceBridge u v mink;
278  clear angle xCenter yCenter xMult yMult xPrim;
279  clear yPrim xRotated yRotated xa ya;
280  %
```

Code 9.1 *nesting2D.m*.

The program draws (see Fig. 9.16) the polygon A in the initial position corresponding to the coordinates read from the file *polygonA.txt* at line 9.

Then the Minkowski sum polygon is drawn and its center is pointed. The polygon A is drawn once again, this time in the rotated position, that corresponds to the maximum value of the material consumption coefficient. At the same time, the polygon A is drawn in several successive positions corresponding to the optimal nesting, according to the obtained results. In addition to the pitch distance between the successive positions of the profile on the material strip, at line 252 a *pieceBridge* value can be presumed.

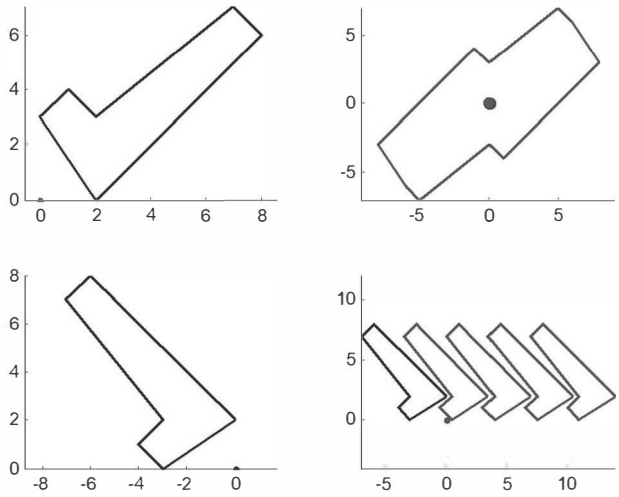

Fig. 9.16 Graphical display of *nesting2D.m* results

If a width limit of the material strip is required, denoted *maxSTripWidth*, the code line 19 will be activated. Then lines 183 and 184 will be activated and line 185 will be disabled, deleting or inserting the percent character '%' at the beginning of the respective line. In this way, the program will choose the optimal position of the polygon on the material strip so that the material consumption coefficient is maximum, but at the same time not exceed the *width* imposed for the material strip. To run the program from Code 9.10, the next functions are called: *angleCalculation.m* (see Code 9.11), *checkSegmentDirection.m* (see Code 9.12), *minkowskiSumConvex.m* (see Code 9.13), *polygonSegmentSlope.m* (see Code 9.14) and *translate_m_k.m* (see Code 9.15).

```
%
%            angleCalculation.m
%
% This function calculates the angle of a point
% relative to the origin of the coordinate system
%
function [ angle ] = angleCalculation( x,y )
%
% case 1
    if x>0 && y == 0
        angle = 0;
    end
    %
    % case 2
    if x>0 && y>0
        angle = atan(y/x);
    end
    %
    % case 3
    if x==0 && y>0
        angle = pi/2;
    end
    %
    % case 4
    if x<0 && y>0
        angle = atan(abs(y/x));
        angle = pi - angle;
    end
    %
    % case 5
    if x<0 && y==0
        angle = pi;
    end
    %
    % case 6
    if x<0 && y<0
        angle = atan(y/x);
        angle = pi + angle;
    end
    %
    % case 7
    if x==0 && y<0
        angle = 3*pi/2;
    end
    %
```

```
46      % case 8
47      if x>0 && y<0
48          angle = atan(abs(y/x));
49          angle = 2*pi - angle;
50      end
51  end
52  %
```

Code 9.11 *angleCalculation.m*.

```
1   %
2   %         checkSegmentDirection.m
3   %
4   % this function checks the orientation of
5   % segment b(j)
6   % if the direction is to the right, then
7   % the segment is added to m(k)
8   %
9   function [ z,j,m,k ] = ...
10          checkSegmentDirection( z,j,m,k,n )
11  %
12      while (strcmp(z(j).direction,'right') == 1 ...
13          && z(j).marker == 0)
14          m(k) = z(j);
15          z(j).marker = 1;
16          j = j +1;
17          %
18          if j > n
19              j = n;
20          end
21          k = k + 1;
22      end
23  end
24  %
```

Code 9.12 *checkSegmentDirection.m*.

```
1   %
2   %             minkowskiSumConvex.m
3   %
4   % This function assumes that the polygon
5   % is convex.
6   % If the  polygon is concave, the function
7   % is able to run, but in some cases does
8   % not work correctly.
9   % This is why, you should first use
10  %      MinkowskiConcave.m
```

```
% to calculate the polygon of the Minkowsky sum.
%
function [a,m] = minkowskiSumConvex(a,b,m,n)
i = 1;
j = 1;
k = 1;
%
while k <= 2*n
    %
    if (j<n && a(i).alfa == b(j).alpha && ...
            strcmp(b(j+1).direction,'right') == 1)
        m(k) = b(j);
        b(j).marker = 1;
        j = j + 1;
        %
        if j > n
            j = n;
        end
        k = k + 1;
        %
        [b,j,m,k] = ...
checkSegmentDirection(b,j,m,k,n);
    %
    elseif (i<n && a(i).alpha == b(j).alpha && ...
            strcmp(a(i+1).direction,'right') == 1)
        m(k) = a(i);
        a(i).marker = 1;
        i = i + 1;
        %
        if i > n
            i = n;
        end
        k = k + 1;
        %
        [a,i,m,k] = ...
checkSegmentDirection(a,i,m,k,n);
    end
    %
    if (i<=n && a(i).alpha==b(j).alpha &&
a(i).marker==0)
        m(k) = a(i);
        a(i).marker = 1;
        i = i + 1;
        %
        if i > n
```

```
            i = n;
        end
        k = k + 1;
        %
        [a,i,m,k] = ...
checkSegmentDirection(a,i,m,k,n);
        %
        if (b(j).marker == 0)
            m(k) = b(j);
            b(j).marker = 1;
            j = j + 1;
            %
            if j > n
                j = n;
            end
            k = k + 1;
            %
            [b,j,m,k] = ...
checkSegmentDirection(b,j,m,k,n);
        %
        end
    end
    %
    if (a(i).alpha<b(j).alpha && a(i).marker==0 ...
 && i<=n)
        m(k) = a(i);
        a(i).marker = 1;
        i = i + 1;
        %
        if i > n
            i = n;
        end
        k = k + 1;
        %
        [a,i,m,k] = ...
checkSegmentDirection(a,i,m,k,n);
    %
    end
    %
    if (a(i).alpha > b(j).alpha && b(j).marker== ...
0 && j <= n)
        m(k) = b(j);
        b(j).marker = 1;
        j = j + 1;
        %
```

```
101             if j > n
102                 j = n;
103             end
104             k = k + 1;
105             %
106             [b,j,m,k] = ...
107     checkSegmentDirection(b,j,m,k,n);
108         end
109         %
110         if (a(n).marker == 1 || b(n).marker == 1)
111             break
112         end
113         %
114     end
115     %
116     while k<= 2*n
117         if (a(n).marker == 1 && b(j).marker ==0)
118             m(k) = b(j);
119             b(j).marker = 1;
120             j = j + 1;
121             k = k + 1;
122         end
123         %
124         if (b(n).marker == 1 && a(i).marker == 0)
125             m(k) = a(i);
126             a(i).marker = 1;
127             i = i + 1;
128             k = k + 1;
129         end
130     end
131     %
132     % join the segments of the Minkowski sum polygon
133     k = 2;
134     while k<= 2*n
135         m = translate_m_k(m,k);
136         k = k + 1;
137     end
138     %
139     % calculate the coordinates of the center
140     % of Minkowski sum polygon
141     xCenter = (m(1).x_start+m(n).x_final)/2;
142     yCenter = (m(1).y_start+m(n).y_final)/2;
143     %
144     % translate the center of the Minkowski
145     % sum polygon in the origin
```

```
146   for i=1:2*n
147       m(i).x_start = m(i).x_start - xCenter;
148       m(i).x_final = m(i).x_final - xCenter;
149       m(i).y_start = m(i).y_start - yCenter;
150       m(i).y_final = m(i).y_final - yCenter;
151   end
152   %
```

Code 9.13 *minkowskiSumConvex.m*.

```
1   %
2   %          polygonSegmentSlope.m
3   %
4   % This function calculates the slope of a polygon
5   % segments, and fills the structure "a"
6   %
7   function [a] = polygonSegmentSlope(a,pol,n)
8   % the calculation of slope
9   for i = 1:n
10      % case1
11      if pol(i).x_final>0 && pol(i).y_final == 0
12          a(i).alpha = 0;
13          a(i).alpha = a(i).alpha*180/pi;
14      end
15      %
16      % case 2
17      if pol(i).x_final>0 && pol(i).y_final>0
18          a(i).alpha =
19  atan(pol(i).y_final/pol(i).x_final);
20          a(i).alpha = a(i).alpha*180/pi;
21      end
22      %
23      % case 3
24      if pol(i).x_final==0 && pol(i).y_final>0
25          a(i).alpha = pi/2;
26          a(i).alpha = a(i).alpha*180/pi;
27      end
28      %
29      % case 4
30      if pol(i).x_final<0 && pol(i).y_final>0
31          a(i).alpha =
32  atan(abs(pol(i).y_final/pol(i).x_final));
33          a(i).alpha = pi - a(i).alpha;
34          a(i).alpha = a(i).alpha*180/pi;
35      end
36      %
```

```
37        % case 5
38        if pol(i).x_final<0 && pol(i).y_final==0
39            a(i).alpha = pi;
40            a(i).alpha = a(i).alpha*180/pi;
41        end
42        %
43        % case 6
44        if pol(i).x_final<0 && pol(i).y_final<0
45            a(i).alpha =
46    atan(pol(i).y_final/pol(i).x_final);
47            a(i).alpha = pi + a(i).alpha;
48            a(i).alpha = a(i).alpha*180/pi;
49        end
50        %
51        % case 7
52        if pol(i).x_final==0 && pol(i).y_final<0
53            a(i).alpha = 3*pi/2;
54            a(i).alpha = a(i).alpha*180/pi;
55        end
56        %
57        % case 8
58        if pol(i).x_final>0 && pol(i).y_final<0
59            a(i).alpha =
60    atan(abs(pol(i).y_final/pol(i).x_final));
61            a(i).alpha = 2*pi - a(i).alpha;
62            a(i).alpha = a(i).alpha*180/pi;
63        end
64    end
65    return;
66    %
```

Code 9.14 *polygonSegmentSlope.m*.

```
1     %
2     %       translate_m_k.m
3     %
4     % the function translates segment m(k+1)
5     % near segment m(k)
6     %
7     function [ m ] = translate_m_k( m,k )
8     %
9     lx = m(k).x_final - m(k).x_start;
10    ly = m(k).y_final - m(k).y_start;
11    %
12    m(k).x_start = m(k-1).x_final;
13    m(k).y_start = m(k-1).y_final;
```

```
14    m(k).x_final = m(k-1).x_final + lx;
15    m(k).y_final = m(k-1).y_final + ly;
16    end
17    %
```

Code 9.15 *translate_m_k.m.*

9.7 Conclusions

●ptimal nesting is an actual issue, with major implications in many fields of industry. During this chapter the theoretical foundations of positioning a profile on the material strip were briefly presented. Simple to complex problems have been presented and various solutions have been proposed to solve the optimal nesting problems. A program has been designed to find the optimal position of a profile on the material strip even when the bandwidth is limited.

10.

FLOWSHOP SCHEDULING PROBLEM

10.1 Introduction

Flowshop Scheduling Problem plays an important role in the field of *NP*[1]*-hard* combinatoric optimization problems and, despite its apparent simplicity, it is very difficult to optimally solve in polynomial time. The problem can be stated in the following way: let us consider a large number of n permutable jobs (or batches of parts) that need to be processed on different m machines in a particular order. The objective is to determine the processing order of jobs so that the total processing time, noted C_{max} (total completion time or makespan) is minimal. The processing times of the jobs on the machines, denoted t_{ij} ($i = 1,2,..., n$, $j = 1,2,..., m$), are known in advance, their values positive constants.

We will admit the following:

a) All jobs are independent and available for processing at time $t = 0$;

b) All machines are always available;

c) Each job can be processed at one time on only one machine;

d) Each machine may at one time execute only a single job;

e) Once a processing of a job has begun, it cannot be interrupted;

f) Preparation-end (set-up) times are included in the manufacturing times;

g) The auxiliary times of organizing the jobs on each of the machines are neglected;

h) If the machine that follows the technological itinerary is unavailable (being busy with another job) the following job in the list will be assigned to a waiting list.

A comprehensive list of these initial assumptions, grouped by category, is given in (Gupta & Stafford, 2006). When the order in which a job passes through the machines is the same for all jobs, the problem is called *Permutation flowshop Scheduling problem* (*PFSP*).

---

[1] Non-deterministic polynomial-time hardness.

As a rule, *PFSP* solving techniques are grouped into two categories, that are: *constructive heuristics* and *improvement heuristics*. The first category of techniques, namely *constructive heuristics*, build a possible solution based on certain decisions, while the techniques of the second category, i.e. *improvement heuristics*, improve through certain methods the solution offered by the first category of techniques.

The algorithm of Johnson (1954) is a classic method that finds the optimal solution, in case of problems with n jobs on two machines, problems being solvable in a polynomial time. However, if the number of machines becomes greater than three, the problems become *NP-Complete*, although there are certain special conditions in which Johnson's algorithm can also be applied to $m = 3$. To solve the general problem *PFSP* (ordering of jobs on m machines), the heuristic *CDS* algorithm (Campbell et al,1970) groups the m machines into two groups, these groups being considered as two virtual machines, after which it applies Johnson's algorithm. A number of $(m-1)$ possible processing sequences are drafted, with the best variant being chosen among them. The *HFC* heuristic algorithm first uses Johnson's algorithm, following in the second part by the improvement of the previously found variant (Koulamas,1988).

Other methods for solving the problem are based on assigning a weighted coefficient (called slope index) to each job, and by sorting the jobs in descending order, using these weighted coefficients as a sorting key, and in the end building the schedule based on the list of the jobs thus sorted (Palmer, 1965; Gupta, 1971; Hundal & Rajgopal, 1988). The *RA* (*Rapid Access*) algorithm is a combination of the previous methods based on Johnson's algorithm and the *Palmer* algorithm (Dannenbring, 1977). Two virtual machines are defined, as in the case of the heuristic *CDS* algorithm. Two weighting schemes are constructed, one for each of the virtual machines, and finally the algorithm of Johnson is applied.

The *NEH* heuristic algorithm of Nawaz, Enscore & Ham (1983) is considered as one of the most powerful heuristic algorithms. It is based neither on Johnson's algorithm, nor on the idea of the weight indices assigned in a particular way to the jobs. Instead, in the *NEH* algorithm, jobs are sorted in descending order, using the total manufacturing time corresponding to each job taken individually, as a sorting key. The first two jobs in the sorted list are then taken, and their total processing time is evaluated in the two possible variants. The best choice between the two variants is retained. Then the third job is taken from the list and the total processing time of the three jobs is evaluated, by placing it in each of the three possible situations. The order corresponding to the minimum manufacturing time shall be retained again. Analogous to the way the third

job was placed, the algorithm continues, in turn, with all other remaining jobs, in the order corresponding to the list sorted at the first stage. In this way, the entire processing sequence is constructed, by placing each job j_k $(2 < k \leq n)$ in the most favourable position towards the $j_1, j_2, ..., j_{k-1}$ already ordered. Therefore, $n(n-1)/2-1$ sequence assessments are required to reach the final result. Also, several other heuristic methods are based on *NEH*, which, unlike this one, suggest different starting sequences (Framinan et al, 2003). The heuristic *SPIRIT* algorithm (Widmer & Hertz, 1989) is also of the *constructive* type, the method approached to construct the schedule being similar to that of an *insertion* type of algorithm. The first two jobs j_i, j_k in the list are chosen so that the processing time needed for these two jobs is minimal. Next, the following jobs, taken in random order, are placed temporarily in that position in the list corresponding to a minimum increase in the manufacturing time.

Unlike the *constructive euristics* algorithms, the *improvement heuristic* algorithms start from an already constructed processing sequence and through different techniques try to improve the result. The techniques are usually based on the simple permutation of some neighboring jobs (Damenbring, 1977) such as *Rapid Access with Close Order Search (RACS)* and *Rapid Access with Extensive Search (RAES)*, both variants built from the original sequence with the *RA*, or, using *Cmax* as an optimization criterion, minimizing intermediate times (Ho & Chang, 1991). Agarwal (2006) starts from three known heuristic methods *Palmer*, *CDS* and *NEH*, and use them for generating a good start solution. They use a weight parameter for optimizing the start solution. The weight parameter is continuously adjusted on the basis of an adaptive learning strategy that also involves backtracking techniques, similar to strategies involving neuronal networks.

It is worth mentioning a number of other heuristic methods whose principles are used in solving combinatorical optimization problems and are also applied to problems of flowshop scheduling. The best known methods are the methods based on the simulation of recrystallization - *Simulated annealing* (Osman & Potts, 1989; Taillard, 1990; Ogbu & Smith, 1990), *Tabu Search* (Moccellin, 1995; Nowicki & Smutnicki, 1996) or methods based on *genetic algorithms* (Reeves & Yamada, 1998; Ponnambalam et al, 2001) etc. A comprehensive study on the performance of these methods can be found in (Ruiz & Morato, 2005).

The flowshop scheduling problems fall into the wide range of operational research. This area of research is vast and important at the same time. A decisive argument in this direction is evident in the large number of ISI-quoted journals, dealing with problems in this area and

quoting this topic even in the journal title. In this sense it is worth to mention journals such as *European Journal of Operational Research*, *Computers & Operations Research* or *Operations Research Letters* from *Elsevier*, *Annals of Operations Research*, *Operations Management Research-Advancing Practice through Theory*, *Mathematical Methods of Operations Research* from *Springer*, *International Journal of Operations and Production Management* from *Emerald* and so on. Another important aspect in solving flowshop scheduling problems is ascertaining the performances of the designed algorithms. A good way to check the algorithm performances is to solve the set of instances proposed by Eric Taillard from the University of Applied Siences of Western Switzerland or the proposed instances in *OR-Library*. Each problem is presented as a matrix with m lines and n columns. Each column of the matrix contains production times of a job on each machine. These values of productive times are randomly generated. The algorithm for generating these values is also indicated. The number of matrix lines is equal to the number of machines. For each set an estimated value of the optimum is known – *lower bound*, and the best value found is deemed the *upper bound*. So, in evaluating the algorithms' performances, references can be made to these upper and lower bounds.

10.2 Total inactivity time calculation

In order to find the optimal processing order of n different jobs on p machines, so that the manufacturing time is minimum, we must take into account the following considerations:

1. Note with T the total time between the start of processing the first job on the M_1 machine and the end of processing the last job on the M_p machine;
2. Let us consider $(P_{j1},P_{j2},...,P_{jn})$ an arbitrary jobs flow order on the p machines;
3. We denote by X_{jq} the waiting time elapsed between the end of the job P_{jq-1} on the M_i machine and the beginning of the job P_{jq} on the M_i machine.

We have:

$$T = \sum_{r=1}^{n} A_{pjr} + \sum_{r=1}^{n} X_{jr}; \qquad (10.1)$$

where $A_{pjr} = t_{jr}$ is the processing time of the job P_{jr} on the M_p machine. In the following, we will use the notations of Johnson (1954), and we will develop, step by step, the necessary equations to calculate the waiting

times, depending only on the processing times. Since the A_{jr} time values are constant for $r = 1, 2,..., n$, in order to minimize T is sufficient to minimize the amount:

$$\sum_{r=1}^{n} X_{jr}; \tag{10.2}$$

Let us consider the example in Fig. 10.1 of the processing of n jobs on 3 machines. According to Fig. 10.1, the processing time of the job P_{j1} product on the M_1 machine is A_{1j1}. On the machine M_2, the waiting time will be:

$$X_{j1}^{(2)} = A_{j1}; \tag{10.3}$$

On the machines M_3, M_4,..., M_p, the waiting times will be determined by the relationships:

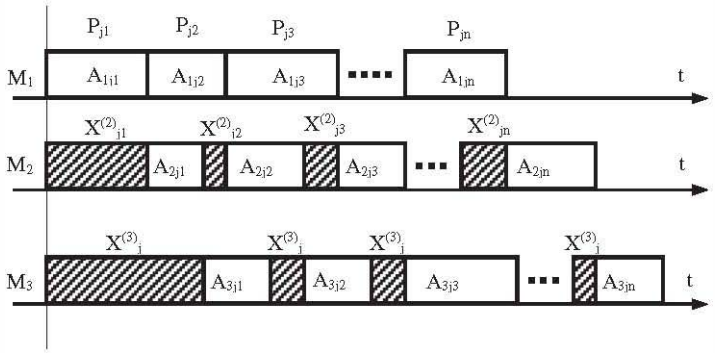

Fig. 10.1 *Gantt* diagram for processing n parts on three machines.

$$X_{j1}^{(3)} = A_{j1}^{(1)} + A_{j1}^{(2)}; \tag{10.4}$$

$$X_{j1}^{(4)} = A_{j1}^{(1)} + A_{j1}^{(2)} + A_{j1}^{(3)}; \tag{10.5}$$

$$\cdots$$

$$X_{j1}^{(p)} = A_{j1}^{(1)} + A_{j1}^{(2)} + \cdots + A_{j1}^{(p-1)}; \tag{10.6}$$

The waiting time X_{j2} on machine M_2 is calculated as follows:

$$X_{j1}^{(2)} = A_{j1}^{(1)} + A_{j2}^{(1)} - A_{j1}^{(2)} - X_{j1}^{(2)}; \tag{10.7}$$

if

$$A_{j1}^{(1)} + A_{j2}^{(1)} \geq A_{j1}^{(2)} + X_{j1}^{(2)}; \tag{10.8}$$

or

$$X_{j1}^{(2)} = 0; \tag{10.9}$$

if

$$A_{j1}^{(1)} + A_{j2}^{(1)} < A_{j1}^{(2)} + X_{j1}^{(2)}; \tag{10.10}$$

For machine M_3:

$$X_{j2}^{(3)} = A_{j1}^{(1)} + A_{j2}^{(1)} + A_{j2}^{(2)} - A_{j1}^{(3)} - X_{j1}^{(3)}; \tag{10.11}$$

if

$$A_{j1}^{(1)} + A_{j2}^{(1)} + A_{j2}^{(2)} \geq A_{j1}^{(3)} + X_{j1}^{(3)}; \tag{10.12}$$

or

$$X_{j2}^{(3)} = 0; \tag{10.13}$$

if

$$A_{j1}^{(1)} + A_{j2}^{(1)} + A_{j2}^{(2)} < A_{j1}^{(3)} + X_{j1}^{(3)}; \tag{10.14}$$

For machine M_4:

$$X_{j2}^{(4)} = A_{j1}^{(1)} + A_{j2}^{(1)} + A_{j2}^{(2)} + A_{j2}^{(3)} - A_{j1}^{(4)} - X_{j1}^{(4)}; \tag{10.15}$$

if

$$A_{j1}^{(1)} + A_{j2}^{(1)} + A_{j2}^{(2)} + A_{j2}^{(3)} \geq A_{j1}^{(4)} + X_{j1}^{(4)}; \tag{10.16}$$

or

$$X_{j2}^{(4)} = 0; \tag{10.17}$$

if

$$A_{j1}^{(1)} + A_{j2}^{(1)} + A_{j2}^{(2)} + A_{j2}^{(3)} < A_{j1}^{(4)} + X_{j1}^{(4)}; \tag{10.18}$$

For machine M_p, the waiting time $X^{(p)}{}_{j2}$ is calculated with the following relationships:

$$X_{j2}^{(p)} = A_{j1}^{(1)} + A_{j2}^{(1)} + \sum_{i=1}^{p-1} A_{j2}^{(i)} - A_{j1}^{(p)} - X_{j1}^{(p)}; \tag{10.19}$$

if

$$A_{j1}^{(1)} + A_{j2}^{(1)} + \sum_{i=1}^{p-1} A_{j2}^{(i)} \geq A_{j1}^{(p)} + X_{j1}^{(p)}; \tag{10.20}$$

or

$$X_{j2}^{(p)} = 0; \tag{10.21}$$

if

$$A_{j1}^{(1)} + A_{j2}^{(1)} + \sum_{i=1}^{p-1} A_{j2}^{(i)} < A_{j1}^{(p)} + X_{j1}^{(p)}; \tag{10.22}$$

Eq. 10.7 to Eq. 10.22 can be written condensed in the form:

$$\begin{cases} X_{j2}^{(2)} = \max\left(A_{j1}^{(1)} + A_{j2}^{(1)} - A_{j1}^{(2)} - X_{j1}^{(2)}, 0\right); \\ X_{j2}^{(3)} = \max\left(A_{j1}^{(1)} + A_{j2}^{(1)} + A_{j2}^{(2)} - A_{j1}^{(3)} - X_{j1}^{(3)}, 0\right); \\ X_{j2}^{(4)} = \max\left(A_{j1}^{(1)} + A_{j2}^{(1)} + A_{j2}^{(2)} + A_{j2}^{(3)} - A_{j1}^{(4)} - X_{j1}^{(4)}, 0\right); \\ \cdots \\ X_{j2}^{(p)} = \max\left(A_{j1}^{(1)} + \sum_{i=1}^{p-1} A_{j2}^{(i)} - A_{j1}^{(p)} - X_{j1}^{(p)}, 0\right); \end{cases} \tag{10.23}$$

$$\begin{cases} X_{j3}^{(2)} = \max\left(\sum_{k=1}^{2} A_{jk}^{(1)} - \sum_{k=1}^{2} A_{jk}^{(2)} - \sum_{k=1}^{2} X_{jk}^{(2)}, 0\right); \\ X_{j3}^{(3)} = \max\left(\sum_{k=1}^{2} A_{jk}^{(1)} + \sum_{i=1}^{3} A_{j3}^{(i)} - \sum_{k=1}^{2} A_{jk}^{(3)} - \sum_{k=1}^{2} X_{jk}^{(3)}, 0\right); \\ \cdots \\ X_{j3}^{(p)} = \max\left(\sum_{k=1}^{2} A_{jk}^{(1)} + \sum_{i=1}^{p} A_{j3}^{(i)} - \sum_{k=1}^{2} A_{jk}^{(p)} - \sum_{k=1}^{2} X_{jk}^{(p)}, 0\right); \end{cases} \tag{10.24}$$

$$\begin{cases} X_{jn}^{(2)} = \max\left(\sum_{k=1}^{n-1} A_{jk}^{(1)} - \sum_{k=1}^{n-1} A_{jk}^{(2)} - \sum_{k=1}^{n-1} X_{jk}^{(2)}, 0\right); \\ X_{jn}^{(3)} = \max\left(A_{jn}^{(2)} + \sum_{i=1}^{n} A_{jk}^{(i)} - \sum_{k=1}^{n-1} A_{jk}^{(3)} - \sum_{k=1}^{n-1} X_{jk}^{(3)}, 0\right); \\ \cdots \\ X_{jn}^{(p)} = \max\left(\sum_{k=1}^{n-1} A_{jk}^{(1)} + \sum_{i=1}^{p} A_{jn}^{(i)} - \sum_{k=1}^{n-1} A_{jk}^{(p)} - \sum_{k=1}^{n-1} X_{jk}^{(p)}, 0\right); \end{cases} \tag{10.25}$$

The total waiting time on machine M_2, denoted $X^{(2)}$ is:

$$\begin{aligned} X^{(2)} = max\Big\{A_{j1}^{(1)}, A_{j1}^{(1)} + A_{j2}^{(1)} - A_{j1}^{(2)} - X_{j1}^{(2)}, \sum_{k=1}^{3} A_{jk}^{(1)} - \\ \sum_{k=1}^{2} A_{jk}^{(2)} - \sum_{k=1}^{2} X_{jk}^{(2)}, \ldots, \sum_{k=1}^{n-1} A_{jk}^{(1)} - \sum_{k=1}^{n-1} A_{jk}^{(2)} - \sum_{k=1}^{n-1} X_{jk}^{(2)}, 0\Big\}; \end{aligned} \tag{10.26}$$

In a similar way, on machine M_3, the total waiting time is:

$$\begin{aligned} X^{(3)} = max\Big\{A_{j1}^{(1)} + A_{j1}^{(2)}, A_{j1}^{(1)} + A_{j2}^{(1)} + A_{j1}^{(2)} - A_{j1}^{(3)} - \\ X_{j1}^{(3)}, \sum_{k=1}^{2} A_{jk}^{(1)} + \sum_{i=1}^{3} A_{j3}^{(i)} - \sum_{k=1}^{2} A_{jk}^{(3)} - \sum_{k=1}^{2} X_{jk}^{(3)}, \ldots, A_{jn}^{(2)} + \\ \sum_{k=1}^{n} A_{jk}^{(1)} - \sum_{k=1}^{n-1} A_{jk}^{(3)} - \sum_{k=1}^{n-1} X_{jk}^{(3)}, 0\Big\}; \end{aligned} \tag{10.27}$$

On machine M_p, the total waiting time is:

$$X^{(p)} = max\left\{A_{j1}^{(1)} + A_{j1}^{(2)} + \cdots + A_{j1}^{(p-1)}, A_{j1}^{(1)} + \sum_{i=1}^{p-1} A_{j2}^{(i)} - A_{j1}^{(p)} - X_{j1}^{(p)}, \sum_{k=1}^{2} A_{jk}^{(1)} + \sum_{i=1}^{p} A_{j3}^{(i)} - \sum_{k=1}^{2} A_{jk}^{(2)} - \sum_{k=1}^{2} X_{jk}^{(2)}, \ldots, \sum_{k=1}^{n-1} A_{jk}^{(1)} + \sum_{i=1}^{p} A_{jn}^{(i)} - \sum_{k=1}^{n-1} A_{jk}^{(p)} - \sum_{k=1}^{n-1} X_{jk}^{(p)}, 0\right\}; \quad (10.28)$$

Making the sum of Eq. 10.26 to Eq. 10.28, will get the entire waiting time corresponding to all jobs. So:

$$\sum_{r=1}^{n} X_{jr} = \sum_{i=1}^{p} \sum_{r=1}^{n} X_{jr}^{(i)}; \quad (10.29)$$

The advantage of the Eq. 10.29 is that it expresses the waiting times in the form of expressions dependent exclusively on the machine's processing times. It takes a minimal value when the processing order:

$$S = \left(P_{j1}, P_{j2}, \ldots, P_{jn}\right) \quad (10.30)$$

is optim. Code 10.1 shows a short Matlab program, called *findMatrixX.m*, that calculates the waiting times in the form of a matrix. This matrix is saved in a text file. Each line contains the waiting times that ensue due to processing the jobs, in the order that was read from the input data file. If the reading order of processing times is changed, then another matrix of waiting times is obtained.

```
%                    findMatrixX.m
% This program calculates the waiting times of
% processing n jobs on m machines. The result
% is presented as a matrix, whose lines are
% assigned to machines
% read the processing time matrix
time = load('m5j20_1.txt');
% open a file to save results
fp = fopen('results.txt','w');
% find the number of machines 'm' and jobs 'n'
[m,n] = size(time);
% initialize "tp" as a copy of 'time' matrix
tp = zeros(m,n);
% Initialize the times x on the last machine
totalTx = 0.0;
```

```
for i=1:m
    for j=1:n  tp(i,j) = time(i,j);  end
end
% calculate the waiting times
x = timeX (m,n,tp);
% write matrix X on the file results.txt
for i = 1:m
    for j = 1:n
        fprintf(fp,'%3d    ',x(i,j));
    end
    fprintf(fp,'\n');
end
% waiting times sum on the last machine
for i=1:n
    totalTx = totalTx + x(m,i);
end
% close the file results.txt
fclose(fp);
%
```

Code 10.1 *findMatrixX.m*.

The program *findMatrixX.m* makes a call to a function called *timeX.m* that calculates the waiting times. This function requires as input data the number of machines, m, the number of jobs, n, and the matrix, t_p, of the processing times.

```
%
%                    timeX.m
% This function calculates the 'dead' times matrix. It
% need as input data the number of machines 'm',
% the number of jobs 'n' and the matrix of times. The
% values in X matrix depends on the order of times
% in the 'tp' matrix.
function [x] = timeX(m,n,tp)
% Initialize the waiting time "totalTx" on the last
machine
x = zeros(m,n);
for i=1:n
    x(1,i) = 0.0;
end
for j=2:m
    x(j,1) = x(j-1,1) + tp(j-1,1);
    if(x(j,1) < 0)
        x(j,1) = 0.0;
    end
```

```
end
%
for k=2:m
    for j=2:n
        t1 = 0.0;
        t2 = 0.0;
        t3 = 0.0;
        t4 = 0.0;
        for i=1:j
            t1 = t1 + tp(k-1,i);
            t2 = t2 + x(k-1,i);
        end
        %
        for i=1:j-1
            t3 = t3 + tp(k,i);
            t4 = t4 + x(k,i);
        end
        x(k,j) = t1 + t2 - t3 - t4;
        if(x(k,j) < 0)
            x(k,j) = 0.0;
        end
    end
end
%
```

Code 10.2 *timeX.m*.

10.3 The algorithm of Johnson

Let us consider the process of n different jobs, on two machines. These jobs will follow the same technological itinerary, meaning that each job needs to be processed first on the machine *M1*, then on the machine *M2*. The matrix:

$$T_{2n} = \begin{pmatrix} t_{11} & t_{12} & \cdots & t_{1n} \\ t_{21} & t_{22} & \cdots & t_{2n} \end{pmatrix}; \tag{10.31}$$

represents the times necessary for processing the jobs $(P_1, P_2, \ldots, P_n)$ on two machines. The elements of the matrix T_{2n} on the first line represent the times necessary for processing the jobs on the first machine, while the elements on the second line are the times necessary for processing the jobs on the second machine. For convenience, it is admitted that the auxiliary times are included in the processing times.

The algorithm of Johnson consists of the following steps:

Step 1. In the matrix T_{2n}, look for the minimum element. Assume that the minimum element is t_{ij}.

a) if $i = 1$, which means that the minimum element is found on the first line of the matrix, then we will start processing with job P_j;

b) if $i = 2$, which means that the minimum element is found on the second line of the matrix, then we will end the processing with job P_j.

Step 2. Remove column j from the matrix T_{2n};

Step 3. Repeat step 1 until the optimum order is completed for all n jobs.

The program *johnson.m* in Code 10.3 follows just the above steps. The input data in this program are represented by the elements of the processing times matrix. Let us consider the case of processing five different pieces, on two machines, for which the matrix of the manufacturing times is:

$$T_{2,5} = \begin{pmatrix} 4 & 7 & 1 & 7 & 5 \\ 2 & 3 & 4 & 6 & 9 \end{pmatrix}; \tag{10.32}$$

```
%                          johnson.m
% this program finds the optimal order for processing
% n jobs on two machines, using Johnson's algorithm
%
% read the matrix of times
time = load('m2j5a.txt');
% open the file 'results.txt'
fp = fopen('results.txt','w');
fprintf(fp,'     the algorithm of Johnson\n\n');
fprintf(fp,'  the matrix of processing times:\n');
% the size of the matrix of times
[m,n] = size(time);
%
for i=1:m
    for j = 1:n
        fprintf(fp,'%3d   ',time(i,j));
    end
    fprintf(fp,'\n');
end
%
% removed(1,n) is a vector that shows which are the
% columns that were removed from time(m,n)
removed = zeros(1,n);
% at start all elements in removed(1,n) are null
for i = 1:n
    removed(1,i) = 0;
```

```
end
optimalOrder = zeros(1,n);
startIndex = 1;
finalIndex = n;
% initially 'minElement' takes the
% maximum value of the matrix 'time'
minElement = max(max(time));
index = 1;
while index <= n
    for j = 1:2
        for i = 1:n
            if (removed(1,i) < 1)
                if (time(j,i) < minElement)
                    minElement = time(j,i);
                    line = j;
                    column = i;
                end
            end
        end
    end
    if line == 1
        optimalOrder(1,startIndex) = column;
        startIndex = startIndex + 1;
    else
        optimalOrder(1,finalIndex) = column;
        finalIndex = finalIndex - 1;
    end
    index = index + 1;
    removed(1,column) = 1;
    minElement = max(max(time));
end
% print to file the optimal order of jobs
fprintf(fp,'\n  the optimal order of jobs is:\n');
for i = 1:n
    fprintf(fp,'%3d   ',optimalOrder(1,i));
end
% new line
fprintf(fp,'\n');
% arrange matrix 'time' using optimal order
optimalTime = zeros(m,n);
for i = 1:n
    for j = 1:2
        optimalTime(j,i) = time(j,optimalOrder(i));
    end
end
```

```
% calculate the waiting times matrix 'x' and
% the total sum of waiting times on the last machine
[x,tx] = timeXplus(m,n,optimalTime);
% print to file the matrix of waiting times
fprintf(fp,'\n  the matrix of waiting times:\n');
for i=1:m
    for j = 1:n
        fprintf(fp,'%3d   ',x(i,j));
    end
    fprintf(fp,'\n');
end
% calculate the totalTime on the second machine (this
% includes the waiting times on the last machine)
totalTime = 0;
for i = 1:n
    totalTime = totalTime + time(2,i);
end
totalTime = totalTime + tx;
% print to file the total processing time
fprintf(fp,'\n  the total processing time:\n');
fprintf(fp,'     totalTime = %6.2f\n',totalTime);
% draw the Gantt diagram
ganttDrawTwoMachines(n,optimalTime);
% close the file 'results.txt'
fclose(fp);
%
```

Code 10.3 *johnson.m*.

To calculate the waiting times on each of the two machines, the program *johnson.m* makes a call to the function *timeXplus.m* (see Code 10.4). From the same program there is a call to the function *ganttDrawTwoMachines.m* (see Code 10.5), function that draws the *Gantt* diagram of the optimal solution.

```
%                 timeXplus.m
% this function calculates the matrix of waiting
times
% as well as the total waiting time 'x' on the last
% machine. This function requires as input data:
% m - the machine number
% n - the number of jobs
% time - the matrix of times
function [x,tx_total] = timeXplus(m,n,time)
%
x = zeros(m,n);
```

```
tp = zeros(m,n);
tx_total = 0.0;
%
for i=1:m
    for j=1:n
        tp(i,j) = time(i,j);
    end
end
%
for i=1:n
    x(1,i) = 0.0;
end
%
for j=2:m
    x(j,1) = x(j-1,1) + tp(j-1,1);
    if(x(j,1) < 0)
        x(j,1) = 0.0;
    end
end
%
for k=2:m
    for j=2:n
        t1 = 0.0;
        t2 = 0.0;
        t3 = 0.0;
        t4 = 0.0;
        for i=1:j
            t1 = t1 + tp(k-1,i);
            t2 = t2 + x(k-1,i);
        end
        for i=1:j-1
            t3 = t3 + tp(k,i);
            t4 = t4 + x(k,i);
        end
        x(k,j) = t1 + t2 - t3 - t4;
        if(x(k,j) < 0)
            x(k,j) = 0.0;
        end
    end
end
for i=1:n
    tx_total = tx_total + x(m,i);
end
return
```

Code 10.4 *timeXplus.m*.

```
%
%          ganttDrawTwoMachines.m
%
% this function draws the Gantt diagram on
% two machines only
function [] = ganttDrawTwoMachines(n,optimalTime)
% color codes
c1 = rand;
c2 = rand;
c3 = rand;
axis equal;
m1 = 0;
%
rectangle('Position',[m1,15,optimalTime(1,1),3],...
    'LineWidth',2,'FaceColor',[c1 c2
c3],'LineStyle','-');
m2 = optimalTime(1,1);
%
rectangle('Position',[m2,10,optimalTime(2,1),3],...
    'LineWidth',2,'FaceColor',[c1 c2
c3],'LineStyle','-');
for i = 2:n
    c1 = rand;
    c2 = rand;
    c3 = rand;
    m1 = m1 + optimalTime(1,i-1);

rectangle('Position',[m1,15,optimalTime(1,i),3],...
        'LineWidth',2,'FaceColor',[c1 c2
c3],'LineStyle','-');
    m2 = max(m1 + optimalTime(1,i),m2 +
optimalTime(2,i-1));

rectangle('Position',[m2,10,optimalTime(2,i),3],...
        'LineWidth',2,'FaceColor',[c1 c2
c3],'LineStyle','-');
end
return
```

Code 10.5 *ganttDrawTwoMachines.m*.

The results of the program *johnson.m* are saved in a text file called *results.txt*, which contains the input data, the optimal order of jobs, the matrix of waiting times and the total manufacturing time. For the example in Eq. 10.32, the content of the *results.txt* file can be seen in Table 10.1.

Table 10.1 The results of the program *johnson.m*.

```
    the algorithm of Johnson

the matrix of processing times:
4     7     1     7     5
2     3     4     6     9

the optimal order of jobs is:
3     5     4     2     1

the matrix of waiting times:
0     0     0     0     0
1     1     0     0     0

the total processing time:
   totalTime =  26.00 (min)
```

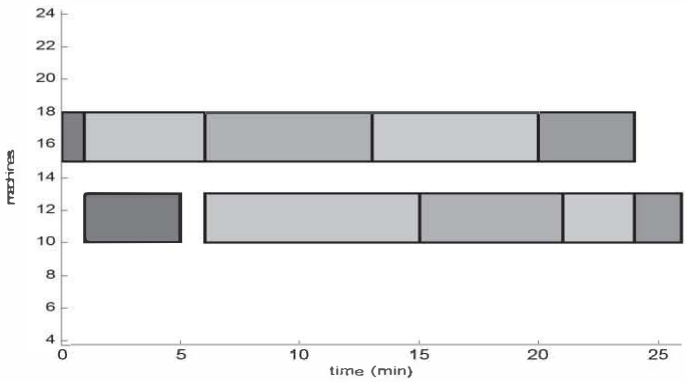

Fig. 10.2 The *Gantt* diagram of the optimal order of jobs.

10.4 Constructive heuristic algorithm

For the design of a constructive heuristic algorithm, we need to imagine a set of rules to find the optimal order of job processing. We can start from the basic idea that the total manufacturing time is the sum of the productive times plus the amount of waiting times on the last machine. As the sum of processing times is a constant, the total manufacturing time will

be shortest when the amount of waiting times on the last of the machines will be minimal. We assume that the order of two jobs $j_i j_k$ is better than the order of $j_k j_i$, if the waiting time $x_{m,2}$ on the last machine is lower in the first order, in relation to the second order (see Fig. 10.3).

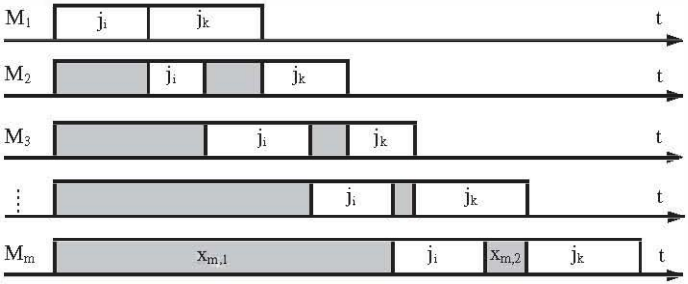

Fig. 10.3 *Gantt* diagram of two jobs on m machines.

This kind of analysis of the waiting time assigns a score 1 to the job j_i, and a score 0 to job j_k, because the job j_i is in front of the job j_k. If the value of the waiting time $x_{m,2}$ is the same for the job pair (j_i, j_k) regardless of their order, both jobs will receive score 0.5. With this rule adopted, we will analyze all possible job pairs and determine the score of each of the jobs in relation to all others. The number of steps necessary for the analysis is $n(n-1)/2$. The jobs will be sorted in descending order of the score thus obtained. According to this set of rules, the job that has the highest score is placed in the front of the ordered list of jobs, and has the highest chance to be followed by a job for which $x_{m,2}$ is zero. However, if the order of two jobs $j_i j_k$ has produced $x_{m,2} = 0$ during the initial analysis, it does not mean that this situation is preserved with their insertion in the manufacturing order. The constructive heuristic algorithm based on these rules follows in Code 10.6.

```
1   %            constructiveHeuristic.m
2   %
3   % read the manufacturing time matrix
4   time = load('m6j20.txt');
5   % find the number of machines 'm' and jobs 'n'
6   [m,n] = size(time);
7   % open a text file to save results
8   fp = fopen('results.txt','w');
9   fprintf(fp,'       Constructive Heuristic
10  Algorithm\n\n');
```

```
11  fprintf(fp,'     Input data:\n');
12  % print the input data to text file
13  for i = 1:m
14      for j = 1:n
15          fprintf(fp,'%3d  ',time(i,j));
16      end
17      fprintf(fp,'\n');
18  end
19  % make a copy 'tp' of matrix 'time'
20  tp = zeros(m,n);
21  for i=1:m
22      for j=1:n
23          tp(i,j) = time(i,j);
24      end
25  end
26  % initialize the line vector of job weights
27  weight = zeros(1,n);
28  % initialize two matrices 'tij' and 'tji' with
29  % m lines and two columns
30  % column i contains the times of job i
31  % column j contains the times of job j
32  tij = zeros(m,2);
33  tji = zeros(m,2);
34  for i = 1:(n-1)
35      for j = i+1:n
36          % generate the matrix with those two jobs i
37  and j
38          tij = tp(:,[i,j]);
39          % calculate the waiting times of the
40          % last line "x(m,2)"
41          x = timeX (m,2,tij);
42          x_ij = x(m,2);
43          % generate the matrix tji, reversing the
44  columns
45          % of the mmatrix tij
46          tji = tp(:,[j,i]);
47          % calculate the dead times of the last line
48  "x(m,2)"
49          x = timeX (m,2,tji);
50          x_ji = x(m,2);
51          % calculate the weight of job i according to
52  job j
53          if x_ij > x_ji
54              weight(1,i) = weight(1,i) + 1;
55          elseif x_ij < x_ji
```

```
            weight(1,j) = weight(1,j) + 1;
        else
            weight(1,i) = weight(1,i) + 0.5;
            weight(1,j) = weight(1,j) + 0.5;
        end
        %
    end
end
% initialize the order of jobs index
index = zeros(1,n);
for i = 1:n
    index(i) = i;
end
% sort the jobs according to their weights
[index] = weightSort(index,weight,n);
% reorder times according to 'index' in tpOrd
tpOrd = zeros(m,n);
for i = 1:n
    for j = 1:m
        tpOrd(j,i) = tp(j,index(1,i));
    end
end
% calculate the matrix of waiting times
x = timeX (m,n,tpOrd);
% 'sumX' is the sum of waiting times on the
% last machine
sumX = 0;
for i = 1:n
    sumX = sumX + x(m,i);
end
% 'sumTu' is the sum of times on the last machine
sumTu = 0;
for i = 1:n
    sumTu = sumTu + time(m,i);
end
% 'totalTime' is the total manufacturing time
totalTime = sumX + sumTu;
% draw the Gantt diagram
ganttDrawing(m,n,tpOrd);
% print to file the optimal order
fprintf(fp,'\n  The optimal order:\n');
for i = 1:n
    fprintf(fp,'%3d   ',index(1,i));
end
fprintf(fp,'\n');
```

```
% print the matrix of the waiting times
fprintf(fp,'\n   The matrix of waiting times:\n');
for i = 1:m
    for j = 1:n
        fprintf(fp,'%3d  ',x(i,j));
    end
    fprintf(fp,'\n');
end
% print the total manufacturing time:
fprintf(fp,'\n   Total manufacturing time:\n');
fprintf(fp,'\n        C_max = %d\n', totalTime);
% close the results.txt file
fclose(fp);
%
```

Code 10.6 *constructiveHeuristic.m*.

The program *constructiveHeuristic.m* calls the function *weightSort.m* (see Code 10.7) that sort the jobs according to their score. The waiting times on each machine are calculated by the function *timeX.m* in Code 10.8. Finally, the function *ganttDrawing.m*, this time a generalized version for m machines, draws the *Gantt* diagram (see Code 10.9).

```
%                    weightSort.m
% this function sorts the jobs according to their
% calculated weights. It takes as actual parameters
% the index of jobs, the calculated weights and the
% job number
function [ index ] = weightSort( index,weight,n )
%
for i = 1:n
    for j = 1:n
        if (weight(1,i) > weight(1,j))
            temp1 = weight(1,i);
            temp2 = index(1,i);
            weight(1,i) = weight(1,j);
            index(1,i) = index(1,j);
            weight(1,j) = temp1;
            index(1,j) = temp2;
        end
    end
end
end
%
```

Code 10.7 *weightSort.m*.

```
%
%                    ganttDrawing.m
%
% This function draws the Gantt diagram of the
% flowshop scheduling solution. The function takes
% as input data the machine number 'm', the job
% number 'm' and the matrix of times 'timeOpt' that
% contains job times as they were ordered.
function [] = ganttDrawing(m,n,timeOpt)
%
% x coordinate of the left-lower point of rectangle
x = zeros(1,m);
% y coordinate of the left-lower point of rectangle
y = zeros(1,m);
y(1,1) = 200;
incremPos = 20;
% the width of rectangles
rectangleWidth = 18;
for i = 2:m
    y(1,i) = y(1,i-1) - incremPos;
end
%
c1 = rand;
c2 = rand;
c3 = rand;
axis equal;
%
x(1,1) = 0;
for k = 2:m
x(1,k) = x(1,k-1)+timeOpt(k-1,1);
end
%
for k = 1:m
rectangle('Position',[x(1,k),y(1,k),timeOpt(k,1),...
    rectangleWidth], 'LineWidth',1,'FaceColor',...
    [c1 c2 c3],'LineStyle','-');
end
%
for i = 2:n
    % set the rectangles' color
    c1 = rand;
    c2 = rand;
    c3 = rand;
    x(1,1) = x(1,1) + timeOpt(1,i-1);
    %
```

```
46      for k = 2:m
47      x(1,k) = max(x(1,k-1) + timeOpt(k-1,i),x(1,k) +...
48          timeOpt(k,i-1));
49      end
50      %
51      for k = 1:m
52          rectangle('Position',...
53          [x(1,k),y(1,k),timeOpt(k,i),...
54          rectangleWidth], 'LineWidth',1,'FaceColor',...
55          [c1 c2 c3],'LineStyle','-');
56      end
57      %
58  end
59  %
```

Code 10.8 *ganttDrawing.m*.

If we consider the *m6n20* problem with 20 jobs on six machines, whose matrix of processing time is:

```
54 83 15 71 77 36 53 38 27 87 76 91 14 29 12 77 32 87 68 94
79  3 11 99 56 70 99 60  5 56  3 61 73 75 47 14 21 86  5 77
16 89 49 15 89 45 60 23 57 64  7  1 63 41 63 47 26 75 77 40
66 58 31 68 78 91 13 59 49 85 85  9 39 41 56 40 54 77 51 31
58 56 20 85 53 35 53 41 69 13 86 72  8 49 47 87 58 18 68 28
79  3 11 99 56 70 99 60  5 56  3 61 73 75 47 14 21 86  5 77
```

Running the constructive heuristic on this example, will result in a C_{max} = 1627 for the following order of jobs:

```
3 9 13 17 12 19 8 11 15 16 20 2 14 10 1 7 6 18 5 4
```

Fig. 10.4 displays the *Gantt* diagram for the problem above. Beside the flow of jobs in the rule based calculated order, this diagram shows many gaps (i.e. waiting times) especially on the last machine. This fact tells two important things to take into consideration. Firstly, attention must be paid to the set of rules that might be improved. Then, another important thing to take into consideration is performing a local search that lead to a better order. Any of these two procedures, or both, are applied, to lead to an improvement of the final result. If a better set of rules is applied, then a better result will emerge, without increasing the running time on the computer significantly. On the other hand, if a local search is performed, this will conduct to a better result, for sure, but the running time on the computer will increase much more that in the first case.

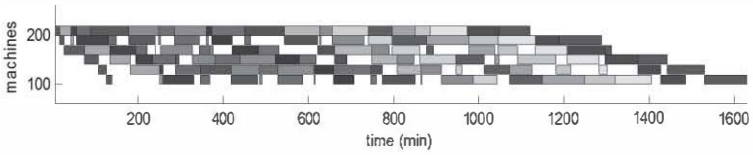

Fig. 10.4 *Gantt* diagram of the *m6n20* problem.

10.5 Improvement heuristic algorithm

An improvement algorithm begins with an existing order of jobs, which is calculated, for example, by a constructive heuristic algorithm, and, using a local search procedure, attempts to improve the existing order. Let us consider the following order of jobs:

$$p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8, p_9, \ldots$$

Let us permute the jobs p_1 and p_2. If this permutation provides a new order of jobs with a lower total manufacturing time, then this permutation is accepted and the new order will be:

$$p_2, p_1, p_3, p_4, p_5, p_6, p_7, p_8, p_9, \ldots$$

Otherwise, the position of jobs p_1 and p_2 remains unchanged. Whatever the result of this permutation is, the procedure continues with the permutation of p_2 and p_3 and so on. Code 10.10 shows the program *improvementHeuristic.m*. This program follows the principle of the *constructiveHeuristic.m* (see Code 10.6) until line 86, where the local search begins. At this line, the function *jobsPermutation.m* is called (see Code 10.11).

```
1   %                improvementHeuristic.m
2   %
3   % read the manufacturing time matrix
4   time = load('m6j20.txt');
5   % find the machine 'm' and job 'n' numbers
6   [m,n] = size(time);
7   % open a text file to save results
8   fp = fopen('results.txt','w');
9   fprintf(fp,'  Improvement Heuristic Algorithm\n\n');
10  fprintf(fp,'     Input data:\n');
11  % print the input data to text file
12  for i = 1:m
```

```
    for j = 1:n
        fprintf(fp,'%3d  ',time(i,j));
    end
    fprintf(fp,'\n');
end
% make a copy 'tp' of matrix 'time'
tp = zeros(m,n);
for i=1:m
    for j=1:n
        tp(i,j) = time(i,j);
    end
end
% initialize the line vector of jobs weights
weight = zeros(1,n);
% initialize two matrices 'tij' and 'tji' with
% m lines and two columns
tij = zeros(m,2);
tji = zeros(m,2);
for i = 1:(n-1)
    for j = i+1:n
        % generate the matrix with those two jobs i
and j
        tij = tp(:,[i,j]);
        % calculate waiting times of the last line
"x(m,2)"
        x = timeX (m,2,tij);
        x_ij = x(m,2);
        % generate the matrix tji, reversing the
columns
        % of the matrix tij
        tji = tp(:,[j,i]);
        % calculate waiting times of the last line
"x(m,2)"
        x = timeX (m,2,tji);
        x_ji = x(m,2);
        % calculate the weight of job i according to
job j
        if x_ij > x_ji
            weight(1,i) = weight(1,i) + 1;
        elseif x_ij < x_ji
            weight(1,j) = weight(1,j) + 1;
        else
            weight(1,i) = weight(1,i) + 0.5;
            weight(1,j) = weight(1,j) + 0.5;
        end
```

```
58          %
59      end
60  end
61  % initialize the order of jobs index
62  index = zeros(1,n);
63  for i = 1:n
64      index(i) = i;
65  end
66  % sort the jobs according to their weights
67  [index] = weightSort(index,weight,n);
68  % reorder times according to 'index' in tpOrd
69  tpOrd = zeros(m,n);
70  for i = 1:n
71      for j = 1:m
72          tpOrd(j,i) = tp(j,index(1,i));
73      end
74  end
75  % calculate the matrix of dead times
76  x = timeX (m,n,tpOrd);
77  % 'sumX' is the sum of waiting times on the last
78  machine
79  sumX = 0;
80  for i = 1:n
81      sumX = sumX + x(m,i);
82  end
83  % 'sumTu' is the sum of times on the last machine
84  sumTu = 0;
85  for i = 1:n
86      sumTu = sumTu + tpOrd(m,i);
87  end
88  timeWithoutLocalSearch = sumX + sumTu;
89  % local search by jobs permutation
90  [index,tpOrd,sumX] = jobsPermutation (m,n,tpOrd,...
91      index,sumX);
92  % 'totalTime' is the total manufacturing time
93  totalTime = sumX + sumTu;
94  % draw Gantt diagram
95  ganttDrawing(m,n,tpOrd);
96  % print to file the optimal order
97  fprintf(fp,'\n  The optimal order:\n');
98  for i = 1:n
99      fprintf(fp,'%3d    ',index(1,i));
100 end
101 fprintf(fp,'\n');
102 % print the matrix of the waiting times
```

```
fprintf(fp,'\n   The matrix of waiting times:\n');
for i = 1:m
    for j = 1:n
        fprintf(fp,'%3d  ',x(i,j));
    end
    fprintf(fp,'\n');
end
% print the total manufacturing time:
fprintf(fp,'\n   Total manufacturing time:\n');
fprintf(fp,'\n        C_max = %d\n', totalTime);
% close the results.txt file
fclose(fp);
%
```

Code 10.9 *improvementHeuristic.m.*

```
%              jobsPermutation.m
%
% This function takes as actual parameters the number
% of machine 'm', the number of job 'n', the order of
% jobs 'tpOrd' calculated on the base of weights, the
% index of jobs and the sum of waiting times on the
last
% machine.
function [index,tpOrd,sumX] = jobsPermutation
(m,n,tpOrd,index,sumX)
% jobs 'two by two' permutation
for i = 1:(n-1)
    % permutation of job i by job (i+1)
    for k = 1:m
        temp = tpOrd(k,i);
        tpOrd(k,i) = tpOrd(k,i+1);
        tpOrd(k,i+1) = temp;
    end
    % index permutation
    tempIndex = index(1,i);
    index(1,i) = index(1,i+1);
    index(1,i+1) = tempIndex;
    % calculate the waiting time matrix for the new
order
    x = timeX (m,n,tpOrd);
    % 'sumXc' is the current sum of waiting times on
the
    % last machine
    sumXc = 0;
    for j = 1:n
```

```
31          sumXc = sumXc + x(m,j);
32      end
33      if (sumXc < sumX)
34          sumX = sumXc;
35      else
36          for k = 1:m
37              temp = tpOrd(k,i);
38              tpOrd(k,i) = tpOrd(k,i+1);
39              tpOrd(k,i+1) = temp;
40          end
41          % if the permutation does not produce
42          % improvement then remove the change
43          % (back to initial order of jobs)
44          tempIndex = index(1,i);
45          index(1,i) = index(1,i+1);
46          index(1,i+1) = tempIndex;
47      end
48  end
49  end
50  %
```

Code 10.10 *jobsPermutation.m*.

To run the program *improvementHeuristic.m*, the functions *weightSort.m* (see Code 10.7), *timeX.m* (see Code 10.8) and *ganttDrawing.m* (see Code 10.9) are also required.

If we reconsider the *m6n20* problem with 20 jobs on six machines, after execution the program in Code 10.10, one will get the order:

```
3 9 13 17 12 19 8 11 15 16 20 2 14 1 10 7 6 5 18 4
```

with the value of C_{max} = 1587. Thus, even a simple local search made by permutations can lead to improvement. In the case of *m6n20* problem, the reduction of C_{max} from 1627 to 1587 means 2.45% improvement. Looking at the *Gantt* diagram in Fig. 10.5, one can easily notice the multitude of gaps between jobs, especially on the last machine.

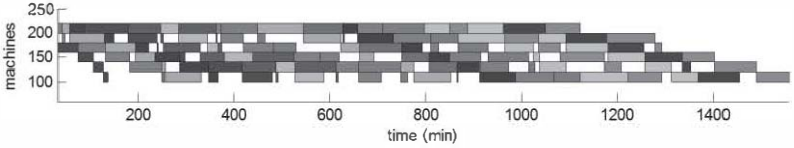

Fig. 10.5 *Gantt* diagram of the *m6n20* problem, after performing local search.

This fact leaves room for designing a better local search procedure. The literature abounds with such local search design ideas. Of the very good books in this respect, the work of Mikalewitz&Fogel (2004) is worth mentioning.

10.6 Conclusions

The problem of organizing manufacture, regardless of production nature, was and will remain a matter of greatest importance. This chapter dealt with solving the flowshop scheduling problem. Although the problem requires compliance with some initial conditions and thus narrows the field of practical applicability, it constitutes a starting point for much more general problems. Through the way ideas are presented, this chapter represents a design tutorial of constructive and improvement heuristics algorithms.

BIBLIOGRAPHY

Adamowicz, M. & Albano, A. (1976), Nesting two-dimensional shapes in rectangular modules, *Computer Aided Design* 8, 27-33.

Agarwal, A., Colak, S. & Eryarsoy, E. (2006), Improvement heuristic for the flow-shop scheduling problem: An adaptive-learning approach, *European Journal of Operational Research* 169, 801-815.

Ancău, M. & Ancău, D.M. (2011), *Metode numerice* [Numerical Methods], Editura Universităţii Tehnice din Cluj-Napoca UTPress.

Ancău, M. & Caizar, C. (2010), The Computation of Pareto-optimal Set in Multicriterial Optimization of Rapid Prototyping Processes, *Computers & Industrial Engineering* 58(4), 696-708.

Ancău, M. & Caizar, C. (2014), The Effectiveness of Multi-Criteria Optimization in Rapid Prototyping, in *Proceedings of the Romanian Academy, Series A* 15(4), 362-370.

Ancău, M. & Camilleri, L. (2007), A Hybrid Heuristic Solving the Traveling Salesman Problem, *Academic Journal of Manufacturing Engineering* 5(4), 17-25.

Ancău, M. & Camilleri, L. (2007), A New Hybrid Heuristic Algorithm for Solving Flowshop Scheduling Problem, *Academic Journal of Manufacturing Engineering* 5(4), 6-16.

Ancău, M. (1999), *Optimizarea proceselor tehnologice* [Optimization of Technological Processes], Editura Universităţii Tehnice din Cluj-Napoca UTPress.

Ancău, M. (2005), *Optimizare numerică. Algoritmi şi programe în C* [Numerical Optimization. Algorithms and programs in C], Editura Casa Cărţii de Ştiinţă, Cluj-Napoca.

Ancău, M. (2008), The optimization of printed circuit board manufacturing by improving the drilling process productivity, *Computers & Industrial Engineering* 55(2), 279-294.

Ancău, M. (2009), Numerical Results Accuracy vs. CPU Time in Solving Flowshop Scheduling Problem, *Academic Journal of Manufacturing Engineering* 7(3), 84-89.

Ancău, M. (2009), The processing time optimization of printed circuit board, *Circuit World* 35(3), 21-28.

Ancău, M. (2012), Main Aspects Concerning PCB Manufacturing Optimization, *Circuit World* 38(2), 75-82.

Ancău, M. (2012), On Solving Flowshop Scheduling Problems, in *Proceedings of the Romanian Academy, Series A* 13(1), 71-79.

Ancău, M. (2015), The importance of Minkowski sum in stamping operation design, *Academic Journal of Manufacturing Engineering* 13(3), 6-11.

Angéniol, B. et al. (1988), Self-organizing feature maps and the traveling salesman problem, *Neural Networks* 1, 289-293.

Aoyama, E. et al. (2004), Optimizing drilling conditions in printed circuit board by considering hole quality Optimization from viewpoint of drill-movement time, *Journal of Materials Processing Technology* 155-156, 1544-1550.

Applegate, D. & Cook, W. (1993), Solving large-scale matching problems, in D. Johnson, & C.C. McGeoch eds., *Network Flows and Matching* 12, 557-576.

Applegate, D.L. et al. (2006), *The Traveling Salesman Problem: A Computational Study*, Princeton University Press, New Jersey.

Bennell, J.A. & Song, X. (2008a), A comprehensive and robust procedure for obtaining the nofit polygon using Minkowski sums, *Computers & Operations Research* 35, 267-281.

Bennell, J.A. & Oliveira, J.F. (2008b), The geometry of nesting problems: A tutorial, *European Journal of Operational Research* 184, 397-415.

Bentley, J.L. (1990b), *K -d* trees for semidynamic point sets, in *Proc. 6th Ann. Symp. on Computational Geometry, ACM,* New York, 187-197.

Bentley, J.L. (1992), Fast algorithms for geometric traveling salesman problem, *ORSA Journal on Compututing* 4, 387-411.

Bentley, J.L. "Experiments on traveling salesman heuristics," in *Proc. 1st Ann. ACMSIAM Symp. on Discrete Algorithms*, SIAM, Philadelphia, PA (1990a): 91-99.

Bonnans, F.J., et al. (2006), *Numerical Optimization: Theoretical and Practical Aspects*, Springer-Verlag, Berlin.

Brandimarte, P. (2006), *Numerical Methods in Finance and Economics. A MATLAB-Based Introduction* (2nd ed.), John Wiley & Sons, New Jersey.

Burke, E.K. et al. Complete and robust no-fit polygon generation for the irregular stock cutting problem, *European Journal of Operational Research* 179 (2007): 27-49.

Butenko, S. & Pardalos, P.M. (2014), *Numerical Methods and Optimization: An Introduction*, CRC Press, Boca Raton.

Campbell, H.G., Dudek, R.A. & Smith, M.L. (1970), A heuristic algorithm for the n job, m machine sequencing problem, *Management Science* 16(10), B630-B637.

Cardenas-Montes, M. (2018), Creating hard-to-solve instances of travelling salesman problem, *Applied Soft Computing* 71, 268-276.

Chakraborty, U.K. & Laha, D. (2007), An improved heuristic for permutation flowshop scheduling, *International Journal of Information and Communication Technology* 1(1), 89-97.

Changa, P.C., Hsiehb, J.C. & Wanga, C.Y. (2007), Adaptive multi-objective genetic algorithms for scheduling of drilling operation in printed circuit board industry, *Applied Soft Computing* 7(3), 800-806.

Chergui , A., Hadj-Hamou , K. & Vignat, F. (2018), Production scheduling and nesting in additive manufacturing, *Computers & Industrial Engineering* 126, 292-301.

Choi, J.C. et al. (1998), A compact and practical CAD system for blanking or piercing of irregular-shaped metal products of stator and rotor parts, *International Journal of Machine Tools and Manufacture*, 38, 931-963.

Chow, W.W. (1979), Nesting of a Single Shape on a Strip, *International Journal of Production Research* 17, 305-322.

Christofides, N. (1976), Worst-case analysis of a new heuristic for the travelling salesman problem, *Research Report No. 388, GSIA*, Camegie-Mellon University, Pittsburgh, PA.

Clarke, G. & Wright, J.W. (1964), Scheduling of vehicles from a central depot to a number of delivery points, *Operations Research* 12, 568-581.

Cook, W.J. (2012) *In Pursuit of the Traveling Salesman: Mathematics at the limits of Computation*, Princeton University Press, New Jersey.

Cormen, T.H., et al. (2003), *Introduction to algorithms* (2nd ed.) Cambridge, Massachusetts: The MIT Press.

Croes, G.A. (1958), A method for solving traveling salesman problems, *Operations Research* 6, 791-812.

Crowder, H. & Padberg, M. (1980), Solving large-scale symmetric travelling salesman problems to optimality, *Management Science* 26, 495-509.

Dancea, I. (1976), *Metode de optimizare* [Optimization Methods], Editura Dacia, Cluj.

Dannenbring, D.G. (1977), An evaluation of flow shop sequencing heuristics, *Management Science* 23(11), 1174-1182.

de Berg, M. et al. (2000), *Computational Geometry: Algorithms and applications*, Springer-Verlag, Berlin.

Dodescu, Gh. et al. (1986), *Simularea sistemelor* [Systems Simulation], Editura Militară, Bucharest.

Dori, D. & Bassat, B. (1984), Efficient nesting of congruent convex figures, *Communications of the ACM* 27, 228-235.

Ehrgott, M. (2005), *Multicriteria Optimization*, Springer Berlin-Heidelberg.

Evtushenko, Yu.G. (1985), *Numerical Optimization Techniques*, Optimization Software Inc., New York.

Fletcher, R. (1987), *Practical methods of optimization* (2nd ed.) John Wiley & Sons, New York.

Fox, R.L. (1971), *Optimization Methods for Engineering Design*, Addison-Wesley, Massachusetts.

Framinan, J.M. & Perez-Gonzales, P. (2015), On heuristic solutions for the stochastic flowshop scheduling problem, *European Journal of Operational Research* 246(2), 423-420.

Framinan, J.M., Leisten, R. & Rajendran, C. (2003), Different initial sequences for the heuristic of Nawaz, Enscore and Ham to minimize makespan, idle time or flowtime in the static permutation flowshop sequencing problem, *International Journal of Production Research* 41(1), 121-148.

Frieze, A.M. (1979), Worst-case analysis of algorithms for traveling salesman problems, *Methods of Operations Research* 32, 97-112.

Garrido, A., et al. (2000), Heuristic methods for solving job-shop scheduling problems. *Proceedings of the Workshop on New Results in Planning, Scheduling and Design (PUK'00), ECAI00 Berlin*, 44-51.

Georgescu, I. (1981), Elemente de inteligenţă artificială [Elements of Artificial Intelligence], Editura Academiei R.S.R., Bucharest.

Gilat, A. (2008), *MATLAB: An Introduction With Applications*, John Wiley & Sons, New Jersey.

Glover F. & Rego C. (2018), New assignment-based neighborhoods for traveling salesman and routing problems, *Networks* 71(3), 170-187.

Glover, F. (1986), Future paths for integer programming and links to the artificial intelligence, *Computers and Operations Research* 13, 533-549.

Goertzel, B. (1993), *The structure of intelligence*, Berlin: Springer-Verlag.

Goldberg, D.E. (1989), *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley, Boston MA.

Gorges-Schleuter, M. (1991), *Genetic Algorithms and Population Structures – A Massively Parallel Algorithm*, PhD thesis, Fachbereich Informatik, Universitat Dortmund.

Gottfried, B.S., et al. (1973), *Introduction to Optimization Theory*, Prentice-Hall, Englewood Cliffs, N.J.

Grötschel, M. & Holland, O. (1991), Solution of large-scale symmetric travelling salesman problems, *Mathematical Programming* 51(1-3), 141-202.

Gupta, J.N.D. & Stafford Jr., E.F. (2006), Flowshop scheduling research after five decades, *European Journal of Operational Research* 169, 699-711.

Gupta, J.N.D. (1971), A functional heuristic algorithms for the flowshop scheduling problem, *Operational Research Quarterly* 22(1), 39-47.

Gutin, G. & Punnen, A.P. (2007), *The Traveling Salesman Problem and Its Variations*, Springer, New York.

Himmelblau, D.M. (1972), *Applied Nonlinear Programming*, McGraw-Hill Book Co., New York.

Ho, J.C. & Chang, Y.L. (1991), A new heuristic for the n-job, m-machine flow-shop problem, *European Journal of Operational Research* 52, 194-202.

Hsieh, J.F. (2005), Mathematical model for helical drill point, *International Journal of Machine Tools and Manufacture*, 45(7-8), 967-977.

Hundal, T.S. & Rajgopal, J. (1988), An extension of Palmer's heuristic for the flow shop scheduling problem, *International Journal of Production Research* 26(6), 1119-1124.

Ishibuchi, H., Misaki, S. & Tanaka, H. (1995), Modified simulated annealing algorithms for the flow shop sequencing problem, *European Journal of Operational Research* 81, 388-398.

Ismail, H.S. et al. (1996), Feature-based design of progressive press tools, *International Journal of Machine Tools and Manufacture* 36, 367-378.

Jain, P., Feynes, P. & Richter, R. (1992), Optimal blank nesting using simulated annealing, *Transaction of the ASME Journal of Mechanical Design* 114, 160-165.

Johnson, D.S. & McGeoch, L.A. (1997), The Traveling Salesman Problem: A Case Study in Local Optimization, in E.H.L. Aarts, & J.K. Lenstra, eds., *Local Search in Combinatorial Optimization*, London: John Wiley and Sons, 215-310.

Johnson, D.S., McGeoch, L.A. & Rothberg, E.E. (1996), Asymptotic experimental analysis for the Held-Karp traveling salesman bound, in *Proceedings 7th ACM SIAM Symp. On Discrete Algorithms, Society for Industrial and Applied Mathematics*, Philadelphia York.

Johnson, S.N. (1954), Optimal two - and three - stage production schedules with setup times included, *Naval Research Logistics Quarterly* 1, 61-68.

Joshi, S. & Sudit, M. (1994), Procedures for solving single-pass strip layout problems, *IIE Transactions* 26, 27-37.

Karoupi, F. & Loftus, M. (1991), Accommodation diverse shapes within hexagonal pavers, *International Journal of Production Research* 29, 1507-1519.

Kelley, J.E. (1960), The Cutting Plane Method for Solving Convex Programs, *Journal of the Society for Industrial and Applied Mathematics* 8(4), 703-712.

Kirkpatrick, S. et al. (1983), Optimization by simulated annealing, *Science* 20, 671-680.

Knox, J. (1994), Tabu search performance on the symmetric traveling salesman problem, *Computers and Operations Research* 21, 867-876.

Kochenderfer, M.J. & Wheeler, T.A. (2019), *Algorithms for Optimization*, MIT Press, Cambridge.

Korte, B. & Vygen, J. (2012), *Combinatorial Optimization: Theory and Algorithms*, Springer, Berlin.

Koulamas, C. (1988), A new constructive heuristic for the flowshop scheduling problem, *European Journal of Operational Research* 105, 66-71.

Kudla, L. (2001), Influence of feed motion on small holes drilling process, *Journal of Materials Processing Technology*, 109(3), 236-241.

Kuester, J.L. & Mize, J. (1973), *Optimization Techniques with Fortran*, McGraw-Hill Book Co, New York.

Lawler, E.L. et al. (1985), *The Traveling Salesman Problem*, John Wiley & Sons Ltd., Chichester.

Lawler, E.L. et al. (1995), *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization*, John Wiley & Sons Ltd., Chichester.

Lin, S. & Kernighan, B.W. (1973), An effective Heuristic Algorithm for the Traveling Salesman Problem, *Operations Research* 21, 498-516.

Lin, Z.C. & Hsu, C.Y. (1996), An investigation of an expert system for shearing cut progressive die design, *International Journal of Advanced Manufacturing Technology* 11, 1-11.

Liu, J., Linn, R. & Kowe, P.S.H. (1999), Study on heuristic methods for PCB drilling route optimization, *International Journal of Industrial Engineering: Theory Applications and Practice* 6(4), 289-296.

Marinescu, Gh. (1983), *Analiză matematică* [Mathematical Analysis,], Editura Academiei R.S.R., Bucharest.

Martin, R.R. & Stephenson, P.C. (1988), Putting objects into boxes, *Computer Aided Design* 20, 506-514.

Marusciac, I. (1973), *Metode de rezolvare a problemelor de programare neliniară* [Methods for solving nonlinear programming problems], Editura Dacia, Cluj.

MathWorks (2011a), *Matlab Getting Started Guide (R2011b)*, Retrieved October 2011 from https://www.mathworks.com/help/releases/R2011b/techdoc/

MathWorks (2011b), *Matlab Mathematics (R2011b)*, Retrieved October 2011 from https://www.mathworks.com/help/releases/R2011b/techdoc/

MathWorks (2011c), *Matlab Graphics (R2011b)*, Retrieved October 2011 from https://www.mathworks.com/help/releases/R2011b/techdoc/

MathWorks (2011d), *Matlab Programming Fundamentals (R2011b)*, Retrieved October 2011 from https://www.mathworks.com/help/releases/ R2011b/techdoc/

McKeon, J.J., Meegan, D. & Sprevak, D. (1990), *An Introduction to Unconstrained Optimization*, Adam Hilger ESM, Cambridge.

Messac, A. (2015), *Optimization in Practice with Matlab, for Engineering Students and Professionals*, Cambridge University Press, New York.

Metropolis, N. et al. (1953), Equation of state calculation for fast computing machines, *The Journal of Chemical Physics* 21(6), 1087-1092.

Michalewicz, Z. & Fogel, D.B. (2004), *How to Solve It: Modern Heuristics* (2nd ed.), Springer, Berlin.

Mihăileanu, N. (1974), *Istoria matematicii* [History of mathematics], Editura Enciclopedică Română, Bucharest.

Moccellin, J.a.V. (1995), A new heuristic method for the permutation flow shop scheduling problem, *Journal of the Operational Research Society* 46, 883-886.

Muhlenbein, H., Gorges-Schleuter, M. & Kramer, O. (1988), Evolution algorithms in combinatorial optimization, *Parallel Computing* 7, 65-85.

Murata, T., Ishibuchi, H. & Tanaka, H. (1996), Genetic algorithms for flowshop scheduling problem, *Computers and Industrial Engineering* 30(4), 1061-1071.

Nawaz, M., Enscore Jr., E.E. & Ham, I. (1983), A heuristic algorithm for the m-machine, n-job flow-shop sequencing problem, *Omega, The International Journal of Management Science*, 11(1), 91-95.

Nee, A.Y.C. (1984), Computer Aided Layout of Metal stamping blanks, *Proceedings of the Institution of Mechanical Engineers, Part B* 198(10), 187-194.

Nesterov, Y. (2004), *Introductory Lectures on Convex Optimization*, Springer, New York.

Nesterov, Y. (2018), *Lectures on Convex Optimization*, Springer Nature Switzerland.

Nocedal, J. & Wright, S. (2006), *Numerical Optimization*, Springer, New York.

Nowicki, E. & Smutnicki, C. (1996), A fast tabu search algorithm for the permutation flow-shop problem, *European Journal of Operational Research* 91, 160-175.

Nye, T.J. (2000), Stamping Strip Layout for Optimal Raw Material Utilization, *Journal of Manufacturing Systems* 19(4), 239-248.

Nye, T.J. (2001), Optimal nesting of irregular convex blanks in strips via an exact algorithm, *International Journal of Machine Tools and Manufacture* 41, 991–1002.

O'Rourke, J. (1998), *Computational Geometry in C* (2nd ed.), Cambridge University Press, Cambridge.

Ogbu, F. & Smith, D. (1990), The application of the simulated annealing algorithm to the solution of the $n/m/C_{max}$ flowshop problem, *Computers and Operations Research* 17(3), 243-253.

Osman, I. & Potts, C. (1989), Simulated annealing for the permutation flow-shop scheduling, *Omega, The International Journal of Management Science* 17(6), 551-557.

Osyczka, A. (1992), *Computer Aided Multicriterion Optimization System*, International Software Publisher, Krakow.

Padberg, M.W. & Rinaldi, G. (1987), Optimization of a 532 city Symmetric Traveling Salesman Problem by Branch and Cut, *Operations Research Letters* 6, 1-7.

Padberg, M.W. & Rinaldi, G. (1991), A branch-and-cut algorithm for the resolution of large-scale symmetric travelling salesman problems, *SIAM Review* 33, 60-100.

Palmer, D. (1965), Sequencing jobs through a multi-stage process in the minimum total time - a quick method of obtaining a near optimum, *Operational Research Quarterly* 16(1), 101-107.

Pandiri, V. & Singh, A. (2019), An artificial bee colony algorithm with variable degree of perturbation for the generalized covering traveling salesman problem, *Applied Soft Computing* 78, 481-495.

Peterson, C. & Söderberg, B. (1989), A new method for mapping optimization problems onto neural network, *International Journal of Neural Systems* 1, 3-22.

Peterson, C. (1990), Parallel distributed approaches to combinatorial optimization: Benchmark studies on traveling salesman problem, *Neural Computation* 2, 261-269.

Ponnambalam, S.G., Aravindan, P. & Chandrasekaran, S. (2001), Constructive and improvement flow shop scheduling heuristics: An extensive evaluation, *Production Planning and Control* 12(4), 335-344.

Prasad, Y.K.D.V. & Somasundaram, S. (1992), CAADS: An automated die design system for sheet-metal blanking, *Computing and Control Engineering Journal* 3, 185-191.

Pratap, R. (2006), *Getting Started with MATLAB 7. A Quick Introduction for Scientists and Engineers*, Oxford University Press, New York.

Press, W.H., et al. (1997), *Numerical Recipes in C. The Art of Scientific Computing* (2nd ed.), New York: Cambridge University Press.

Qu, W. & Sanders, J.L. (1987), A nesting algorithm for irregular parts and factors affecting trim losses, *International Journal of Production Research* 25(3), 381-397.

Rao, S.S. (2009), *Engineering Optimization: Theory and Practice* (4th ed.), John Wiley & Sons, New Jersey.

Reeves, C. & Yamada, T. (1998), Genetic algorithms, path relinking and the flowshop sequencing problem, *Evolutionary Computation* 6(1), 45-60.

Reinelt, G. (1991), TSPLIB—A traveling salesman problem library, *ORSA Journal on Computing* 3(4), 376-384.

Rosenkrantz, D.J., Steams, R.E. & Lewis, P.M. (1977), An analysis of several heuristics for the traveling salesman problem, *SIAM Journal of Computing* 6, 563-581.

Ruiz, R. & Morato, C. (2005), A comprehensive review and evaluation of permutation flowshop heuristics, *European Journal of Operational Research* 165, 479-494.

Ruiz, R., Pan, Q.K. & Naderi, B. (2019), Iterated Greedy methods for the distributed permutation flowshop scheduling problem, *Omega* 83, 213-222.

Simmen, M.W. (1991), Parameter sensitivity on the elastic net approach to the traveling salesman problem, *Neural Computation* 3, 363-374.

Stancu-Minasian, I.M. (1980), *Programarea stocastică cu mai multe funcţii obiectiv* [Stochastic programming with multiple objective functions], Editura Academiei R.S.R., Bucharest.

Struckmeier , F. & Leon, F.P. (2019), *Nesting* in the sheet metal industry: dealing with constraints of flatbed laser-cutting machines, *Procedia Manufacturing* 29, 575-582.

Su, Y.L., et al. (2004), Tribological characteristics and cutting performance of $Cr_{x\%}C$-coated carbide tools, *Journal of Materials Processing Technology*, 153-154(1-3), 699-706.

Taillard, E. (1990), Some efficient heuristic methods for the flow-shop sequencing problem, *European Journal of Operational Research* 47, 67-74.

Teodorescu, N., et al. (1976), *Matematici aplicate. Culegere de probleme* [Applied mathematics. Problem collection], Societatea de ştiinţe matematice din R.S.R., Bucharest.

Theodocrates, V.E. & Grimsley, J.L. (1995), The optimal packing of arbitrarily-shaped polygons using simulated annealing and polynomial-time cooling schedules, *Computer Methods in Applied Mechanics and Engineering* 125, 53-70.

Tien, F.C., Yeh, C.H. & Hsieh, K.H. (2004), Automated visual inspection for microdrills in printed circuit board production, *International Journal of Production Research* 42(12), 2477-2495.

Valenzuela, C.L. & Jones, A.J. (1994), Evolutionary divide and conquer (I): A novel genetic approach to the TSP, *Evolutionary Computation* 1, 313-333.

Vanderplaats, G.N. (1989), *Numerical Optimization Techniques for Engineering Design, with Applications*, McGraw-Hill Book Co., New-York.

Venkataraman, P. (2009), *Applied Optimization with Matlab Programming* (2nd ed.), John Wiley & Sons, New Jersey.

Verghese, N. (2004), PCB drillability: A material science approach to resin development, *Circuit World*, 30(4), 44-51.

Widmer, M. & Hertz, A. (1989), A new heuristic method for the flow shop sequencing problem, *European Journal of Operational Research* 41, p.186-193.

Wismer, D.A., et al. (1978), *Introduction to Nonlinear Programming – A Problem solving Approach*, North Holland, New York.

Xu, X. & Tsai, T. (1991), Effective neural algorithms for traveling salesman problem, *Neural Networks* 4, 193-205.

Yang, X.-S. (2018), *Optimization Techniques and Applications with Examples*, John Wiley & Sons, New Jersey.

Ying, K.C. & Lin, S.W. (2018), Minimizing makespan for no-wait flowshop scheduling problems with setup times, Computers & Industrial Engineering 121,73-81.

Yu, C. & Manoochehri, S. (2002), Nesting Arbitrary Shapes Using Geometric Mating, *Journal of Computing and Information Science in Engineering (ASME)* 2, 171-177.

LIST OF SOURCE CODES

Chapter 1

Chapter 2

Chapter 3

Chapter 4

Chapter 5

Chapter 6

Chapter 7

Chapter 8

Code 8.1 *convH.m*
Code 8.2 *convV.m*
Code 8.3 *NN.m*
Code 8.4 *removeIntersections.m*
Code 8.5 *twoOpt1.m*
Code 8.6 *direction.m*
Code 8.7 *segmentsIntersection.m*
Code 8.8 *onSegment.m*
Code 8.9 *translation.m*
Code 8.10 *findLength.m*
Code 8.11 *tsp.m*
Code 8.12 *twoOpt2.m*

Chapter 9

Code 9.1 *minkowskiConvex.m*
Code 9.2 *arrange.m*
Code 9.3 *drawPolygon.m*
Code 9.4 *theConvexHull.m*
Code 9.5 *minkowskiConcave.m*
Code 9.6 *minkowskiSum.m*
Code 9.7 *shapeLmatch1.m*
Code 9.8 *eraseMultiplePoints.m*
Code 9.9 shapeLmatch2.m
Code 9.10 *nesting2D.m*
Code 9.11 *angleCalculation.m*
Code 9.12 *checkSegmentDirection.m*
Code 9.13 *minkowskiSumConvex.m*
Code 9.14 *polygonSegmentSlope.m*
Code 9.15 *translate_m_k.m*

Chapter 10

Code 10.1 *findMatrixX.m*
Code 10.2 *timeX.m*
Code 10.3 *johnson.m*
Code 10.4 *timeXplus.m*
Code 10.5 *ganttDrawTwoMachines.m*
Code 10.6 *constructiveHeuristic.m*
Code 10.7 *weightSort.m*
Code 10.8 *ganttDrawing.m*
Code 10.9 *improvementHeuristic.m*
Code 10.10 *jobsPermutation.m*