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

CONTENTS

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 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 columns 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).

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

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

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:

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

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

where

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

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 appliccable 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.

```
%          whileStatement.m
%
% This is an example of a cycle with an
% unknown number of steps
%
% open a text file to save results
fp = fopen('results.txt','w');
x = 0;
while x<=5
    fprintf(fp,'x = %2d\n',x);
    % increment x
    x = x + 1;
end
% close the text file
fclose(fp);
%
```

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 istructions 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).

```
% 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
    %
```

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);
```

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

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.

```
%        continueStatement1.m
%
% 'continue' is inside a 'for' loop
% open the file ,results.txt’
fp = fopen('results.txt','w');
for i = 1:3
    for j = 1:3
        for k = 1:3
            if i==2
                continue
            end
            fprintf(fp,'i=%2d    j=%2d   k=%2d\n',...
                i,j,k);
        end
    end
end
% close the file ’results.txt’
fclose(fp);
%
```

Code 1.7 *continueStatement1.m*.

```
%        continueStatement2.m
%
% 'continue' is inside a 'while' loop
% open the file ,results.txt’
fp = fopen('results.txt','w');
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 *continueStatement2.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, meanning 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*.

```matlab
%        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*.

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

Code 1.10 *equationDiscriminant.m*.

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