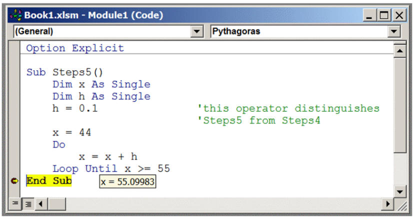
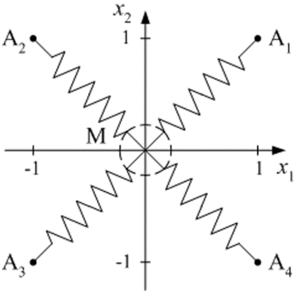
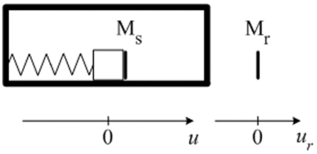
G. Z. Garber

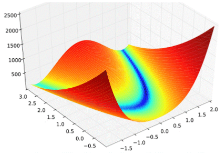
Foundations of Excel VBA Programming and Numerical Methods

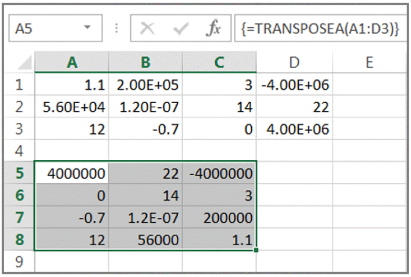


```
Option Explicit

Sub Steps5()
    Dim x As Single
    Dim h As Single
    h = 0.1                     'this operator distinguishes
                                'Steps5 from Steps4
    x = 44
    Do
        x = x + h
    Loop Until x >= 55
End Sub   x = 55.09983
```



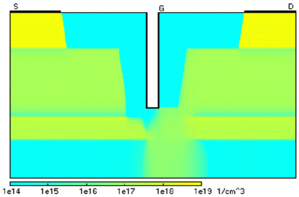






A5 {=TRANSPOSEA(A1:D3)}

	A	B	C	D	E
1	1.1	2.00E+05	3	-4.00E+06	
2	5.60E+04	1.20E-07	14	22	
3	12	-0.7	0	4.00E+06	
4					
5	4000000	22	-4000000		
6	0	14	3		
7	-0.7	1.2E-07	200000		
8	12	56000	1.1		
9					



G. Z. Garber

Foundations of Excel VBA Programming and Numerical Methods. — Moscow: PRINTKOM, 2013. — 528 p.

ISBN 978-5-91146-894-1

Intended for university students studying computer science, applied mathematics and information technology, as well as for post-graduate students, scientific workers and other readers wishing to refine their skill in solving problems by using tabular processor Microsoft Office Excel 2007 – 2013.

Elements of an environment for developing programs (macros) and base constructs of programming languages Visual Basic and VBA are considered. Modern and classical numerical methods and their program realization are also considered. As examples, Excel macros are developed for solving mathematical, physical, engineering and economic problems. A compact disk containing program modules and other text information is enclosed. No preliminary programming experience is required for grasping the material.

This book is based on the author’s two previous books approved by the Scientific and methodical council in computer science at the Ministry of Education and Science of the Russian Federation as a manual on discipline “Computer science” for university students.

ISBN 978-5-91146-894-1

Many things are incomprehensible to us not because our comprehension is weak, but because those things are not within the frames of our comprehension.

Kozma Prutkov, 1854

About the author

Gennadiy Z. Garber graduated from the faculty of applied mathematics at the Moscow Institute of Electronic Machinery in 1972. Professor in Computer Science, Doctor of Science in Microelectronics, principal scientific worker of Pulsar R&D Manufacturing Company, Moscow, participant of the International Conference on “Computer as a Tool”, IEEE EUROCON 2005 and 2007.

Area of interests: mathematical modeling of semiconductor devices and integrated circuits for radio-, micro- and nanoelectronics; development of teaching techniques in applied mathematics and programming for Excel.

About the prototype books published in Russian

Sample programs and descriptions of their usage allow to learn programming from scratch.

Using concrete examples with a minimum portion of theoretical introduction, the author has managed to show the first (and, I must say, main) steps of the object-oriented programming technology in VBA.

The author’s approach is very successful, in which a theoretical material on each numerical method is accompanied by a description of the program realization, by a scenario of the computing experiments for applied problems and by an analysis of the calculation results.

The student can obtain not only mathematical knowledge from the manual, but also acquire practical skills, which are very important in a training course on numerical methods.

From the reviews of the expert of the Scientific and methodical council in computer science at the Ministry of Education and Science of the Russian Federation on author’s books [1, 2]

Contents

Introduction 8

Chapter 1. Programming in Visual Basic 12

1.1. Elements of Visual Basic Environment. 13

1.2. Main commands of the program debugger. 19

1.3. Variables. Data types 23

1.4. Two main functions for conversion of data types 27

1.5. Constants 29

1.6. Obtaining information 32

1.7. Assignment operator 36

1.8. Arithmetic expression. 38

1.9. Mathematical functions. Functions of date and time 45

1.10. Logical expression 48

1.11. GoTo operator. 52

1.12. Decision-making constructs. 53

1.13. Cycles. 58

1.14. Manifestation of the error of real numbers’ computer representation 65

1.15. Arrays. 68

1.16. User-defined procedures 77

1.17. Built-in procedures. Usage of standard windows. 85

1.18. Records 90

1.19. Work with strings. 94

1.20. Work with text files. 101

1.21. Matrix terminology. Formulation of demonstration tasks 109

1.22. Program for transposing a matrix relative to its auxiliary diagonal . 111

1.23. User-defined forms 116

1.24. Digression. Developing programs with the form in Microsoft Visual Studio. 129

Chapter 2. Programming in VBA 133

2.1. Loading the form from the Excel window. Running the program executable file. 134

2.2. Layout of the control elements on the Excel worksheet 136

2.3. User-defined functions of Excel 139

2.4. Two methods for developing Excel macros 144

2.5. Excel Macro Recorder 145

2.6. VBA code generated by Excel Macro Recorder and its editing . . . 148

2.7. Objects and events 151

2.8. Object Application 154

2.9. Objects Workbook, Workbooks and ActiveWorkbook. 161

2.10. Objects Worksheet, Worksheets and ActiveSheet 167

2.11. Objects Range, Selection and ActiveCell. 171

2.12. Study of objects. 177

2.13. Using the Excel table as the user interface of programs 180

2.14. Two more Excel macros. Personal Macro Workbook 182

2.15. One more user-defined function of Excel. 188

2.16. Digression. Change of Excel options 193

Chapter 3. Finite Difference Method for Solving Differential Equations. . . 195

3.1. Finite difference analogs of derivatives for a uniform grid. 196

3.2. Finite difference scheme for the linear differential equation. The decomposition method 199

3.3. Sufficient stability conditions for the decomposition method 204

3.4. Simplification of the second-order linear differential equation. . . . 208

3.5. Program realization of the decomposition method 210

3.6. Examples of using the decomposition method 212

3.7. Examples of the computing error. Instability and loss of accuracy. . 217

3.8. Solving the system of linear algebraic equations by using Excel functions 223

3.9. Solving the system of linear algebraic equations by the Gaussian elimination method 225

3.10. Two subroutines for solving the system of linear algebraic equations 228

3.11. Reduction of the computing error. 235

3.12. Solving the nonlinear differential equation by the quasilinearization method. 238

3.13. Solving the Shockley-Poisson equation. 241

3.14. Finite difference analogs of derivatives for a nonuniform grid. . . . 249

3.15. The decomposition method for a nonuniform grid 252

3.16. Solving the Shockley-Poisson equation on a nonuniform grid. . . . 255

3.17. Use of solution symmetry. 261

3.18. The cyclic decomposition method 267
3.19. Program realization of the cyclic decomposition method 271
3.20. Solving the oscillation equation. 273
Chapter 4. Cubic Spline 281
4.1. Definition of cubic spline. Spline moments 282
4.2. Spline interpolation 288
4.3. Use of cubic spline for processing transistor electrical characteristics 292
4.4. Spline integration 298
4.5. Iterative methods for solving the nonlinear algebraic equation. . . . 303
4.6. Noniterative method for solving the nonlinear algebraic equation. . 314
4.7. Calculating the charge storage capacity 321
4.8. Subroutine for automatic creation of graphs 327
4.9. Cubic spline usage for solving the second-order linear differential equation 329
4.10. Program realization of the cubic spline method for solving the linear differential equation 334
4.11. Solving the linear differential equation by the cubic spline method . 337
4.12. Modeling of heating of a geophysical cable. Locally one-dimensional scheme 340
Chapter 5. Quadratic and Linear Splines 352
5.1. Definition of quadratic spline. Spline slopes 353
5.2. Method for solving the initial value problem for the system of differential equations. 356
5.3. Program for solving the initial value problem 359
5.4. Solving the system of nonlinear algebraic equations by the Newton method. 362
5.5. Newton and Newton-like methods for solving the single nonlinear algebraic equation 365
5.6. Modeling of the piano mechanism linking a key with hammer. . . 372
5.7. Definition of linear spline. 383
5.8. The least-squares method. 386
5.9. Program to determine the dependence of the wheat productivity on the land quality. 390
5.10. The forward and backward Fourier transforms of a periodic function 395
5.11. Subroutines for the forward and backward discrete Fourier transforms. 400

5.12. Solving the sound insulation problem 406
Chapter 6. Numerical Methods for Nonlinear Programming 415
6.1. Minimizing linear and nonlinear functions of several variables by the Solver add-in. 417
6.2. Method for minimizing a nonlinear function of one variable. 428
6.3. The coordinate-descent method. 432
6.4. Examples of using the minimization methods 437
6.5. The Powell minimization method. 446
6.6. Determining the equilibrium state of a four-spring system. 456
6.7. Minimization with nonlinear constraints 462
6.8. Minimization of the multimodal function. 475
6.9. Minimization of the tabular function 483
6.10. Solving the nonlinear differential equation by the shooting method . 490
6.11. Modeling of the hammer motion in the piano mechanism 493
6.12. Nonlinear programming and the least-squares method. 501
Instead of Conclusions. 510
Appendix 1. Data Types of Visual Basic and VBA 513
Appendix 2. Greek and Russian Alphabets Denoted by Latin Letters. . . . 515
Appendix 3. The Main Mathematical Functions 517
Appendix 4. Material for Tasks 518
Appendix 5. Analytical Method for Solving the Cubic Algebraic Equation . 520
Appendix 6. Realization of the Tangent Method by Using the Excel Circular Reference. 521
References List 523
Subject Index 525

Introduction

Because of many advantages (above all, availability), tabular processor Excel, which is a part of Microsoft Office, is used in various areas of human activity: in economics and finances, electrical engineering and electronics, medicine, building construction, etc. This book is about Excel usage in applied mathematics.

While writing this book, the author pursued the following goals:

- to teach the reader to program in the modern programming language, Visual Basic (VB), and its extension, Visual Basic for Applications (VBA);
- on the basis of these programming languages, to give the reader full enough representation about numerical methods aimed at obtaining a solution of a task in the form of numbers (instead of formulas that are a result of using analytical methods);
- to show that Excel with programs (macros), written by the reader in VBA, is convenient for solving applied tasks by numerical methods.

The book is intended for the reader familiar with Excel, Windows Explorer, Windows Clipboard and text editor Notepad for Windows. Besides, the reader should be conversant with higher mathematics and general physics. No preliminary programming experience is necessary.

Learning this book is possible only by using a computer equipped with tabular processor Excel.

According to the author's opinion, there is no difference between terms "program" and "macro" when these terms concern the programming for Excel. Therefore, words "program" and "macro" are synonyms in this book.

The book contains six chapters, six appendices, a list of references and a subject index.

In the first two chapters, we consider elements of Visual Basic Environment and main facilities of programming languages VB and VBA. The standard window of operating system Windows, text file, user-defined form and Excel table are considered as the program user interface — the facility of dialogue between the user and program. We consider the creation of Excel user-defined functions. Besides, we demonstrate how to work with the program debugger, reference systems, Excel Macro Recorder and Personal Macro Workbook.

In the third chapter, we consider the finite difference method for solving the second-order linear differential equation with two kinds of conditions on the solution, namely, the boundary and periodicity conditions. This is followed by a review of two versions of the decomposition method for solving systems of linear algebraic equations of special form called finite difference schemes. The simplest scheme is also solved by the Gaussian elimination method. The question of stability of the decomposition and Gaussian methods is investigated in respect of not increasing the computing error during solving the scheme. Using the Shockley-Poisson equation as an example, we consider the quasilinearization method for solving the nonlinear differential equation with boundary conditions. To demonstrate the possibilities of the finite difference method, we develop subroutines and programs for solving mathematical and applied problems. We use the Excel scatter diagrams for visualization of calculation results.

Chapter 4 is devoted to the use of the third-degree (cubic) spline:

- for interpolation, differentiation and integration of tabular (grid) functions;
- for solving the nonlinear algebraic and linear differential equations.

Besides, we consider:

- two classical methods for solving the nonlinear algebraic equations, namely, the bisection and secant methods;
- the locally one-dimensional scheme for solving the heat equation with two spatial coordinates.

We solve a series of applied problems to demonstrate the possibilities of the cubic spline construction. In addition to the macros and user-defined procedures (subroutines and function) realizing the numerical methods, a subroutine for automatic creation of graphs is developed.

In Chapter 5, we review the use of the second-degree (quadratic) spline for solving the initial value problem (of Cauchy) for the system of differential equations. The first-degree (linear) spline is used in the least-squares method intended for determining parameters of a function. Besides, we review the following methods:

- the Newton method for solving the system of nonlinear algebraic equations;
- the tangent, secant and Steffensen methods (called Newton-like methods) for solving a single nonlinear algebraic equation;
- methods for the forward and backward discrete Fourier transforms of a periodic function.

Based on this theoretical material, we develop procedures and programs for solving applied problems.

Chapter 6 is mainly devoted to nonlinear programming, more precisely, to the question of finding the minimum of a nonlinear function of one or several variables without calculating the function derivative or partial derivatives. This chapter begins with the use of the Solver add-in for Excel to minimize concrete linear and nonlinear functions of several variables. Further, we develop subroutines for finding the local minimum of a nonlinear function of general form, which are based on the coordinate-descent and Powell methods.

We review the following applications of the developed minimization subroutines:

- for optimizing the dimensions of a one-liter tin can;
- for determining the equilibrium state of a four-spring mechanical system;
- for minimizing a nonlinear function with nonlinear constraints and a tabular function of two variables;
- for determining the local minima of a multimodal function of two variables (with several local minima);
- in the shooting method intended for solving the nonlinear differential equation with boundary conditions;
- in the least-squares method.

Appendix 1 presents the data types of Visual Basic and VBA.

Appendix 2 contains the Greek alphabet with English names of the letters and the Russian alphabet denoted by Latin letters. The inclusion of this appendix is justified by the possible lack of Greek and Russian letters on the computer keyboard. English names of Greek letters are used in texts of program modules and source data for programs. Russian letters in Latin are mainly used in the references list.

Appendix 3 contains the main mathematical functions of Visual Basic. In addition, this appendix contains operators allowing the use of mathematical functions not included in the programming language.

Appendix 4 contains data for tasks intended to consolidate the book material and check up its understanding.

Appendix 5 presents an analytical method for solving the cubic algebraic equation. We use this method in Chapter 4.

Appendix 6 demonstrates the use of circular reference in Excel for solving the nonlinear algebraic equation by the tangent method.

The subject index contains the main terms and designations with numbers of pages, on which their sense is uncovered. It will allow using the book as a reference manual.

The present book is based on author’s books [1, 2] approved by the Scientific and methodical council in computer science at the Ministry of Education and Science of the Russian Federation as a manual on discipline “Computer science” for university students.

In the book, we often speak about mathematical (computer, numerical) modeling. The essence of mathematical modeling lies in the replacement of an object, in particular of a process, by an appropriate mathematical model and in its further study by using a computer. Operation with the model, instead of the object, allows to obtain operatively detailed information, showing internal connections of the object and its qualitative and quantitative characteristics. The mathematical modeling is so popular that, when speaking about it, adjective “mathematical” may be omitted, as in this book.

In writing the book, we used a personal computer equipped with 32-bit version of the Windows 7 operating system and Microsoft Office Professional Plus 2013 Preview. For obtaining information, shown in Fig. 1.8, 2.23 and 3.6, the reference system of Excel 2010 was used. The system disk name is *C* and the computer user name is *usr* in the book.

We will need the *Developer* tab in Excel Ribbon (a part of the Excel window), among tabs *Home*, *Insert*, *Page Layout*, etc. If such a tab does not exist, we fulfill the following:

1) click on the *File* button in the top left corner of the Excel window (in Excel 2007, click on the *Office* button);
2) click on the *Options* button;
3) in the *Excel Options* window opened, click on button *Customize Ribbon*;
4) in area *Customize the Ribbon*:
   - set *Main Tabs* by using the drop-down list;
   - then turn on option *Developer*;
5) click on the *OK* button.

This operational sequence can be written as the following formula: *File* > *Options* > *Customize Ribbon* > *Main Tabs* > turn on *Developer* > *OK*. We will frequently use such formulas.

When opening an Excel workbook containing a macro, the *Security Warning* panel can appear. To allow the macro to work, we must click on the *Enable Content* button of this panel.

When executing a macro, cycling is possible. To interrupt it, we must press the *Esc* key on the computer keyboard.

The enclosed compact disk (CD) contains text files with program modules and with source data for programs. The texts on the CD correspond to the numbered listings in the book. A method of work with these files is described on pp. 26 and 245.

The program texts on the CD may be used as templates when developing programs for solving other tasks with the same mathematical formulation as the tasks considered in the book.

For contact with the author, the following internet resources can be used: gzgarber@gmail.com, http://gzgarber.narod.ru/.

Chapter 1.
Programming in Visual Basic

We review elements of Visual Basic Environment, a part of Microsoft Office, and constructs of the Visual Basic programming language. The standard window of operating system Windows, text file and form are used as the user interface of programs.

In addition, we demonstrate how to work with the program debugger and reference systems.

1.1. Elements of Visual Basic Environment

For writing and debugging programs, we will use Visual Basic Environment, which is a part of Microsoft Office.

Program debugging involves detection and correction of errors that, as a rule, are present in a program text just written.

To go to Visual Basic Environment, we must fulfill the following two operations:

1) in the Excel window (with the active workbook by name Book1), activate the *Developer* tab by clicking on it;

2) click on the *Visual Basic* button in area *Code*.

As a result, the Visual Basic Environment window is displayed (Fig. 1.1). In this window, we can perform various actions: entering and editing the program text, as well as debugging and executing the program. Further, we will use a shorter name for this window and call it “the VB window”.

The program is also called an application or project. It will be in the Excel workbook (by name Book1).

Let us consider the elements of the VB window.

1. Menu bar. There are standard menus, like in many windows of the operating system: *File*, *Edit*, *View*, *Tools* and *Help*. The *Insert* menu is used for organizing a place for program storage (in the workbook). Menus *Debug* and *Run* are respectively used for debugging and running the program.

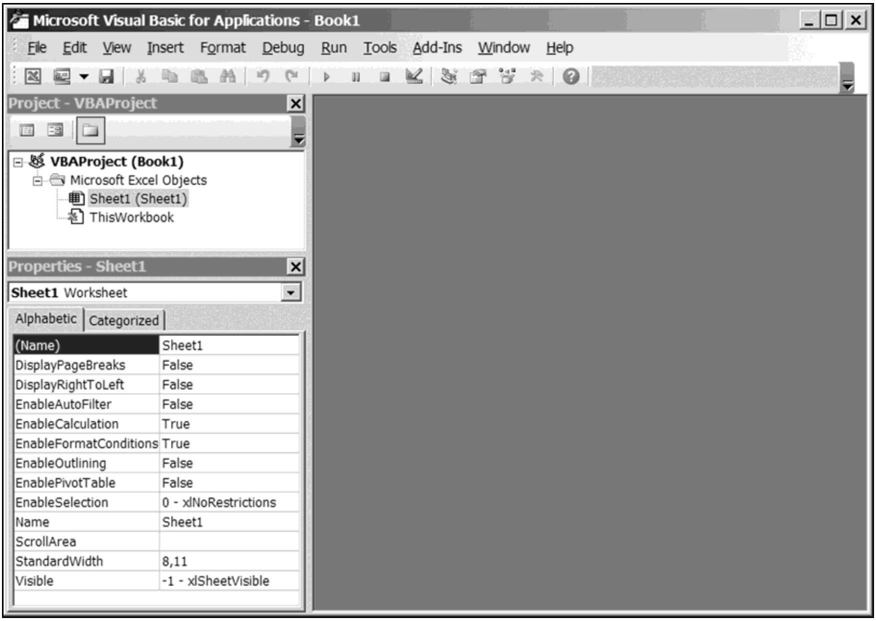
2. Context menu. It serves for convenience of work in the area (of the VB window), in which the mouse pointer is located.

For using the context menu:

1) place the mouse pointer in the necessary area of the screen and make the right click;

2) click (by the left mouse button) on the required command of the displayed menu.

3. Toolbars: *Standard*, *Edit*, *Debug* and others. Only the standard toolbar is displayed by default. To add or remove any toolbar, we have to fulfill *View* > *Toolbars* and to click on the required command of the displayed menu. The check (tick) mark against the command testifies to the presence of the corresponding toolbar on the display screen.



(Name)	Sheet1
DisplayPageBreaks	False
DisplayRightToLeft	False
EnableAutoFilter	False
EnableCalculation	True
EnableFormatConditions	True
EnableOutlining	False
EnablePivotTable	False
EnableSelection	0 - xlNoRestrictions
Name	Sheet1
ScrollArea	
StandardWidth	8,11
Visible	-1 - xlSheetVisible

Fig. 1.1. The Visual Basic Environment window including the standard toolbar, the project explorer window and the properties window

Let us consider the toolbars.

1. Toolbar *Standard* is displayed by default. It allows us to perform a wide spectrum of actions.

This toolbar is usually located under the menu bar, however, we can move it to other areas of the VB window by using the mouse.

2. Toolbar *Edit* is intended for work with the program text. It realizes possibilities of an elementary text editor:

- copying and moving a text fragment to Windows Clipboard;
- inserting the text fragment from Windows Clipboard;
- search and replacement of words and phrases, etc.

3. Toolbar *Debug* is intended for debugging the program. Many provisions are made for debugging:

- observation of the current values of the program variables;
- step-by-step program execution, in which one operator (statement, instruction) or its part is performed on each step, etc.

As a rule, we will review programs without a user-defined form as the program user interface. Development of such program begins with inserting a module into the active Excel workbook. For inserting a module, let us fulfill the following sequence of operations.

1. In the Excel window, *Developer* > *Visual Basic* in area *Code*. As a result, the VB window appears, including the project explorer window and the properties window (Fig. 1.1).

If the project explorer window is not displayed, we have to click on *Project Explorer* in the *View* menu. We will need the properties window only in Section 1.23.

2. Select line *VBAProject (Book1)* by clicking on it in the project explorer window.

3. *Insert* > *Module*.

As a result, a line corresponding to the inserted module, Module1, appears in the project explorer window. Besides, an empty window opens; it is the code window corresponding to Module1 (Fig. 1.2). In this window, we will create the program text by using the computer keyboard.

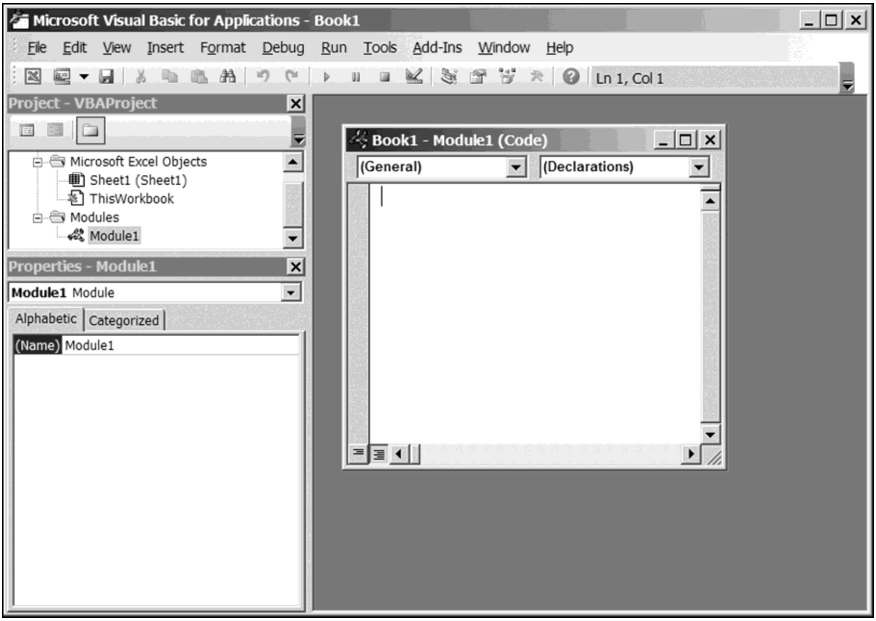


Fig. 1.2. The VB window, including the code window, after inserting Module1 into the Excel workbook

For opening the code window corresponding to the module inserted earlier, we have to click twice on the name of this module in the project explorer window.

To delete a module:

1) make the right click on the module name (for example *Module1*) in the project explorer window;

2) in the context menu opened, click on the *Remove* command;

3) click on the *No* button in the open window with a question about exporting the module before removing it.

Before the computer will execute a program, we (as the program developer) must form its text in the code window. The first and last lines (operators) of the program are standard:

```
Sub name()

End Sub
```

On p. 79, we will consider the origin of word `Sub`. Word *`name`* means the program name appointed by us.

The name must satisfy the following conditions:

- the first character should be a letter;
- the name must include only letters, figures and the underscore character;
- the name must include less than 256 characters.

As we see, the name cannot include the space character. To use *`name`* consisting of several words, we have:

- to begin each word with a capital (uppercase) letter;
- to use the underscore character instead of the space character.

Examples of the program name follow:

```
MyProgram13
my_program
MyProgram_13
```

Between the first and last lines of the program, we have to place other lines (operators) of this program. For that, it is possible to use Windows Clipboard and habitual commands of editing (as in Notepad). After typing a new line, we have to press the *Enter* key on the keyboard.

Let us start with a simple program based on the Pythagoras theorem,

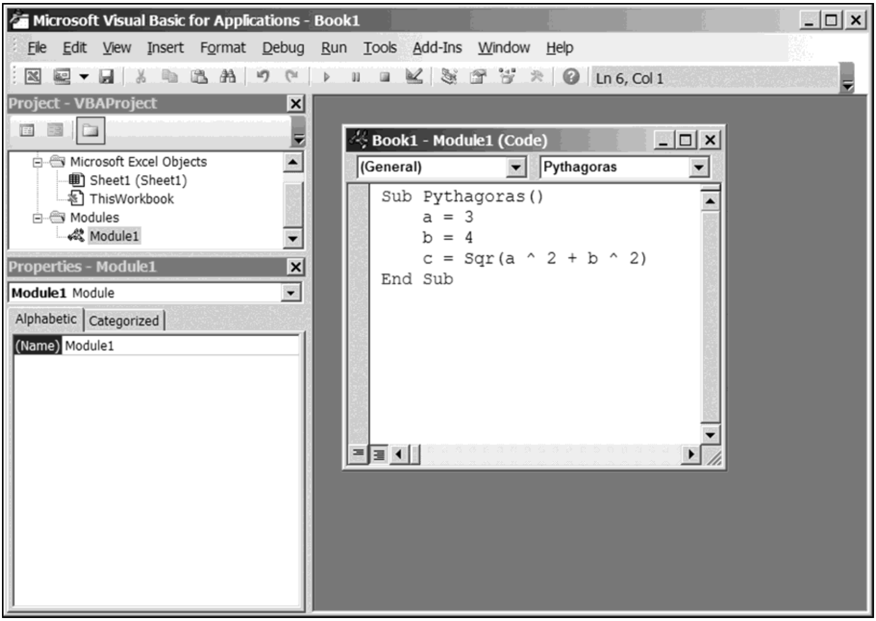
$$c = \sqrt{a^2 + b^2}\ , \qquad (1.1)$$

for calculating length c of the hypotenuse of a right-angled triangle with legs $a = 3$ and $b = 4$.

In the code window, we type the following program text (Fig. 1.3):

```
Sub Pythagoras()
    a = 3
    b = 4
    c = Sqr(a ^ 2 + b ^ 2)
End Sub
```

In this text, `Pythagoras` is the program name, `a`, `b` and `c` are names of variables, `Sub` is a keyword, `End Sub` is a keyword combination. The program and variable names are appointed by us.



Module1 Module

(Name) Module1

(General) | Pythagoras

```
Sub Pythagoras()
    a = 3
    b = 4
    c = Sqr(a ^ 2 + b ^ 2)
End Sub
```

Fig. 1.3. The VB window with the `Pythagoras` program in the code window

In a programming language, words used only in the language constructs are called keywords. We cannot use keywords as names of programs and variables in programs. By default, Visual Basic Environment is tuned in such manner that all keywords are highlighted in blue color (at formation of the program text in the code window), comments are highlighted in green, syntactic errors — in red.

It is visible that the power operation (^) is written as in Excel, `Sqr` is the square root function (in Excel, the square root function is SQRT).

For convenience, if we need to place an operator on several lines, for carrying over, we have to type the space character with the subsequent underscore character. At the finish of typing these characters, we have to press the *Enter* key on the keyboard.

If we need to place several operators on one line, we have to type a colon between these operators.

An apostrophe means that information following it (up to the line end) is a comment, i.e., a character set, which does not influence the program execution.

Thus, our program can be written as follows:

```
Sub Pythagoras()
    a = 3: b = 4
    c = _
        Sqr(a ^ 2 + b ^ 2)    'according to Pythagoras
End Sub
```

If a comment occupies several lines, each line must be preceded by an apostrophe. For example, program

```
Sub Pythagoras()
    a = 3: b = 4
    c = Sqr(a ^ 2 + b ^ 2)    'according to Pythagoras:
                              'pythagorean pants are
                              'equal in all directions
End Sub
```

is equivalent to the previous program.

To save the program, we fulfill the following:

1) in the Excel window, *File* > *Save As* > *Browse*;

2) in the *Save As* window opened, choose a folder intended for saving the Excel workbook, for example, *My Documents*;

3) by means of drop-down list *Save as type*, set the following file type: *Excel Macro-Enabled Workbook*;

4) click on the *Save* button.

As a result, the `Pythagoras` program is saved as a part of the Excel workbook by name Book1 with extension .xlsm.

For returning to the `Pythagoras` program:

1) open the Book1 workbook;

2) if the *Security Warning* panel appears, click on button *Enable Content*;

3) go to Visual Basic Environment;

4) click twice on *Module1* in the project explorer window.

We will execute the `Pythagoras` program in the next section.

1.2. Main commands of the program debugger

After typing the program text, the detection and correction of errors in the program follows. At this stage, we can use the debugger.

Let us consider the main commands of the debugger; we can see them in the *Debug* menu of the VB window.

1. *Step Into* — the execution of one program operator or its part. The click on *Step Into* is equivalent to pressing the *F8* key on the keyboard. This command is used for the step-by-step program execution.

2. *Run To Cursor* — the execution of the program up to the blinking cursor. The click on *Run To Cursor* is equivalent to pressing *Ctrl* + *F8*.

For setting the blinking cursor in the proper place of the program, we have to click on this place.

If we speak about key presses, the plus sign means the synchronism of these presses, i.e., “pressing *Ctrl* + *F8*” means “simultaneous pressing the *Ctrl* and *F8* keys”.

3. *Toggle Breakpoint* — the installation or liquidation of the breakpoint at the place, where the blinking cursor is located. The breakpoint marks the program line, where the program execution stops. This command can also be performed by pressing the *F9* key.

For the installation or liquidation of the breakpoint, we can click on the left border of the code window against the proper line.

4. *Clear All Breakpoints* — the liquidation of all breakpoints. This command can also be performed by pressing *Ctrl* + *Shift* + *F9*.

5. *Add Watch* — the current visualization of the value of a variable. We will review the command usage in Section 1.15.

In addition to commands *Step Into* and *Run To Cursor*, two more commands for the program execution, *Step Over* and *Step Out*, are in the *Debug* menu. We will review them in Section 1.16.

Let us consider commands *Run* and *Reset* located in the *Run* menu of the VB window.

1. *Run* — the start of the program execution (or shorter, of the program) and transition from one breakpoint to another. If the breakpoints are absent, the program is executed completely. This command is represented by arrow ► on the toolbars of the VB window, in particular, on the standard toolbar.

The program can be started from the Excel window; we will consider this possibility later (p. 113).

2. *Reset* — the discontinuation of the program execution. This command is represented by square ■ on the toolbars.

During the program execution stops (in particular, at the breakpoints), yellow color highlights the operator, which is not executed yet. If we place the mouse pointer on a variable, its value is displayed.

For obtaining the hypotenuse length by means of the `Pythagoras` program from the previous section, we fulfill the following:

1) click on the left border of the code window against the last line of the program for marking this line by the breakpoint (Fig. 1.4a);

2) click on arrow ► for executing the `Pythagoras` program up to the breakpoint;

3) if window *Macros* appears, successively click on the *Pythagoras* line and the *Run* button in this window;

4) in the code window, whose state is depicted in Fig. 1.4b, place the mouse pointer on the `c` variable in the program text; *c = 5* appears (Fig. 1.5);

5) click on arrow ► for terminating the program execution.

After starting the program execution, a window containing message *Can't execute code in break mode* may appear, indicating that we forgot to terminate (or to discontinue) the previous program execution. For correcting this error, we fulfill the following:

1) click on the *OK* button in the message window;

2) click on arrow ► (or on square ■) for termination (or discontinuation) of the program execution;

3) restart the program.

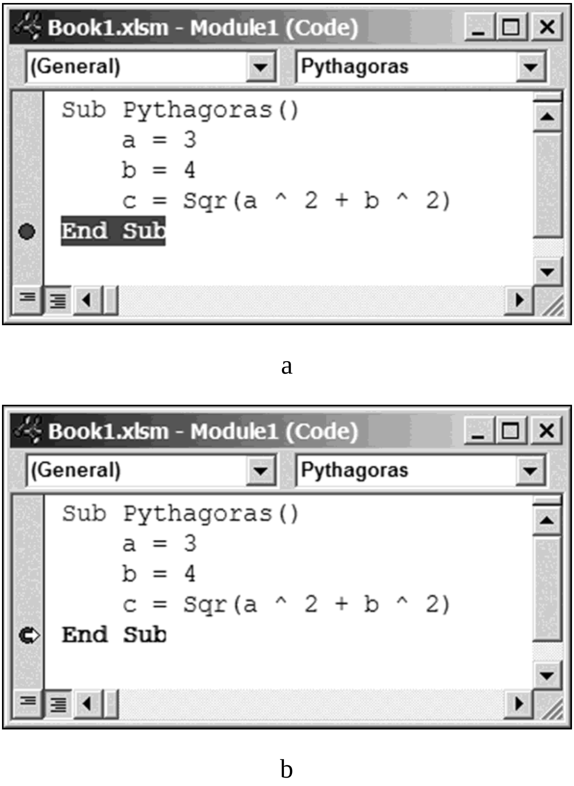
The program execution means consecutive execution of its operators: at first, the computer sets the values of variables `a` and `b`, and then calculates the value of `c`. To be convinced, we have to liquidate the breakpoint (by clicking on it) and to execute the `Pythagoras` program step-by-step by the *F8* key, watching the change in variables `a`, `b` and `c`.

Let us assume that an error was committed: when typing operator

```
c = Sqr(a ^ 2 + b ^ 2)
```

we pressed the minus key on the keyboard instead of the plus key. In this case, the program execution stops, and a window appears with the following message: *Run-time error '5': Invalid procedure call or argument.*

To understand our error, we click on button *Debug* in the message window. As a result, the place, where the stop occurred, is highlighted in yellow color. Looking at the values of `a` and `b`, we can understand the reason of the stop — the negative value of the argument of the square root function.



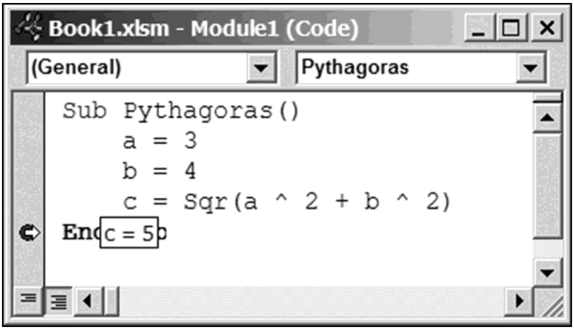
```
Sub Pythagoras()
    a = 3
    b = 4
    c = Sqr(a ^ 2 + b ^ 2)
End Sub
```

a

```
Sub Pythagoras()
    a = 3
    b = 4
    c = Sqr(a ^ 2 + b ^ 2)
End Sub
```

b

Fig. 1.4. The `Pythagoras` program with the breakpoint (a) before and (b) after starting the program execution



```
Sub Pythagoras()
    a = 3
    b = 4
    c = Sqr(a ^ 2 + b ^ 2)
End Sub
```

Fig. 1.5. The visualization of the value of `c` during the execution stop

For fuller information on the possible reasons of the stop, we fulfill the following:

1) remaining in Visual Basic Environment, start the Excel help system by pressing the *F1* key;

2) type *Error 5* in the text box of the *Excel Help* window;

3) click on the *Search* button;

4) click on the following line of the open list: *Invalid procedure call or argument (Error 5)*.

After correcting our error (that is, after changing minus to plus), we have to restart the program.

During the execution, our program (as a set of zeros and units) is located in the main memory of the computer. This program is being executed by the processor that performs different operations including, among others, arithmetic operations.

Now we will pass to the Visual Basic programming language.

1.3. Variables. Data types

Variables in programming have about the same meaning as variables in algebra. We recommend declaring a variable before its usage.

The declaration operator has the following syntax:

```
Dim variable [As type]
```

In this language construct:

- `Dim` (from “dimension”) is the keyword testifying the appearance of a new variable;
- *`variable`* is the variable name;
- `As` is the keyword;
- *`type`* is the data type (Appendix 1) of the declared variable.

Here and below, the square brackets indicate an optional part of the syntax, i.e., the part that may be absent. Such usage of the brackets is acceptable because the square and curly brackets are not used in constructs of Visual Basic.

In other words, the declaration operator has the following two versions:

```
Dim variable As type
Dim variable
```

Initially, we will consider the first version.

When (during the program execution) the computer meets the `Dim` operator, it allots a memory cell for the variable by name *`variable`*. The cell size (in bytes) is defined by *`type`* — the data type of the variable; *`type`* is keyword `Boolean`, `Byte`, `Integer`, `Long`, `Currency` or so on (Appendix 1).

According to the second column of the table in Appendix 1, the memory cell sizes, corresponding to different variables, can strongly differ.

To understand how much information is contained in one byte, let us notice that three bytes are usually enough for storing information on one pixel (the color point on the display screen) — one byte each for intensities of the red, green and blue colors.

One `Dim` operator allows declaring several variables if we list them through a comma. The example operators follow:

```
Dim my_variable As Double
Dim i As Byte, j As Integer, k As Integer
```

The restrictions on names of variables are the same as on names of programs (p. 16), at that, the upper or lower case of letters does not matter. For example, if the `Alpha` variable is declared and we try to declare the `alpha` variable, `Alpha` will be automatically replaced by `alpha`.

Alpha is the English name of a Greek letter. If the keyboard does not support the Greek language, we recommend to use English names of Greek letters according to Appendix 2 when naming variables and programs.

For reducing programming errors, we recommend to tune Visual Basic Environment so that it demands the declaration of variables. For this purpose, we fulfill the following:

1) open menu *Tools* in the VB window;
2) click on *Options*;
3) activate the *Editor* tab;
4) turn on option *Require Variable Declaration*;
5) click on the *OK* button.

As a result, the code window corresponding to a new module will contain the following first line:

```
Option Explicit
```

We can also put this line into the code window (or remove it) manually, as a usual program line.

In the presence of line `Option Explicit`, the computer diagnoses the use of an undeclared variable in the program text: during the program execution, the computer displays message *Variable not defined*.

Data types `Byte`, `Integer`, `Long`, `Currency`, `Single` and `Double` (Appendix 1) are called numerical data types.

According to the third column of the table in Appendix 1:

- in a memory cell, corresponding to a variable of the `Byte` data type, non-negative integers (up to 255) can only be stored;
- in a cell, corresponding to a variable of the `Integer` or `Long` data type, integers can be stored;
- in a cell, corresponding to a variable of the `Currency`, `Single` or `Double` data type, decimal numbers can be stored.

Let us pass to the second version of the `Dim` operator (p. 23).

If we do not specify the data type when declaring a variable (for example, by operator `Dim W`), the variable (by name `W`) automatically receives the `Variant` data type. It means that any information can be stored in a memory cell, corresponding to this variable; i.e., the Variant data type is similar to the general format of Excel.

Let us consider operator

```
Dim i, j As Integer
```

This operator is equivalent to the following:

```
Dim i As Variant, j As Integer
```

If we need the Integer data type for both variables, i and j, we should declare them as follows:

```
Dim i As Integer, j As Integer
```

or

```
Dim i As Integer
Dim j As Integer
```

Later we will consider ways of declaring variables without the use of the Dim keyword (p. 81).

As an example of using the declaration operator, let us consider the following program for calculating the number of days in the 20th century and defining the current date and time:

Listing 1.1

```
Sub Century_20()
    Dim D1 As Date, D2 As Date, D3 As Date
    Dim N As Long
    D1 = #1 Jan 1900#        'beginning date of century
    D2 = #31 Dec 1999#       'ending date of century
    N = D2 - D1 + 1          'number of days in century
    D1 = Time                'current time
    D2 = Date                'current date
    D3 = Now                 'date and time
End Sub
```

In the second and third lines of this program, Date and Long are the data types (Appendix 1); #1 Jan 1900# and #31 Dec 1999# mean dates on January 1, 1900 and December 31, 1999. The subtraction of the first date from the second date determines the number of days between these dates.

Word Time means determination of the current time by means of the VB function by name Time. In other words:

- Time is the call of the Time function of VB;
- the value of this function is the current time of the day or, that is the same, the Time function returns the current time of the day into the program ("into the program" may be omitted).

In operator

```
D2 = Date
```

word Date means the call of the Date function returning the current date into the program.

We see that Date is a name of the VB function and a name of the data type. Thanks to various contexts, it does not lead to confusion.

Word Now means the call of the Now function returning the current date and time together.

Regarding the VB functions, we will talk in more details later, in particular, in Sections 1.4 and 1.9.

To be convinced of the operational capability of the Century_20 program, we fulfill the following operations:

1) insert a module into the active Excel workbook (p. 15);

2) enter the Century_20 program text into the code window of the new module;

3) make the step-by-step execution of this program by means of the *F8* key, watching the values of variables D1, D2, D3 and N.

It should be emphasized that, before the first press of key *F8*, we must place the blinking cursor inside the program text, not in line Option Explicit.

Text Listing 1.1 of the Century_20 program can be entered into the code window by means of the keyboard. We can also copy it from the enclosed CD.

For copying:

1) open file Listing_1_01.txt with the Notepad editor, for example, by double click on the pictogram of this file in Windows Explorer;

2) in the Notepad window opened, highlight the program text and copy it into Windows Clipboard, for example, by pressing *Ctrl* + *C*;

3) by the click, locate the blinking cursor in the code window of Visual Basic Environment;

4) paste the program text from Windows Clipboard into the code window, for example, by pressing *Ctrl* + *V*;

5) close the Notepad window with file Listing_1_01.txt.

1.4. Two main functions for conversion of data types

A string is a quoted sequence of characters. The example strings follow:

```
"Hello, World!"
"13.333"
"37 RUR"
"$ 37"
" "
```

The last string contains only the space character.

Let us supplement the string definition by the empty string, `""`, which does not contain any characters.

Strings `" "` and `""` are used in program Listing 1.7. In Section 1.19, we will expand the string definition even more.

Converting string to number is often necessary. For this purpose, the `Val` function is used. It converts the numerical beginning of string to number. If the `Val` function cannot perform this, it returns zero into the program. The argument of the `Val` function is a string; this function returns a number.

For the backward conversion (that is, number to string), the `Str` function is used. The argument of this function is a number, variable of numerical data type (p. 24) or arithmetic expression (Section 1.8). The `Str` function returns a string.

Let us make the step-by-step execution of the program below.

Listing 1.2

```
Sub StrVal()
    Dim strA As String
    Dim curB As Currency
    strA = "45.77"
    curB = Val(strA)           'result: curB = 45.77
    strA = Str(curB)           'result: strA = " 45.77"
    curB = Val("4.7 = X")      'result: curB = 4.7
    curB = Val("4,7 = X")      'result: curB = 4
    curB = Val("X = 4.7")      'result: curB = 0
    curB = Val("")             'result: curB = 0
End Sub
```

The first comment corresponds to the case when operator

```
strA = Str(curB)
```

is highlighted in yellow color during the step-by-step program execution. This comment means the following: if the mouse pointer is located on `curB`, information *curB = 45.77* appears.

Information *curB = 45.77* is the result of executing operator

```
curB = Val(strA)
```

which is in the same line with the comment.

The second comment corresponds to the case when operator

```
curB = Val("4.7 = X")
```

is highlighted in yellow color during the step-by-step execution. This comment means the following: if the mouse pointer is located on `strA`, information *strA = " 45.77"* appears. And so on.

All comments, which begin with word "result" or "Returns" (p. 33), have similar sense.

Let us remind that, during the stops of the program execution, yellow color highlights the operator, which is not executed yet.

In addition to the `Val` and `Str` functions, there are other functions for conversion of data types. We will review them in Section 1.8.

1.5. Constants

Constants are similar to variables. However, unlike a variable, the content of the memory cell, corresponding to a constant, cannot be changed during the program execution. There are two versions of constants in Visual Basic, named built-in and user-defined constants.

The user-defined constant can be declared by means of operator

```
Const invariable [As type] = value
```

In this operator:

- `Const` is the keyword testifying the appearance of a new constant;
- *`invariable`* is the constant name;
- `As` is the keyword, as in the `Dim` operator (p. 23);
- *`type`* is the data type (Appendix 1) of the declared constant;
- *`value`* is a value of the declared constant.

The restrictions on names of constants are the same as on names of variables and programs (p. 16).

Examples of the constant declaration follow:

```
Const e As Double = 2.718281828
                            'base of natural logarithm
Const e = 2.718281828
Const phi = 1.618033989                 'gold relation
Const Flag As Boolean = False
Const Message = "End of Work"
Const Millennium As Date = #1 Jan 2000#
Const beta As Currency = 2 ^ 0.5
```

In the fourth example operator, `Boolean` is the so-called logical data type (Appendix 1).

When executing the last operator, the rounded square root of 2 (that is, 1.4142) is assigned to constant `beta`. This example shows that *`value`* in the `Const` operator can be an elementary arithmetic or logical expression (Sections 1.8 and 1.10). In this case, the content of the memory cell, corresponding to the constant, is determined by the expression value rounded according to the `type` data type.

One `Const` operator allows declaring several constants if we list them through a comma. The example operator follows:

```
Const Min = 0, Max = 1000, tau As Double = 6.283185307
```

As an example of using constants, let us consider the following program for conversion of an angle from degrees to radians:

```
Sub deg2rad()
    Dim angleD As Double
    Dim angleR As Double
    Const pi As Double = 3.141592654     'pi = tau / 2
    angleD = 270                 'angle equals 270 degrees
    angleR = angleD * pi / 180
                                 'result: angle in radians
End Sub
```

As we see, operator

```
Const pi As Double = 3.141592654
```

declares the `pi` constant before its usage in operator

```
angleR = angleD * pi / 180
```

The `rad2deg` program below, which converts an angle from radians to degrees, is similar to the above program.

```
Sub rad2deg()
    Dim angleD As Double
    Dim angleR As Double
    Const pi180 As Double = 3.141592654 / 180
    angleR = 4.5                 'angle equals 4.5 radians
    angleD = angleR / pi180
                                 'result: angle in degrees
End Sub
```

On p. 82, we will consider the declaration of a constant by means of keyword combination `Public Const`.

The built-in constant does not need any declaration. Names of the built-in constants of Visual Basic begin with prefix vb, for example, vbFriday (this constant equals 6).

For names (in particular, names of constants), the developers of Windows accepted the following agreement: names of similar data begin with the same short prefix. In particular, the built-in constants of Visual Basic have prefix vb, the built-in constants of Excel have prefix xl.

In addition to vbFriday, we will come across the following built-in constants: vbYesNo, vbYes, vbTab, vbCrLf, vbCr, vbLf, xlR1C1, xlCalculationAutomatic, xlCalculationManual, xlDialogOpen, xlDialogSaveAs, xlCenter, etc.

1.6. Obtaining information

For obtaining information on a built-in constant, we must press the *F2* key when the VB window is active. As a result, the object browser window appears (Fig. 1.6). In the top box of this window, we should set <*All Libraries*> by means of the drop-down list. In the text box below, we should type what is interesting for us, for example, *vbFriday*. Then we click on the binoculars pictogram. The answer is in the *Search Results* area (Fig. 1.7).

For closing the object browser window, we have to click on the little cross at the right end of the menu bar.

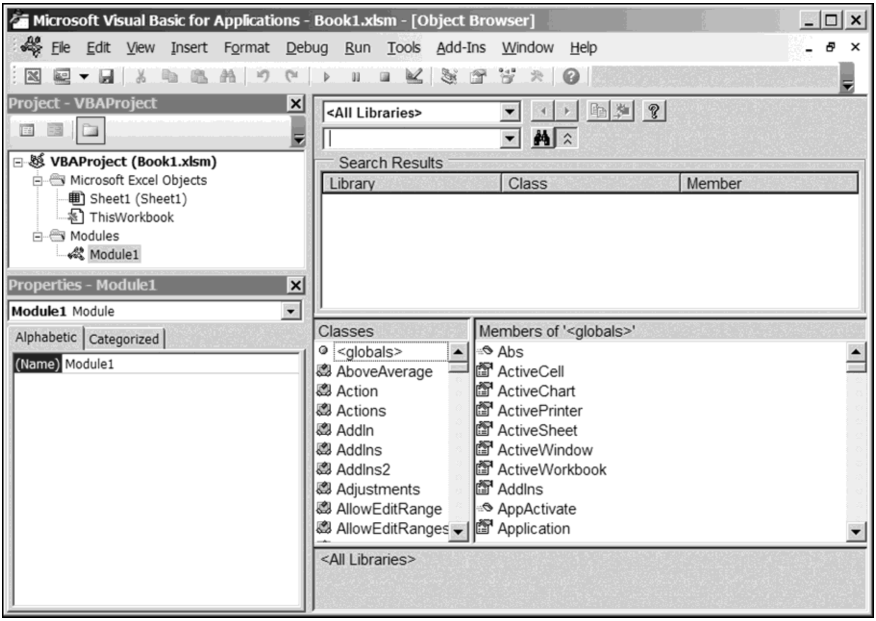
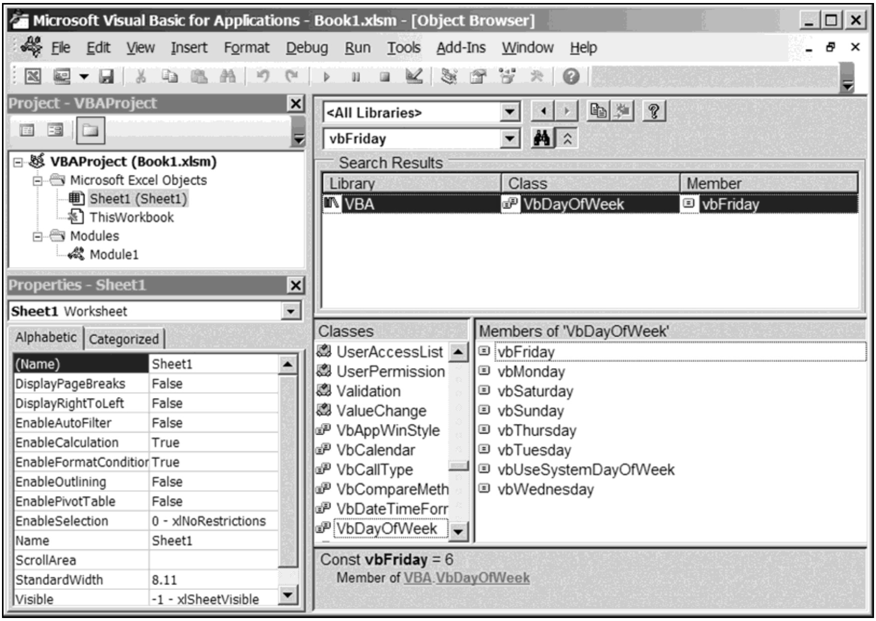


Fig. 1.6. The VB window with the object browser window instead of the code window



(Name)	Sheet1
DisplayPageBreaks	False
DisplayRightToLeft	False
EnableAutoFilter	False
EnableCalculation	True
EnableFormatConditior	True
EnableOutlining	False
EnablePivotTable	False
EnableSelection	0 - xlNoRestrictions
Name	Sheet1
ScrollArea	
StandardWidth	8.11
Visible	-1 - xlSheetVisible

Library	Class	Member
VBA	VbDayOfWeek	vbFriday

Classes: UserAccessList, UserPermission, Validation, ValueChange, VbAppWinStyle, VbCalendar, VbCallType, VbCompareMeth, VbDateTimeForr, VbDayOfWeek

Members of 'VbDayOfWeek': vbFriday, vbMonday, vbSaturday, vbSunday, vbThursday, vbTuesday, vbUseSystemDayOfWeek, vbWednesday

Const **vbFriday** = 6
Member of VBA.VbDayOfWeek

Fig. 1.7. Information on the built-in vbFriday constant

The Excel help system, started by pressing the *F1* key, is useful too (p. 22). For accelerating the process of finding the necessary information, the blinking cursor must be preliminarily located on the word of interest.

By means of the Excel help system, we will study the `Val` function. For this purpose, let us fulfill the following.

1. Enter the `StrVal` program (p. 27) into the code window.
2. Locate the blinking cursor on the `Val` word by clicking on it.
3. Press the *F1* key. As a result, the *Excel Help* window, containing the full information on the `Val` function, is displayed (Fig. 1.8).
4. After studying the last information, copy fragment

```
Dim MyValue
MyValue = Val("2457")     ' Returns 2457.
MyValue = Val(" 2 45 7")     ' Returns 2457.
MyValue = Val("24 and 57")     ' Returns 24.
```

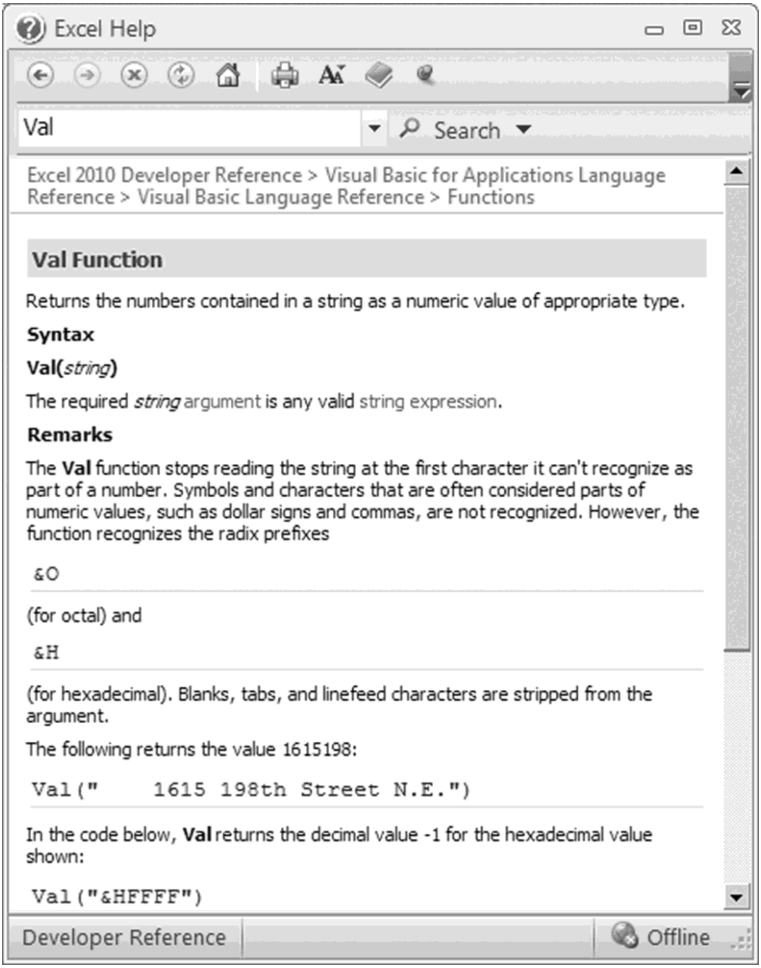
from the bottom part of the *Excel Help* window into the `StrVal` program as follows:

1) highlight this fragment by the mouse, as in Notepad;

2) copy it into Windows Clipboard by pressing *Ctrl* + *C*;

3) locate the blinking cursor in the code window, against the last line of the `StrVal` program;

4) paste the fragment from Windows Clipboard into the program text by pressing *Ctrl* + *V*.



Excel 2010 Developer Reference > Visual Basic for Applications Language Reference > Visual Basic Language Reference > Functions

Val Function

Returns the numbers contained in a string as a numeric value of appropriate type.

Syntax

Val(*string*)

The required *string* argument is any valid string expression.

Remarks

The **Val** function stops reading the string at the first character it can't recognize as part of a number. Symbols and characters that are often considered parts of numeric values, such as dollar signs and commas, are not recognized. However, the function recognizes the radix prefixes

```
&O
```

(for octal) and

```
&H
```

(for hexadecimal). Blanks, tabs, and linefeed characters are stripped from the argument.

The following returns the value 1615198:

```
Val("    1615 198th Street N.E.")
```

In the code below, **Val** returns the decimal value -1 for the hexadecimal value shown:

```
Val("&HFFFF")
```

Fig. 1.8. The *Excel Help* window with information on the `Val` function

As a result, the `StrVal` program takes the following form:

```
Sub StrVal()
    Dim strA As String
    Dim curB As Currency
    strA = "45.77"
    curB = Val(strA)           'result: curB = 45.77
    strA = Str(curB)           'result: strA = " 45.77"
    curB = Val("4.7 = X")      'result: curB = 4.7
    curB = Val("4,7 = X")      'result: curB = 4
    curB = Val("X = 4.7")      'result: curB = 0
    curB = Val("")             'result: curB = 0
Dim MyValue
MyValue = Val("2457")     ' Returns 2457.
MyValue = Val(" 2 45 7")     ' Returns 2457.
MyValue = Val("24 and 57")     ' Returns 24.
End Sub
```

We advise the reader to execute this program step-by-step (by means of the *F8* key), watching the value of the `MyValue` variable.

1.7. Assignment operator

The assignment operator has the following syntax:

```
variable = expression
```

Here, *`variable`* is the variable name, *`expression`* is an arithmetic or logical expression (Sections 1.8 and 1.10) or string, which can be considered as an expression (Section 1.19). A separately taken number, constant, variable or function is also an arithmetic expression. In the assignment operator, = is the so-called assignment sign.

The assignment operator works as follows:

1) the value of *`expression`* is calculated;

2) the resulting value is assigned to *`variable`*, i.e., is written into the corresponding memory cell.

If the data type of the variable in the left-hand side of the assignment operator does not coincide with the type of the expression value in the right-hand side, the value type is generally converted (transformed) during the execution of the assignment operator.

Let us make the step-by-step execution of the following program for converting strings `"78.8"`, `"78,8"` and `"78;8"` to numbers of the `Currency` data type.

```
Sub Conversion()
    Dim curA As Currency
    Dim curB As Currency
    Dim curC As Currency
    curA = "78.8"                 'result: curA = 78.8
    curB = "78,8"                 'result: curB = 78.8
    curC = "78;8"                 'result is absent
End Sub
```

Because of executing the first and second assignment operators, strings `"78.8"` and `"78,8"` are successfully converted to number `78.8`.

When executing assignment operator

```
curC = "78;8"
```

the stop occurs with the following information: *Run-time error '13': Type mismatch*. It speaks about the following:

- the types of variable on the left and of value on the right of the assignment sign (=) are different;
- the computer cannot convert string `"78;8"` to a number.

We will continue considering the data type conversion in the next section of the book.

Unlike other programming languages, for example C++, multiple assignments, for example `x = y = z = 1.3`, are inadmissible in VB. We have to use several assignment operators with the same right-hand side, i.e., language construct

```
x = 1.3
y = 1.3
z = 1.3
```

or

```
x = 1.3: y = 1.3: z = 1.3
```

is correct.

1.8. Arithmetic expression

In Visual Basic, an integer is represented by a sequence of figures with the minus sign or without any sign. Examples of integers are

```
-18      32      0
```

If a number has the fractional part, it separates from the integral part by a point. In this case, we may omit the integral part if it equals zero. Examples of decimal numbers follow:

```
0.5      -5.68      -.12      .035      3.
```

In the last example, `3` with a point means a number with zero fractional part, i.e., an integer, in fact.

We reviewed the main form of decimal numbers.

Decimal numbers may also be represented in exponential form. For example, `-1.6E-19` is the electron charge, $-1.6 \cdot 10^{-19}$ C (its absolute value is figured in Section 3.13). Instead of E, letter D may be used in the exponential representation, i.e., the electron charge, $-1.6 \cdot 10^{-19}$ C, may be written as `-1.6D-19`

One of the main constructs of any programming language is an arithmetic expression similar to the algebraic expression in mathematics. However, we cannot omit the multiplication sign in the arithmetic expression. Table 1 below contains equivalent algebraic and arithmetic expressions.

Table 1. Examples of expressions

Algebraic expression	Arithmetic expression of VB
$5x+12y$	`5 * x + 12 * y`
$\frac{x}{y}$	`x / y`
y^x	`y ^ x`
x	`x`
$19.55 \cdot 10^{-6}$	`19.55E-6` or `19.55D-6`

1.8. Arithmetic expression

The arithmetic expression contains the invariables (numbers and constants), variables and/or functions related by arithmetic operations. As we already mentioned in the previous section, an individual number, constant, variable or function is also an arithmetic expression.

The expressions in Table 1 do not include functions. We will review arithmetic expressions with functions in the next section.

Arithmetic operations are denoted as follows: + (addition), **–** (subtraction or sign change operation), `*` (multiplication), `/` (division), `^` (power operation), `\` (integer division, i.e., division of integers neglecting the integer remainder), `Mod` (modulus operation, i.e., determining the integer remainder after division of integers).

According to the assignment operator syntax, the arithmetic expression is on the right of sign =. For example, assignment operator

```
z = 5 * x + 12 * y
```

includes arithmetic expression

```
5 * x + 12 * y
```

Let us consider the following program:

```
Sub Arithmetic1()
    Dim m As Integer
    Dim n As Integer
    Dim x As Double
    m = 5
    n = 2
    x = m / n                          'result: x = 2.5
    x = m \ n                          'result: x = 2
    x = m Mod n                        'result: x = 1
End Sub
```

To verify the correctness of this program, ***we advise the reader*** to make the step-by-step execution (by means of the *F8* key), watching the change in the value of `x`.

If the arithmetic expression contains several operations, the order of their performance is defined by the following rules of priorities of arithmetic operations:

1) first of all, the power operation (`^`) is performed;

2) next, the multiplication and division (`*`, `/`) are performed in that sequence as they are in the expression;

3) the integer division (\ );

4) the modulus operation (`Mod`);

5) the sign change operation (−);

6) finally, the addition and subtraction (+, −) are performed in that sequence as they are in the expression.

As we see, the power operation (^) has the highest priority in VB.

The rules of priorities of arithmetic operations in VB differ from the rules of priorities in Excel regarding the power operation, multiplication, division, the sign change operation (so-called negation), addition and subtraction: the negation (−) is performed first, i.e., has the highest priority in Excel.

To be convinced, we put

```
=-1^2
```

into the Excel formula box and click on the tick button of the Excel formula bar (or press the *Enter* key). As a result, value 1 appears in the active cell on the worksheet. After the execution of VB operators

```
Dim i As Integer
i = -1 ^ 2
```

the `i` variable is equal to -1.

In both VB and Excel, parentheses are used for changing the sequence of the operations: the values of the parenthesized arithmetic expressions are calculated at first.

We will not see the square and curly brackets in the arithmetic expressions of Table 2 below because, as it was already told, such brackets are not used in the VB constructs.

When calculating the expression value, results of performance of intermediate arithmetic operations remain in the processor or are written into cells of the cache memory (from where they are read when required).

The cache memory has a short time of reference (that is, of writing and reading information); this time is much shorter than the time of reference for the main memory. The cache memory is intended for temporary storage of intermediate results and contents of memory cells often used. The program fragment, which is being executed, may also be stored in the cache memory.

Let us consider the following assignment operator with the arithmetic expression in the right-hand side:

```
z = 5 * x + 12 * y
```

The processor is performing this operator approximately thus:

1) multiplies `5` by the content of cell `x`;
2) writes the result into a cache memory cell, for example, `cache1`;
3) multiplies `12` by the content of cell `y`;
4) adds the result to the value of `cache1`;
5) writes the result into cell `z`.

Table 2. Examples of expressions

Algebraic expression	**Arithmetic expression of VB**
$7\{3[a+b(c+d)]+8\}+2$	`7 * (3 * (a + b * (c + d)) + 8) + 2`

$-a^b$	`-a ^ b`
	or
	`-(a ^ b)`

a^{-b}	`a ^ (-b)`
a^{b-c}	`a ^ (b - c)`
$10^{-4.7}$	`10 ^ (-4.7)`
$10^{4.7}$	`10 ^ 4.7`
$A\cdot B$	`A * B`

$A(-B)$	`A * (-B)`
	`-A * B`
	or
	`-(A * B)`

a^{b^c}	`a ^ (b ^ c)`

$(a^b)^c$	`a ^ b ^ c`
	or
	`(a ^ b) ^ c`

$\frac{a\cdot b}{c\cdot d}$	`(a * b) / (c * d)`
	or
	`a * b / (c * d)`

$a\cdot 10^4$	`a * 1E4`
	`a * 10E3`
	or
	`a * 10000`

In mathematics and programming, participants of operations are called operands, both in the case of arithmetic operations and in the case of logical operations (Section 1.10). For example, arithmetic expression `5 * x` includes the following two operands: integer `5` and variable `x`.

For a better understanding of the rules of priorities of arithmetic operations, let us consider the following program:

```
Sub Arithmetic2()
    Dim m As Integer
    Dim n As Integer
    Dim x As Single
    Dim y As Single
    x = 3
    m = 2
    n = -1
    y = (-3) ^ m                          'result: y = 9
    y = -(3 ^ m)                          'result: y = -9
    y = -3 ^ m                            'result: y = -9
    y = 10 + (x + 7) ^ (m + n)            'result: y = 20
    y = 10 + x + 7 ^ m + n                'result: y = 61
End Sub
```

We advise the reader to make the step-by-step execution of this program, watching the value of the `y` variable and explaining the value origin.

In addition, ***we advise the reader*** to verify that

```
(m \ n) * n + m Mod n
```

equals `m` for arbitrary integers `m` and `n` (naturally, `n` ≠ 0). For that, the reader has to write a program, which is similar to `Arithmetic1`, and to execute the new program step-by-step.

Arithmetic expressions may contain variables and invariables of different types. If the type of the value of arithmetic expression in the right-hand side of the assignment operator (on the right of sign =) does not coincide with the data type of the variable in the left-hand side of the assignment operator (on the left of sign =), the type of the value is converted during the assignment.

Let us consider the situation when the value of arithmetic expression on the right of sign = has a fractional part and the variable on the left of sign = is of the `Integer` or `Long` data type. During the assignment, the value is transformed according to the following rules for rounding off:

- if the fractional part of the value is equal to 0.5, this value is rounded up to the even number from two nearest integers;
- otherwise, the value is rounded up to the nearest integer.

Because operations `\` and `Mod` are applicable only to integers, the execution of these operations over numbers with a fractional part begins with the rounding off the operands to integers according to the formulated rules. The results of operations \ and Mod are integers.

An operation with one operand is called a unary operation. Among the arithmetic operations, only the sign change operation (−) is unary. An operation with two operands (^, *, /, \, Mod, +, − as subtraction) is called a binary operation.

VB includes special functions for converting data types. Two of these functions (Str and Val) were reviewed in Section 1.4; the remaining functions are listed in Table 3 below.

Table 3. Functions for converting data types

Function name	Resulting data type
CBool	Boolean
CByte	Byte
CCur	Currency
CDate	Date
CDbl	Double
CInt	Integer
CLng	Long
CSng	Single
CStr	String
CVar	Variant

Requirements to the argument of these functions and examples of their usage are given in the Excel help system. For accelerating the process of finding the necessary information, we must press the *F1* key when the VB window is active.

Let us make the step-by-step execution of the program below.

```
Sub Functions()
    Dim intN As Integer
    Dim strN As String
    Dim curN As Currency
    intN = -15
    strN = Str(intN)             'result: strN = "-15"
    strN = CStr(intN)            'result: strN = "-15"
    intN = 15                    '8th operator
    strN = Str(intN)             '9th operator
                                 'result: strN = " 15"
    strN = CStr(intN)            '10th operator
                                 'result: strN = "15"
    curN = 25.5                  '11th operator
    intN = 1 + CInt(curN)          '12th operator
                                   'result: intN = 27
    intN = CInt(1 + curN)          '13th operator
                                   'result: intN = 26
    intN = CInt("78.8")            '14th operator
                                   'result: intN = 79
    intN = CInt("78,8")            '15th operator
                                   'result: intN = 79
End Sub
```

Note the following:

- if the argument of the `Str` and `CStr` functions is a non-negative number, these functions return different strings (see the 9th and 10th operators);
- the `CInt` function rounds the argument value according to the rules for rounding off (see the 12th and 13th operators);
- a string may be the `CInt` function argument (see the 14th and 15th operators).

1.9. Mathematical functions. Functions of date and time

Let us start with an analysis of the mathematical functions given in the table of Appendix 3.

We already used the square root function, `Sqr(x)`, in our first program — `Pythagoras` on p. 17.

The argument of trigonometric functions (cosine, sine and tangent) is an angle in radians, not in degrees.

Function `Atn(x)` is an inverse trigonometric function, $\arctan x$. The arctangent returns (into the program) the angle in radians from -π/2 to π/2 whose tangent is equal to the value of `x`. Such angle is called the principal angle of $\tan x$.

The sign function, `Sgn(x)`, returns -1, 0, 1 at `x < 0`, `x = 0`, `x > 0`, respectively.

The `Log(x)` function is the natural logarithm of `x`, $\ln x$. According to the logarithm properties [3], the following expressions are valid for the decimal logarithm: $\lg x = \ln x / \ln 10 = \ln x / 2.302585093$.

In Appendix 3, in addition to the main mathematical functions of Visual Basic, the VB operators are given for counting the values of trigonometric function $\cot x$, of inverse trigonometric functions $\arcsin x$, $\arccos x$ and $\mathrm{arccot}\, x$ and of decimal logarithm $\lg x$.

In addition to the mathematical functions of Appendix 3, let us consider function `Round(x[, n])` intended for rounding off numbers with a fractional part. As we know, the square brackets separate an optional part of the construct. In other words, this function of VB has the following two versions:

- the `Round(x, n)` function returns the value of `x`, rounded up to `n` decimal places;
- the `Round(x)` function returns the integer obtained by rounding off the value of `x` according to the rules formulated on p. 42; this function is identical to the `Round(x, 0)` and `CInt(x)` functions.

Arguments of the mathematical functions are arithmetic expressions. Assignment operator

```
c = Sqr(a ^ 2 + b ^ 2)
```

in the Pythagoras program (p. 17) is an example of this.

The mathematical functions are used in arithmetic expressions, and the functions have a priority as compared to the arithmetic operations when calculating the expression value. For example, during the execution of operator

```
e = b / c * Cos(a) ^ 3 - d
```

the processor first calculates the cosine value, and then:

1) cubes this value;

2) writes the result into the cache1 cell of the cache memory;

3) successively performs the division, multiplication and subtraction for calculating the value of arithmetic expression

```
b / c * cache1 - d
```

4) writes the result into the memory cell corresponding to variable e.

To learn the use of the mathematical functions, let us do the following:

1) put operator block

```
Dim V As Single, W As Single
V = intN + Round(Sqr(2 * intN), 2)
W = Round(V) ^ 2
```

into the code window containing the Functions program (p. 43), above the last line;

2) mark the last line of the program by the breakpoint;

3) click on arrow ► for the program execution up to the breakpoint;

4) make sure that the calculated values of V and W are equal to 33.21 and 1089, respectively;

5) explain these results;

6) click on arrow ► for terminating the program execution.

In addition to the considered functions with one and two arguments, there is the Rnd function (from “random”) without arguments. This function is intended for generation of random numbers used for modeling random phenomena. The simplest random phenomenon can be described as follows: there is a 50 % chance that the “head” or “tail” will be the result of a coin flip.

The idea of modeling random phenomena is known for a long time. Following the advent of the electronic computers, this idea was developed in the 1950s under the name of the Monte Carlo method.

The Monte Carlo method is used for modeling financial risks, semiconductor devices and evolution of stars. It is only a part of the problems demanding generation of random numbers.

The use of the `Rnd` function is described in the Excel help system, which must be started by pressing the *F1* key when the VB window is active. The `RandomNumbers` program in Section 1.15 and code Listing 6.11 in Section 6.8 are examples of the `Rnd` function's usage.

Functions of date and time, `Time`, `Date` and `Now`, are without arguments too. These functions return the following values of the `Date` data type: current time, date and date together with time, respectively. They appeared in program `Century_20` (p. 25). We will encounter these functions of date and time more than once.

1.10. Logical expression

In addition to arithmetic expressions, logical expressions are also important in Visual Basic.

The logical expression uses the following well known comparison signs: < (less than), > (greater than), = (equal to), <= (less than or equal to, ≤ in mathematics), >= (greater than or equal to, ≥ in mathematics), <> (unequal to, ≠ in mathematics).

The logical expression accepts one of the two Boolean (logical) values, `True` (logical unit) or `False` (logical zero). Separately taken `True` or `False` is also a logical expression.

Examples of the logical expression follow:

```
5 >= 3
5 < 3
False
```

The first logical expression accepts `True`; the second and third logical expressions accept `False`.

Let us consider the following program:

```
Sub Logic1()
    Dim x As Integer
    Dim y As Integer
    Dim blnA As Boolean
    x = 5: y = 2
    blnA = x > y                'result: blnA = True
    blnA = x = y                'result: blnA = False
End Sub
```

In this program, we see four assignment operators and two logical expressions (`x > y` and `x = y`). According to the assignment operator syntax, the logical expressions are on the right of the assignment sign. The values of the logical expressions for `x = 5` and `y = 2` are given in the corresponding comments.

In complicated logical expressions, logical operations are used. We will consider three of them: `Not`, `And`, `Or`.

Operation Not is the so-called logical negation. It is defined as follows:

- if A equals True, then Not A equals False;
- if A equals False, then Not A equals True.

Operation Not has one operand (A), i.e., it is a unary operation.

Definitions of logical operations And and Or are given by the following two tables.

Definition of the And operation

A	B	A And B
True	True	True
True	False	False
False	True	False
False	False	False

Definition of the Or operation

A	B	A Or B
True	True	True
True	False	True
False	True	True
False	False	False

According to these tables, the And and Or operations have two operands (A and B), i.e., they are binary operations.

Logical expression A And B is equal to True only in that case when both operands are equal to True. In all other cases, expression A And B is equal to False. The And operation is called conjunction or logical multiplication.

Expression A Or B is equal to False only in that case when both operands are equal to False. In all other cases, expression A Or B is equal to True. The Or operation is called disjunction or logical addition.

In the presence of several logical operations in a logical expression, the order of their performance is defined by the following rules of priorities:

1) first of all, operation Not (logical negation) is performed;
2) further, And (logical multiplication);
3) in last turn, Or (logical addition) is performed.

For change of sequence of the operations’ performance, parentheses are used, as in arithmetic expressions. Parentheses may be also used for readability of logical expressions.

For a better understanding of the logical operations, let us consider the following program:

```
Sub Logic2()
    Dim x As Double
    Dim y As Double
    Dim z As Double
    Dim blnA As Boolean
    x = 1
    y = 2.87
    z = 3.12
    blnA = (x > y) And (y < z)    'result: blnA = False
    blnA = x < y And y < z        'result: blnA = True
    blnA = x > y Or y > z         'result: blnA = False
    blnA = Not (x < y Or Not y < z)
                                  'result: blnA = False
    blnA = Not x > y And x > y    'result: blnA = False
    blnA = Not (x > y And x > y)  'result: blnA = True
End Sub
```

In operator

```
blnA = (x > y) And (y < z)
```

parentheses are used for readability of logical expressions `x > y` and `y < z`. These parentheses may be omitted.

We advise the reader to verify the correctness of the `Logic2` program by means of the step-by-step execution.

Note that double logical expressions, for example `0 < x <= 1`, are inadmissible in VB. Instead of `0 < x <= 1`, we have to write

```
0 < x And x <= 1
```

or

```
x > 0 And x <= 1
```

For grasping the material of this and the previous sections, ***we advise the reader*** to write a program allowing to define the values of `y`, for which `CInt(y)`, `Fix(y)`, `Int(y)`, `Round(y)` are equal to each other if `y` accepts the following values:

- -1.8, -1.25, 1.27, 1.68;
- $f(a)$, $f(0.5a+0.5b)$, $f(b)$, $0.5f(a)+0.5f(b)$.

Here, functions `Fix(y)` and `Int(y)` of VB are described in Appendix 3, functions `CInt(y)` and `Round(y)` are from Sections 1.8 and 1.9, function $f(x)$ is from Appendix 4, *a* and *b* are the boundaries of the $f(x)$ function's domain.

In addition, ***we advise the reader*** to replace condition "`CInt(y)`, `Fix(y)`, `Int(y)`, `Round(y)` are equal to each other" of the previous task by one of the following conditions:

1) `Fix(y) = CInt(y) And Fix(y) <> Int(y)`
2) `Int(y) = CInt(y) And Int(y) <> Fix(y)`
3) `Fix(y) = Int(y) Or Fix(y) <> CInt(y)`
4) `Fix(y) = CInt(y) Or Fix(y) <> Int(y)`
5) `Int(y) = CInt(y) Or Int(y) <> Fix(y)`
6) `Fix(y) = Int(y) Or Fix(y) = CInt(y)`
7) `Int(y) = CInt(y) Or Int(y) = Fix(y)`
8) `CInt(y) = Fix(y) Or CInt(y) = Int(y)`
9) `Fix(y) = Int(y) And Fix(y) <> CInt(y)`

1.11. GoTo operator

Operators of the previous programs are executed by turn. Such programs are called linear programs.

The `GoTo` operator is used to change the order of execution of the program operators. It has the following syntax:

```
GoTo lbl
```

In this syntax, the so-called label, *`lbl`*, may be one of the following:

- a non-negative integer without a sign (`0`, `1`, `2`, `3`, …);
- a sequence of letters, figures and underscores beginning with a letter, for example, `start_53a`.

We have to place the *`lbl`* label in front of the operator, to which the jump must be performed (or, that is the same, to which the control must be transferred). We have to type a colon behind the label.

After executing the operator with *`lbl`* in front, the next operator will be executed if the labeled operator is not `GoTo`.

Examples of `GoTo` usage are given on p. 54: we see two labels in the `IT2` program, `2` and `LastLine`.

If the label is a non-negative integer, this label may be called an operator's (line's) number. For example, operator

```
If X > 9 And X < 12 Then GoTo LastLine
```

in the `IT2` program with label `2` in front may be named as operator `2` (line `2`).

The `GoTo` operator is often called ***the unconditional jump operator***. In the next section, we will consider `GoTo` as a part of the so-called conditional jump operator.

1.12. Decision-making constructs

This is a typical situation when, in a certain place of a program, it is necessary to execute those or other operators depending on some conditions. The choice of the operators is performed by means of one of two decision-making constructs, `If…Then` or `Select Case`.

The first decision-making construct, `If…Then`, is called ***the conditional operator***. Below are several versions of this operator.

The simplest conditional operator follows:

```
If condition Then statement1
```

where *`condition`* is a logical expression.

The computer calculates the value of *`condition`*. If `True` (`False`) is the result, we will say that the condition is true (false).

If the condition is true, operator *`statement1`* is executed; if the condition is false, operator *`statement1`* is not executed.

Further, the next operator (following the `If…Then` construct) is executed, regardless of whether or not *`statement1`* was executed (if *`statement1`* is not the `GoTo` operator).

Let us consider the following example program:

```
Sub IT1()
    Dim X As Byte
    X = 12                              'initial value of X
    If (X > 9 And X < 12) Then X = X + 1
    X = X + 2
    X = X * 2                           'final value of X
End Sub
```

For understanding this program, ***we advise the reader*** to do the following:

1) install the breakpoint against the `End Sub` line;
2) click on arrow ► for the program execution up to the breakpoint;
3) make sure that the calculated value of `X` is equal to 28 during the stop of the program execution;

4) explain this result;
5) click on arrow ► for terminating the program execution.

A special case of the reviewed `If…Then` construct is ***the conditional jump operator*** with the following syntax:

```
If condition Then GoTo lbl
```

According to this construct including `GoTo`, if the condition is true, the jump is performed to the operator labeled by *`lbl`*.

The program below is an example of using the conditional jump operator.

```
Sub IT2()
    Dim X As Integer
    X = 12
2:  If X > 9 And X < 12 Then GoTo LastLine
    X = X - 2
    GoTo 2
LastLine:
End Sub
```

We advise the reader to make the step-by-step execution of this program and to explain why the value of `X` changes so, instead of differently.

The quantity of operators, which must be executed when the condition is true, may be greater than one. In this case, the following construct is used:

```
If condition Then
    statements
End If
```

where *`statements`* is an operator block. This conditional operator is performed as follows.

If the condition is true, block *`statements`* (below the `Then` keyword) is executed. If the condition is false, block *`statements`* is not executed.

Further, the jump is performed to the operator following the `If…Then` construct, regardless of whether or not block *`statements`* was executed (if this block does not include the `GoTo` operator).

Let us consider the following `If…Then` construct:

```
If condition Then statement1 Else statement2
```

If the condition is true, operator *`statement1`* (behind keyword `Then`) is executed. If the condition is false, operator *`statement2`* (behind keyword

Else) is executed. Further, the jump is performed to the operator following the If…Then construct, without dependence of what operator was executed earlier, *statement1* or *statement2* (if these operators are not GoTo).

The program below is an example of using the last construct.

```
Sub IT3()
    Dim X As Byte
    X = 12                           'initial value of X
    If (X > 9 And X < 12) Then X = X + 3 _
        Else X = X + 2
    X = X * 2                        'final value of X
End Sub
```

We will assume that operator blocks *statements1*, *statements2*, ..., *statementsN*, *statements* do not include GoTo (for brevity).

Let us consider the following construct, which is similar to the previous If…Then construct, but contains blocks instead of operators:

```
If condition Then
    statements1
Else
    statements2
End If
```

If the condition is true, block *statements1* (which is below keyword Then) is executed. If the condition is false, block *statements2* (which is below keyword Else) is executed. Further, the jump is performed to the operator following the End If keyword combination, without dependence of what block was executed earlier, *statements1* or *statements2*.

Let block *statementsN* be executed after checking not one but several conditions. In this case, we use the following If…Then construct:

```
If condition1 Then
    statements1
ElseIf condition2 Then
    statements2
    .  .  .  .
ElseIf conditionN Then
    statementsN
[Else
    statements]
End If
```

If the first condition is true (that is, logical expression `condition1` accepts `True`), then block `statements1` is executed. If this condition is false, the second condition (which is behind keyword `ElseIf`) is checked. If this condition is true (`condition2` = `True`), then block `statements2` is executed, and so on. If `conditionN` = `True`, then block `statementsN` is executed. If none of the *N* conditions is true, then block `statements` (located below the `Else` keyword) is executed.

Further, the jump is performed to the operator following the `End If` keyword combination, without dependence of what block was executed earlier, `statements1`, `statements2`, ..., `statementsN` or `statements`.

An example of using the last construct is in codes Listings 5.7 and 5.8 of Section 5.11.

We reviewed several versions of the conditional operator. In addition, let us consider function

```
IIf(condition, expression1, expression2)
```

Arguments of this function have the following sense:

- `condition` is a logical expression;
- `expression1`, `expression2` are arithmetic or logical expressions (Sections 1.8 and 1.10) or strings, which can be considered as expressions (Section 1.19).

Depending on the value of `condition` (`True` or `False`), the `IIf` function returns into the program the value of `expression1` or `expression2`, respectively.

The program below is an example of the function usage.

```
Sub IT4()
    Dim intA As Integer, strA As String
    intA = 6
    strA = IIf(intA Mod 2 = 0, "Even", "Odd")
End Sub
```

The second decision-making construct, `Select Case`, is called ***the case operator***. The syntax of this construct is as follows:

```
Select Case expression
    Case value1
        statements1
    Case value2
```

```
        statements2
        .  .  .  .
    Case valueN
        statementsN
    [Case Else
        statements]
End Select
```

where *expression* is an arithmetic or logical expression or string, *value1*, *value2*, ..., *valueN* are given numbers, Boolean values or strings. The case operator is performed as follows.

If *expression* accepts the value of *value1*, block *statements1* is executed. If *expression* accepts the value of *value2*, block *statements2* is executed, and so on. If *expression* is not equal to any of the *N* values (*value1*, *value2*, ..., *valueN*), then block *statements* (located below the `Case Else` keyword combination) is executed.

Further, the jump is performed to the operator following the case operator, without dependence of what block was executed earlier, *statements1*, *statements2*, ..., *statementsN* or *statements*.

When the same block must be executed at several values of *expression*, we have to enumerate these values (through a comma) behind the `Case` keyword.

As an example, let us consider the following program:

```
Sub Choice()
    Dim x As Integer
    x = 1
    Select Case 2 * x + 1
        Case 1
            x = x + 1
        Case 2, 3, 4
            x = 10
        Case Else
            x = 20
    End Select
    x = x Mod 3
End Sub
```

We advise the reader to execute this program step-by-step and to explain why the value of `x` changes so, instead of differently.

1.13. Cycles

To execute repeatedly an operator block, we can use one of three cycle operators, `For…Next`, `While…Wend` and `Do…Loop`.

The For…Next cycle is used when the number of the block's executions is known in advance, i.e., before the first execution of this block. This construct has the following syntax:

```
For counter = beginning To ending [Step growth]
    statements
Next [counter]
```

where *`counter`* is a variable of numerical data type (p. 24), *`beginning`* and *`ending`* are the boundaries of the *`counter`* change, *`growth`* is the step of this change; *`beginning`*, *`ending`* and *`growth`* are the cycle parameters.

Let us consider the `For…Next` cycle at a positive value of *`growth`*.

At first, the value of *`beginning`* is assigned to the *`counter`* variable. Further, condition *`counter`* > *`ending`* is checked. If the result is `True`, the cycle is completed, at that, block *`statements`* is not executed even once.

If the result of checking condition *`counter`* > *`ending`* is equal to `False`, then block *`statements`* is executed for the first time. After that, the jump occurs to the cycle beginning. Further, the *`counter`* variable's value increases by *`growth`*, and condition *`counter`* > *`ending`* is checked again. If the result is equal to `False`, then block *`statements`* is executed for the second time, and so on.

The cycle is completed when the check of condition *`counter`* > *`ending`* gives `True`. In this case, the operator following the cycle is executed.

As an example of the cycle usage, let us consider the following program for calculating the factorial of number 6:

```
Sub Factorial1()
    Dim I As Byte
    Dim F As Long
    F = 1
    For I = 1 To 6 Step 1
        F = F * I
    Next I
End Sub
```

According to handbook [3], factorial of natural number *n* (*n*! is the designation) is the product of the positive integers from 1 to *n*: $n! = 1 \cdot 2 \cdot ... \cdot n$ $(1! = 1)$.

We advise the reader to do the following:

1) execute the `Factorial1` program step-by-step, watching the `F` variable’s value;

2) replace `6` (parameter *`ending`*) by `13` in the cycle operator;

3) run the resulting `Factorial1` program for calculating 13!;

4) explain the reason of the stop with information *Run-time error ‘6’: Overflow*, using the description of the `Long` data type in Appendix 1.

At a negative value of the *`growth`* parameter, the `For…Next` cycle works as at a positive value, but:

- condition *`counter`* < *`ending`* is being checked;
- the *`counter`* variable cannot be of the `Byte` data type.

The following 2nd version of the program for calculating 6! is an example of using a negative value of the *`growth`* parameter.

```
Sub Factorial2()
    Dim I As Integer
    Dim F As Long
    F = 1
    For I = 6 To 1 Step -1
        F = F * I
    Next I
End Sub
```

If the `Step` keyword is omitted, the step of the *`counter`* change is equal to unity by default.

Arithmetic expressions may be used as the cycle parameters (*`beginning`*, *`ending`* and *`growth`*). It is important that all variables in these arithmetic expressions had numerical values before the `For…Next` cycle work.

As an example of such usage of arithmetic expressions, let us consider the following 3rd version of the program for calculating 6!:

```
Sub Factorial3()
    Const e As Double = 2.718281828
    Dim J As Byte
    Dim N As Byte
    Dim F As Long
    N = Round(e)
    F = 1
    For J = 1 To N ^ 2 - 3
        F = F * J
    Next J
End Sub
```

In this program, arithmetic expression `N ^ 2 – 3` is used as parameter *`ending`*. The `N` variable is equal to `3`, and `N ^ 2 – 3` is equal to `6`.

The absence of the `Step` keyword says that the step of the `J` variable change is equal to `1`, and `J` changes from 1 to 6.

Frequently it is required to leave the cycle before completion of its execution. In this case, the `For…Next` cycle has the following syntax:

```
For counter = beginning To ending [Step growth]
    [statements1]
    If condition Then Exit For
    [statements2]
Next [counter]
```

The `Exit For` operator is used for immediate exit from the cycle. It is a part of the simplest conditional operator. The last cycle works as follows.

For each value of the *`counter`* variable, after executing the *`statements1`* block, the computer calculates the value of logical expression *`condition`*. If this value is `False`, the cycle continues to work. Otherwise, the jump is performed to the operator following the cycle construct (without executing block *`statements2`*).

As an example of using the `Exit For` operator, let us consider the following 4th version of the program for calculating 6!:

```
Sub Factorial4()
    Dim I As Byte
    Dim F As Long
    F = 1
    For I = 1 To 13
        F = F * I
        If I = 6 Then Exit For
    Next I
End Sub
```

The While…Wend cycle is used when the number of the block's executions is not known in advance. The syntax of this cycle follows:

```
While condition
    statements
Wend
```

The `While…Wend` cycle work begins with calculating the value of logical expression *condition*. If *condition* = `False`, the cycle is completed, i.e., the jump is performed to the operator following the `Wend` keyword. If *condition* = `True`, block *statements* is executed. After that, the value of logical expression *condition* is calculated again, and so on.

The 5th version of the program for calculating 6! follows:

```
Sub Factorial5()
    Dim I As Byte
    Dim F As Long
    F = 1
    I = 1
    While I <= 6
        F = F * I
        I = I + 1
    Wend
End Sub
```

The Do…Loop cycle, as well as the `While…Wend` cycle, is used when the number of the block's executions is not known in advance. Four versions of this construct exist.

The first version is the `Do While…Loop` cycle with the following syntax:

```
Do While condition
    statements
Loop
```

The `Do While…Loop` cycle work begins with calculating the value of logical expression *condition*. If *condition* = `False`, the cycle is completed, i.e., the jump is performed to the operator following the `Loop` keyword. If *condition* = `True`, block *statements* is executed. After that, the value of logical expression *condition* is calculated again, and so on.

The `Do While…Loop` cycle is equivalent to the `While…Wend` cycle reviewed above.

The 6th version of the program for calculating 6! follows:

```
Sub Factorial6()
    Dim I As Byte
    Dim F As Long
    F = 1
    I = 1
    Do While I <= 6
        F = F * I
        I = I + 1
    Loop
End Sub
```

One more example of using the `Do While…Loop` cycle is the following program for the movement along the *x* axis:

```
Sub Steps1()
    Dim x As Single
    Dim h As Single
    h = 0.5                 'step equals 0.5
    x = 44                  'initial value of x equals 44
    Do While x < 55         'final value of x equals 55
        x = x + h           'value of x increases by h
    Loop
End Sub
```

We advise the reader to make the step-by-step execution of the `Steps1` program, watching the `x` variable change.

The `Do Until…Loop` cycle, which is the second version of the `Do…Loop` construct, has the following syntax:

```
Do Until condition
    statements
Loop
```

The `Do Until…Loop` cycle work begins with calculating the value of logical expression *`condition`*. If *`condition`* = `True`, the cycle is completed, i.e., the jump is performed to the operator following the `Loop` keyword. If *`condition`* = `False`, block *`statements`* is executed. After that, the value of logical expression *`condition`* is calculated again, and so on.

The program with the `Do Until…Loop` cycle for the movement along the *x* axis has the following form:

```
Sub Steps2()
    Dim x As Single
    Dim h As Single
    h = 0.5              'step equals 0.5
    x = 44               'initial value of x equals 44
    Do Until x >= 55     'final value of x equals 55
        x = x + h        'value of x increases by h
    Loop
End Sub
```

There is a situation when, during the work of the `Do While…Loop` and `Do Until…Loop` cycles, block *`statements`* is not executed even once because the condition of completing the cycle is checked before the block execution. Sometimes it is inconvenient.

The `Do…Loop While` cycle, which is the third version of the `Do…Loop` construct, has the following syntax:

```
Do
    statements
Loop While condition
```

The `Do…Loop While` cycle work begins with executing operator block *`statements`*. After that, the value of logical expression *`condition`* is calculated. If *`condition`* = `False`, the cycle is completed. Otherwise, block *`statements`* is executed again, and so on.

The program with the `Do…Loop While` cycle for the movement along the *x* axis has the following form:

```
Sub Steps3()
    Dim x As Single
    Dim h As Single
    h = 0.5              'step equals 0.5
    x = 44               'initial value of x equals 44
    Do
        x = x + h        'value of x increases by h
    Loop While x < 55    'final value of x equals 55
End Sub
```

The `Do…Loop Until` cycle, which is the fourth version of the `Do…Loop` construct, has the following syntax:

```
Do
    statements
Loop Until condition
```

The `Do…Loop Until` cycle work begins with executing operator block *`statements`*. After that, the value of logical expression *`condition`* is calculated. If *`condition`* = `True`, the cycle is completed. Otherwise, block *`statements`* is executed again, and so on.

The program with the `Do…Loop Until` cycle for the movement along the *x* axis has the following form:

```
Sub Steps4()
    Dim x As Single
    Dim h As Single
    h = 0.5                  'step equals 0.5
    x = 44                   'initial value of x equals 44
    Do
        x = x + h            'value of x increases by h
    Loop Until x >= 55       'final value of x equals 55
End Sub
```

During the work of the `Do…Loop While` and `Do…Loop Until` cycles, block *`statements`* is executed at least once because the condition of completing the cycle is checked after the block execution.

All four versions of the `Do…Loop` cycle can contain the `Exit Do` operator intended for immediate exit from the cycle. In the usage, this operator is similar to the `Exit For` operator of the `For…Next` cycle.

For the `While…Wend` cycle, there is no operator similar to the `Exit For` and `Exit Do` operators.

1.14. Manifestation of the error of real numbers' computer representation

In the last program of the previous section, we will reduce the value of h to one-fifth. We can expect that the number of repeated executions of operator x = x + h will increase fivefold as a result and the final value of x will remain equal to 55. However, it is not so: the final value of x is equal to 55.09983. In order to verify this assertion, we do the following:

1) enter program

```
Sub Steps5()
    Dim x As Single
    Dim h As Single
    h = 0.1                    'this operator distinguishes
                               'Steps5 from Steps4
    x = 44
    Do
        x = x + h
    Loop Until x >= 55
End Sub
```

into the code window;

2) install the breakpoint against the End Sub line;

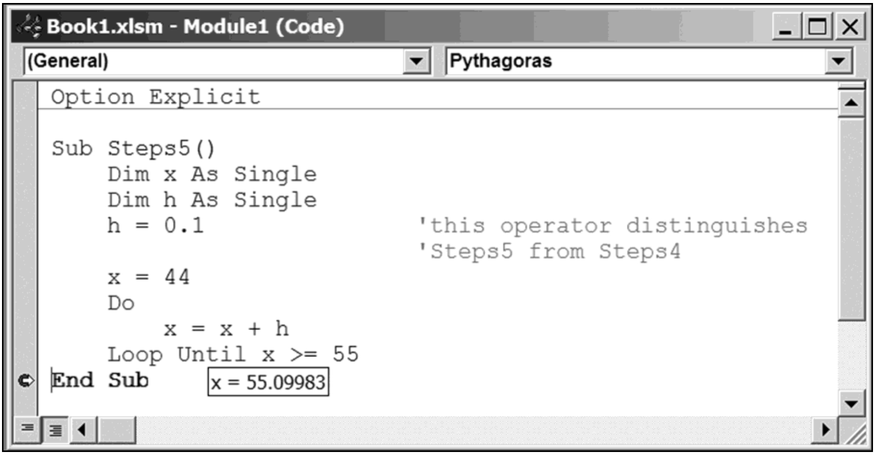
3) click on arrow ► for the program execution up to the breakpoint;

4) place the mouse pointer on the x variable in the program text; as a result, *x = 55.09983* appears (Fig. 1.9);

5) click on arrow ► for terminating the program execution.

We see "the phenomenon" of changing the final value of x after reducing the value of h because "almost all" real numbers in the computer are represented with an error.

In our case, the value of variable h of the Single data type is slightly less than 0.1. Therefore, there is the "superfluous" execution of the x = x + h operator during the cycle. This explains the difference between the observed value of 55.09983 and the expected value of 55.



```
Option Explicit

Sub Steps5()
    Dim x As Single
    Dim h As Single
    h = 0.1                 'this operator distinguishes
                            'Steps5 from Steps4
    x = 44
    Do
        x = x + h
    Loop Until x >= 55
End Sub     x = 55.09983
```

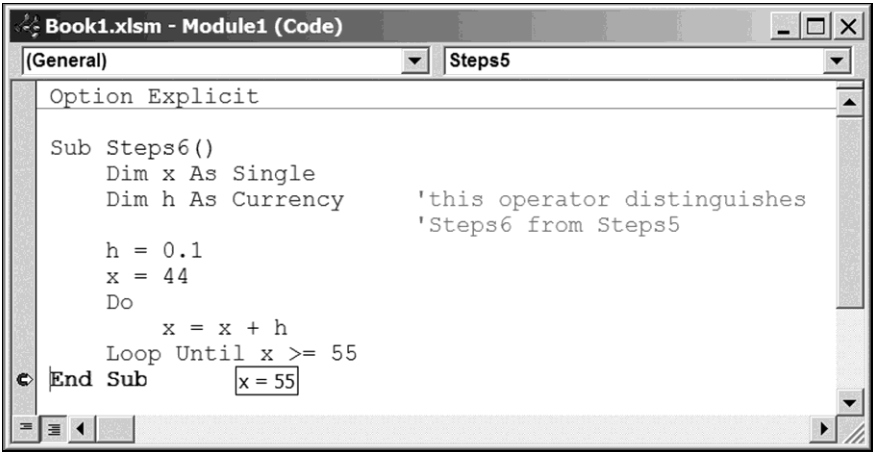
Fig. 1.9. The code window with the `Steps5` program and with the final value of `x`

Let us replace `Single` by `Currency` in the declaration of the `h` variable. The new version of the program follows:

```
Sub Steps6()
    Dim x As Single
    Dim h As Currency          'this operator distinguishes
                               'Steps6 from Steps5
    h = 0.1
    x = 44
    Do
        x = x + h
    Loop Until x >= 55
End Sub
```

The final value of `x`, calculated by means of the `Steps6` program, is equal to `55` (Fig. 1.10). This is because the value of variable `h` of the `Currency` data type is exactly equal to `0.1`, therefore, there is not the superfluous execution of the `x` = `x` + `h` operator during the cycle.

According to what has been said above, when working with real numbers, we have to consider possible inaccuracy of their computer representation because the error can lead to unexpected results.



```
Option Explicit

Sub Steps6()
    Dim x As Single
    Dim h As Currency      'this operator distinguishes
                           'Steps6 from Steps5
    h = 0.1
    x = 44
    Do
        x = x + h
    Loop Until x >= 55
End Sub        x = 55
```

Fig. 1.10. The code window with the `Steps6` program and with the final value of `x`

1.15. Arrays

An array is a sequence or table (two-dimensional, three-dimensional, and so on) with variables of the same data type, which are called elements of the array. Every reference to an element includes the array name and one index (the element number in the sequence) or several indices (coordinates of the element in the table). If there are two indices, they determine the coordinates of the element in the two-dimensional table, i.e., the numbers of row and column whose intersection is the element.

Before usage, the array must be declared. Besides, we have to specify the lower and upper boundaries of each index's change.

There are two kinds of arrays — static and dynamic. For each of them, we will use the following scheme: we will consider the one-dimensional array (with one index) followed by the multidimensional array (with several indices) together with the one-dimensional array.

The static array is used when the quantity of its elements is known in advance. In the declaration of a static one-dimensional array, for the index, we have to specify the lower and upper boundaries defining the quantity of elements, and these boundaries cannot be changed during the program execution.

Static arrays are declared as variables, i.e., by means of keywords `Dim` and `As`. The boundaries are integers in parentheses. The `To` keyword must be between the lower and upper boundaries. Examples of the declaration follow:

```
Dim arrB(1 To 10) As Currency
Dim A(-10 To 10) As String
```

If a single integer is given in the parentheses, it is the upper boundary. In this case, the lower boundary is equal to zero by default. For example, operator

```
Dim arrA(9) As Byte
```

is equivalent to operator

```
Dim arrA(0 To 9) As Byte
```

If the lower boundary of indices must be equal to unity by default, we have to type

```
Option Base 1
```

above the first line of the program. In this case, operator

```
Dim arrA(9) As Byte
```

is equivalent to operator

```
Dim arrA(1 To 9) As Byte
```

Further, we will consider that line `Option Base 1` is absent.

The values of the boundaries should be between the limits for the `Long` data type (Appendix 1), i.e., from -2147483648 to 2147483647.

Let us consider the following program:

```
Sub StaticArrays()
    Dim B1(1 To 6) As Byte, S1 As Byte
    Dim B2(1 To 6) As Currency, S2 As Currency
    Dim B3(1 To 6) As Byte, S3 As Byte
    Dim I As Byte
'Determination of first five elements of arrays:
    For I = 1 To 5                      'first cycle
        B1(I) = I                       'first power of I
        B2(I) = I ^ .333                'cubic root of I
        B3(I) = I ^ 3                   'I cubed
    Next
'Determination of sixth elements of arrays:
    S1 = 0
    S2 = 0
    S3 = 0
    For I = 1 To 5                      'second cycle
        S1 = S1 + B1(I)
        S2 = S2 + B2(I)
        S3 = S3 + B3(I)
    Next
    B1(6) = S1
    B2(6) = S2
    B3(6) = S3
End Sub
```

In this program:

- the first cycle is used for determination of the first five elements of arrays B1, B2 and B3; upon completing the cycle, these elements of the arrays contain the numerical values, which are equal to the 1st, 1st/3rd and 3rd powers of I: B1(1) = 1, B1(2) = 2, ..., B1(5) = 5; B2(1) = 1, B2(2) = 1.2596, ..., B2(5) = 1.7091; B3(1) = 1, B3(2) = 8, ..., B3(5) = 125;
- the second cycle is used for summation of the earlier determined elements of the arrays; upon completing the cycle, the sums of the first five elements of arrays B1, B2 and B3 are respectively equal to S1 = 15, S2 = 6.9971 and S3 = 225;
- the remaining operators of the program assign the calculated values of the sums to the sixth elements of the arrays: B1(6) = 15, B2(6) = 6.9971 and B3(6) = 225.

It was noted above that one-dimensional and multidimensional arrays exist. We reviewed one-dimensional arrays, which are similar to rows and columns on the Excel worksheet and to vectors in mathematics.

For declaration of multidimensional arrays (with several indices), we use a construct similar to operator Dim for one-dimensional arrays. The difference consists in that several boundaries are given through a comma.

For example, operators

```
Dim A(4, 6) As Byte
Dim B(1 To 5, -7 To -1) As Byte
```

declare two-dimensional arrays A and B, which contain identical quantities of elements. This quantity is equal to 5 × 7 = 35.

A two-dimensional array is similar to a range of cells on the Excel worksheet and to a matrix in mathematics. A three-dimensional array is similar to a range of cells on several worksheets of the same Excel workbook. A four-dimensional array is similar to a range of cells on several worksheets of several open workbooks.

Open Excel workbooks are represented by buttons on the taskbar of Windows Desktop.

The reference to an element of a multidimensional array includes the array name and the indices listed through a comma. Examples of the reference are figured in the following assignment operators:

```
A(i, j + 1) = 17
D(K) = A(i, 0)
```

As we see, the reference may be on the left and/or right of the assignment sign.

Operator

```
Dim C(1 To 5, -5 To -1, 4) As Byte
```

declares a three-dimensional array containing 5 × 5 × 5 = 125 elements.

The number of indices is called the dimension of an array. The dimension of the above `C` array is equal to 3. The largest dimension equals 60.

An array occupies S × Q + 4 × D + 20 bytes of the main memory, where:

- S is the memory size (in bytes) occupied by one element;
- Q is the quantity of the elements;
- D is the dimension.

It concerns both static and dynamic arrays. The last array is reviewed below.

The dynamic array is used when the quantity of its elements is not known in advance and must be defined during the program execution. When finishing work with the dynamic array, it is possible to free the memory cells occupied by this array. It is important for problems demanding large size of the main memory.

The declaration of the dynamic array has the following two parts.

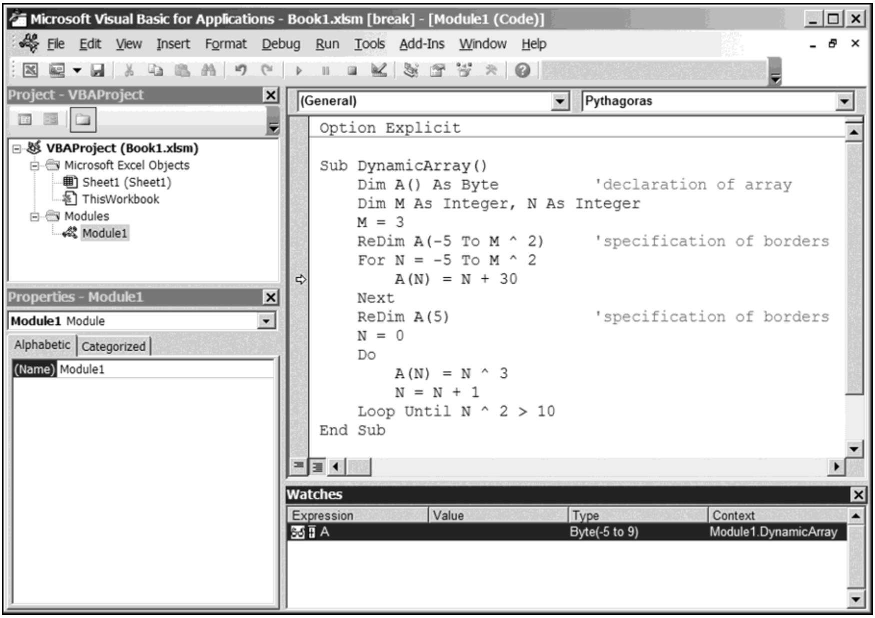
1. The array is declared by means of the `Dim` operator without boundaries of indices. A pair of parentheses must follow the array name.
2. The boundaries of indices are specified by means of the `ReDim` operator in a proper place of the program, at that, we can use not only integers as boundaries, but also arithmetic expressions. It is important that all variables in these expressions have numerical values before the `ReDim` operator execution.

The following program is an example of using the dynamic array.

```
Sub DynamicArray()
    Dim A() As Byte        'declaration of array
    Dim M As Integer, N As Integer
    M = 3
    ReDim A(-5 To M ^ 2)   'specification of boundaries
    For N = -5 To M ^ 2
        A(N) = N + 30
    Next
    ReDim A(5)             'specification of boundaries
    N = 0
    Do
        A(N) = N ^ 3
        N = N + 1
    Loop Until N ^ 2 > 10
End Sub
```

After typing this program in the code window, let us do the following.

1. By clicking, set the blinking cursor on variable A, more precisely, in front of or behind A. Fulfill operations *Debug* > *Add Watch*. The *Add Watch* window, containing the variable name in text box *Expression*, appears.
2. Click on *OK*. The *Watches* window, intended for the current visualization, appears with a line corresponding to the A array.
3. Set the blinking cursor in any place of the program text.
4. Execute the program step-by-step (Fig. 1.11), watching the A array values by means of the *Watches* window.



```
Option Explicit

Sub DynamicArray()
    Dim A() As Byte          'declaration of array
    Dim M As Integer, N As Integer
    M = 3
    ReDim A(-5 To M ^ 2)     'specification of borders
    For N = -5 To M ^ 2
        A(N) = N + 30
    Next
    ReDim A(5)               'specification of borders
    N = 0
    Do
        A(N) = N ^ 3
        N = N + 1
    Loop Until N ^ 2 > 10
End Sub
```

Expression	Value	Type	Context
A		Byte(-5 to 9)	Module1.DynamicArray

Fig. 1.11. The step-by-step execution of the DynamicArray program

For visualization of the A array values in the *Watches* window, we must click on the plus sign in front of A in this window. As a result, plus turns to minus and the A array values appear below.

We can edit the contents of the *Watches* window.

Let us admit that we want to watch the value of variable n instead of the A array values. For this purpose, during the step-by-step program execution (between presses of the *F8* key), we must:

1) click on the minus sign in front of A; as a result, the plus sign appears but the A array values disappear;

2) highlight A by clicking on it (in the *Watches* window);

3) type n instead of A highlighted;

4) press the *Enter* key;

5) continue the step-by-step program execution, watching the value of n.

To remove the line, corresponding to the A array, from the *Watches* window, we must:

1) highlight this line by clicking on the glasses pictogram;

2) press the *Delete* key.

To add a line, we must fulfill *Debug* > *Add Watch*, and so on.

To close the *Watches* window, we must click on the little cross in the top right corner of this window. To open the *Watches* window, we must fulfill *View* > *Watch Window*.

The ReDim operator can be used for changing the array's dimension, as in the following program:

```
Sub Dimension()
    Dim arrA() As Byte
    ReDim arrA(1, 1)          'two-dimensional array
    arrA(0, 0) = 13: arrA(0, 1) = 14
    arrA(1, 0) = 15: arrA(1, 1) = 16
    ReDim arrA(1 To 3, 3, 3)  'three-dimensional array
    arrA(1, 0, 0) = 17
End Sub
```

Note that, at repeated execution of the ReDim operator, the array values will be lost because the ReDim operator nulls all elements of the array.

To keep the array values, we have to insert the Preserve keyword between ReDim and the array name. As an example, see operators 0, 1 and 2 in the following program:

```
Sub Conservation1()
    Dim J As Integer
0:  Dim arrA() As Integer        'declaration of array
1:  ReDim arrA(-5 To 1)   'specification of boundaries
    For J = -5 To 1
        arrA(J) = J ^ 2
    Next J
2:  ReDim Preserve arrA(-5 To 4)
                          'specification of boundaries
    For J = 2 To 4
        arrA(J) = J ^ 3
    Next J
End Sub
```

Labels `0`, `1` and `2` may be omitted.

We advise the reader to execute the `Conservation1` program step-by-step, watching the `arrA` array values by means of the *Watches* window.

In the `Conservation1` program, the `Preserve` keyword is used to change the upper boundary of the one-dimensional array's index without nulling the array values. When changing the lower boundary, we cannot use this keyword.

In the case of a multidimensional array, the `Preserve` keyword can be used only when changing the upper boundary of the last index. As an example of such usage of the `Preserve` keyword, see operators `0`, `1` and `2` in the following program:

```
Sub Conservation2()
    Dim I As Integer, J As Integer
0:  Dim arrA() As Integer        'declaration of array
1:  ReDim arrA(2, -5 To 1)
                                'specification of boundaries
    For I = 0 To 2
        For J = -5 To 1
            arrA(I, J) = (I + 1) * J ^ 2
        Next J
    Next I
2:  ReDim Preserve arrA(2,-5 To 4)
                                'specification of boundaries
    For I = 0 To 2
        For J = 2 To 4
            arrA(I, J) = (I + 1) * J ^ 3
        Next J
    Next I
End Sub
```

We advise the reader to make the step-by-step execution of the last program, watching the `arrA` array values by means of the *Watches* window.

The values of the lower and upper boundaries of any index can be returned to the program. For this purpose, functions `LBound` and `UBound` are respectively used. We can look the description of these functions in the Excel help system started by pressing the *F1* key when the VB window is active.

If arrA is a one-dimensional array, the LBound and UBound functions are used as in the operator block below:

```
Dim lower As Long, upper As Long
lower = LBound(arrA)          'lower boundary of index
upper = UBound(arrA)          'upper boundary of index
```

Function UBound is necessary, for example, when the value of the upper boundary is unknown and, at the same time, we have to increase this value by certain number.

As it was already told, advantage of the dynamic array over the static is that we can free the memory cells, earlier occupied by the dynamic array. Operator Erase is used for this, as in the following example program:

```
Sub Memory()
    Dim A() As Byte
    Dim B() As Byte
    ReDim A(8)  'memory for A: 9 + 4 + 20 = 33 bytes
    Erase A     'memory for A: 0 bytes
    ReDim B(2, 3)
                'memory for B: 12 + 8 + 20 = 40 bytes
End Sub
```

After inputting the Memory program into the code window, let us make the following:

1) generate the lines corresponding to arrays A and B in the *Watches* window, fulfilling *Debug* > *Add Watch* twice;

2) click on any place of the program text to set the blinking cursor there;

3) execute the program step-by-step, watching the memory distribution by means of the *Watches* window.

For solving a series of mathematical problems, arrays of random numbers are required. They can be generated by means of the Rnd function (p. 46).

Let us consider the following program:

```
Sub RandomNumbers()
    Dim N As Long
    Dim I As Long
    Dim S() As Single
    N = 20
    ReDim S(1 To N)
    Randomize          'it must be before calls of Rnd
    For I = 1 To N
        S(I) = Rnd
    Next I
End Sub
```

This program calculates 20 random real numbers from 0 to 1 and writes them into dynamic array `S`. The `Rnd` function is being called in cycle `For…Next`.

Function `Rnd` is the built-in generator of random real numbers uniformly distributed on segment $[0, 1]$: a number, being returned by the `Rnd` function, appears in any place of segment $[0, 1]$ with equal probability.

Before a series of the `Rnd` function calls, the `Randomize` operator must be executed for preparing the random-number generator for work.

We advise the reader to execute the `RandomNumbers` program step-by-step, watching the `S` array by means of the *Watches* window.

Operator `Randomize` and function `Rnd` are used in code Listing 6.11 intended for minimization of the multimodal function (Section 6.8).

1.16. User-defined procedures

A user-defined procedure allows executing an operator block in different places of our program.

According to this VB construct:

1) we have to write the block, intended for multiple executions, only once;

2) a name with formal parameters or without them must be written in the header of the block.

This expanded block is called a procedure declaration.

Generally, the user-defined procedure has:

- input parameters, which are considered given;
- output parameters, which are determined while executing the block.

After forming the procedure declaration, we locate the calls of this procedure with the actual parameters in those places of the program, which should be occupied by the block being the procedure prototype.

The procedures are divided into functions and subroutines (subprograms). The function, returning a value, is intended for usage in arithmetic and logical expressions and in strings. The subroutine usage in expressions and strings is impossible. This is the main difference between the function and subroutine.

The declaration of ***the user-defined function*** has the following syntax:

```
Function name([formal_parameters]) [As type]
    statements
End Function
```

where *name* is the function name, *type* is the data type (Appendix 1) of *name*, i.e., of the function value, *formal_parameters* are the parameter (argument) names listed through a comma, *statements* is the operator block. The parameter names can be accompanied by keywords.

Block *statements* must include at least one assignment operator whose left-hand side (on the left of sign =) is the function name.

The function call looks like

```
name([actual_parameters])
```

where *actual_parameters* are variables, arrays, expressions (arithmetic, logical) and/or strings listed through a comma.

As a result of calling the *name* function, the function value, corresponding to *actual_parameters*, is returned into the program.

Records and arrays of records may be among formal and actual parameters. We will review the record construct in Section 1.18.

Let us consider the following code:

```
Sub Program1()
    Dim L As Long
    Dim W As Double
    L = Fact(12)
    W = 4.2 + Fact(10) / 2
End Sub

Function Fact(N) As Long                              'N < 13
    Dim I As Byte
    Dim J As Long
    J = 1
    For I = 1 To N
        J = J * I
    Next I
    Fact = J
End Function
```

The first group of operators is program `Program1`; the second group is the declaration of function `Fact`. They are in one or different modules of the same Excel workbook, i.e., we can type the program and the function declaration in one or different code windows.

The `Fact` function calculates *N*!, i.e., factorial of natural number `N`, which is the formal parameter.

We see two calls of the `Fact` function in the program, with `12` and `10` as the actual parameter.

1. The `Fact` function call is located in the right-hand side of assignment operator

```
L = Fact(12)
```

As a result of the operator execution, the `Fact` function value (that is, the value returned by the `Fact` function into the program when `N` is equal to `12`) is assigned to the `L` variable.

2. The `Fact` function call is figured in arithmetic expression

```
4.2 + Fact(10) / 2
```

The value of this arithmetic expression is assigned to the `W` variable.

The declaration of ***the user-defined subroutine*** has the following syntax:

```
Sub name([formal_parameters])
    statements
End Sub
```

where *name* is the subroutine name, *formal_parameters* are the parameter names listed through a comma, as in the above declaration of the user-defined function, *statements* is the operator block.

Keyword `Sub` occurs from word “subroutine”. It is in the beginning and end not only of the subroutine declaration (or simply of the subroutine), but also of the program because “main subroutine” is synonym of “program”.

There are two equivalent operators of calling the subroutine:

```
Call name([actual_parameters])
name [actual_parameters]
```

where *actual_parameters* is the list of actual parameters, as in the above call of the user-defined function. In the presence of the `Call` keyword, the actual parameters are in parentheses; in the absence of `Call`, the parentheses are not used.

The example program and subroutine follow:

```
Sub Program2()
    Dim aa As Single, bb As Single
    Dim cc1 As Single, cc2 As Single, cc3 As Single
    aa = 3
    bb = 4
    Call Hypotenuse(aa, bb, cc1)
                           '1st call of subroutine
    Call Hypotenuse(3, 4, cc2)
                           '2nd call of subroutine
    Hypotenuse aa, bb, cc3
                           '3rd call of subroutine
End Sub
```

```
Sub Hypotenuse(ByVal A, ByVal B, ByRef C)
    C = Sqr(A ^ 2 + B ^ 2)
End Sub
```

The first group of operators is program Program2, the second group is the declaration of subroutine Hypotenuse for calculating the length of the hypotenuse of a right-angled triangle. They are located in one or different modules.

We see three operators of calling the Hypotenuse subroutine in program Program2, and two of them contain the Call keyword.

Formal parameters A and B (in the subroutine declaration) are the input parameters, lengths of the legs. The ByVal keyword in front of A and B in the first line of the subroutine declaration (i.e., in the header of the operator block) means that these parameters must be passed by value (when calling Hypotenuse). In this case, the values of parameters A and B (3 and 4, respectively) are transferred to the Hypotenuse subroutine at all three calls.

Formal parameter C is the output parameter, i.e., the hypotenuse length. The ByRef keyword in front of C means that this parameter must be passed by reference (when calling Hypotenuse). In this case, the address of the memory cell, corresponding to variable cc1 (at the first call), cc2 (at the second call) or cc3 (at the third call), is transferred to the Hypotenuse subroutine.

Keyword ByRef may be omitted.

In the code under consideration, the parameters of the Hypotenuse subroutine are simple variables (not arrays). If the parameters are arrays, both input and output parameters must be passed by reference.

Let us consider the following code located in one module:

```
Dim N1 As Integer

Sub Program3()
    Dim xx(50) As Double
    Dim yy(50) As Double
    Dim i As Integer
    N1 = 3
    For i = N1 To 30
        xx(i) = 0.1 * i
    Next i
    Call XSINX(30, xx, yy)          'call of subroutine
End Sub

Sub XSINX(ByVal N2, ByRef X() As Double, _
    ByRef F() As Double)
    Dim j As Integer
    For j = N1 To N2
        F(j) = X(j) * Sin(X(j))
    Next j
End Sub
```

In addition to usage of arrays as the subroutine parameters, we see something new in the last code: the N1 variable is declared above the first line of the program. This is done for making N1 visible (in respect of the usage possibility) in both the program and subroutine. Let us return to the variable declaration.

So far, we have been speaking about how to declare variables, but have not mentioned where to declare. We can declare them in two places:

- inside the program or user-defined procedure;
- in the general declarations area occupying the top part of the code window.

The place of the variable declaration defines the area of the variable usage. For example, if a variable is declared in the user-defined procedure (as variable j in the last code), only this procedure "sees" this variable. Other procedures (if they exist) and the program cannot use this variable's value and change it.

Such variable is called ***a local variable***. We can also say that the variable is visible at the procedure level.

For letting a variable's value be accessible to all user-defined procedures of the given module, we have to declare this variable in the general declarations area (as variable N1 in the last code). In this case, the program and all user-defined procedures, declared in the given module, can use this variable's value and change it.

Such variable is called ***a module variable***. The Dim keyword in front of N1 may be replaced by the Private keyword:

```
Private N1 As Integer
```

All that was said about variables also concerns the user-defined constants, but the constant's value, naturally, cannot be changed.

If the declaration of the XSINX subroutine is located in the separate module, the first line of the last code should be as follows:

```
Public N1 As Integer
```

Such variable is called ***a public variable***. The declaration of public variable N1 can be in each of two modules or only in one, the first (with text of Program3) or second (with text of XSINX).

At the similar declaration of a constant, instead of a variable, the corresponding line of the general declarations area should begin with keyword combination `Public Const`.

The debugger command, being fulfilled by pressing the *F8* key, is called *Step Into*: during the step-by-step program execution by means of the *F8* key, the entrance into the user-defined procedure takes place. If we do not need the step-by-step execution inside the user-defined procedure, we use the following two commands of the *Debug* menu.

1. *Step Over* — the step-by-step program execution without entrance into the user-defined procedure. This command can also be performed by pressing *Shift* + *F8*.
2. *Step Out* — exit from the user-defined procedure. It is used when the procedure remainder should be executed in the automatic mode. This command can also be performed by pressing *Ctrl* + *Shift* + *F8*.

We advise the reader to execute programs `Program1`, `Program2` and `Program3` by pressing *F8, Ctrl* + *F8, Shift* + *F8* and *Ctrl* + *Shift* + *F8*. Before the execution, the blinking cursor must be located inside the program text (not in the procedure declaration and not in the general declarations area).

One or several last parameters of the list of parameters of the user-defined procedure (function or subroutine) can be optional, i.e., they can be absent in the procedure call.

If the parameter is not obligatory, the `Optional` keyword must be in front of this parameter's name in the first line of the procedure declaration. The optional parameter must have the `Variant` data type (Appendix 1).

Let us consider the following example code:

```
Sub Program4()
    Dim bytA As Byte, bytB As Byte
    Dim intC As Integer
    bytA = 5
    bytB = 10
    intC = Apt(bytA, bytB)
                          'result: bytB = 6, intC = 625
    bytB = 10
    intC = Apt(bytA)      'result: bytB = 10, intC = 625
End Sub

Function Apt(ByVal a, Optional b As Variant)
    If Not IsMissing(b) Then b = a + 1
    Apt = a ^ 4
End Function
```

In this example, the b parameter of the Apt function is optional; keyword Optional in front of this parameter tells about it. Keyword combination As Variant behind b may be omitted.

Other features of the Apt function declaration:

- Not is the logical negation;
- IsMissing is the following function.

The value of IsMissing(b) is equal to True or False as follows:

- True in the absence of the optional parameter in the Apt function call;
- False in the presence of the optional parameter.

We advise the reader to execute the Program4 program step-by-step, watching the values of variables bytB and intC.

We can specify a value of the optional parameter in the absence of this parameter in the call of the user-defined procedure.

The example program and procedure follow:

```
Sub Program5()
    Dim bytA As Byte, bytB As Byte, bytC As Byte
    bytA = 5: bytB = 2
    Call Ept(bytA, bytC, 5 * bytB)  'result: bytC = 15
    Call Ept(bytA, bytC)            'result: bytC = 8
End Sub

Sub Ept(ByVal a, c, Optional b = 3)
    c = a + b
End Sub
```

In this code:

- the b parameter of the Ept subroutine is optional;
- if the call of Ept contains only two actual parameters, b = 3 is used when executing operator c = a + b.

We advise the reader to make the step-by-step execution of the Program5 program, watching the value of bytC.

Operators

```
Exit Sub
Exit Function
```

are intended for immediate terminating the procedure execution. The first operator should be in the subroutine declaration, the second — in the function declaration. At their execution, the jump is being performed to the same point, as upon the normal terminating the procedure execution.

The programs in this section use only one user-defined procedure; for example, program Program1 uses only the Fact function. However, several user-defined procedures may be used by one program.

According to the computer terminology, as a program, we can consider the program itself (the main subroutine) together with the declarations of the user-defined procedures (used by the program) and/or together with the general declarations area. For example, program Program3 together with operator

```
Dim N1 As Integer
```

can also be called program Program3. The following can also be called program Program3: the program itself, the XSINX subroutine declaration and the general declarations area. Sometimes term “program” is used in the extended sense, which is equivalent to “code”.

1.17. Built-in procedures. Usage of standard windows

Visual Basic includes a considerable quantity of the built-in procedures, which differ from the user-defined procedures in the following: developers of the Visual Basic features programmed their declarations. These declarations are hidden from us as the program developer.

The built-in procedures, as well as the user-defined procedures, are divided into functions and subroutines.

We have already encountered the built-in functions of VB. They are:

- functions reviewed in Section 1.9;
- `IIf` in Section 1.12;
- `IsMissing` in Section 1.16, etc.

Below is considered built-in function `InputBox` intended for input of information (into the program) by means of the standard windows of operating system Windows.

An example of the built-in subroutine is the `MsgBox` procedure intended for output of information (from the program) into the standard windows.

We will use `InputBox` and `MsgBox` in the program from Section 1.1, intended for calculating the hypotenuse length, with which we began studying VB.

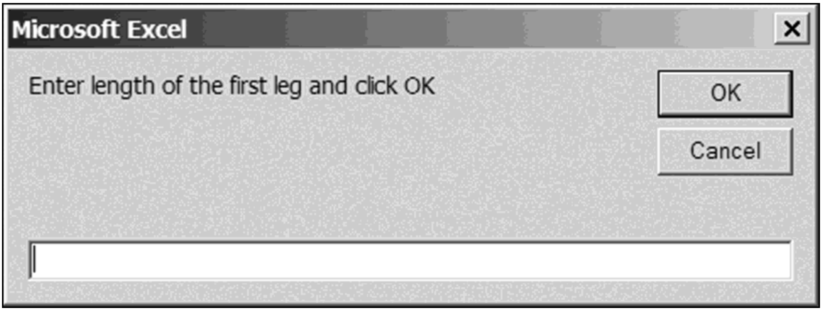
1. Let us enter the following program into the code window:

Listing 1.3

```
Sub Pythagoras1()
    Dim a As Single
    Dim b As Single
    Dim c As Single
    Dim s As String
1:  s = InputBox( _
        "Enter length of the first leg and click OK")
2:  a = Val(s)
3:  s = InputBox( _
        "Enter length of the second leg and click OK")
4:  b = Val(s)
5:  c = Sqr(a ^ 2 + b ^ 2)     'according to Pythagoras
6:  s = Str(c)
```

```
7:  MsgBox s
End Sub
```

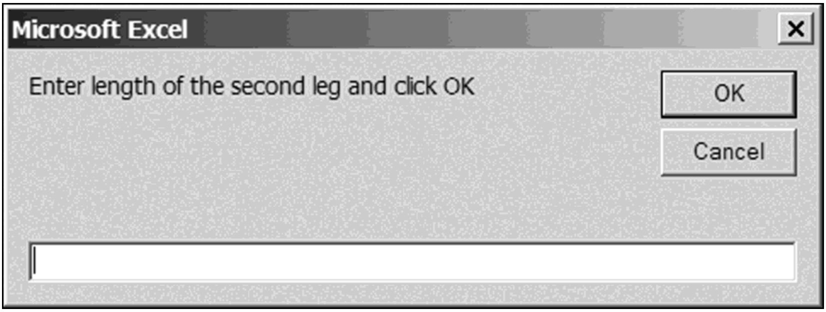
2. Let us run the `Pythagoras1` program. The window appears (Fig. 1.12), offering to input the length of the first leg of a right-angled triangle. Let us put, for example, 400 (without inverted commas) into the text box of this window by means of the keyboard and click on the *OK* button.


Microsoft Excel

Enter length of the first leg and click OK

Fig. 1.12. The first window (on the Excel worksheet)
for inputting the source data

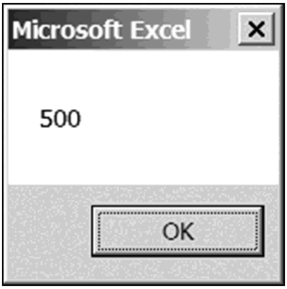
3. The window appears (Fig. 1.13), offering to input the length of the second leg. Let us put, for example, 300 and click on *OK*.


Microsoft Excel

Enter length of the second leg and click OK

Fig. 1.13. The second window for inputting the source data

4. The window appears (Fig. 1.14), containing the hypotenuse length calculated. Let us click on the *OK* button to close this window and terminate the program execution.



Microsoft Excel

500

Fig. 1.14. The window with the result

In the above program, operator 1 includes the InputBox function call, which is used for entering information from the keyboard. This function returns the string, which was entered into the text box of the window in Fig. 1.12, i.e., "400". This string is assigned to the s variable of the String data type.

Operator 2 converts the s string's value to number 400 and assigns this number to the a variable of the Single data type.

Operator 3 assigns string "300", which was entered into the text box of the window in Fig. 1.13, to the s variable. Operator 4 converts the s string's value to number 300 and assigns this number to the b variable.

Operator 5 calculates the hypotenuse length according to the Pythagoras theorem. The calculated value of 500 is assigned to variable c. Operator 6 converts 500 to string; as a result, the s string's value becomes "500".

Operator 7, which is the call of the built-in MsgBox subroutine, opens the window with value 500 calculated (Fig. 1.14). As in the case of the user-defined subroutine, operator 7 may be written in the following form:

```
Call MsgBox(s)
```

The calls of built-in procedures InputBox and MsgBox have only one parameter (of the String data type), which is obligatory. However, these procedures also have optional parameters, whose appointment may be looked in the Excel help system started by pressing the *F1* key when the VB window is active. Before pressing this key, we recommend to locate the blinking cursor on the required word (InputBox or MsgBox) in the code window containing program Pythagoras1.

In the Excel help system, the MsgBox procedure is termed as a function, instead of a subroutine, because the call of MsgBox may be a part of arithmetic expressions. In this case, MsgBox returns (into the program) the integer value depending on the button, on which the user clicked. In the example below, we will consider using the MsgBox procedure as a function.

For calculating the area of a right-angled triangle by the Pythagoras1 program, we replace operator 7 by three operators labeled by 7, 8 and 9. The following program is the result:

```
Sub Pythagoras2()
    Dim a As Single
    Dim b As Single
    Dim c As Single
    Dim s As String
1:  s = InputBox( _
        "Enter length of the first leg and click OK")
2:  a = Val(s)
3:  s = InputBox( _
        "Enter length of the second leg and click OK")
4:  b = Val(s)
5:  c = Sqr(a ^ 2 + b ^ 2)    'according to Pythagoras
6:  s = Str(c)
7:  Dim Ret As Integer
8:  Ret = MsgBox(s, vbYesNo, _
        "Do you want to calculate area?")
9:  If Ret = vbYes Then MsgBox Str(a * b / 2)
End Sub
```

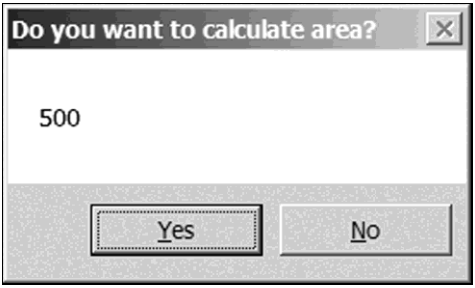
In operator 8, procedure MsgBox is a function. In this case, the procedure parameters are in parentheses.

When executing the Pythagoras2 program, the window with the hypotenuse length calculated has other content (Fig. 1.15). After clicking on the *Yes* button, the window with the triangle area calculated is displayed (Fig. 1.16). By clicking on the *OK* button, we terminate the program execution.

Operators 7 — 9 of the last program may be replaced by operator

```
If MsgBox(s, vbYesNo, _
    "Do you want to calculate area?") = vbYes _
    Then MsgBox Str(a * b / 2)
```

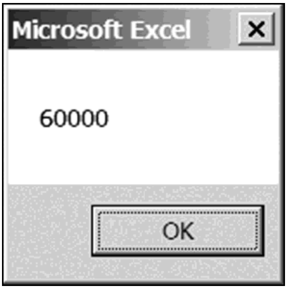
This also gives the results shown in Fig. 1.15 and 1.16.



Do you want to calculate area?

500

Fig. 1.15. The first resulting window



Microsoft Excel

60000

Fig. 1.16. The second resulting window

1.18. Records

A record or user-defined data type is a collection of variables, possibly of different data types, grouped together under a single name for convenient handling. Each variable of the record is called a field.

Before usage of a record, we must create it by means of the `Type` operator with the following syntax:

```
Type struct
    field1 As type1
    field2 As type2
    .   .   .   .   .
    fieldN As typeN
End Type
```

where *struct* is a name of the record, *field1*, *field2*, …, *fieldN* are the field names, *type1*, *type2*, …, *typeN* are data types (Appendix 1) for the corresponding fields.

The record creation operator must be placed in the general declarations area of the program, which uses this record.

For example, we have to work up results of a university session, where the following three subjects were evaluated: physics, mathematics and informatics. In this case, the following fields are necessary:

1) name of the student;
2) number of the test book;
3) mark in physics;
4) mark in mathematics;
5) mark in informatics.

Let the name of the record be `Session`, and let the field names be `Name`, `Number`, `Physics`, `Math` and `Inform`. In this case, the record creation operator has the following form:

```
Type Session
    Name As String
    Number As Long
    Physics As Byte
    Math As Byte
    Inform As Byte
End Type
```

To get access to the created record (by name *struct*), we have to declare one or several variables in the program by means of the Dim operator, in the same way as we declared variables in Section 1.3.

The declaration operator has the following syntax:

```
Dim variable As struct
```

where *variable* is the variable name, *struct* is the variable's data type.

For example, operator

```
Dim Sess As Session, BestSess As Session
```

declares variables (records) Sess and BestSess of the Session data type.

For the reference to the record field, we use the variable and field names separated by a point.

For example, assignment operator

```
Sess.Name = "Maksim Zakharkin"
```

contains the reference in the left-hand side. As a result of the operator execution, string

```
"Maksim Zakharkin"
```

is assigned to the Name field of the Sess variable (of type Session).

For filling the fields, it is convenient to use the With operator, which has the following syntax:

```
With variable
    .field1 = expression1
    .field2 = expression2
    .  .  .  .  .  .  .
    .fieldN = expressionN
End With
```

In this operator, *expression1*, *expression2*, ..., *expressionN* are arithmetic or logical expressions (Sections 1.8 and 1.10) or strings, which can be considered as expressions (Section 1.19).

For example:

```
With Sess
    .Name = "Maksim Zakharkin"
    .Number = 02237
    .Physics = 4
    .Math = 5
    .Inform = 5
End With
```

The assignment operator can be applied both to fields and to entire records, as in the following examples:

```
BestSess.Number = Sess.Number
BestSess = Sess
```

Arrays of records may be used. For example, it is natural to store the marks obtained by students of group E13 in array `SessE13`. If this array of records is static, it is declared as follows:

```
Dim SessE13(1 To 15) As Session
```

For example, let us consider the following code for calculating distance between Tushino and Ostankino on the Moscow map.

Listing 1.4

```
Type Point
    Name As String
    x As Single
    y As Single
End Type

Sub TushinoOstankino()
    Dim P() As Point
    ReDim P(0 To 1)
    With P(0)
        .Name = "Tushino"
        .x = 17.6
        .y = 29.7
    End With
    With P(1)
        .Name = "Ostankino"
        .x = 44.1
        .y = 37.5
    End With
    MsgBox Str(Distance(P()))      'output of 27.62408
End Sub

Function Distance(positions() As Point) As Single
    Distance = _
    Sqr((positions(0).x - positions(1).x) ^ 2 + _
    (positions(0).y - positions(1).y) ^ 2)
End Function
```

Code Listing 1.4 includes the operator creating the `Point` record, program `TushinoOstankino` and the `Distance` function. This code has the following peculiarities:

- `P` is the dynamic array of records of the `Point` data type;
- the `Distance` function argument is the array of records of type `Point`.

To understand the work of program `TushinoOstankino`, ***we advise the reader*** to execute it step-by-step, watching the `P` array in window *Watches*. Before the first press of the *F8* key, the blinking cursor should be located in the `TushinoOstankino` program, not in the operator creating the `Point` record and not in the `Distance` function.

1.19. Work with strings

The string is not only a pair of quotation marks `""` and a sequence of characters enclosed in quotes (p. 27), but also a variable of the `String` data type, declared by means of the `String` keyword. For example, in operator block

```
Dim A As String
Dim B As String * 15
A = "Informatics"
B = "Informatics"
```

we see the following strings: `A`, `B`, `"Informatics"`.

The string as a variable may have inconstant or constant length (Appendix 1).

- The variable-length string occupies a part of the main memory, which can change during the program execution. In the above example, `A` is a string of variable length. According to Appendix 1, after performing assignment operator

```
A = "Informatics"
```

the `A` string occupies 21 bytes of the main memory.

- The fixed-length string occupies a fixed part of the main memory. At the end of the string declaration (behind an asterisk), we must specify the size of the main memory (in bytes) for this string. In the above example, `B` is a string of fixed length, which occupies 15 bytes of the main memory.

The quantity of characters of the value, assigned to the fixed-length string, may differ from the quantity specified in the declaration, i.e., may be less or greater than 15. In the first case, instead of missing characters, spaces will be automatically added to the end of the string. In the second case, superfluous characters will be automatically removed.

For association of two or more strings, we must use one of signs & (ampersand) and + (plus). The resulting string, as a quoted sequence of characters, does not depend on the sign.

The program below is an example of using signs & and + as the string connector.

```
Sub Strings1()
    Dim strA As String, strB As String, strC As String
    strA = "String ": strB = "variable"
    strC = strA & strB
                     'result: strC = "String variable"
    strC = "String " + strB
                     'result: strC = "String variable"
End Sub
```

Owing to the presence of the connector, strings

```
strA & strB
"String " + strB
```

are called compound strings. They can be considered as expressions similar to arithmetic and logical expressions.

Term “substring” is used below. It is a string, not containing the space and tabulation characters. Substrings are put together into a string by means of the space character or vbTab — the built-in constant corresponding to the tabulation character. The space character and vbTab (or the tabulation character) are called the substring connectors.

Let us complete the last program by operators

```
strA = "String variable"
                'result: strA = "String variable"
strB = "String" & " " & "variable"
                'result: strB = "String variable"
strC = "String" & vbTab & "variable"
                'result: strC = "String|variable"
```

According to the comments, the resulting first and second strings are the same quoted set of characters. This string differs from the resulting third string by only the character between substrings "String" and "variable". It is:

- the space character in the first and second strings;
- the tabulation character in the third string.

If we place the mouse pointer on strC in the program text (after execution of the last operator), information *String□variable* or *String variable* appears.

It should be emphasized that the connection result is a string, not a substring.

When working with strings, three functions of removing spaces are used:

- Trim deletes the beginning and ending spaces of the string that is the function argument;
- LTrim deletes the beginning spaces of the string (on the left);
- RTrim deletes the ending spaces of the string (on the right).

The following program is an example of using these functions.

```
Sub Strings2()
    Dim strA As String, strB As String
    strA = "   String variable   "
    strB = Trim(strA)
                  'result: strB = "String variable"
    strB = LTrim(strA)
                  'result: strB = "String variable   "
    strB = RTrim(strA)
                  'result: strB = "   String variable"
End Sub
```

As already mentioned in Sections 1.4 and 1.8, for converting number to string, the `CStr` or `Str` function is used; for the inverse conversion, the `Val` function is used.

Function `Space` returns a string of spaces into the program; the quantity of spaces is determined by the function argument.

The following program uses functions `CStr`, `Val` and `Space`.

```
Sub Strings3()
    Dim strA As String, curB As Currency
    Dim strC As String
    strA = "X = "
    curB = 45.77
    strC = strA & CStr(curB)
                   'result: strC = "X = 45.77"
    curB = Val("45.77 = X")      'result: curB = 45.77
    curB = Val(strC)             'result: curB = 0
    strC = "String" & Space(3) & "variable"
                   'result: strC = "String   variable"
End Sub
```

It is possible to transform a string so that all letters in it become uppercase or lowercase. For this purpose, functions `UCase` and `LCase` are used, respectively, as in the following example program:

```
Sub Strings4()
    Dim strA As String, strB As String
    strA = "Pavel Ivanov"
    strB = UCase(strA)  'result: strB = "PAVEL IVANOV"
    strB = LCase(strA)  'result: strB = "pavel ivanov"
End Sub
```

To replace any part of a string by certain characters, the `Replace` function is used. The example program follows:

```
Sub Strings5()
    Dim strA As String
    Dim strB As String
    strA = "Pavel Ivanov"
    strB = Replace(strA, "Ivanov", "Gusev")
                         'result: strB = "Pavel Gusev"
End Sub
```

To determine the quantity of characters in a string (without considering inverted commas), function `Len` (from “length”) is used, at that, its argument is the string. Here is the example program:

```
Sub Strings6()
    Dim strA As String
    Dim intA As Integer
    strA = "String variable"
    intA = Len(strA)                'result: intA = 15
End Sub
```

Functions `Replace` and `Len` have additional possibilities, which can be studied by means of the Excel help system started by pressing the *F1* key when the VB window is active.

Quite often, we have to extract a part from a string. For this purpose, functions `Left`, `Right` and `Mid` (from “middle”) are used.

The calls of functions `Left` and `Right` are as follows:

```
Left(string, quantity)
Right(string, quantity)
```

These functions return the string containing the specified *`quantity`* of characters of the beginning and end of *`string`*, respectively.

The call of function `Mid` follows:

```
Mid(string, number[, quantity])
```

This function returns the string containing *`quantity`* characters of *`string`*, starting from the character whose number equals *`number`*. If *`quantity`* is omitted, the `Mid` function returns all characters up to the end of *`string`*.

Functions `Left`, `Right` and `Mid` are used in the following program:

```
Sub Strings7()
    Dim strA As String
    Dim strB As String
    strA = "My string variable"
    strB = Left("My string variable", 9)
                               'result: strB = "My string"
    strB = Right(strA, 8)   'result: strB = "variable"
    strB = Mid(strA, 4, 6) 'result: strB = "string"
    strB = Mid(strA, 11)    'result: strB = "variable"
End Sub
```

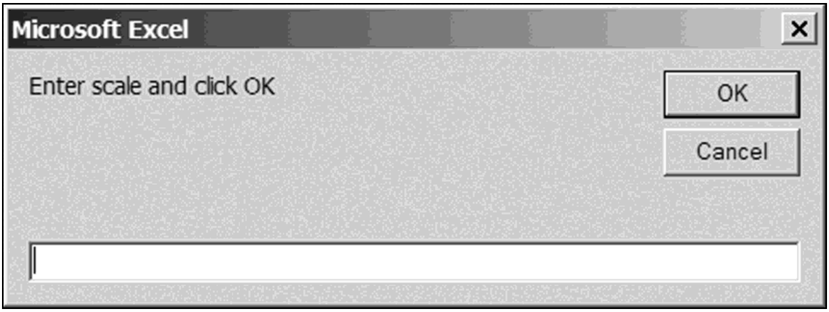
Let us consider an example of using the compound string in the call of the `MsgBox` procedure.

1. In Listing 1.4, we replace operator `MsgBox Str(Distance(P()))` on p. 93 by the following operator block:

```
Dim L As Single
Dim S As String
Dim M As Single
L = Distance(P())
S = InputBox("Enter scale and click OK")
                    'input of scale, Fig. 1.17
M = Val(S)
L = M * L
MsgBox "Distance from " & P(0).Name & " to " & _
    P(1).Name & vbCrLf & "equals" & Str(L) & " km"
                    'output of distance, Fig. 1.18
```

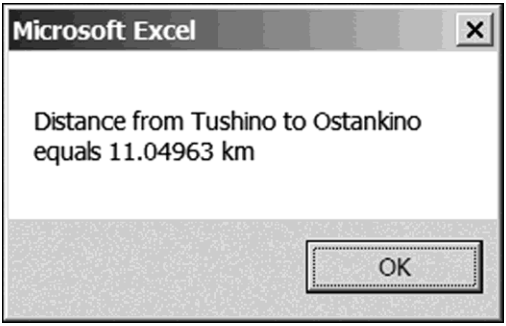
2. Let us click on arrow ► to start the program execution.
3. We put 0.4 into the text box of the displayed window (Fig. 1.17) and click on the *OK* button.
4. Let us click on the *OK* button in the emerging window (Fig. 1.18) to terminate the program execution.

In the above operator block, the calls of `InputBox` and `MsgBox` appeared. As already mentioned, the built-in `InputBox` function is used to input information from the keyboard; this function returns (into the program) the string, which was put into the text box. The build-in `MsgBox` procedure is used to depict information on the display screen. The parameter of this procedure is a compound string, which includes built-in constant `vbCrLf`.



Enter scale and click OK

Fig. 1.17. The window for inputting the scale



Distance from Tushino to Ostankino
equals 11.04963 km

Fig. 1.18. The window with the calculation result

Because of using the vbCrLf constant, we see two lines in the resulting window (Fig. 1.18). The vbCrLf constant can be replaced by association vbCr & vbLf of built-in constants "return" vbCr (from "Carriage return") and "new line" vbLf (from "Linefeed").

The TimeValue function is useful, which converts time from the String data type to the Date data type. As an example of using this function, let us consider the following program:

```
Sub TV()
    Dim dTime1 As Date
    Dim dTime2 As Date
    Dim dTime3 As Date
    dTime1 = #2:30:45# + TimeValue("00:15:00")
                                '1st assignment operator
                                'result: dTime1 = 2:45:45
    dTime2 = Now + TimeValue("00:25:00")
                                 '2nd assignment operator
    dTime3 = Time + TimeValue("00:00:10")
                                 '3rd assignment operator
End Sub
```

Because of executing the 2nd assignment operator, the `dTime2` variable has a value, which is equal to the current date and time plus 25 minutes. Because of executing the 3rd assignment operator, `dTime3` is equal to the current time plus 10 seconds.

In Listing 2.10 on p. 170, we will use the `Format` function intended for converting a value to a string of the given form. To see the full information on this function, we must press the *F1* key when the blinking cursor is located on `Format` in the code window.

1.20. Work with text files

A file is an area on a hard disk, compact disk, USB flash drive (UFD) or any other medium that:

- contains single-type information;
- has a name.

When working with files, we will use operations for reading information from a file and writing information into a file.

Several types of files exist; we will consider text files. The contents of such file are lines of characters with a combination of characters “return” and “new line” at the end of each line.

For viewing text files, we will use the Notepad editor. At its window, we will not see characters “return” and “new line”.

To start working with a text file, operator `Open` is used for opening this file. This operator has the following syntax:

```
Open name For purpose As number
```

In this construct, *`name`* is the file full name, i.e., the file name together with its path (in the file system of Windows) and extension, *`number`* is the file number, *`purpose`* is keyword `Input`, `Output` or `Append` (from “appending”).

The last three keywords have the following sense:

- `Input` means that the file must be opened for reading information from this file;
- `Output` — the file must be opened for writing information into it;
- `Append` — the file must be opened for adding information into it.

As the file number, we recommend to use the *`number`* variable of the `Integer` data type whose value is the result of executing the following assignment operator:

```
number = FreeFile
```

where `FreeFile` is the built-in function that returns (into the program) the free file number.

The last assignment operator must be placed above the `Open` operator, more precisely, it must be executed before the `Open` operator.

After finishing the work with the file, it must be closed by the `Close` operator as follows:

```
Close number
```

For addition of new lines into the file, operator `Print` is used, which has the following syntax:

```
Print #number, line1
```

where *`line1`* is a string (the digit may be different).

Let the file with specified *`number`* be opened by means of the `Output` keyword. When performing the `Print` operator, string *`line1`* (with a combination of characters “return” and “new line” at the end, not in inverted commas) is written into the file beginning. When repeated performing the `Print` operator, string *`line2`* is added into the file, and so on.

Let the file with specified *`number`* be opened by means of the `Append` keyword. When performing the `Print` operator, *`line1`* is added into the file.

Below, we will consider two ways of extracting information from the file with specified *`number`*, which is opened by means of the `Input` keyword.

1. Extracting information by using operator `Line Input` with the following syntax:

```
Line Input #number, variable
```

This operator reads the next line from the file, at that, this line (without “return” and “new line” at the end) is assigned to *`variable`* of the `String` data type.

2. Extracting information by means of built-in function

```
Input(quantity, number)
```

This function returns (into the program) the string, which contains the subsequent characters from the file. The quantity of these characters is specified by *`quantity`*.

In programs `Creation` and `Addition`, given below, we will use the reviewed operators and built-in functions intended for work with text files. Using these programs as examples, we will also consider other useful operators and functions.

The first program follows:

Listing 1.5

```
Sub Creation()
    Dim FName1 As String
    Dim FName2 As String
    Dim FNum1 As Integer
    Dim FNum2 As Integer
    Dim n1 As Long
    Dim n2 As Long
    Dim strA As String       'auxiliary string
1:  MkDir("c:\Users\usr\texts")
                             'creating folder texts
2:  FName1 = _
        "c:\Users\usr\texts\a.txt"
                             'full name of 1st file
3:  FName2 = _
        "c:\Users\usr\texts\b.txt"
                             'full name of 2nd file
'Creating file a.txt:
    FNum1 = FreeFile
    Open FName1 For Output As FNum1
                             'opening file a.txt
    strA = "Text file is created,"
    Print #FNum1, strA
    Print #FNum1, _
        "it contains several strings."
'Determining quantity of characters in file a.txt:
    n1 = LOF(FNum1)
    Close FNum1              'closing file a.txt
    MsgBox "In file a.txt" & Str(n1) & " characters"
'Copying information from file a.txt to file b.txt:
    FNum1 = FreeFile
    Open FName1 For Input As FNum1
                             'opening file a.txt
    FNum2 = FreeFile
    Open FName2 For Output As FNum2
                             'opening file b.txt
    Do Until EOF(FNum1)      'cycle of reading-writing
        Line Input #FNum1, strA
        Print #FNum2, strA
    Loop
    Close FNum1              'closing file a.txt
'Adding new string into file b.txt:
    strA = "New string is added."
    Print #FNum2, strA
'Determining quantity of characters in file b.txt:
    n2 = LOF(FNum2)
    Close FNum2              'closing file b.txt
    FNum2 = FreeFile
    Open FName2 For Input As FNum2
                             'opening file b.txt
    strA = Input(1, FNum2)   'reading character
    Close FNum2              'closing file b.txt
    MsgBox "In file b.txt" & Str(n2) & _
        " characters, " & _
        "and first character is " & _
        strA
End Sub
```

In operators 1, 2 and 3, usr is the computer user name. Before the program execution, the reader should type his concrete user name instead of usr in these operators.

During the program execution, operator

```
MkDir("c:\Users\usr\texts")
```

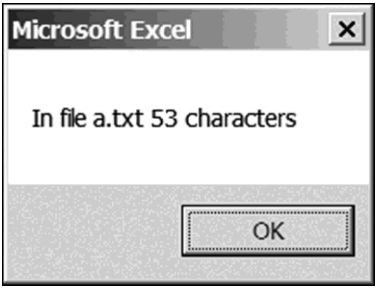
creates folder texts inside folder usr; MkDir is the abbreviation of “make directory”.

The built-in LOF function returns into the program the quantity of characters in the file, at that, characters “return” and “new line”, which are at the end of each line, are taken into consideration. The argument of this function is the file number; LOF is the abbreviation of “length of file”.

The argument of the built-in EOF function is the file number too. This function, figuring in the condition of the Do Until…Loop cycle termination, returns (into the program) True at achievement of the file end. The function name is the abbreviation of “end of file”.

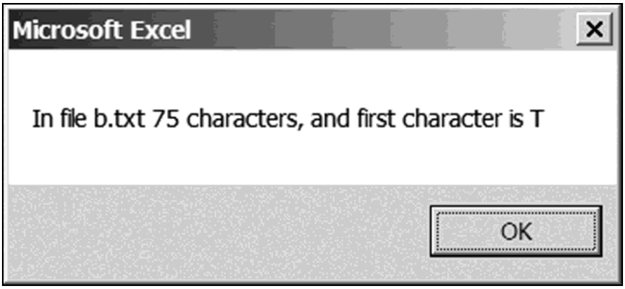
The MsgBox procedure is intended for depicting the string, which is its parameter, on the display screen. Two calls of MsgBox are in the Creation program, therefore, two windows, represented in Fig. 1.19 and 1.20, appear during the program execution.

After termination of the program execution, files a.txt and b.txt have the following contents.



In file a.txt 53 characters

Fig. 1.19



In file b.txt 75 characters, and first character is T

Fig. 1.20

File `a.txt`:

```
Text file is created,
it contains several strings.
```

File `b.txt`:

```
Text file is created,
it contains several strings.
New string is added.
```

We advise the reader to look through the contents of files `a.txt` and `b.txt` by using the Notepad editor to be convinced of the correctness of the program work.

Before restarting the `Creation` program, the `MkDir` operator of creating the `texts` folder must be omitted, for example, by means of an apostrophe.

The liquidation of files a.txt and b.txt and folder texts may be required at the end of the program execution. For this purpose, the following operators are used:

```
Kill(FName1)
Kill(FName2)
RmDir("c:\Users\usr\texts")
```

where RmDir is the abbreviation of “remove directory”. We must insert these operators above operator End Sub.

At the end of this section, we will consider how to tune Windows Explorer for displaying not only the file names, but also the extension of these names, for example, extension .txt for the above text files.

The following second program adds a line into file b.txt and runs the Notepad editor.

Listing 1.6

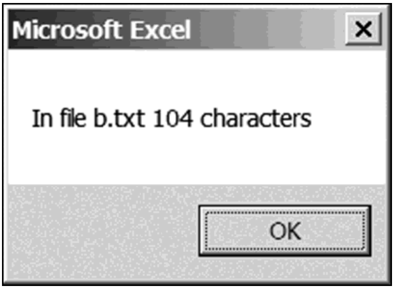
```
Sub Addition()
    Dim FNum As Integer, n As Long
    Dim RetVal As Integer          'for function Shell
    FNum = FreeFile
'Opening file b.txt and adding new string:
    Open "c:\Users\usr\texts\b.txt" _
        For Append As FNum
    Print #FNum, "Second new string is added."
'Determining quantity of characters in file b.txt:
    n = LOF(FNum)
    Close FNum                     'closing file b.txt
    MsgBox "In file b.txt" & Str(n) & " characters"
'Starting editor Notepad:
0:  RetVal = Shell("c:\Windows\notepad.exe",1)
End Sub
```

The Shell function (operator 0) is intended for running any executable file whose name has extension .exe. The first argument of the Shell function is the full name of the executable file (here, the full name must be without spaces). The second argument, which may be omitted, defines the style of the window being a result of calling the Shell function.

To see the full information on the Shell function, we must press the *F1* key when the blinking cursor is located on Shell in the code window.

We will run the Addition program after the execution of the Creation program (without the operators liquidating the files and folder).

The window, shown in Fig. 1.21, appears during the Addition program execution. After clicking on *OK* in this window, operator 0 is executed, which starts the Notepad editor (Fig. 1.22), and then operator End Sub is executed, i.e., the Addition program is terminated.



Microsoft Excel

In file b.txt 104 characters

Fig. 1.21

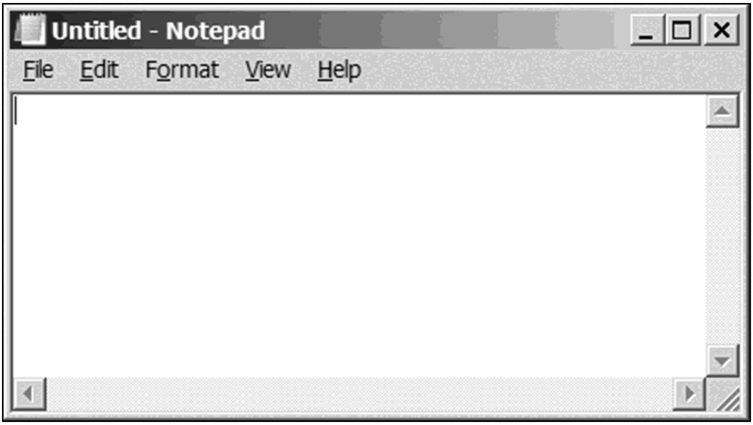


Fig. 1.22. The Notepad window

After terminating the Addition program, the Notepad window remains open. By means of this window, we can view the contents of a text file (in particular, b.txt) and, if necessary, create a new file or edit an existing file.

After the Addition program execution, file b.txt has the following contents:

```
Text file is created,
it contains several strings.
New string is added.
Second new string is added.
```

If a file is opened for adding information (by using keyword `Append`), this file may not exist. In this case, the file is created. If a file is opened for reading information (by using keyword `Input`), this file, naturally, should exist.

To see file names with extension in Windows Explorer, we must fulfill the following:

1) in Windows Explorer, open the folder, which contains files of interest;

2) on the menu bar of Windows Explorer, fulfill *Organize* > *Folder and search options*;

3) in open window *Folder Options*, activate tab *View*;

4) in list *Advanced settings*, turn off option *Hide extensions for known file types*;

5) successively click on buttons *Apply* and *OK*.

To expand the chosen operation mode of Windows Explorer to all folders, we must click on button *Apply to Folders* before clicking on the *Apply* button. In open window *Folder Views*, we must click on the *Yes* button.

We advise the reader to write a program for creating a text file including $n+1$ lines of the following form:

$$\mathrm{f}(x_i) = f_i \,,$$

where x_i, f_i are the values of argument and function, $0 \le i \le n$; $a = x_0 < x_1 < x_2 < ... < x_{n-2} < x_{n-1} < x_n = b$. Function $f(x)$ and corresponding values of a and b from Appendix 4 must be used.

The constructs of Visual Basic considered above can be used for solving sufficiently complicated tasks, two of which will be formulated in the next section.

1.21. Matrix terminology. Formulation of demonstration tasks

In the previous sections, we were solving the task of calculating the hypotenuse length. However, this task is very simple, and it is impossible to show all possibilities of Visual Basic on it. Therefore, we will also solve two tasks of transposing a numerical matrix, relative to the main and auxiliary diagonals. Let us formalize the concept of matrix transposition.

Let **A** be a matrix containing *m* rows and *n* columns, **B** be a matrix containing *n* rows and *m* columns:

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1j} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2j} & \dots & a_{2n} \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ a_{i1} & a_{i2} & \dots & a_{ij} & \dots & a_{in} \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ a_{m1} & a_{m2} & \dots & a_{mj} & \dots & a_{mn} \end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix} b_{11} & b_{12} & \dots & b_{1i} & \dots & b_{1m} \\ b_{21} & b_{22} & \dots & b_{2i} & \dots & b_{2m} \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ b_{j1} & b_{j2} & \dots & b_{ji} & \dots & b_{jm} \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ b_{n1} & b_{n2} & \dots & b_{ni} & \dots & b_{nm} \end{bmatrix}.$$

If elements of matrix **B** “are calculated” according to formula

$$b_{ji} = a_{ij} \tag{1.2}$$

at $1 \le i \le m$, $1 \le j \le n$, mathematicians say that matrix **B** is obtained by transposing matrix **A** ***relative to the main diagonal*** (or simply, by transposing matrix **A**).

The main (left-to-right) diagonal of the matrix is an imaginary straight line from the top left corner of the matrix to the bottom right corner.

If formula (1.2) is replaced by formula

$$b_{ji} = a_{m+1-i,\, n+1-j}, \tag{1.3}$$

then matrix **B** is obtained by transposing **A** ***relative to the auxiliary diagonal***.

The auxiliary (right-to-left) diagonal of the matrix is an imaginary straight line from the top right corner to the bottom left corner.

To solve the task of transposing matrix **A**, we must generate matrix **B** according to formula (1.2) or (1.3). Note that $\mathbf{A}^{\mathrm{T}}$ is the usual designation of the **A** matrix transposed relative to the main diagonal, i.e., according to (1.2).

Let us notice the following obvious fact: when transposing the matrix relative to the main diagonal, its rows and columns interchange their positions.

If $m = n$, matrix **A** is called a square matrix. In this case, elements a_{ii} are located on the main diagonal, elements $a_{i,\,m+1-i}$ are located on the auxiliary diagonal, $1 \le i \le m$.

When transposing square matrix **A** relative to the main (auxiliary) diagonal, the mirror reflection of **A** relatively the main (auxiliary) diagonal takes place.

Further, we will use the following terms and definitions.

The **A** matrix, containing m rows and n columns, is named as "the **A** matrix of size $m \times n$" or "the **A** matrix $m \times n$". The matrix, containing one column or one row, is called a vector.

The product of the **A** matrix $m \times n$ and the **C** matrix $n \times s$ is the **AC** matrix $m \times s$, which is a result of the scalar multiplication of the rows of **A** and the columns of **C**, namely:

$$\mathbf{AC} = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \cdot & \cdot & \cdot & \cdot \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix} \times \begin{bmatrix} c_{11} & c_{12} & \dots & c_{1s} \\ c_{21} & c_{22} & \dots & c_{2s} \\ \cdot & \cdot & \cdot & \cdot \\ c_{n1} & c_{n2} & \dots & c_{ns} \end{bmatrix} =$$

$$= \begin{bmatrix} a_{11}c_{11} + \dots + a_{1n}c_{n1} & a_{11}c_{12} + \dots + a_{1n}c_{n2} & \dots & a_{11}c_{1s} + \dots + a_{1n}c_{ns} \\ a_{21}c_{11} + \dots + a_{2n}c_{n1} & a_{21}c_{12} + \dots + a_{2n}c_{n2} & \dots & a_{21}c_{1s} + \dots + a_{2n}c_{ns} \\ \dots & \dots & \dots & \dots \\ a_{m1}c_{11} + \dots + a_{mn}c_{n1} & a_{m1}c_{12} + \dots + a_{mn}c_{n2} & \dots & a_{m1}c_{1s} + \dots + a_{mn}c_{ns} \end{bmatrix}.$$

More precisely, the elements of **R** = **AC** are defined by the following formula:

$$r_{iq} = \sum_{j=1}^{n} (a_{ij} c_{jq}),$$
$$1 \le i \le m,\ 1 \le q \le s.$$

The **A** matrix $m \times n$ and the **D** matrix $m \times n$ are called equal if the corresponding elements of these matrices are equal: $a_{ij} = d_{ij}$, $1 \le i \le m$, $1 \le j \le n$.

Square matrix **E** is called the unit matrix if the main diagonal of **E** contains only units, and all remaining elements of **E** are equal to zero.

The inverse **A** matrix is the $\mathbf{A}^{-1}$ matrix of size $n \times m$, which satisfies the following condition: $\mathbf{A}^{-1}\mathbf{A} = \mathbf{E}$, where **E** is the unit matrix of size $n \times n$.

1.22. Program for transposing a matrix relative to its auxiliary diagonal

The program for solving one of the two tasks formulated in the previous section is given below. This program uses file `a.txt` with the source data (values of *m* and *n* and matrix **A**) and creates file `b.txt` with the result (the **B** matrix).

Listing 1.7

```
Sub TRANSPA()
Dim FNameA As String, FNameB As String
Dim FNum As Integer
Dim strC As String, strD As String, strE As String
Dim m As Integer, n As Integer
Dim i As Integer, j As Integer
Dim k As Integer, l As Integer
Dim A() As Double, B() As Double
1:  FNameA = _
        "c:\Users\usr\texts\a.txt"
2:  FNameB = _
        "c:\Users\usr\texts\b.txt"
3:  FNum = FreeFile
'Opening file a.txt:
4:  Open FNameA For Input As FNum
'Reading values of m and n from file a.txt:
5:  Line Input #FNum, strC
6:  strC = Mid(strC, 3)
7:  m = Val(strC)
8:  Line Input #FNum, strC
9:  strC = Mid(strC, 3)
10: n = Val(strC)
'Setting size of matrices:
11: ReDim A(1 To m, 1 To n)
12: ReDim B(1 To n, 1 To m)
'Reading matrix A from file a.txt:
13: For i = 1 To m
14:    Line Input #FNum, strC
```

```
15:    j = 0
16:    strD = ""
                 'string "" is not equal to string " "
17:    l = Len(strC)
18:    For k = 1 To l
19:       strE = Mid(strC, k, 1)
20:       If strE <> " " Then strD = strD & strE
21:       If strE = " " Or k = l Then
22:          j = j + 1
23:          A(i, j) = Val(strD)
24:          strD = ""
25:       End If
26:    Next k
27: Next i
'Closing file a.txt:
28: Close FNum
29: FNum = FreeFile
'Opening file b.txt:
30: Open FNameB For Output As FNum
'Forming matrix B, its writing into file b.txt:
31: For j = 1 To n
32:    strC = ""
33:    For i = 1 To m
34:       B(j, i) = A(m + 1 - i, n + 1 - j)
35:       strC = strC & Str(B(j, i)) & " "
36:    Next i
37:    Print #FNum, strC
38: Next j
'Closing file b.txt:
39: Close FNum
End Sub
```

In operators 1 and 2, usr is the user name.

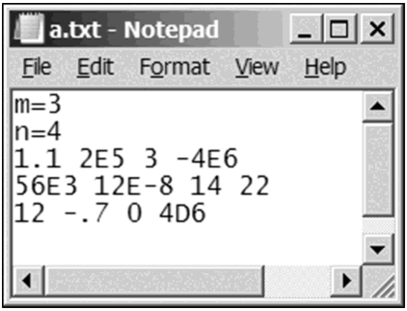
Program TRANSPA is intended for transposing a numerical matrix relative to its auxiliary diagonal. In operators 16, 24 and 32, we see string "", not containing any character. Operators 20, 21 and 35 include string " ", which contains only the space character.

Operator 34 corresponds to formula (1.3), which defines the operation of transposing a matrix relative to its auxiliary diagonal.

File

```
c:\Users\usr\texts\a.txt
```

contains the source data, i.e., values of *m* and *n* and matrix **A**. This file may be created by means of the Notepad editor whose window is represented in Fig. 1.23.



```
m=3
n=4
1.1 2E5 3 -4E6
56E3 12E-8 14 22
12 -.7 0 4D6
```

Fig. 1.23. The source data: elements of the matrix's rows separate from each other by one space

For starting the execution, we can use one of the following two ways:

- clicking on arrow ▶ in the VB window;
- fulfilling the following operations in the Excel window: *Developer* (or *View*) > *Macros* > line *TRANSPA* > *Run*.

It is possible to appoint a combination of keys starting the program execution. For that, being in the Excel window, we fulfill operations *Developer* (or *View*) > *Macros* > line *TRANSPA* > *Options*. In open window *Macro Options*, we fulfill the following:

- appoint the startup key combination, for example, *Ctrl* + *t*;
- click on the *OK* button;
- close the window by clicking on the little cross in the top right corner.

After that, the simultaneous press of the *Ctrl* and *t* keys runs the `TRANSPA` program.

The result of the `TRANSPA` program execution is matrix **B**, which is in file

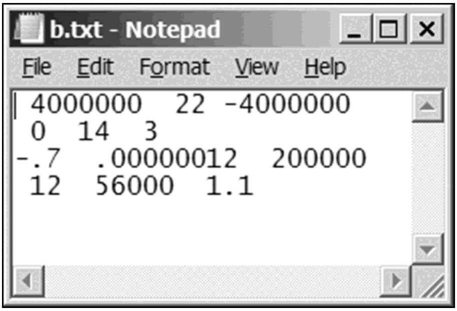
```
c:\Users\usr\texts\b.txt
```

The Notepad window with matrix **B** is represented in Fig. 1.24.

Let us consider ways of transition from the VB window to the Excel window and back.

The transition from the VB window to the Excel window is possible in one of the following four ways:

- clicking on the Excel button at the left end of the standard toolbar in the VB window;
- simultaneous pressing the *Alt* and *F11* keys;
- rolling the VB window down by means of the underscore button in the top right corner;
- clicking on the Excel button on the taskbar of Windows Desktop.



```
 4000000   22 -4000000
 0  14   3
-.7   .00000012   200000
 12   56000  1.1
```

Fig. 1.24. The calculation result

For returning to the VB window from the Excel window, we should fulfill *Developer* (or *View*) > *Macros* > line *TRANSPA* > *Edit*. As a result, the VB window with the `TRANSPA` program text in the code window appears. Further, we can correct the program, for example, change names of files with the source data and for the calculation result.

If we click on button *Step Into* (instead of button *Edit*), the VB window will also be opened. Further, the step-by-step execution of the `TRANSPA` program (by means of the *F8* key) is possible.

Other ways of transition from the Excel window to the VB window follow:

- pressing *Alt* + *F11*;
- rolling the Excel window down by means of the underscore button;
- clicking on the VBA button on the taskbar of Windows Desktop.

The `TRANSPA` program text almost coincides with the text of the program, which will be reviewed in the next section. Therefore, the `TRANSPA` program will still be required.

For further usage of the `TRANSPA` program, we have to save it on the computer's hard disk by fulfilling the following operations similar to the operations described on p. 18:

1) in the Excel window, *File* > *Save As* > *Browse*;

2) in the *Save As* window, choose a folder intended for saving the Excel workbook, for example, *c:\Users\usr*;

3) enter *BookTRANSPA* into text box *File name*;

4) set file type *Excel Macro-Enabled Workbook* by means of drop-down list *Save as type*;

5) click on the *Save* button.

The `TRANSPA` program is saved as a part of the BookTRANSPA.xlsm workbook of Excel.

As we see, the user interface of the `TRANSPA` program is a pair of text files.

1.23. User-defined forms

As the program user interface, the form is sometimes convenient. For creating this rectangle with text boxes, buttons and other control elements, Form Designer exists in Visual Basic Environment.

Let us create the form for the program intended for calculating the length of the hypotenuse of a right-angled triangle.

The following control elements are suitable for the form:

- place for displaying the calculation result;
- text box for inputting (into the program) the length of the first leg;
- text box for inputting the length of the second leg;
- button for starting the calculation.

Let Book1 be the name of the active Excel workbook. Developing the program (project) with the form begins with the following sequence of operations for inserting the form into the active Excel workbook (instead of inserting the module).

1. In the Excel window, fulfill *Developer* > *Visual Basic* in area *Code*. As a result, the VB window (Fig. 1.1) containing the project explorer window and the properties window appears.

In the absence of the properties window, we can open it by fulfilling *View* > *Properties Window*.

2. In the project explorer window, highlight line *VBAProject (Book1)* by clicking on it.

3. Fulfill *Insert* > *UserForm*.

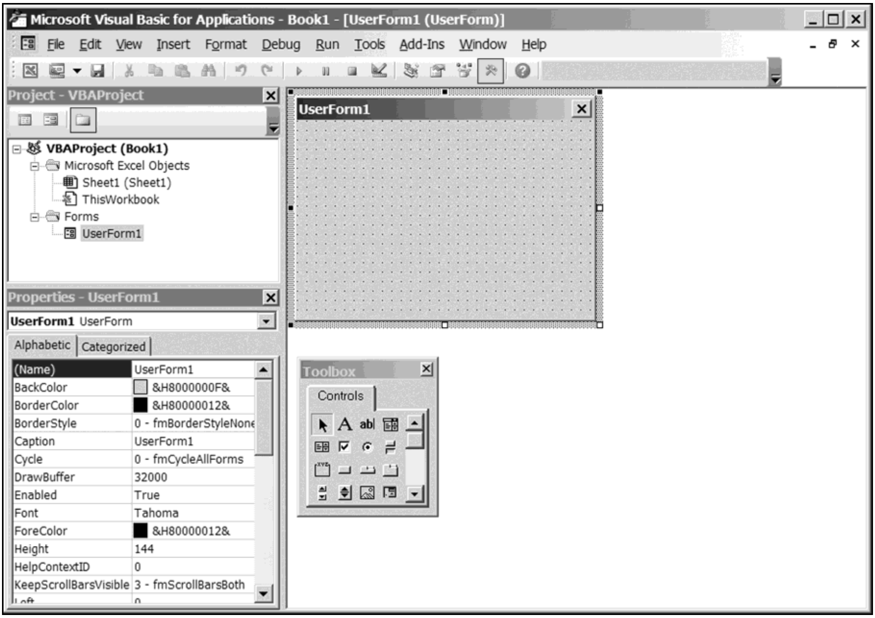
As a result, the following features appear (Fig. 1.25):

- blank form UserForm1, a rectangle with points;
- line *UserForm1* in the project explorer window;
- the *Toolbox* window, containing ***the control elements***.

If the last window is not displayed, for displaying it, we must click on the form and then fulfill *View* > *Toolbox*.

The properties window “is tied” to the form because this element is selected by the frame with 8 markers (five black and three white, Fig. 1.25).

To change the width of the UserForm1 form, we have to drag the form’s right border by using the mouse, having seized the white marker in the center of this border. Thus, the value of the Width property (in the properties window) also changes.



(Name)	UserForm1
BackColor	□ &H8000000F&
BorderColor	■ &H80000012&
BorderStyle	0 - fmBorderStyleNone
Caption	UserForm1
Cycle	0 - fmCycleAllForms
DrawBuffer	32000
Enabled	True
Font	Tahoma
ForeColor	■ &H80000012&
Height	144
HelpContextID	0
KeepScrollBarsVisible	3 - fmScrollBarsBoth
Left	0

Fig. 1.25. The VB window after inserting UserForm1 into the Excel workbook

Expression “to seize a point” means the following: to place the mouse pointer on this point, and then to press the left button on the mouse.

The value of property Width (that is, the form width) can also be set in the properties window.

Similarly, we can change the height of the UserForm1 form (property Height).

After clicking on property Caption (in the properties window), we will change it to the following: *My first form*.

The *Toolbox* window (Fig. 1.25) contains pictograms of the control elements, which can be simply used in the project. We will use the following four elements: Label, TextBox, CommandButton and CheckBox. To see the element name, we have to place the mouse pointer on this element in *Toolbox*.

To insert an element into the form and to edit this element, we fulfill the following operations.

1. Select the form by clicking on it.
2. Click on the required element of the *Toolbox* window.
3. Move the mouse pointer into the form. At that, the pointer becomes a crosshair.
4. Place the crosshair into the required part of the form and press the left button on the mouse. After that, release the button.

Or in another way, without releasing the left button, drag the mouse pointer, for example, downwards and to the right. After that, release the button.

5. Move the element (if needed), having seized its center.
6. Move the element's borders (if needed), by turn having seized the white markers.
7. Set the element's properties by means of the properties window, which is tied to this element (selected by the frame with 8 white markers).

Upon termination of the element editing, we must remove its selection by clicking on area outside the element limits. To return to the element editing, we must select this element by clicking on its image in the form. For removing the selected element from the form, we must press the *Delete* key.

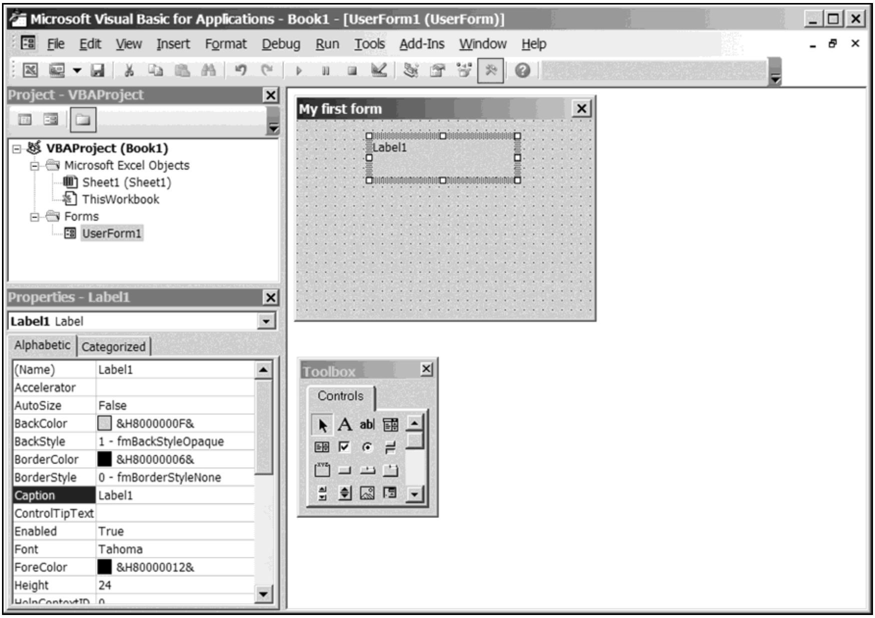
In the form, we should create ***the message place***, where the calculation result will be displayed during the program execution. For this purpose, we will use element Label.

To insert element Label into the form and to change several properties of this element, we fulfill the following.

1. Click on the Label element, which is in the *Toolbox* window.
2. Move the mouse pointer into the form. At that, the pointer becomes a crosshair.
3. Place the crosshair into the top left part of the form and press the left button on the mouse. Without releasing the button, drag the mouse pointer downwards and to the right. After that, release the button.

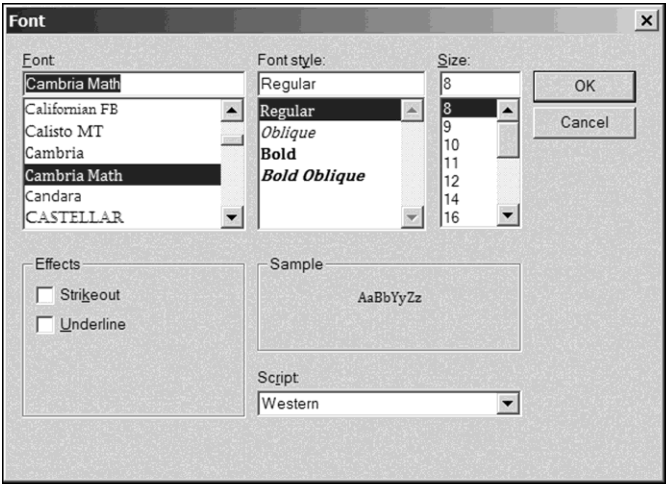
The message place with default name Label1 (that is, the Label1 element) is the result (Fig 1.26).

4. Use the Font property for setting the font fashion and size. When clicking on this property, a button appears in the properties window. The dots on this button mean that a window exists for setting the Font property.
5. Click on the dots image button. At that, the *Font* window appears for setting the message parameters.
6. Click on line *Cambria Math* (Fig. 1.27) for setting this font. In the *Sample* area, the font fashion is pictured.
7. Set the font size at 10.
8. Click on the *OK* button.



(Name)	Label1
Accelerator	
AutoSize	False
BackColor	&H8000000F&
BackStyle	1 - fmBackStyleOpaque
BorderColor	&H80000006&
BorderStyle	0 - fmBorderStyleNone
Caption	Label1
ControlTipText	
Enabled	True
Font	Tahoma
ForeColor	&H80000012&
Height	24
HelpContextID	0

Fig 1.26. The VB window after inserting element Label1 into form UserForm1



Font: Cambria Math

Font style: Regular

Size: 8

□ Strikeout

□ Underline

Script: Western

Fig. 1.27. The window for setting font

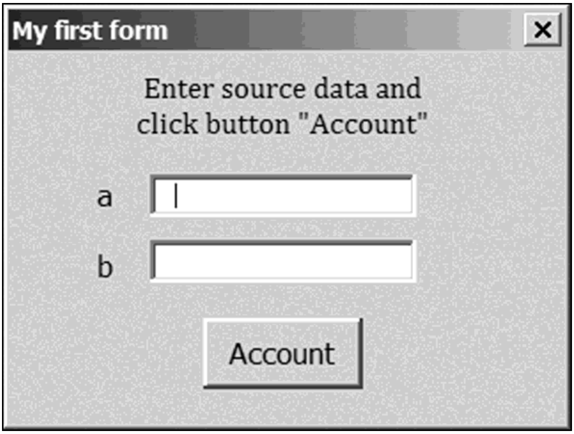
9. Set the following values of other properties of the message place.

TextAlign: *2* (horizontal center alignment, taken from the drop-down list)

Caption: *Enter source data and click button "Account"*

Name: *LabelMessage* (we change the element name)

10. By means of the mouse, change the height and width of the message place and the height and width of the form to get the message image as in Fig. 1.28.



Enter source data and click button "Account"

Fig. 1.28. The form after the program start

Similarly, we can insert ***the text box*** into the form. For that, the TextBox element is used.

We will insert two text boxes (Fig. 1.28), with names TextBoxA and TextBoxB. On the left of the text boxes, we will create inscriptions *a* and *b* by using the Label element.

To insert ***the button*** into the form, we fulfill the following.

1. Click on the CommandButton element, which is in the *Toolbox* window.
2. Depict the button in the form.
3. Set the following properties of this button.

Caption: *Account*

Name: *ComButCalc*

4. Remove the button selection by clicking on area outside its limits.

The development of the form by means of Form Designer is completed. Now we have to develop ***the program***.

Let us click twice on the *Account* button, which is in the UserForm1 form. At that, there appears the code window (corresponding to the form) with the blank of the program, being started by clicking on the *Account* button.

The following first and last lines of the program are generated automatically:

```
Private Sub ComButCalc_Click()

End Sub
```

We already encountered the `Private` keyword on p. 81. Here, it means the following: if `ComButCalc_Click` is used as a subroutine, its call should be in the same code window, in which the subroutine declaration is located. However, we are not going to use `ComButCalc_Click` as a subroutine. Therefore, the `Private` keyword may be deleted.

The `ComButCalc_Click` program for calculating the hypotenuse length follows:

```
Private Sub ComButCalc_Click()
    Dim a As Single, b As Single, c As Single
    Dim s As String
1:  s = TextBoxA.Text
2:  a = Val(s)
3:  s = TextBoxB.Text
4:  b = Val(s)
5:  c = Sqr(a ^ 2 + b ^ 2)     'according to Pythagoras
6:  s = Str(c)
7:  LabelMessage.Caption = "c =" & s
End Sub
```

This program is similar to the `Pythagoras1` program (p. 85).

Unlike program `Pythagoras1`, the `ComButCalc_Click` program contains the following three constructs similar to the record: `TextBoxA`, `TextBoxB` and `LabelMessage`.

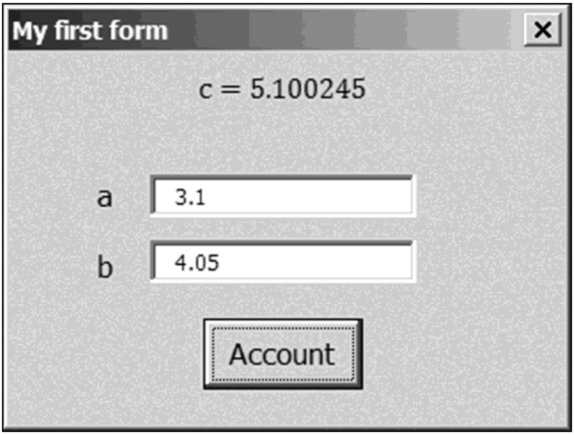
In operator `1`, “field” `Text` of “record” `TextBoxA` (of “type” `TextBox`), more precisely, property `Text` of element `TextBoxA` is a variable of the `String` data type whose value coincides with the string being entered into the *a* text box of the form during the program usage. In operator `3`, property `Text` of element `TextBoxB` coincides with the string being entered into the *b* text box.

The values of `a` and `b` are the results of executing operators `1` — `4`.

Operator `5` calculates length `c` of the hypotenuse; operator `6` converts this value to string `s`.

Operator `7` assigns the string (which includes the `s` string) to property `LabelMessage.Caption` (that is, to property `Caption` of element `LabelMessage`). The value of this property is put into the form.

We can start the developed program by clicking on arrow ► of the VB window. The form (Fig. 1.28), loaded into the Excel window, is the result of this click. Fig. 1.29 shows the form's state upon finishing the calculation, i.e., after putting the lengths of the triangle's legs into text boxes *a* and *b* and clicking on the *Account* button.



c = 5.100245

a 3.1

b 4.05

Fig. 1.29. The form upon finishing the calculation

Further, the calculation may be repeated for other source data, i.e., other lengths of the triangle's legs. Upon termination of the series of calculations, we have to close the form by clicking on the little cross in the top right corner (Fig. 1.29).

We can simplify the `ComButCalc_Click` program as follows:

```
Private Sub ComButCalc_Click()
    Dim a As Single, b As Single, c As Single
1:  a = TextBoxA.Value
2:  b = TextBoxB.Value
3:  c = Sqr(a ^ 2 + b ^ 2)     'according to Pythagoras
4:  LabelMessage.Caption = "c =" & Str(c)
End Sub
```

In this program, we used property Value of element TextBox. At that, when performing operator `1`, the information, which is in the corresponding text box of the form, is interpreted according to the data type of the variable on the left of sign =, that is, as a number. The same can be said about operator `2`.

1.23. User-defined forms

In the previous section, we developed the TRANSPA program, Listing 1.7, for transposing a number matrix relative to its auxiliary diagonal. This program is inconvenient to use because names of files are a part of the program text (see operators 1 and 2). To eliminate this drawback, we will add the form, already designed, to the TRANSPA program. In this case, the form elements have the following sense (from top to down, Fig. 1.28):

- place for displaying messages;
- text box for inputting the full name of the source data file;
- text box for inputting the full name of the result file;
- button for starting the calculation.

The basis of the new ComButCalc_Click program is the TRANSPA program text, which is in the BookTRANSPA workbook. To gain access to this text, we fulfill the following (p. 18):

1) open the BookTRANSPA workbook with Excel, for example, by means of the context menu;

2) go to Visual Basic Environment in the standard way;

3) open the code window with the TRANSPA program text by clicking twice on the corresponding module name in the project explorer window.

The TRANSPA program text is also contained in file Listing_1_07.txt of the enclosed CD.

By using the TRANSPA program text, Listing 1.7, we will transform the ComButCalc_Click program developed above to the following view by means of Windows Clipboard.

```
Private Sub ComButCalc_Click()
Dim FNameA As String, FNameB As String
Dim FNum As Integer
Dim strC As String, strD As String, strE As String
Dim m As Integer, n As Integer
Dim i As Integer, j As Integer
Dim k As Integer, l As Integer
Dim A() As Double, B() As Double
1:  FNameA = TextBoxA.Text
2:  FNameB = TextBoxB.Text
3:  FNum = FreeFile
'Opening file a.txt:
4:  Open FNameA For Input As FNum
'Reading values of m and n from file a.txt:
5:  Line Input #FNum, strC
6:  strC = Mid(strC, 3)
7:  m = Val(strC)
```

```
8:  Line Input #FNum, strC
9:  strC = Mid(strC, 3)
10: n = Val(strC)
'Setting size of matrices:
11: ReDim A(1 To m, 1 To n)
12: ReDim B(1 To n, 1 To m)
'Reading matrix A from file a.txt:
13: For i = 1 To m
14:    Line Input #FNum, strC
15:    j = 0
16:    strD = ""
                 'string "" is not equal to string " "
17:    l = Len(strC)
18:    For k = 1 To l
19:       strE = Mid(strC, k, 1)
20:       If strE <> " " Then strD = strD & strE
21:       If strE = " " Or k = l Then
22:          j = j + 1
23:          A(i, j) = Val(strD)
24:          strD = ""
25:       End If
26:    Next k
27: Next i
'Closing file a.txt:
28: Close FNum
29: FNum = FreeFile
'Opening file b.txt:
30: Open FNameB For Output As FNum
'Forming matrix B and its writing into file b.txt:
31: For j = 1 To n
32:    strC = ""
33:    For i = 1 To m
34:       B(j, i) = A(m + 1 - i, n + 1 - j)
35:       strC = strC & Str(B(j, i)) & " "
36:    Next i
37:    Print #FNum, strC
38: Next j
'Closing file b.txt:
39: Close FNum
40: LabelMessage.Caption = _
        "Account is terminated. Input..."     'message
End Sub
```

Property `Text` of element `TextBoxA` is a variable whose value coincides with the full name of the text file, containing values of *m* and *n* and matrix **A**; this full name is entered into the *a* text box of the UserForm1 form during the program usage. Property `Text` of element `TextBoxB` is a variable whose value coincides with the full name of the text file for matrix **B**; this full name is entered into the *b* text box. When executing the program, operators `1` and `2` specify the files' full names. Operators `4` and `30` open the corresponding files.

When executing operator `40`, string

```
"Account is terminated. Input..."
```

is assigned to property `LabelMessage.Caption`, i.e., this string is put into the message place of the form.

We can start the developed program by clicking on arrow ► of the VB window. As a result, the form (Fig. 1.28) is loaded. Fig. 1.30 shows the form's state upon finishing the calculation, i.e., after inputting

```
c:\Users\usr\texts\a.txt
c:\Users\usr\texts\b.txt
```

into text boxes *a* and *b*, respectively, and clicking on button *Account* (`usr` is the user name in these two full names).

Further, the calculation may be repeated for other names of the files and for other contents of the source data file. Upon termination of the series of calculations, we have to close the form, depicted in Fig. 1.30, by clicking on the little cross in the top right corner.

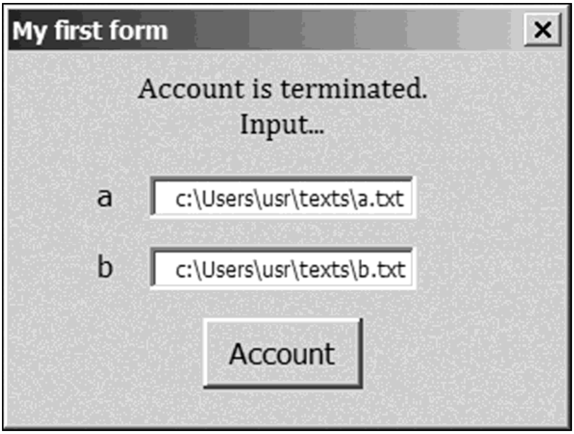
The source data file, which contains values of *m* and *n* and matrix **A**, can be created and/or edited by using the Notepad editor. We can view the calculation result by means of the same editor. Fig. 1.23 and 1.24 show the Notepad window with the source data and calculation result.

We can use element TextBox not only for input of information but also for output. In this case, property Text of the corresponding TextBox element (for example, property TextBox3.Text) must be in the left-hand side of the assignment operator.

We advise the reader to modify the form and the text of the last program so that *"Account is terminated. Input..."* is being put into the text box of the form.

We learned how to use the following three elements of the *Toolbox* window: Label, TextBox and CommandButton. Let us also consider element CheckBox, which is used in the following cases:

- an option should either be turned on or turned off;
- one of two alternatives must be chosen.



Account is terminated.
Input...

a c:\Users\usr\texts\a.txt

b c:\Users\usr\texts\b.txt

Fig. 1.30. The form upon termination of the matrix transposition

In the form, the CheckBox element looks like a little square field. At the click on this field, the check (tick) mark appears in it; at the repeated click, the check mark disappears.

Property Value of element CheckBox has value `True` or `False` as follows:

- `True` when the check mark is present in the field;
- `False` when the check mark is absent.

Let us expand the form, depicted in Fig. 1.28, by the CheckBox element placed on the left of the button and return to the program, which calculates the hypotenuse length.

The new version of the `ComButCalc_Click` program (p. 121) follows:

```
Private Sub ComButCalc_Click()
    Dim a As Single, b As Single, c As Single
    Dim s As String, out As String
    a = TextBoxA.Value
    b = TextBoxB.Value
    c = Sqr(a ^ 2 + b ^ 2)     'according to Pythagoras
    out = "c =" & Str(c)
    If CheckBox1.Value Then
        s = Str(a * b / 2)
        out = out & vbCrLf & "s =" & s
    End If
    LabelMessage.Caption = out
End Sub
```

In this program, `CheckBox1` is the CheckBox element's name. In the presence of the check mark, the triangle area is calculated in addition to the hypotenuse length.

Fig. 1.31 and 1.32 show the initial and final states of the form. To get the result, depicted in Fig. 1.32, we must fulfill the following operations:

1) put 3 and 4 into text boxes *a* and *b* of the form in Fig. 1.31;
2) set the check mark by clicking on element *s*;
3) click on the *Account* button.

Let us save the Excel workbook on the hard disk of the computer under name BookForm, having done *File* > *Save As* > *Browse*, and so on (p. 114). At that, the project (with the form), which is a part of the workbook, is saved too.

For returning to the project, the double click on line *UserForm1* in the project explorer window is required. We can delete the form, as well as the module, by means of the context menu. For that:

1) right click on the form name in the project explorer window;
2) in the open context menu, fulfill the *Remove* command;
3) click on the *No* button in the open window with a question about exporting the form before removing it.

We advise the reader to use the form with the CheckBox element for the matrix transposition and to modify the program on p. 123 so that:

- in the presence of the check mark, the matrix transposition relative to the main diagonal would be performed according to formula (1.2) on p. 109;
- in the absence of the check mark, the matrix transposition relative to the auxiliary diagonal would be performed according to formula (1.3).

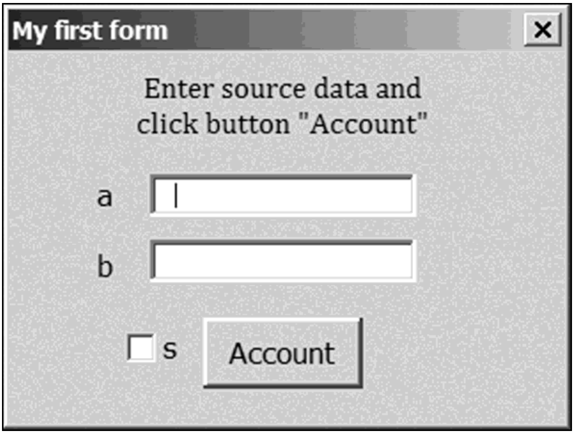
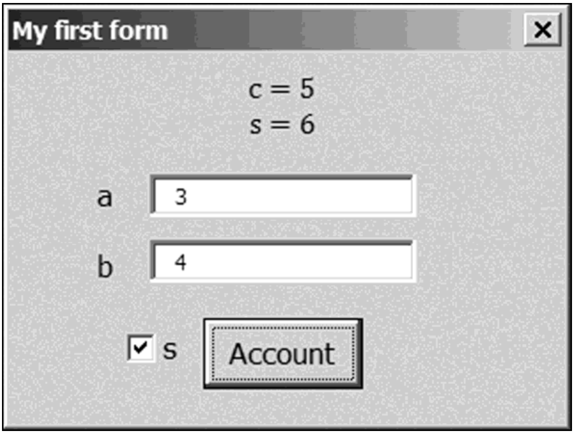


Fig. 1.31. The form after the program start



c = 5
s = 6

a 3

b 4

Fig. 1.32. The form upon finishing the calculation

1.24. Digression. Developing programs with the form in Microsoft Visual Studio

The programs, which were reviewed above and will be reviewed below, have the following drawback: we must equip our computer with tabular processor Excel for calculations by means of these programs. If we want to execute a program without using Excel, this program should be developed in Visual Basic Environment, which is not a part of Microsoft Office. Let us consider singularities of developing the program (project) with the form in Visual Basic Environment, which is a part of package Microsoft Visual Studio 2010.

Microsoft Visual Studio 2010 is installed on the computer in the standard way. To start Visual Studio, we fulfill the following operations on Windows Desktop: *Start > All Programs > Microsoft Visual Studio 2010 > Microsoft Visual Studio 2010*. The *Start Page* window is the result.

For setting the necessary operation mode of Visual Studio, we must fulfill the following operations:

1) click on the *New Project* hyperlink;

2) in the left area of the open *New Project* window, click on the plus sign against *Other Languages*;

3) in the open list, click on the plus sign against *Visual Basic*;

4) in the open list, click on line *Windows*;

5) in the list of the central area of the *New Project* window, click on line *Windows Forms Application*.

The necessary operation mode of Visual Studio is the result. The following information in the right area of the *New Project* window speaks about it: *A project for creating an application with a Windows user interface*.

Further, the project name and the folder, intended for the project, must be given by means of text boxes *Name* and *Location*. By default, the project and folder have the same name, *WindowsApplication1* (the digit in the name may be different).

After clicking on the *OK* button, the window of Visual Basic Environment is displayed. Blank form *Form1* is a part of this window. Fig. 1.33 shows the window after fulfilling *View > Toolbox*. To open the properties window, we must fulfill the following operations: *View > Other Windows > Properties Window*. The properties window (Fig. 1.34) appears in place of the *Toolbox* window.

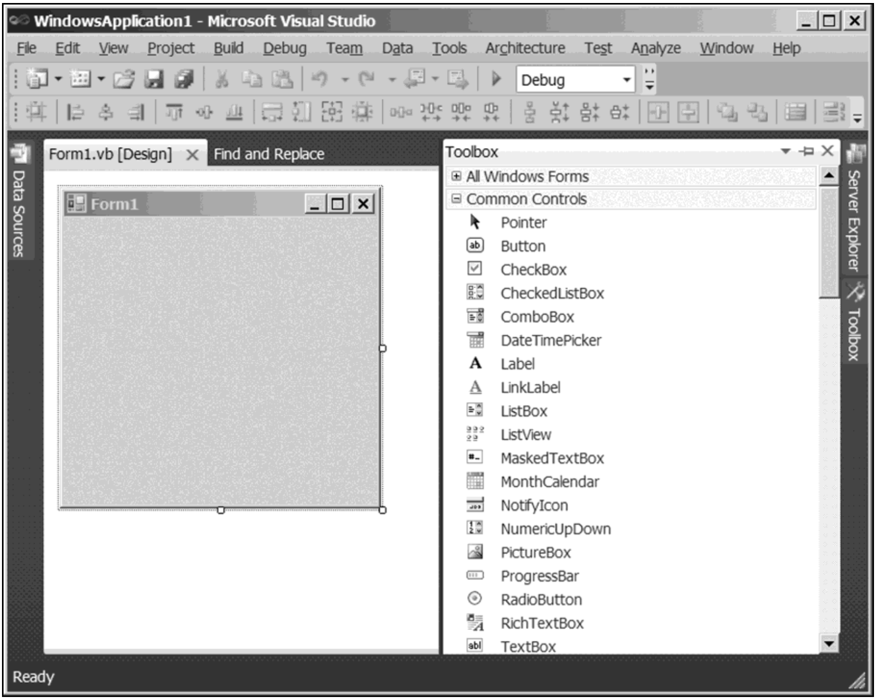


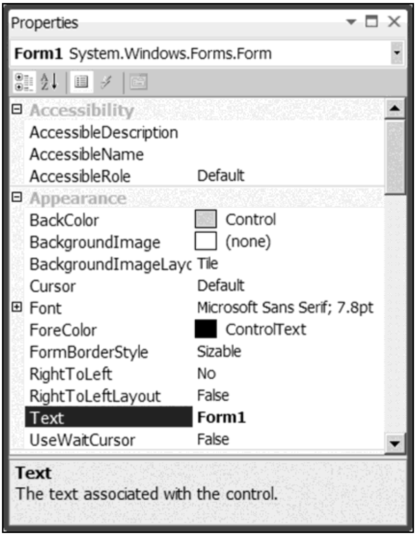
Fig. 1.33. The window of Visual Basic Environment (a part of Microsoft Visual Studio 2010) with the *Toolbox* window

The development of forms and corresponding programs is similar to the development described in the previous section.

The text of the programs, developed in Microsoft Visual Studio 2010, is close to the text of the programs, developed in Excel. For example, program `ComButCalc_Click` for calculating the hypotenuse length (p. 121) becomes as follows:

```
Public Class Form1
    Private Sub ComButCalc_Click(ByVal sender _
    As System.Object, ByVal e As System.EventArgs) _
    Handles ComButCalc.Click
        Dim a As Single, b As Single, c As Single
        Dim s As String
        s = TextBoxA.Text
        a = Val(s)
        s = TextBoxB.Text
        b = Val(s)
        c = Math.Sqrt(a ^ 2 + b ^ 2)
                                'according to Pythagoras
        s = Str(c)
        LabelMessage.Text = "c=" & s
    End Sub
End Class
```

The beginning and end of the program were generated automatically when we clicked twice on the button inserted into the form.



Properties

Form1 System.Windows.Forms.Form

⊟ Accessibility	
AccessibleDescription	
AccessibleName	
AccessibleRole	Default
⊟ Appearance	
BackColor	Control
BackgroundImage	(none)
BackgroundImageLayc	Tile
Cursor	Default
⊞ Font	Microsoft Sans Serif; 7.8pt
ForeColor	ControlText
FormBorderStyle	Sizable
RightToLeft	No
RightToLeftLayout	False
Text	**Form1**
UseWaitCursor	False

Text
The text associated with the control.

Fig. 1.34. The properties window as a part
of Visual Basic Environment

The project contains several files and folders. By default, it is located in the folder with the following full name:

```
c:\Users\usr\WindowsApplication1
```

where `usr` is the user name.

To save the current state of the project, we must fulfill *File* > *Save*. For returning to the project, we can use one of the following two ways:

- the double click on the *WindowsApplication1.vbproj* file in the project folder;
- the click on the *WindowsApplication1* hyperlink in area *Recent Projects* of the *Start Page* window of Visual Studio.

When we execute the program during its development, the debug version of the program executable file is created automatically. The full name of this file follows:

```
c:\Users\usr\WindowsApplication1\bin\Debug\ _
    WindowsApplication1.exe
```

Upon termination of the program development, we must create the release (working) version of the executable file. For that, we fulfill the following operations:

1) *Build* > *Configuration Manager*;
2) in the *Configuration Manager* window opened, enter *Release* into the *Active solution configuration* box by means of the drop-down list;
3) click on the *Close* button;
4) *Build* > *WindowsApplication1*.

The release version of the executable file is the result. Its full name follows:

```
c:\Users\usr\WindowsApplication1\bin\Release\ _
    WindowsApplication1.exe
```

The release version of the executable file is more efficient (in respect of the execution rate) in comparison with the debug version.

For loading the form, we must click twice on the program executable file in Windows Explorer. Further, we can use the form for the calculation.

If we want to transfer the program with the form to another user, it is enough to transfer the program executable file.

Let us return to programming in tabular processor Excel.

Chapter 2.
Programming in VBA

We review the main objects of the VBA programming language, which is the Visual Basic extension in the sense that VBA includes the VB constructs. The Excel table is considered as the user interface of macros.

Besides, we also consider creating Excel user-defined functions and working with Excel Macro Recorder, Personal Macro Workbook and the reference systems.

2.1. Loading the form from the Excel window. Running the program executable file

The user-defined forms, developed in Section 1.23, were loading from the VB window, for example, by clicking on arrow ► of the standard toolbar. However, it is desirable to load the form from the Excel window.

Below, we will consider the form-loading program.

Let us open the BookForm workbook of Excel (p. 127), go to the VB window and fulfill *Insert* > *Module*.

We enter the following simple program into the code window of the inserted module:

```
Sub LoadingForm()
    UserForm1.Show
End Sub
```

Here, `UserForm1` is the form name.

Now the form, pictured in Fig. 1.31, can be loaded from the Excel window as follows: *Developer* (or *View*) > *Macros* > line *LoadingForm* > *Run*. By fulfilling *Developer* (or *View*) > *Macros* > line *LoadingForm* > *Options* > …, we can appoint a combination of keys for loading the form.

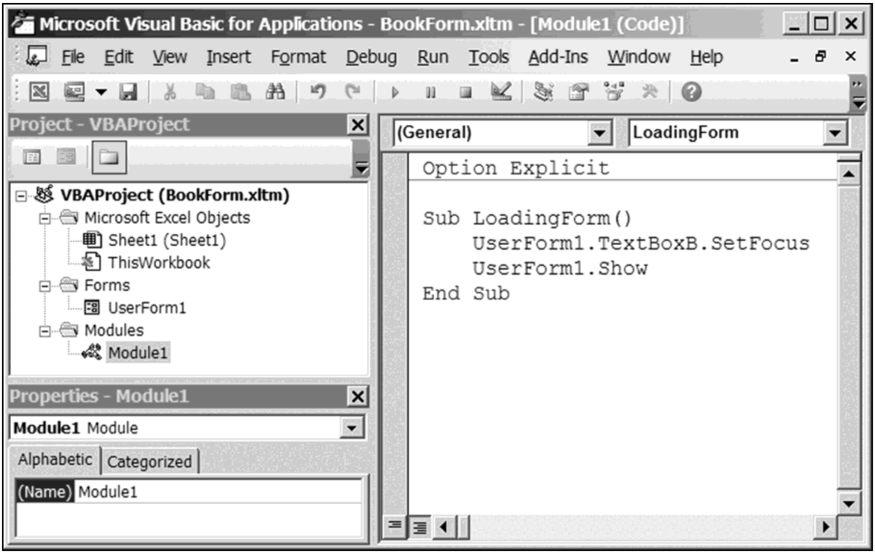
The loaded form may be used for transposing a numerical matrix or calculating the length of the hypotenuse of a right-angled triangle and its area.

If we want the blinking cursor to be, for example, in text box *b* of the loaded form, it is necessary to insert operator

```
UserForm1.TextBoxB.SetFocus
```

above operator `UserForm1.Show`. The `LoadingForm` program becomes as follows (Fig. 2.1):

```
Sub LoadingForm()
    UserForm1.TextBoxB.SetFocus
    UserForm1.Show
End Sub
```



```
Option Explicit

Sub LoadingForm()
    UserForm1.TextBoxB.SetFocus
    UserForm1.Show
End Sub
```

Fig. 2.1. The VB window with lines *UserForm1*
and *Module1* in the project explorer window

Let us save the Excel workbook with the `LoadingForm` program under the old name of BookForm.

The following program (in the code window) allows opening the Notepad window in Excel:

```
Sub StartEXE()
    Dim RetVal As Integer           'for function Shell
    RetVal = Shell("c:\Windows\notepad.exe",1)
End Sub
```

In this program, we see the call of the `Shell` function, as in the `Addition` program on p. 106.

To open the Notepad window, we must run the `StartEXE` program by fulfilling *Developer* (or *View*) > *Macros* > line *StartEXE* > *Run*. Besides, we can appoint a combination of keys for opening the Notepad window.

By using the `StartEXE` macro, we can run any executable file, in particular, created in Visual Basic Environment, which is a part of Microsoft Visual Studio (Section 1.24). For that, `c:\Windows\notepad.exe` must be replaced by the full name (without spaces) of this executable file.

2.2. Layout of the control elements on the Excel worksheet

The control elements of Section 1.23 can be placed on the Excel worksheet. For example, to create the form-loading button, we must fulfill the following.

1. In the Excel window with open workbook BookForm, fulfill *Developer* > *Insert* in area *Controls*. The window with control elements appears (Fig. 2.2).

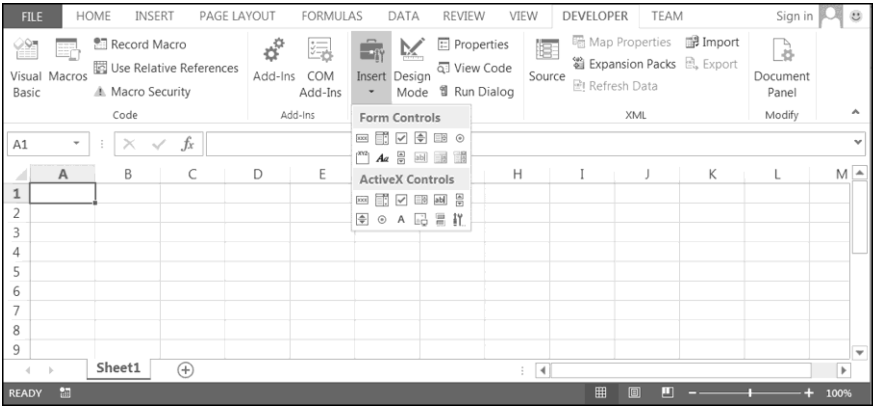


Fig. 2.2. The Excel window with control elements below the *Insert* button

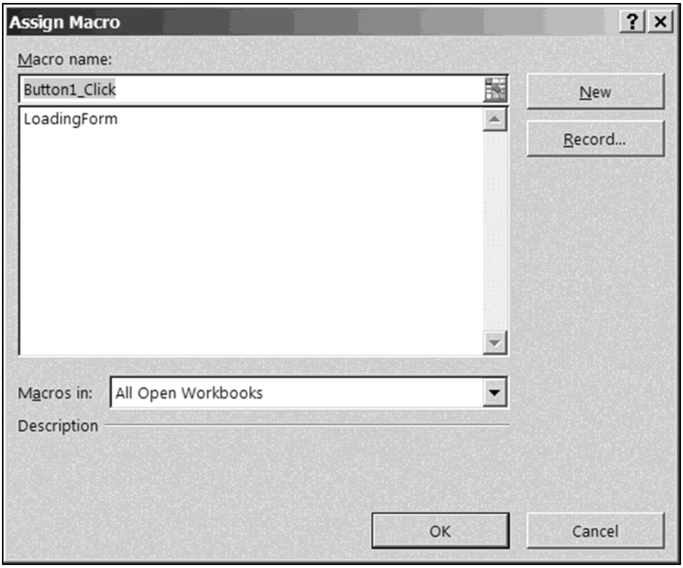
2. Click on the Button element in the *Form Controls* area of the last window, and then click on any cell of the Excel worksheet. The *Assign Macro* window opens (Fig. 2.3).

3. Click on the *New* button. The VB window opens (Fig. 2.4), and button *Button1* selected by 6 markers appears on the worksheet (Fig. 2.5).

4. Insert operator `UserForm1.Show` into the program blank in the code window depicted in Fig. 2.4. The following program is the result:

```
Sub Button1_Click()
    UserForm1.Show
End Sub
```

5. Click on area outside *Button1* to remove the button selection.



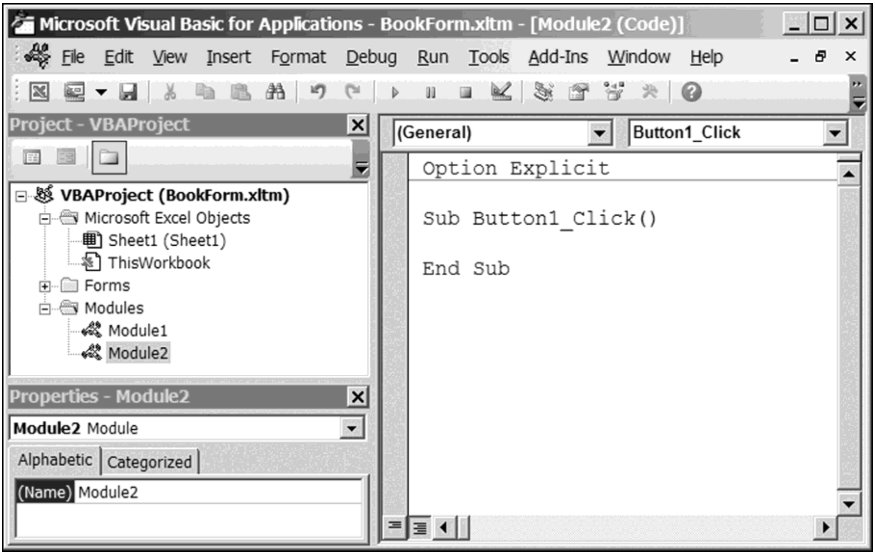
Macro name: Button1_Click

LoadingForm

Macros in: All Open Workbooks

Description

Fig. 2.3



(General) Button1_Click

```
Option Explicit

Sub Button1_Click()

End Sub
```

Module2 Module

(Name) Module2

Fig. 2.4. The VB window with *Module1* and *Module2* in the project window

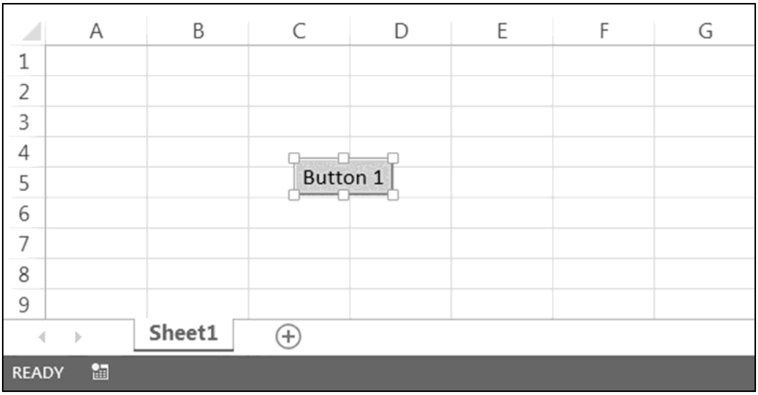


Fig. 2.5. The created button on the worksheet

Now, for loading the form pictured in Fig. 1.31, all we have to do is click on button *Button1*.

To change the inscription on the button:

1) select the button as follows:
- right click on it;
- then perform the *Cancel* command in the context menu opened;

2) edit the inscription as usual text;
3) click on area outside the button to remove its selection.

To remove the selected button, we must press the *Delete* key.

The Excel worksheet with the control elements plays the role of the form. For beauty of this "form", it is possible to remove gridlines from the Excel worksheet. For that, in the Excel window, we must fulfill *View* > turn off *Gridlines* in area *Show*.

If we want to use the button for opening the Notepad window, program `Button1_Click` must be changed as follows:

```
Sub Button1_Click()
    Dim RetVal As Integer          'for function Shell
    RetVal = Shell("c:\Windows\notepad.exe",1)
End Sub
```

In this program, we can replace `c:\Windows\notepad.exe` by the full name of any other executable file (p. 135). In this case, the above program is suitable for running this executable file from the Excel window.

2.3. User-defined functions of Excel

Our study of Visual Basic began with the Pythagoras program (p. 17) intended for calculating the hypotenuse length. Let us consider the following function for solving this simple task:

```
Function Hypotenuse(a, b)
    Hypotenuse = Sqr(a ^ 2 + b ^ 2)
End Function
```

This function declaration should be entered into the code window after inserting a module, for example Module1, into the active Excel workbook. As a result, the Hypotenuse function appears in the *User Defined* category of the Excel functions library. To verify this, we must fulfill the following two operations:

1) click on the *fx* button of the Excel formula bar;

2) in the *Insert Function* window opened (the first window of Function Wizard), enter *User Defined* into box *Or select a category* by means of the drop-down list.

We see line *Hypotenuse* in list *Select a function* (Fig. 2.6), i.e., the considered function is available in category *User Defined* of the Excel functions library.

Let us interrupt the operation of Function Wizard by clicking on the *Cancel* button in the *Insert Function* window.

The created function is used, for example, as follows:

1) into two cells on the Excel worksheet, enter the lengths of the triangle's legs, for example, 30.02 and 40 into A2 and B2, respectively;

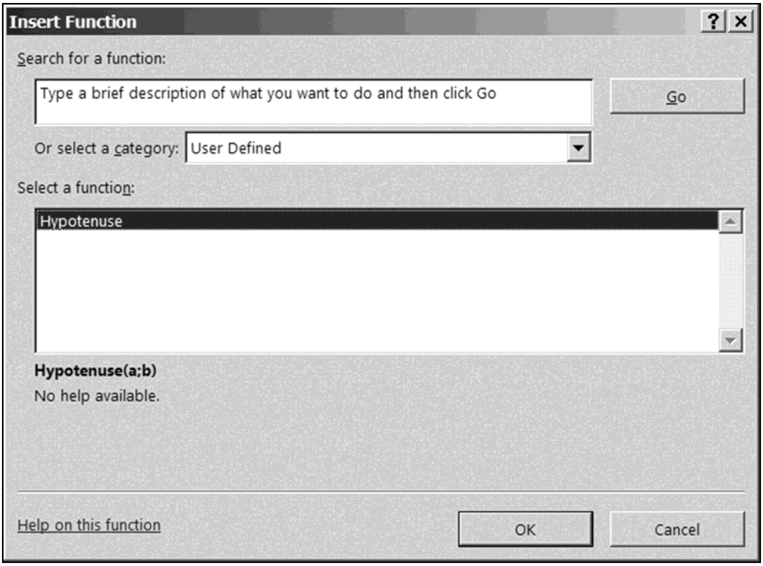
2) select a cell for the hypotenuse length, for example, C2;

3) in the Excel formula box after =, put the Hypotenuse function whose arguments are the addresses of the cells with the lengths of the triangle's legs (we must type a semicolon between the arguments instead of a comma accepted in Visual Basic);

4) calculate in one of the following two ways:

- by clicking on the tick button of the Excel formula bar;
- by pressing the *Enter* key.

As a result, the hypotenuse length appears in the selected cell (Fig. 2.7).



Insert Function

Search for a function:

Type a brief description of what you want to do and then click Go

Or select a category: User Defined

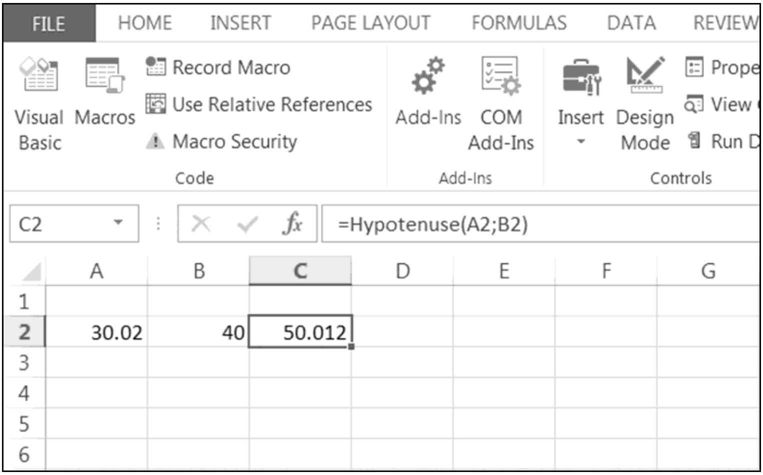
Select a function:

Hypotenuse

Hypotenuse(a;b)

No help available.

Fig. 2.6. The first window of Function Wizard



C2 =Hypotenuse(A2;B2)

	A	B	C	D	E	F	G
1							
2	30.02	40	50.012				
3							
4							
5							
6							

Fig. 2.7. The use of the `Hypotenuse` function

While using Function Wizard, we have to click on *OK* in the *Function Arguments* window (the second window of Function Wizard) for calculation.

The second window of Function Wizard is intended for inputting arguments of a function, in particular, of a user-defined function. This window appears as a result of clicking on *OK* in the first window of Function Wizard (Fig. 2.6). We will use the second window of Function Wizard in Section 2.15 (Fig. 2.32).

User-defined functions are used in Excel formulas, just like built-in functions.

The `Hypotenuse` function can be considered as the user-defined function of Visual Basic (p. 77). Therefore, we can use the `Hypotenuse` function in a program as follows:

```
Sub Pythagoras()
    Dim a As Single
    Dim b As Single
    Dim c As Single
    a = 3
    b = 4
    c = Hypotenuse(a, b)
End Sub
```

This text must be put into a new module, for example, by name Module2.

By executing the last program step-by-step, we can verify its operational capability.

In Excel, when transposing a matrix (cell range) relative to the main diagonal (Section 1.21), its rows and columns interchange their positions. For such transposition, Excel has the built-in TRANSPOSE function, which is available in category *Lookup & Reference* of the functions library. It is an example of the function returning an array of numbers, instead of one number (the SQRT function returns a number, see p. 17).

The TRANSPOSE function is used as follows:

1) enter a matrix (containing, for example, 3 rows and 4 columns) into Excel cells;

2) select the cell range (containing 4 rows and 3 columns) for the transposition result;

3) into the Excel formula box after =, enter the TRANSPOSE function whose argument is the source range (containing 3 rows and 4 columns);

4) calculate in one of the following two ways:

- by clicking on the tick button of the Excel formula bar when keys *Ctrl* and *Shift* are simultaneously pressed;
- by pressing the *Enter* key when keys *Ctrl* and *Shift* are simultaneously pressed (that is, by pressing *Ctrl* + *Shift* + *Enter*).

The transposition result appears in the selected cell range.

Function Wizard simplifies the use of the TRANSPOSE function. For transposing the matrix, we have to click on *OK* in the *Function Arguments* window when keys *Ctrl* and *Shift* are simultaneously pressed.

It is interesting to create a user-defined function for transposing a matrix (cell range) relative to its auxiliary diagonal according to formula (1.3) on p. 109. This function's declaration follows:

Listing 2.1

```
Function TRANSPOSEA(massive As Variant) As Variant
    Dim m As Integer, n As Integer
    Dim i As Integer, j As Integer
    Dim R() As Variant              'resulting matrix
    m = massive.Rows.Count          'quantity of rows
    n = massive.Columns.Count       'quantity of columns
    ReDim R(1 To n, 1 To m)         'specification of size
    For j = 1 To n
        For i = 1 To m
            R(j, i) = massive(m + 1 - i, n + 1 - j)
        Next i
    Next j
    TRANSPOSEA = R
End Function
```

Formal parameter `massive` may be considered as a record (Section 1.18). More precisely, `massive` is a variable of the Range type, but we will talk about it later.

Upon putting Listing 2.1 into the code window, the TRANSPOSEA function appears in category *User Defined* of the Excel functions library.

TRANSPOSEA is used as the TRANSPOSE function:

1) enter a matrix (intended for transposing relative to its auxiliary diagonal) into Excel cells, for example, A1:D3;

2) select the cell range for the transposition result, for example, A5:C8;

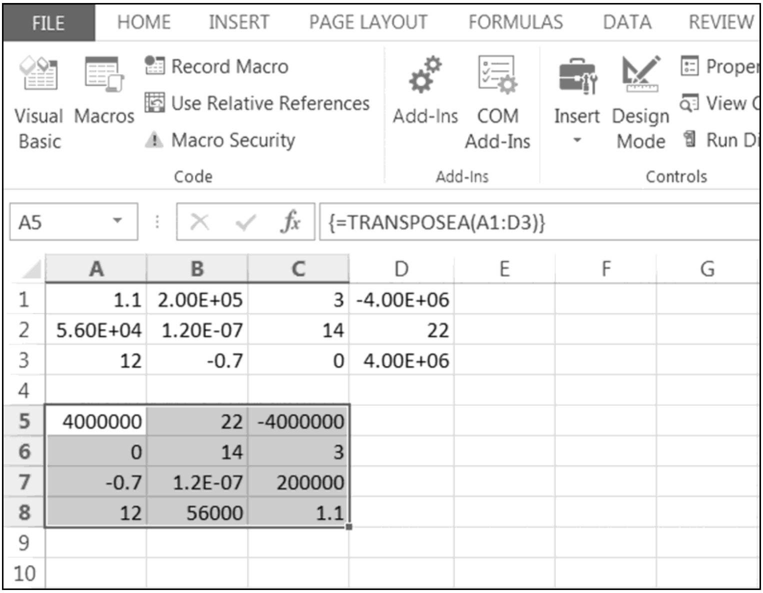
3) into the Excel formula box, enter

```
=TRANSPOSEA(A1:D3)
```

4) calculate in one of the following two ways:

- by clicking on the tick button of the Excel formula bar when keys *Ctrl* and *Shift* are simultaneously pressed;
- by pressing *Ctrl* + *Shift* + *Enter*.

The result of transposing the matrix relative to its auxiliary diagonal appears in the selected A5:C8 range (Fig. 2.8).



A5

{=TRANSPOSEA(A1:D3)}

	A	B	C	D	E	F	G
1	1.1	2.00E+05	3	-4.00E+06			
2	5.60E+04	1.20E-07	14	22			
3	12	-0.7	0	4.00E+06			
4							
5	4000000	22	-4000000				
6	0	14	3				
7	-0.7	1.2E-07	200000				
8	12	56000	1.1				
9							
10							

Fig. 2.8. The result of the TRANSPOSEA function usage

While using Function Wizard for the matrix transposition, we have to click on *OK* in the *Function Arguments* window when keys *Ctrl* and *Shift* are simultaneously pressed.

It is interesting to compare the TRANSPOSEA function with the program of Section 1.22. By the number of operators, the function is four times shorter than the program. Besides, the function can be used when creating Excel macros by means of Excel Macro Recorder (Section 2.5).

Note that the name of the formal parameter of the TRANSPOSEA function can differ from `massive`, for example, name `diapason` may be used.

When operating with the matrices containing a large number of columns, it is convenient to use the R1C1 reference style, in which the Excel columns are numbered by natural numbers instead of letters.

For setting the R1C1 reference style, we must fulfill the following operations:

1) click on the *File* button in Excel;
2) *Options* > *Formulas*;
3) turn on option *R1C1 reference style* in area *Working with formulas* of the *Excel Options* window;
4) click on the *OK* button.

2.4. Two methods for developing Excel macros

An Excel macro is a set of operators, which can be executed automatically. The macro is written in a programming language called Visual Basic for Applications or VBA.

The VBA programming language includes the reviewed constructs of Visual Basic.

The following two methods are used for creating macros:

1) programming in Visual Basic Environment;
2) creating by means of Excel Macro Recorder.

We used the first method in the previous sections, in particular, for developing two macros for loading the UserForm1 form (Sections 2.1 and 2.2).

The second method is simpler but less universal in comparison with the first method; it is normally considered when learning Excel. We will consider the second method because Excel Macro Recorder is an excellent helper in the first method for creating macros: if we do not know how to write down any action (or set of actions) in VBA, we have to create a macro for performing this action (or set of actions) by using Excel Macro Recorder and then study the VBA code generated automatically.

This is how we will develop:

- the operator blocks for automatic creation of graphs in Listings 3.10, 3.13 and 3.18;
- operator `12` in Listing 6.15 for the horizontal center alignment of the Excel cell content.

Excel Macro Recorder also facilitates the use of the first method for creating macros. The fact is that the program in VBA can be created not from scratch, but by starting from a prototype, which was created by means of Excel Macro Recorder.

It is precisely in this way that we will develop the `graph` subroutine for automatic creation of graphs (Section 4.8).

Creating macros by editing the prototype leads to a considerable “thought saving”.

2.5. Excel Macro Recorder

Excel Macro Recorder operation is similar to recording by means of a video-disk recorder. Excel Macro Recorder:

- records the operations being fulfilled by the user in the Excel window;
- then transforms these operations into a set of VBA operators, that is, into a program.

By means of Excel Macro Recorder, we will create a macro for performing the following operational sequence:

1) removal of gridlines from the Excel worksheet;
2) setting the R1C1 reference style;
3) assignment of the Currency format to all cells on the worksheet;
4) selection of the R14C5 cell, that is, E14.

For creation of the macro, we fulfill the following operations.

1. *Developer* > *Record Macro* (in area *Code*) or *View* > arrow *Macros* (in area *Macros*) > *Record Macro*.

2. Enter a name of the macro and other information into text boxes of the *Record Macro* window opened:

Macro name: *MR*

Shortcut key: *Ctrl+m* (it will be used for starting the macro execution)

Store macro in: *This Workbook* (taken from the drop-down list)

Description: *Result of using Excel Macro Recorder*

3. In the window with the filled text boxes (Fig. 2.9), click on button *OK*. At that, the *Record Macro* button changes its status to *Stop Recording*. Macro record is started.

4. Remove gridlines from the active Excel worksheet, for example, by name Sheet1: *View* > turn off *Gridlines* in area *Show*.

5. Set the R1C1 reference style as follows:

1) click on the *File* button;
2) *Options* > *Formulas*;
3) turn on the *R1C1 reference style* option in area *Working with formulas* of the *Excel Options* window;
4) click on the *OK* button.

6. Assign the Currency format to all cells on the active worksheet as follows:

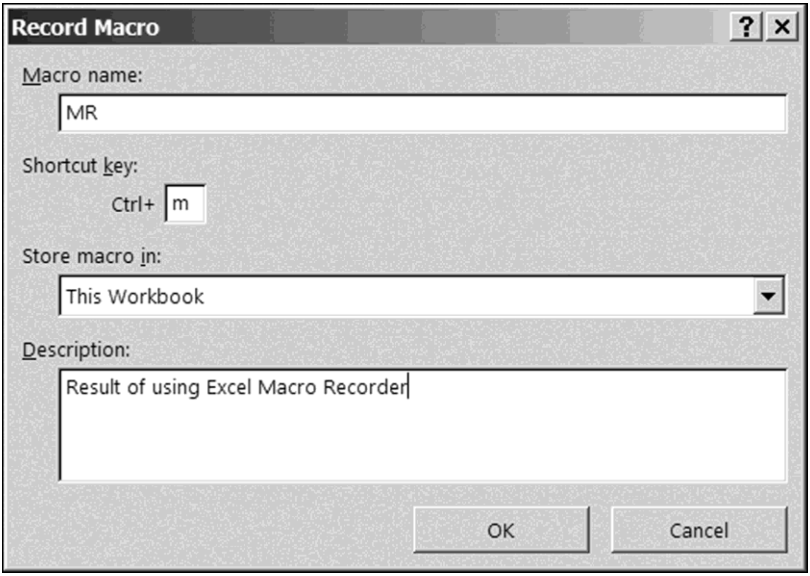
1) select all cells on the worksheet by clicking on the intersection of the top line (with numbers of columns) and the left column (with numbers of rows);

2) activate the *Home* tab;

3) in area *Number*, set the *Currency* format by using the drop-down list.

7. Select the R14C5 cell by clicking on it.

8. *Developer* > *Stop Recording* (in area *Code*) or *View* > arrow *Macros* (in area *Macros*) > *Stop Recording*.



Macro name: MR

Shortcut key: Ctrl+ m

Store macro in: This Workbook

Description: Result of using Excel Macro Recorder

Fig. 2.9. The *Record Macro* window before clicking on the *OK* button

The record is finished; macro MR is the result.

Before verifying the operational capability of the created macro, we return to the A1 reference style. For that:

1) click on the *File* button;

2) *Options* > *Formulas*;

3) turn off the *R1C1 reference style* option in area *Working with formulas* of the *Excel Options* window;

4) click on button *OK*.

Before starting the macro, let us create a worksheet, for example Sheet2, for which operations written in the macro must be performed. For creating Sheet2, we use the *New Sheet* command, depicted by the round plus icon in the bottom part of the Excel window (see ⊕ in Fig. 2.5).

For starting the MR macro from the Excel window, one of the following two ways is used:

- simultaneous pressing keys *Ctrl* and *m*;
- *Developer* (or *View*) > *Macros* > line *MR* > *Run*.

When executing the macro, the computer repeats actions enumerated in items 4 — 7.

After restoring the A1 reference style, let us save the Excel workbook with the MR macro (which is the result of using Excel Macro Recorder) under the name of BookMacrorecorder. During the save, we have to set the following file type: *Excel Macro-Enabled Workbook*.

For removing the MR macro, we must fulfill the following operations: *Developer* (or *View*) > *Macros* > line *MR* > *Delete* > *Yes*. However, we will not delete the macro because it will be required in the next sections.

2.6. VBA code generated by Excel Macro Recorder and its editing

In the previous section, we created the VBA macro corresponding to the sequence of Excel operations performed with Excel Macro Recorder turned on.

For displaying the macro text, we must fulfill the following in the Excel window with workbook BookMacrorecorder: *Developer* (or *View*) > *Macros* > line *MR* > *Edit*. At that, the VB window appears with the code window containing the required text.

The macro code follows:

Listing 2.2

```
Sub MR()
'
' MR Macro
' Result of using Excel Macro Recorder
'
' Keyboard Shortcut: Ctrl+m
'
    ActiveWindow.DisplayGridlines = False
    Application.ReferenceStyle = xlR1C1
    Cells.Select
    Selection.NumberFormat = "#,##0.00$"
    Range("E14").Select
End Sub
```

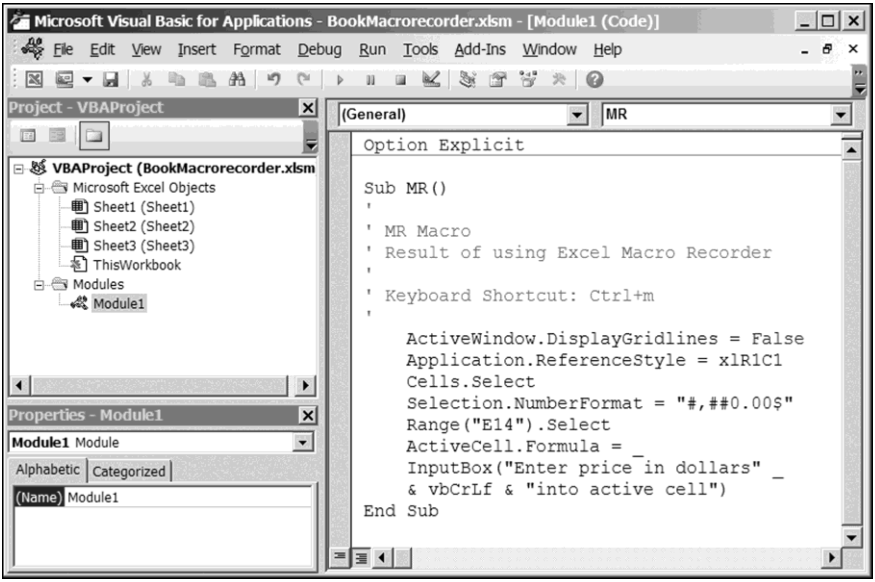
There is the possibility of editing this code created by using Excel Macro Recorder.

Let us assume that we want to set content of cell R14C5 (that is, E14) by means of the standard window. For that, we type the following operator above the last line of Listing 2.2:

```
ActiveCell.Formula = _
InputBox("Enter price in dollars" _
& vbCrLf & "into the active cell")
```

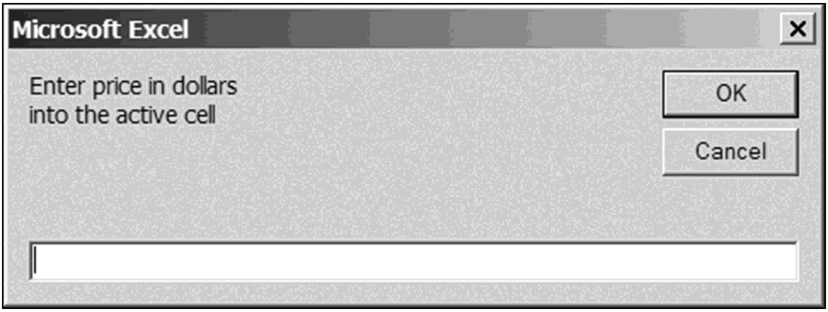
Fig. 2.10 shows the expanded text of the MR macro.

Let us create the Sheet3 worksheet and start the MR macro with the additional operator when Sheet3 is active. At that, the window represented in Fig. 2.11 appears.



```
Option Explicit

Sub MR()
'
' MR Macro
' Result of using Excel Macro Recorder
'
' Keyboard Shortcut: Ctrl+m
'
    ActiveWindow.DisplayGridlines = False
    Application.ReferenceStyle = xlR1C1
    Cells.Select
    Selection.NumberFormat = "#,##0.00$"
    Range("E14").Select
    ActiveCell.Formula = _
    InputBox("Enter price in dollars" _
    & vbCrLf & "into active cell")
End Sub
```

Fig. 2.10. The VB window with the macro text after typing the additional operator



Enter price in dollars
into the active cell

Fig. 2.11. The standard window for inputting information into the selected cell

We put a value, for example 12, into text box *Enter price in dollars into the active cell*. After clicking on the *OK* button, the text box content appears in cell R14C5 (Fig. 2.12). The dollar sign in front of 12.00 is because of the Currency format of the cell.

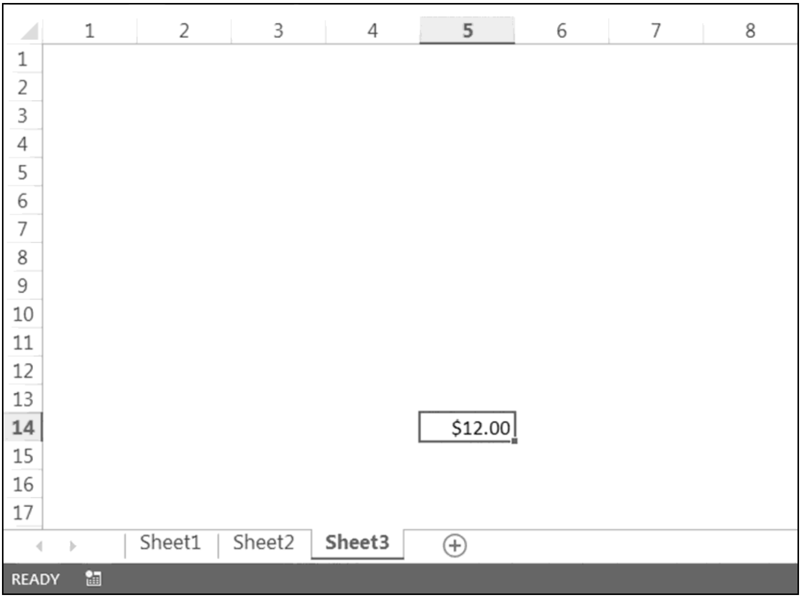


Fig. 2.12. The result of the `MR` macro execution

Let us restore the A1 reference style.

Further, we will consider constructs of VBA, which are absent in Visual Basic.

2.7. Objects and events

All modern programming languages (in particular, Visual Basic and VBA) are object- and event-oriented.

The object orientation is based on partitioning the subject area (for which we are developing a program) and clustering the parts.

- The program’s part, which corresponds to the cluster, is named ***object***.
- The object’s characteristics are named ***properties***.
- Actions performed over the object are named ***methods***.

Examples of the subject area are problems of modeling financial risks, semiconductor devices and evolution of stars (p. 47), as well as Visual Basic and VBA themselves.

Examples of the object are the Visual Basic objects, which were considered in Section 1.23: UserForm, Label, TextBox, CommandButton and CheckBox.

The event orientation is based on the following concepts:

- “event” — the object’s qualitative change, which follows from work of the user or computer;
- “handler for event” — the command set, executed by the computer when the event occurs.

An example of the event is the click on the *Account* button in the user-defined form, for example, depicted in Fig. 1.28. Because of handling this event, the ComButCalc_Click program execution is started.

The VBA programming language is intended for creating programs in the Microsoft Office applications, such as word processor Word, tabular processor Excel, technical editor Visio, database management system Access, etc. VBA differs from Visual Basic in the presence of specific objects of Microsoft Office and of its applications.

We will be interested in the so-called Excel objects intended for Excel VBA Programming. Examples of such objects are Workbook, Worksheet, Range and ActiveCell.

Operators intended for work with an object make the following:

- setting object properties;
- returning object properties into the program;
- applying object methods.

The syntax of ***setting object properties*** follows:

```
object.property = expression
```

where `expression` is an arithmetic or logical expression or string. The computer executes this operator as the normal assignment operator:

1) calculates the value of `expression`;
2) assigns this value to property `object.property`.

The syntax of ***returning object properties*** follows:

```
object.property
```

Property `object.property` may be a part of operators, in particular, it may be in the right-hand side of the assignment operator, i.e., the property is similar to built-in functions of Visual Basic. Often, `object.property` itself is an object.

Note that not all object properties can be returned and set. There are properties, which can be only returned or set. To study possibilities of this or that object property, we must use the reference systems, which are started by pressing the *F1* and *F2* keys when the VB window is active (Sections 1.6 and 2.12).

The operator of ***applying object methods*** has the following form:

```
object.method
```

The operator of applying the Add method is the exception. It creates a new object, `subobject`, and adds it to `object`. The syntax of this operator follows:

```
[Set variable = ]object.Add
```

In this syntax, `Set` is the keyword, `variable` is a variable of the same data type as `subobject`.

Applying the Add method is similar to calling the MsgBox procedure of Visual Basic, which is both a subroutine and function.

An object hierarchy exists. The highest in the hierarchy of the application is the Application object, i.e., all other objects “are included” in it. The Application object reminds the Russian nested doll, but (unlike the nested doll) several objects may be included in each object.

The full object name is a sequence of the object names separated by a point, at that, this sequence begins with Application. For example,

```
Application.Workbooks("Archive").Worksheets("Cod"). _
    Range("A1")
```

in VBA for Excel is the full name of the `Range("A1")` object or the reference to the A1 cell on the Cod worksheet of the Archive workbook.

The use of the full object name is not necessary. Often, we can use the incomplete name, i.e., without names of the objects activated at present. For example, if the Archive workbook is active, then the full name of the `Range("A1")` object may be shortened as follows:

```
Worksheets("Cod").Range("A1")
```

As before, this is the reference to the A1 cell on the Cod worksheet of the Archive workbook.

In the above VBA notations, strings are in the parentheses. These strings may be compound.

An example of using compound strings is the following operator:

```
Range("G" & CStr(CInt(Now - #1 Jan 2000#))).Select
```

Because of its execution, the following cell is selected on the active Excel worksheet: the intersection of the G column and the row whose number is equal to the number of days from the century beginning.

We advise the reader to do the following:

1) type the last operator above line `End Sub` of program `Century_20` (p. 25);
2) execute the obtained program;
3) pay attention to the position of the Excel cell activated.

We see that the object construct is similar to the record (Section 1.18). Therefore, as the first approximation, the object can be considered as the built-in record whose creation operator (the `Type` operator) is hidden from the program developer.

In addition to properties and methods, some objects of VBA are also characterized by events. However, it does not relate to the main Excel objects, which will be considered below.

2.8. Object Application

It was mentioned above that the Application object occupies the top level in the object hierarchy of Excel. This means that the Application object controls the settings of the application, i.e., such settings as are in window *Excel Options* (to open this window, we must fulfill *File* > *Options*). Operator

```
Application.ReferenceStyle = xlR1C1
```

in the `MR` macro text, which was created by means of Excel Macro Recorder (see Listing 2.2 on p. 148), speaks about this role of the Application object.

However, the Application object not only changes parameters of Excel. If we want to use Excel functions when programming in VBA, the Application object is also necessary for us.

Let us use built-in functions AVERAGE and SUM of Excel for processing range A1:A4 on the Sheet1 worksheet.

1. Insert a module into the active Excel workbook. Enter program

Listing 2.3

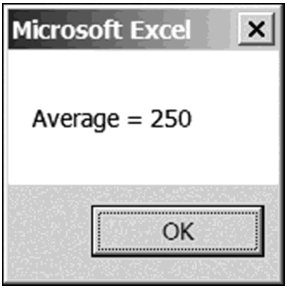
```
Sub BuiltinFunctions()
    Dim W As Single
1:  W = Application. _
        Average(Worksheets("Sheet1").Range("A1:A4"))
    MsgBox "Average = " & CStr(W)
2:  W = Application. _
        Sum(Worksheets("Sheet1").Range("A1:A4"))
    MsgBox "Sum = " & CStr(W)
End Sub
```

into the code window.

2. Go to worksheet Sheet1 of the active workbook.
3. Enter number 100 into cell A1, 200 into A2, 300 into A3 and 400 into A4.
4. Run the `BuiltinFunctions` macro. The window with the average of the entered numbers appears (Fig. 2.13).

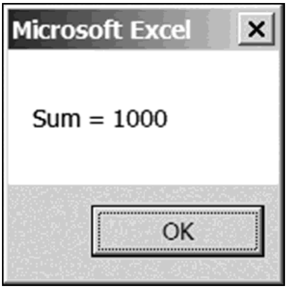
5. To continue the program execution, click on the *OK* button. The window containing the sum of the entered numbers appears (Fig. 2.14).

6. To terminate the program execution, click on the *OK* button.



Average = 250

Fig. 2.13. The window with the first processing result



Sum = 1000

Fig. 2.14. The window with the second processing result

The built-in Excel functions are ***properties*** of the Application object. When executing operators `1` and `2`, properties Average and Sum return with parameter `Worksheets("Sheet1").Range("A1:A4").`

In addition to Excel functions, the Application object has other properties. From the long list of the properties, we will consider the following.

- ActiveWorkbook — the active workbook.
- ActiveSheet — the active worksheet of the active workbook (in our opinion, ActiveWorksheet would be a more suitable name of this property).
- ActiveCell — the active cell on the active worksheet of the active workbook.

As an example of returning the ActiveCell property, let us consider the following program, which sets the italic font for the active cell and puts text *Report for May* into this cell.

```
Sub ItalicFont()
    With Application.ActiveCell
        .Font.Italic = True
        .Value = "Report for May"
    End With
End Sub
```

The `With` operator was defined on p. 91.

Properties ActiveWorkbook, ActiveSheet and ActiveCell are objects. We will consider them in greater detail in the next three sections.

❖ Calculation — the calculation mode (see area *Calculation options* of the *Excel Options* window after fulfilling *File* > *Options* > *Formulas*).

The main values of this property are:

1) `xlCalculationAutomatic` — the automatic calculation: recalculation according to formulas is performed automatically when Excel cells' contents change; it is the default operation mode of Excel;

2) `xlCalculationManual` — the manual calculation, for example, when pressing the *F9* key.

As an example of setting the Calculation property, let us consider the following operator intended for setting the automatic calculation mode:

```
Application.Calculation = xlCalculationAutomatic
```

❖ Dialogs — the collection of the Excel dialog boxes.

In VBA, word "collection" means a group of single-type objects.

By means of the Dialogs property, it is possible to display the Excel dialog boxes. For example, the execution of operator

```
Application.Dialogs(xlDialogOpen).Show
```

leads to displaying window *Open* familiar to us; `xlDialogOpen` is the parameter of the Dialogs property.

In the last operator, `Application.Dialogs(xlDialogOpen)` is the object (of the `Balloon` type), `Show` is the object's method.

We already encountered the `Show` method in Sections 2.1 and 2.2. It is operator

```
UserForm1.Show
```

figuring in the programs of loading the user-defined form.

Let us use the `Show` method for saving an Excel workbook.

1. Above the last line of the BuiltinFunctions macro (p. 154), insert the following operator of displaying the document saving window:

```
Application.Dialogs(xlDialogSaveAs).Show
```

The macro has the following expanded text:

```
Sub BuiltinFunctions()
    Dim W As Single
1:  W = Application. _
        Average(Worksheets("Sheet1").Range("A1:A4"))
    MsgBox "Average = " & CStr(W)
2:  W = Application. _
        Sum(Worksheets("Sheet1").Range("A1:A4"))
    MsgBox "Sum = " & CStr(W)
    Application.Dialogs(xlDialogSaveAs).Show
End Sub
```

2. Go to the Excel window.
3. Enter arbitrary numbers, for example 100, 200, 300 and 400, into range A1:A4 on the Sheet1 worksheet.
4. Run the above macro (the windows, depicted in Fig. 2.13 and 2.14, will appear during the execution).
5. In the displayed *Save As* window, enter file name *BookApplication*.
6. Set file type *Excel Macro-Enabled Workbook* by means of drop-down list *Save as type*;
7. Click on the *Save* button.
8. Make sure that the obtained BookApplication workbook contains macro BuiltinFunctions and the file name has extension .xlsm.

On p. 108, we considered the order of tuning Windows Explorer to see file names with extension.

In the above ItalicFont macro, operator With was used for setting object properties. However, we can use this operator for applying object methods. For example, operator

```
Application.Dialogs(xlDialogSaveAs).Show
```

can be replaced by construct

```
With Application.Dialogs(xlDialogSaveAs)
    .Show
End With
```

The `With` operator may include both properties and methods. As an example, let us consider the following new version of the `BuiltinFunctions` macro:

```
Sub BuiltinFunctions()
    Dim W As Single
1:  W = Application. _
        Average(Worksheets("Sheet1").Range("A1:A4"))
    MsgBox "Average = " & CStr(W)
2:  W = Application. _
        Sum(Worksheets("Sheet1").Range("A1:A4"))
    MsgBox "Sum = " & CStr(W)
    With Application
        .ActiveCell.Value = "Report for May"
                                           'property Value
        .Dialogs(xlDialogSaveAs).Show      'method Show
    End With
End Sub
```

During the execution, text *Report for May* appears in the active cell.

From the list of ***methods*** of the Application object, we will consider the following: Quit, Calculate, OnTime.

❖ Quit — quit Excel.

The operator of applying the Quit method follows:

```
Application.Quit
```

❖ Calculate — the forced calculation.

As examples of using the Calculate method, let us consider the following operators:

```
Application.Calculate
Worksheets("Report7").Calculate
Worksheets("Report7").Range("A1:C10").Calculate
```

If mode “manual calculation” is set in Excel, then:

- the execution of the first operator leads to the calculation according to the formulas in all open Excel workbooks (which are represented by buttons on the taskbar of Windows Desktop); this execution is equivalent to pressing key *F9*;
- the execution of the second operator leads to the calculation on the `Report7` worksheet of the active workbook;

- the execution of the third operator leads to the calculation in the A1:C10 range on the `Report7` worksheet of the active workbook.

As we know, the execution of operator

```
Application.Calculation = xlCalculationManual
```

tunes Excel for the manual calculation.

To tune Excel for the manual calculation, we can also fulfill the following operational sequence in the Excel window: *File* > *Options* > *Formulas* > turn on *Manual* > *OK.*

❖ OnTime — the start of the macro at the given moment of time; the time and the macro name are the method parameters.

As an example of applying the OnTime method, let us consider the following macro, which writes the current time into cell A1 (on the active worksheet) after starting the macro.

Listing 2.4

```
Sub MyMacro()
1:  Range("A1") = Time
2:  Application.OnTime Now + TimeValue("00:00:01"), _
        "MyMacro"
End Sub
```

In the `MyMacro` macro, operator `1` writes the current time into the A1 cell. Operator `2` starts `MyMacro` one second after the current moment; therefore, operator `1` is executed every second, i.e., the A1 cell content is updated every second.

For the initial start of `MyMacro`, we must fulfill *Developer* (or *View*) > *Macros* > line *MyMacro* > *Run*.

Let us depict the time starting from the moment of opening the Excel workbook (containing the corresponding macro). For this purpose, we use name `Auto_Open` instead of `MyMacro`. The new version of the last program follows:

```
Sub Auto_Open()
1:  Range("A1") = Time
2:  Application.OnTime Now + TimeValue("00:00:01"), _
        "Auto_Open"
End Sub
```

We save the workbook, containing this macro, as a macro-enabled workbook, for example, by name BookOnTime. After that, we close the BookOnTime workbook.

Further, let us fulfill the following:

1) open the BookOnTime workbook;

2) to allow the macro to work, click on the *Enable Content* button of the *Security Warning* panel.

At that, the Auto_Open macro is started automatically. So is happened thanks to the macro name (Excel so is arranged).

After the auto start, the Auto_Open macro is working as MyMacro:

- every second, Auto_Open starts;
- therefore, every second, operator 1 writes the current time into cell A1 on the active worksheet.

For consolidating the material of this section, ***we advise the reader*** to understand the work of the following code, which contains the Auto_Open macro with two operators of applying the OnTime method.

Listing 2.5

```
Sub Auto_Open()
    Application.OnTime TimeValue("12:30:00"), "MyMa1"
    Application.OnTime TimeValue("12:31:00"), "MyMa2"
End Sub

Sub MyMa1()
    Range("G1") = 13.333
End Sub

Sub MyMa2()
    Range("G2") = Time
    Application.OnTime Now + TimeValue("00:00:01"), _
        "MyMa2"
End Sub
```

These three macros should be put into one module or different modules of the same Excel workbook.

2.9. Objects Workbook, Workbooks and ActiveWorkbook

Object Workbook follows right after object Application in the object hierarchy. It is easy to understand the following ***properties*** of the Workbook object.

❖ Name — the workbook name with its extension.

The possible extensions of the name follow:

- .xlsm if the workbook contains macros;
- .xlsx if macros are absent in the workbook, etc.

The file name together with its extension frequently is also called the file name. As a rule, it does not lead to any confusion.

❖ Path — the path to the workbook in the file system of Windows.

❖ FullName — the workbook name with its path and extension, i.e., the workbook full name.

Let us consider the following three ***methods*** of the Workbook object:

❖ Close — closing the workbook;

❖ Save, SaveAs — saving the workbook.

The SaveAs method differs from the Save method in that SaveAs has a list of optional parameters, which includes FileName, FileFormat and Password.

Examples of using the properties and methods of the Workbook object are given below, when considering objects of the Workbook data type. Here, “data type” has the same sense as in expression “variables (records) of the Session data type” on p. 91.

Objects of the Workbook type (“data” is omitted for brevity) have the properties and methods of the Workbook object.

The Workbooks object is an object containing all open Excel workbooks. This object is also named as the Workbooks collection.

In VBA,

```
Workbooks("Personnel_department")
```

is the Workbook type object corresponding to the open workbook by name Personnel_department.

Let us consider the main ***methods*** of the Workbooks object.

❖ Activate — activation of the specified workbook (from a number of the open Excel workbooks) when its first worksheet becomes active.

As an example of applying the Activate method, let us consider the following operator of activating the above Personnel_department workbook:

```
Workbooks("Personnel_department").Activate
```

❖ Add — the creation of a new workbook, which becomes active at once.

Applying the Add method is accompanied by return (into the program) of the created workbook, which is the Workbook type object. As an example, see operator `2` in the `NewBook` macro (p. 164).

Note that all collections have the Add method. Its application adds (to the collection) a new object of the corresponding type. In particular, a new object of the Workbook type is added to the Workbooks collection.

All collections, in particular Workbooks, have useful ***property*** by name Count that allows determining the quantity of objects in the collection.

An example of returning the Count property is in the following program:

Listing 2.6

```
Sub NumberofBooks()
    MsgBox Str(Workbooks.Count)
End Sub
```

The window, containing the number of open workbooks and the *OK* button, is displayed when executing the `NumberofBooks` macro. To terminate the execution, we must click on *OK*.

It was mentioned in the previous section that ActiveWorkbook is a property of the Application object, corresponding to the active workbook. As the Workbook type object, the ActiveWorkbook property has the properties and methods of the Workbook object.

To obtain examples of returning properties Name and FullName of the Workbook object and of applying the Save method, let us fulfill the following:

1) in the Excel window with the BookMacrorecorder workbook (p. 147), click on the *Enable Content* button of the *Security Warning* panel to allow the `MR` macro to work;

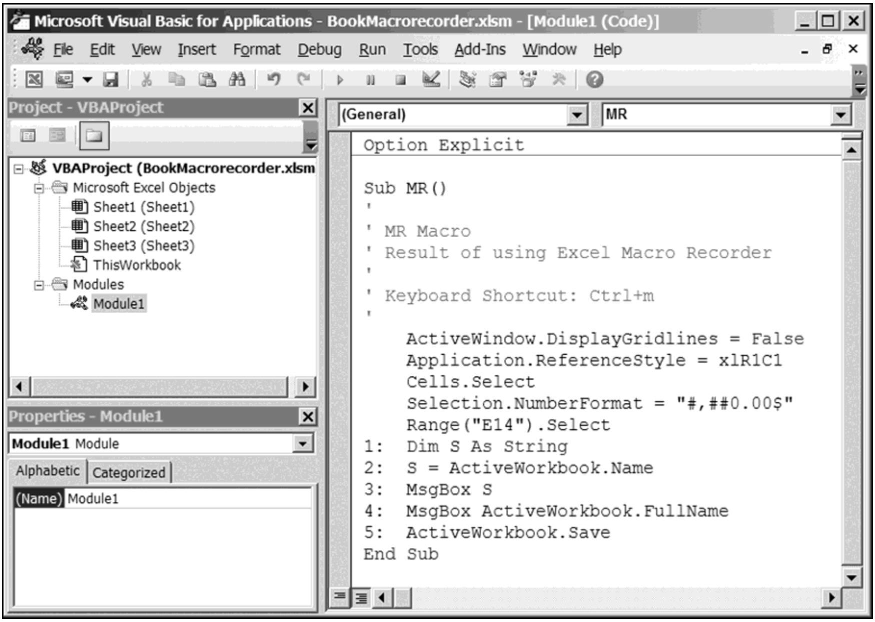
2) *Developer* (or *View*) > *Macros* > line *MR* > *Edit*;

3) put operator block

```
1:  Dim S As String
2:  S = ActiveWorkbook.Name
3:  MsgBox S
4:  MsgBox ActiveWorkbook.FullName
5:  ActiveWorkbook.Save
```

above the last line of the MR macro (Fig. 2.15);

4) start the MR macro execution, for example, by clicking on arrow ► in the VB window.



```
Option Explicit

Sub MR()
'
' MR Macro
' Result of using Excel Macro Recorder
'
' Keyboard Shortcut: Ctrl+m
'
    ActiveWindow.DisplayGridlines = False
    Application.ReferenceStyle = xlR1C1
    Cells.Select
    Selection.NumberFormat = "#,##0.00$"
    Range("E14").Select
1:  Dim S As String
2:  S = ActiveWorkbook.Name
3:  MsgBox S
4:  MsgBox ActiveWorkbook.FullName
5:  ActiveWorkbook.Save
End Sub
```

Fig. 2.15. The VB window with the macro text
after inserting the additional operator block

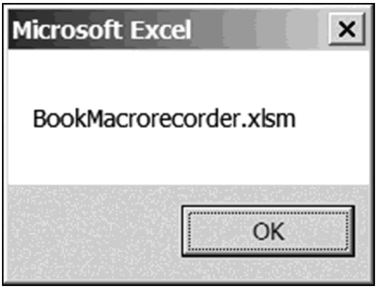
Because of executing operator 2, the S string has a value that is equal to the name of the active workbook. Operator 3 displays this string (Fig. 2.16).

Let us click on the *OK* button (in the window depicted in Fig. 2.16) to continue executing the MR macro. Operator 4 displays the active workbook's full name (Fig. 2.17). After clicking on the *OK* button, operator 5 saves the active workbook.

To obtain an example of returning the Path property of the Workbook object, we will execute the MR macro with the following form of the operator intended for displaying the full name:

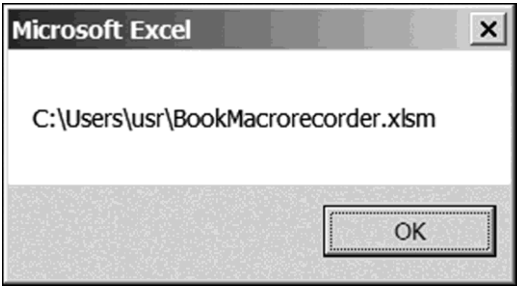
```
4:  MsgBox ActiveWorkbook.Path & "\" & _
        ActiveWorkbook.Name
```

The window, depicted in Fig. 2.17, appears during the execution, as in the case of the previous version of operator 4.



BookMacrorecorder.xlsm

Fig. 2.16. The window with the active workbook's name



C:\Users\usr\BookMacrorecorder.xlsm

Fig. 2.17. The window with the full name of the active workbook

In the following example, a workbook is created, a numerical value is entered into it, this workbook is saved and closed, and the Excel window is closed too.

1. Insert a module into a blank workbook, for example Book1, and put the following text into the code window (Fig. 2.18):

Listing 2.7

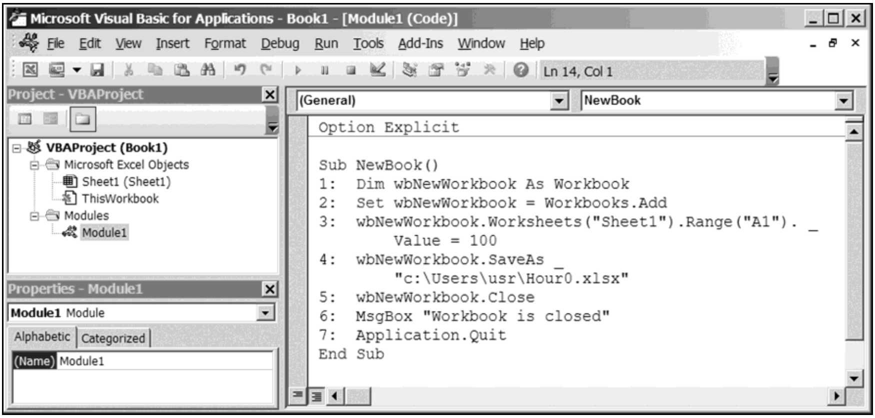
```
Sub NewBook()
1:   Dim wbNewWorkbook As Workbook
2:   Set wbNewWorkbook = Workbooks.Add
3:   wbNewWorkbook.Worksheets("Sheet1").Range("A1"). _
         Value = 100
4:   wbNewWorkbook.SaveAs _
         "c:\Users\usr\Hour0.xlsx"
5:   wbNewWorkbook.Close
```

```
6:  MsgBox "Workbook is closed"
7:  Application.Quit
End Sub
```

2. Start the `NewBook` macro execution.
3. When the window with message *Workbook is closed* appears (Fig. 2.19), click on the *OK* button in it. At that, the `NewBook` macro closes the Excel window. When closing the Excel window, we may disagree with the offer to save Book1.
4. Open the Hour0 workbook in folder

```
c:\Users\usr
```

Number 100 is in cell A1 on worksheet Sheet1.



```
Option Explicit

Sub NewBook()
1:  Dim wbNewWorkbook As Workbook
2:  Set wbNewWorkbook = Workbooks.Add
3:  wbNewWorkbook.Worksheets("Sheet1").Range("A1"). _
        Value = 100
4:  wbNewWorkbook.SaveAs _
        "c:\Users\usr\Hour0.xlsx"
5:  wbNewWorkbook.Close
6:  MsgBox "Workbook is closed"
7:  Application.Quit
End Sub
```

Fig. 2.18. The VB window with the `NewBook` macro text

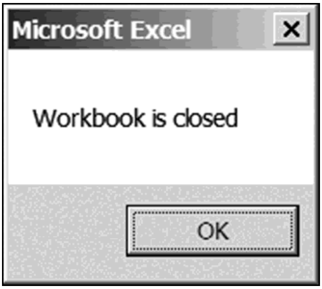
The `NewBook` macro begins with the declaration of the `wbNewWorkbook` variable (operator `1`). Because of executing operator `2`, a workbook is created and assigned to the `wbNewWorkbook` variable. Operator `3` puts number 100 into the A1 cell on the Sheet1 worksheet of this workbook.

When creating a workbook, tabular processor Excel gives it a default name, for example, Book2. Because the workbook name is known inexactly, the `wbNewWorkbook` variable is used instead of `Workbooks("Book2")` in the macro. Operator `4` saves this workbook under the following full name:

`c:\Users\usr\Hour0.xlsx`.

Operator 5 closes the Hour0 workbook, and operator 6 displays the message about it (Fig. 2.19). Operator 7 performs exit from Excel.

In operator 1, we may replace the Workbook type by the Object data type (Appendix 1) similar to the Variant data type familiar to us.



Workbook is closed

Fig. 2.19. The window with the *OK* button for terminating the NewBook macro execution

2.10. Objects Worksheet, Worksheets and ActiveSheet

The Worksheet object follows Workbook in the object hierarchy. Let us consider the following ***properties*** of object Worksheet:

❖ Name — the worksheet name;

❖ Cells — one of the following:

- the collection of all cells on the worksheet;
- the single cell if we specify (by two integers in parentheses, through a comma) the numbers of row and column whose intersection defines this cell.

The Cells property will be also considered in the next section, as applied to the Range object.

We will consider the following two ***methods*** of the Worksheet object:

❖ Activate — activating the worksheet;

❖ Delete — deleting the worksheet.

Examples of using the properties and methods of the Worksheet object are given below, when considering objects of the Worksheet type. The last objects have the properties and methods of the Worksheet object.

The Worksheets object is an object containing all worksheets of the Excel workbook. This object is also named as the Worksheets collection.

Its number or name can identify each worksheet of the Worksheets object. For example, Worksheets(1) designates the 1st worksheet of the workbook, and Worksheets(“Sheet1”) is the worksheet by name Sheet1. Worksheets(1) and Worksheets(“Sheet1”) are objects of the Worksheet type.

As examples of applying the Activate and Delete methods of the Worksheet object, let us consider the following operators:

```
Worksheets(3).Activate
Worksheets("Sheet2").Delete
```

The first operator activates the 3rd worksheet of the active workbook, and the second operator deletes worksheet by name Sheet2 of the active workbook.

When working with object Worksheets, the application of the Add ***method*** adds to the collection a new worksheet that becomes active at once. This application is accompanied by return (into the program) of the created worksheet, which is the Worksheet type object. As an example, see operator `2` in the `NewSheet` macro on p. 170.

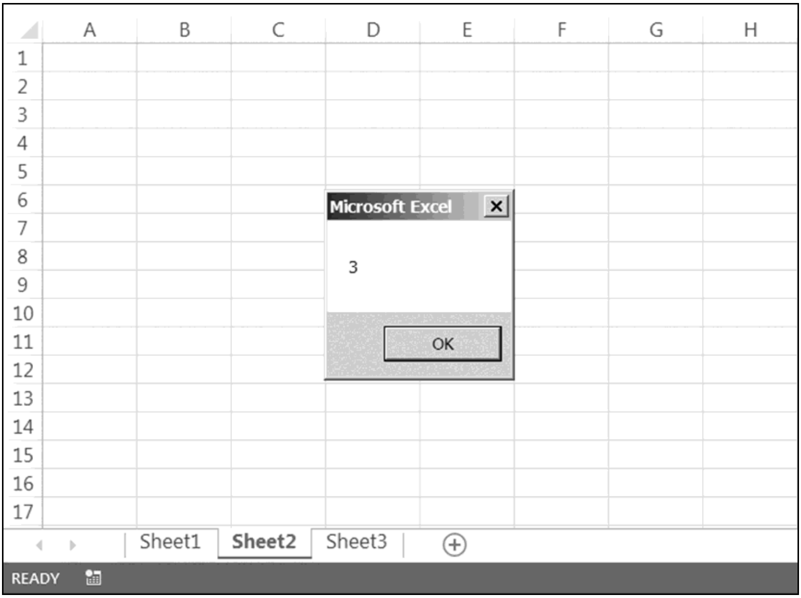
The Worksheets collection (as well as other collections) has the Count ***property*** that allows determining the number of objects in the collection.

An example of returning property Count is in the following macro:

Listing 2.8

```
Sub NumberofSheets()
    MsgBox Str(Worksheets.Count)
End Sub
```

The window, containing the number of worksheets in the active Excel workbook and the *OK* button, is displayed when executing the `NumberofSheets` macro (Fig. 2.20). For terminating the execution, we have to click on *OK*.


Microsoft Excel

3

Fig. 2.20

As we already mentioned in Section 2.8, ActiveSheet is a property of the Application object, corresponding to the active worksheet of the active workbook. As the Worksheet type object, the ActiveSheet property has the properties and methods of the Worksheet object.

An example of setting the Name property of the Worksheet object is the following operator:

```
ActiveSheet.Name = "July"
```

This operator assigns the month name to the active worksheet.

As an example of returning the Cells property of the Worksheet object, let us consider the following program:

Listing 2.9

```
Sub Color()
    Dim i As Integer
    Dim j As Integer
    For i = 1 To 4
        For j = 1 To 5
0:          With ActiveSheet.Cells(i, j)
                If .Value < 0 Then
                    .Font.Color = QBColor(10)
                    .Font.Italic = True
                End If
            End With
        Next j
    Next i
End Sub
```

This program:

1) examines the values in cells A1:E4 on the active worksheet;
2) sets the green italic font when the value is negative.

Note that the object name can be excluded from VBA notation

```
ActiveSheet.Cells(i, j)
```

In this case, line `0` becomes as follows:

```
With Cells(i, j)
```

Besides, note that operator `Cells.Select` in macro Listing 2.2 on p. 148 (programmed by means of Excel Macro Recorder) is equivalent to operator `ActiveSheet.Cells.Select`.

As an example of applying the Add method of the Worksheets object, let us consider the program for inserting a new worksheet into the active Excel workbook. For that, we fulfill the following:

1) put text

Listing 2.10

```
Sub NewSheet()
1:  Dim wsNewWorksheet As Worksheet
2:  Set wsNewWorksheet = Worksheets.Add
3:  wsNewWorksheet.Name = Format(Date, "d mmmm yyyy")
End Sub
```

into the code window;

2) execute the `NewSheet` macro.

The name of the inserted worksheet is the current date. This worksheet is active; it is placed in front of the former active worksheet.

The `NewSheet` macro (which is similar to `NewBook` on p. 164) begins with the declaration of variable `wsNewWorksheet` (operator `1`). The created worksheet is assigned to this variable (operator `2`). Operator `3` sets the worksheet name by means of the `wsNewWorksheet.Name` property.

Operator `3` includes the call of the `Format` function intended for converting a value (in particular, of the `Date` data type) to a string of the given form. The first argument is the `Date` function returning the current date. The second argument is the string, which gives the following format of the date: the day of month, the name of month, the year completely.

Note that the `Set` keyword is used not only in the operator of applying the Add method: any assignment operator, where objects are present, begins with the `Set` keyword. For example, the `NewSheet` program will also work if we replace `Worksheets.Add` by object `Worksheets(1)` in operator `2`. The new version of the program follows:

```
Sub NewSheet1()
1:  Dim wsNewWorksheet As Worksheet
2:  Set wsNewWorksheet = Worksheets(1)
3:  wsNewWorksheet.Name = Format(Date, "d mmmm yyyy")
End Sub
```

As the execution result, the current date becomes the 1st worksheet’s name.

2.11. Objects Range, Selection and ActiveCell

Object Range follows the Worksheet object in the object hierarchy. It allows working with the following elements of Excel:

- range of cells;
- range of columns;
- range of rows;
- single cell.

Let us consider some ***properties*** of the Range object.

❖ Formula — the Excel formula with operands (cell addresses) in the A1 reference style.

For example, operators

```
Range("C2:F8").Formula = "=$A$4+COS($A$10)"
Range("D:E").Formula = "=$A$4+COS($A$10)"
Range("2:2").Formula = "=$A$4+COS($A$10)"
Range("B3").Formula = "=$A$4+COS($A$10)"
```

set the Formula property. These operators are respectively used to put formula

```
=$A$4+COS($A$10)
```

into the following parts of the active worksheet:

- range C2:F8;
- columns D and E;
- the 2nd row;
- the B3 cell.

❖ FormulaR1C1 — the Excel formula with operands (cell addresses) in the R1C1 reference style.

For example, operator

```
Worksheets("Sheet1").Range("G1:H4").FormulaR1C1 = _
    "=SQRT(R5C8)^3+7.3"
```

sets the FormulaR1C1 property. This operator is used for entering formula

```
=SQRT($H$5)^3+7.3
```

into the G1:H4 range on the Sheet1 worksheet (the *R1C1 reference style* option may be turned on or off, see p. 143).

❖ Address — the cell address.

❖ Offset — the range shifted relative to the selected (active) range according to two integers in parentheses.

In the Offset property, the first parameter (in parentheses) is the vertical shift, and the second parameter is the horizontal shift. A comma is placed between these parameters.

❖ Value — one of the following:

- the array of the range values;
- the cell value, if the Range object corresponds to the single cell.

❖ Columns — the collection of the range columns.

❖ Rows — the collection of the range rows.

❖ Cells — the collection of the range cells.

Note that we considered the Cells property regarding the Worksheet object in the previous section.

Below, we will list some ***methods*** of the Range object.

❖ Clear — the removal of the range contents.

For example, the following application of the Clear method clears cells A1:F7 on the Sheet1 worksheet:

```
Worksheets("Sheet1").Range("A1:F7").Clear
```

❖ Select — the selection (activation) of the range.

Objects of the Range type have properties and methods of the Range object.

When working with the Range type objects, it is convenient to use ***the ForEach…Next cycle***, which is similar to cycle `For…Next` (p. 58). The syntax of this operator may be studied by means of the Excel help system started by pressing the *F1* key when the VB window is active.

In the following program example, operator `For Each…Next` is used for squaring the values of range A1:A6 on Sheet1 of the active workbook:

Listing 2.11

```
Sub Square()
    Dim x As Range
    For Each x In Worksheets("Sheet1").Range("A1:A6")
        x.Value = x.Value ^ 2
    Next
End Sub
```

The Selection object allows working with the active (selected) cells. As the Range type object, the Selection object has properties Columns, Rows and Cells. Thus, properties

```
Selection.Columns
Selection.Rows
Selection.Cells
```

are the collections of columns, rows and cells of the selected range, respectively.

The following example program inserts the multiplication table into the selected range on the active worksheet:

Listing 2.12

```
Sub MultiplicationTable()
    Dim m As Integer, n As Integer
    Dim i As Integer, j As Integer
1:  m = Selection.Rows.Count      'quantity of rows
2:  n = Selection.Columns.Count   'quantity of columns
3:  For i = 1 To m
4:      For j = 1 To n
5:          Selection.Cells(i, j).Value = i * j
6:      Next j
7:  Next i
End Sub
```

Operators `1` and `2` contain the Count property whose return allows defining the quantity of objects in the `Selection.Rows` and `Selection.Columns` collections. Because of executing these operators, the quantities of rows and columns in the selected range are assigned to the `m` and `n` variables, respectively.

Operator `5` may have the following form:

```
Selection.Cells(i, j) = i * j
```

Here, `.Value` is present implicitly.

We advise the reader to do the following:

1) enter program `MultiplicationTable` into the code window;
2) go to the Excel window and select the range for the multiplication table;
3) run program `MultiplicationTable`;
4) looking the execution result, verify the program correctness.

The Selection object can be used for recovering the selection. For example, in the `graph` subroutine (Section 4.8):

- in the beginning, operator

```
Set wbOldSelection = Selection
```

assigns the selected range to the `wbOldSelection` variable of the `Range` type;

- in the end, operator

```
wbOldSelection.Select
```

selects the `wbOldSelection` range.

It was mentioned in Section 2.8 that ActiveCell is a property of the Application object, corresponding to the active cell on the active worksheet of the active workbook. As the Range type object, the ActiveCell property has the properties and methods of the Range object.

The `PropofRange` program, given below, contains examples of using the properties and methods of the Range object.

1. Put 100 into cell B1, 200 into B2 and 300 into B3 on the Sheet1 worksheet.
2. Put formula

```
=SUM(B1:B3)
```

into cell B4 on Sheet1, and click on the tick button of the Excel formula bar.

3. Go to Visual Basic Environment and insert a module into the active workbook.
4. Enter the following text into the code window:

Listing 2.13

```
Sub PropofRange()
1:  Worksheets("Sheet1").Range("A1").Select
2:  ActiveCell.Offset(2, 3).Select
3:  MsgBox "Current cell – " & ActiveCell.Address
4:  MsgBox "Value in cell B4 = " & _
        Range("B4").Value
5:  MsgBox "Formula in cell B4: " & _
        Range("B4").Formula
End Sub
```

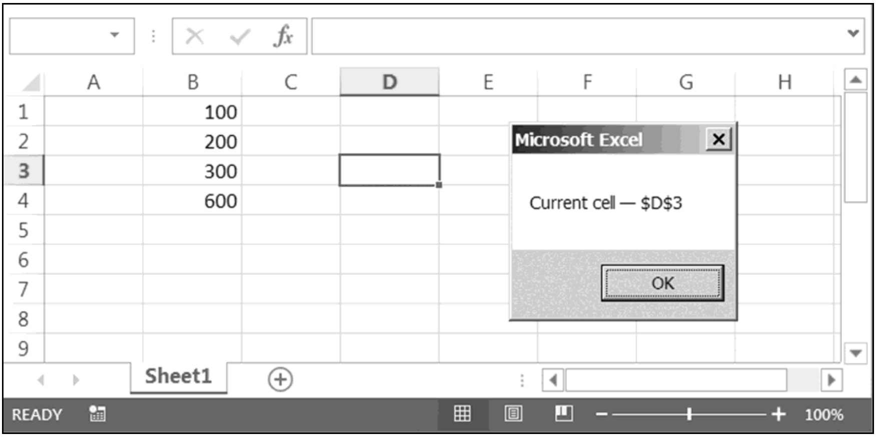
5. Run the `PropofRange` program execution after activating the Sheet1 worksheet. The window with message *Current cell - $D$3* and the *OK* button appears (Fig. 2.21a).

6. Click on the *OK* button. The window with message *Value in cell B4 = 600* and the *OK* button appears (Fig. 2.21b).

7. Click on the *OK* button. The window with message *Formula in cell B4: =SUM(B1:B3)* and the *OK* button appears (Fig. 2.21c).

8. Click on the *OK* button for terminating the execution.

Operator `1` of the `PropofRange` program selects cell A1 on the Sheet1 worksheet. Operator `2` selects the D3 cell, shifted 2 vertically and 3 horizontally relative to the A1 cell (property Offset and method Select of the Range object are figured in this operator). Further, three windows with the message and *OK* button are sequentially displayed (Fig. 2.21).

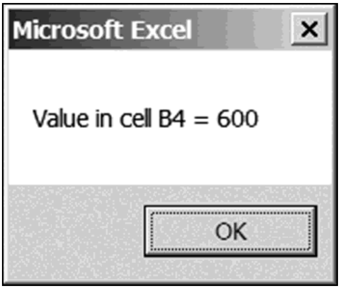


	A	B	C	D	E	F	G	H
1		100						
2		200						
3		300						
4		600						
5								
6								
7								
8								
9								

Microsoft Excel

Current cell — $D$3

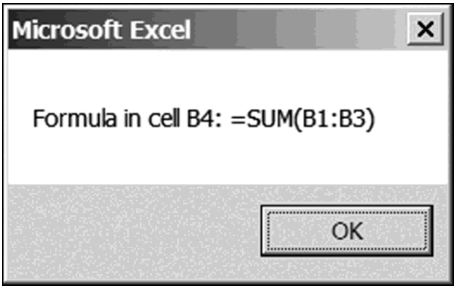
Fig. 2.21a. The Excel worksheet with the first window



Microsoft Excel

Value in cell B4 = 600

Fig. 2.21b. The second window



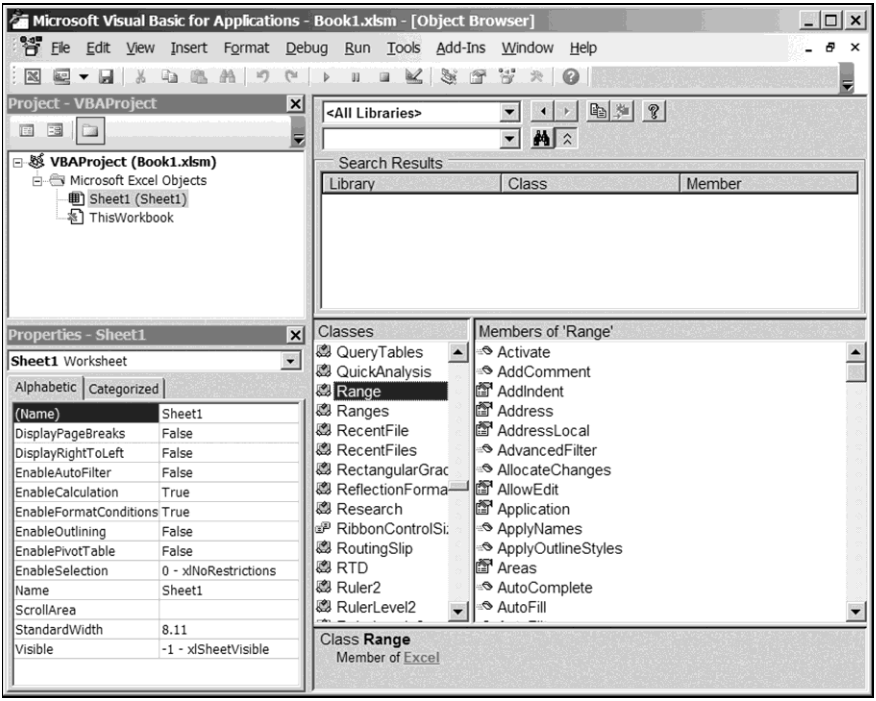
Formula in cell B4: =SUM(B1:B3)

Fig. 2.21c. The third window

2.12. Study of objects

We can obtain information on any object by means of the reference systems started by pressing keys *F1* and *F2*. For that, we press the *F2* key when the VB window is active. As a result, the object browser window appears.

If the Range object is interesting for us, we highlight *Range* in list *Classes* by click. At that, list *Members of 'Range'*, containing properties and methods of object Range, appears in the object browser window (Fig. 2.22).



Sheet1 Worksheet

(Name)	Sheet1
DisplayPageBreaks	False
DisplayRightToLeft	False
EnableAutoFilter	False
EnableCalculation	True
EnableFormatConditions	True
EnableOutlining	False
EnablePivotTable	False
EnableSelection	0 - xlNoRestrictions
Name	Sheet1
ScrollArea	
StandardWidth	8.11
Visible	-1 - xlSheetVisible

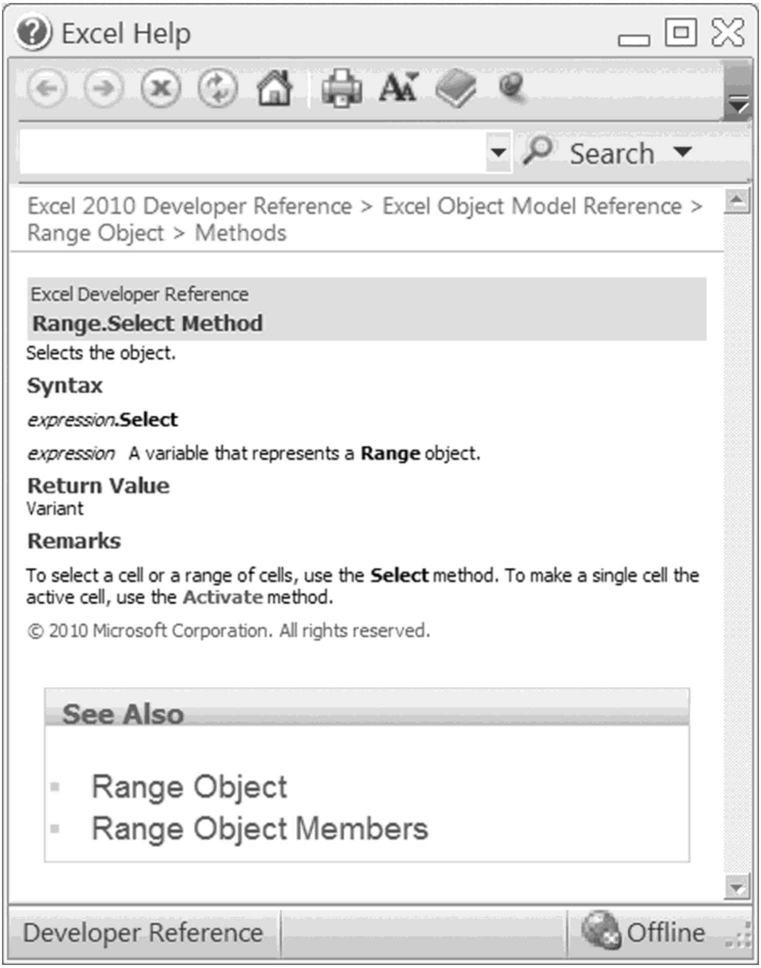
Classes:
QueryTables
QuickAnalysis
Range
Ranges
RecentFile
RecentFiles
RectangularGrac
ReflectionForma
Research
RibbonControlSi:
RoutingSlip
RTD
Ruler2
RulerLevel2

Members of 'Range':
Activate
AddComment
AddIndent
Address
AddressLocal
AdvancedFilter
AllocateChanges
AllowEdit
Application
ApplyNames
ApplyOutlineStyles
Areas
AutoComplete
AutoFill

Class **Range**
Member of Excel

Fig. 2.22. The VB window containing the object browser window

For obtaining information on the Select method of the Range object, we highlight line *Select* in list *Members of 'Range'* and press the *F1* key. As a result, the *Excel Help* window, containing the necessary information, is displayed (Fig. 2.23).



Excel 2010 Developer Reference > Excel Object Model Reference > Range Object > Methods

Excel Developer Reference

Range.Select Method

Selects the object.

Syntax

expression.**Select**

expression A variable that represents a **Range** object.

Return Value

Variant

Remarks

To select a cell or a range of cells, use the **Select** method. To make a single cell the active cell, use the **Activate** method.

© 2010 Microsoft Corporation. All rights reserved.

See Also

- Range Object
- Range Object Members

Fig. 2.23. The *Excel Help* window with the description of the `Range.Select` method

We advise the reader to obtain information on the Add method of object Worksheets in a similar way.

In addition, ***we advise the reader*** to obtain information on the ActiveCell object by means of the binoculars pictogram in the object browser window, such as we obtained information on *vbFriday* on p. 32.

2.13. Using the Excel table as the user interface of programs

We considered the main Excel objects. Now we can use the Excel table as the user interface of programs.

Initially, we will create a macro for calculating the length of the leg of a right-angled triangle with given lengths of the hypotenuse and the other leg. In this macro, we will realize formula

$$b = \sqrt{c^2 - a^2}$$

following from formula (1.1) on p. 16.

Let length *a* of the leg be in the A4 cell, length *c* of the hypotenuse be in C4. Let B5 be intended for resulting length *b*. The macro has the following text:

Listing 2.14

```
Sub Leg()
    Dim a As Single, b As Single, c As Single
    a = Range("A4").Value
    c = Range("C4").Value
    If c < a Then
        Range("B5").Value = "Error"
        End            'immediate termination of macro
    End If
    b = Sqr(c ^ 2 - a ^ 2)    'according to Pythagoras
    Range("B5").Value = b
End Sub
```

We advise the reader to do the following:

1) enter the `Leg` macro text into the code window of a new module;
2) go to the Excel window;
3) enter 4.3 and 5.1 into cells A4 and C4, respectively;
4) run the `Leg` macro;
5) make sure that number 2.742261 appears in the B5 cell on the active worksheet.

As we know, length `c` of the hypotenuse must be greater than or equal to length `a` of the leg. If the C4 cell value is less than the A4 cell value, logical expression `c < a` accepts `True` when executing the `Leg` macro. In this case, message *Error* appears in cell B5 and the `End` operator terminates the macro execution. We encounter the `End` operator for the first time.

It is easy to understand the work of the following macro intended for transposing a numerical matrix relative to its auxiliary diagonal according to formula (1.3) on p. 109:

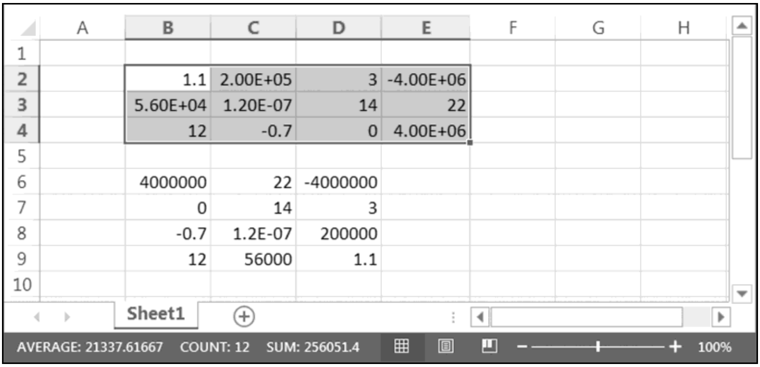
Listing 2.15

```
Sub TRANSPA()
    Dim m As Integer, n As Integer
    Dim i As Integer, j As Integer
    m = Selection.Rows.Count        'quantity of rows
    n = Selection.Columns.Count     'quantity of columns
    For j = 1 To n
        For i = 1 To m
            Selection.Cells(j + m + 1, i) = _
                Selection.Cells(m + 1 - i, n + 1 - j)
        Next i
    Next j
End Sub
```

This macro is used as follows:

1) on the Excel worksheet, select the matrix intended for the transposition;
2) run the `TRANSPA` macro.

The result of the macro execution appears below the original matrix (Fig. 2.24).



	A	B	C	D	E	F	G	H
1								
2		1.1	2.00E+05	3	-4.00E+06			
3		5.60E+04	1.20E-07	14	22			
4		12	-0.7	0	4.00E+06			
5								
6		4000000	22	-4000000				
7		0	14	3				
8		-0.7	1.2E-07	200000				
9		12	56000	1.1				
10								

Fig. 2.24. The Excel worksheet with the initial matrix and the result of its transposing relative to the auxiliary diagonal

2.14. Two more Excel macros. Personal Macro Workbook

Let us create a macro fulfilling the following operations:

1) at the 1st run, the macro writes the value of cell A3 on worksheet Sheet1 into cell B2 on worksheet Sheet2;

2) at the 2nd run, the macro writes the value of the same cell, Sheet1!A3, into cell Sheet2!B4;

3) at the 3rd run, the macro writes the value of cell Sheet1!A3 into cell Sheet2!B6, and so on.

Into cell Sheet1!A3, the values are entered manually before the next run of the macro.

Let us use Sheet1!A4 as an auxiliary cell. We have to write zero into this cell before a series of the macro runs.

Into the code window of a new module, we enter the following text of the macro:

Listing 2.16

```
Sub Macr1()
    Dim i As Integer, M As Range
    Set M = Range("Sheet2!B1:Sheet2!B200")
    i = Range("Sheet1!A4").Value + 2
    M(i) = Range("Sheet1!A3").Value
    Range("Sheet1!A4").Value = i
End Sub
```

In the above `Macr1` macro:

- `i` is an auxiliary variable;
- `M` is the array corresponding to range B1:B200 on worksheet Sheet2;
- `Range("Sheet1!A4").Value`, `Range("Sheet1!A3").Value` are the values in cells A4 and A3 on worksheet Sheet1.

At the 1st run of the `Macr1` macro:

1) number `2` is assigned to variable `i`;

2) the value of cell A3 on worksheet Sheet1, for example `100`, is assigned to variable `M(2)`, i.e., to cell B2 on worksheet Sheet2;

3) the value of `i` (number `2`) is written into cell A4 on worksheet Sheet1.

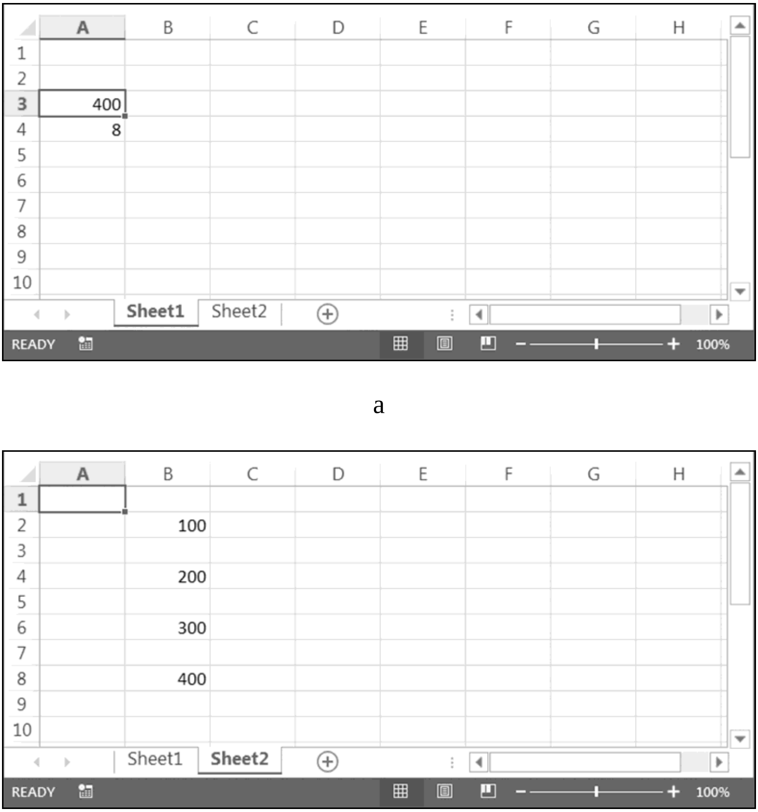
At the 2nd run of the `Macr1` macro:

1) number `4` is assigned to variable `i`;

2) the value of cell A3 on worksheet Sheet1, for example `200`, is assigned to variable `M(4)`, i.e., to cell B4 on worksheet Sheet2;

3) the value of `i` (number `4`) is written into cell A4 on worksheet Sheet1.

And so on. We see that the `Macr1` macro fulfills the necessary operations (Fig. 2.25).



	A	B	C	D	E	F	G	H
1								
2								
3	400							
4	8							
5								
6								
7								
8								
9								
10								

a

	A	B	C	D	E	F	G	H
1								
2		100						
3								
4		200						
5								
6		300						
7								
8		400						
9								
10								

b

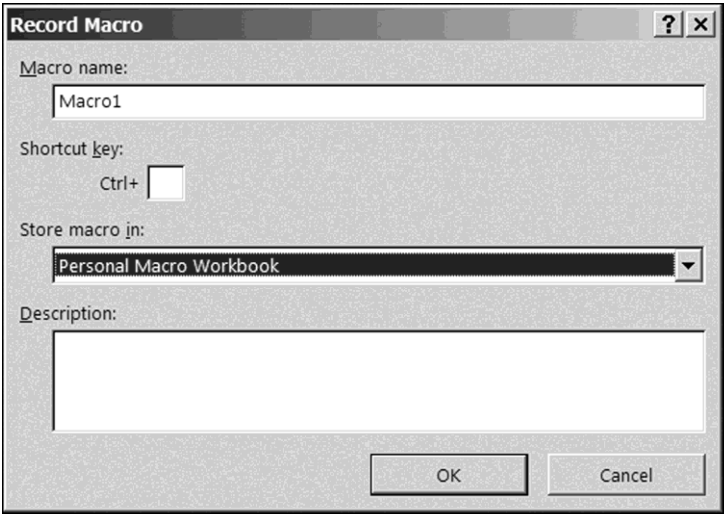
Fig. 2.25. Worksheets Sheet1 (a) and Sheet2 (b) after the fourth run of the `Macr1` macro: numbers 100, 200, 300 and 400 are put into cell Sheet1!A3 before the macro runs

The developed macro, `Macr1`, has two drawbacks:

- we can use this macro only in the Excel workbook, where it was created;
- the number of the macro runs (in one series) should not exceed 100 because the `M` array, containing the fixed quantity of elements, is used in this macro.

For liquidation of the first drawback, we will create Personal Macro Workbook. For that, let us fulfill the following operations.

1. Open a blank workbook.
2. Open window *Record Macro* by fulfilling *Developer* > *Record Macro* (in area *Code*) or *View* > arrow *Macros* (in area *Macros*) > *Record Macro*.
3. Enter *Personal Macro Workbook* into box *Store macro in* by using the drop-down list (Fig. 2.26) and click on *OK*.



Record Macro

Macro name: Macro1

Shortcut key: Ctrl+

Store macro in: Personal Macro Workbook

Description:

Fig. 2.26. The *Record Macro* window before clicking on the *OK* button

4. *Developer* > *Stop Recording* (in area *Code*) or *View* > arrow *Macros* (in area *Macros*) > *Stop Recording*.
5. Open the VB window.
6. Click on line *VBAProject (PERSONAL.XLSB)* in the project explorer window, and fulfill *File* > *Save PERSONAL.XLSB*.
7. Close the Excel window by clicking on the little cross in the top right corner. If the window with a question about saving changes in Personal Macro Workbook is displayed (when closing the Excel window), click on the *Yes* button.

Because of these operations, the PERSONAL.XLSB file is created. This file is Personal Macro Workbook of the computer user by name *usr*.

Personal Macro Workbook opens automatically when starting Excel. Macros and user-defined functions, placed in Personal Macro Workbook, can be used in Excel workbooks that are in folder `c:\Users\usr` (and in folders enclosed in it).

The automatic opening of Personal Macro Workbook does not lead to appearance of any button on the taskbar of Windows Desktop.

Let us fulfill the following operations:

1) open a blank workbook;

2) go to the VB window;

3) click on the *VBAProject (PERSONAL.XLSB)* line in the project explorer window;

4) *Insert* > *Module* (it means the module insertion into Personal Macro Workbook);

5) put macro

Listing 2.17

```
Sub Macr2()
    Dim i As Integer
    i = Range("Sheet1!A4").Value + 2
    Range("Sheet2!B" & CStr(i)) = _
        Range("Sheet1!A3").Value
    Range("Sheet1!A4").Value = i
End Sub
```

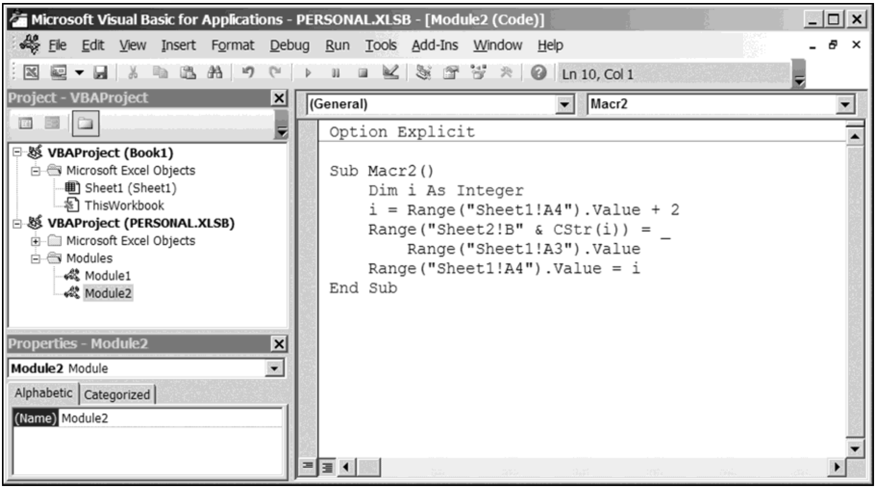
into the code window.

In Fig. 2.27, Module1 in the project explorer window is the result of operation of Excel Macro Recorder that was used for creating Personal Macro Workbook. This module may be removed or used in further work with Personal Macro Workbook, for example, for storage of the Func9 function (Section 2.15). The `Macr2` macro is placed in the Module2 module.

To save changes made in Personal Macro Workbook, we must fulfill the following in the VB window: *File* > *Save PERSONAL.XLSB*. At that, the `Macr2` macro, which is a part of file PERSONAL.XLSB, is saved on the hard disk of the computer.

Macro `Macr2`, used as macro `Macr1`, has the following advantages:

- macro `Macr2` can be used in all Excel workbooks that are in folder `c:\Users\usr`;
- the number of the macro runs is not limited because the `M` array is absent in the macro code.



(Name) Module2

```
Option Explicit

Sub Macr2()
    Dim i As Integer
    i = Range("Sheet1!A4").Value + 2
    Range("Sheet2!B" & CStr(i)) = _
        Range("Sheet1!A3").Value
    Range("Sheet1!A4").Value = i
End Sub
```

Fig. 2.27. The VB window with the `Macr2` macro
in the code window of the Module2 module

It is possible to transfer Personal Macro Workbook to other user. For this purpose, the PERSONAL.XLSB file should be copied into the appropriate folder of this user. Let us consider the location of Personal Macro Workbook on the hard disk of the computer.

The workbooks, which are in the XLSTART folder, are opening automatically when starting Excel. The full name of this folder follows:

```
c:\Users\usr\AppData\Roaming\Microsoft\Excel\XLSTART
```

where `usr` is the user name. File PERSONAL.XLSB is located in this folder.

If we want to transfer our Personal Macro Workbook to the user whose name is *usr2*, we must copy the PERSONAL.XLSB file into folder

```
c:\Users\usr2\AppData\Roaming\Microsoft\Excel\XLSTART
```

of his computer.

Note that folder `AppData` (from “Application Data”) may be hidden in Windows Explorer. To make a folder or file visible, we must fulfill the following operations:

1) open the folder containing the hidden folder or file of interest;
2) *Organize* > *Folder and search options*;

3) in open window *Folder Options*, activate tab *View*;

4) in list *Advanced settings*, turn on option *Show hidden files, folders, and drives*;

5) successively click on buttons *Apply* and *OK*.

To expand the chosen operation mode of Windows Explorer to all folders, we must click on button *Apply to Folders* before clicking on the *Apply* button. In open window *Folder Views*, we must click on the *Yes* button.

2.15. One more user-defined function of Excel

There are value *h* and two-dimensional numerical array (matrix) **A**, whose elements are in cells of the Excel table. We will consider a user-defined function, which returns the number of the **A** array values exceeding *h*.

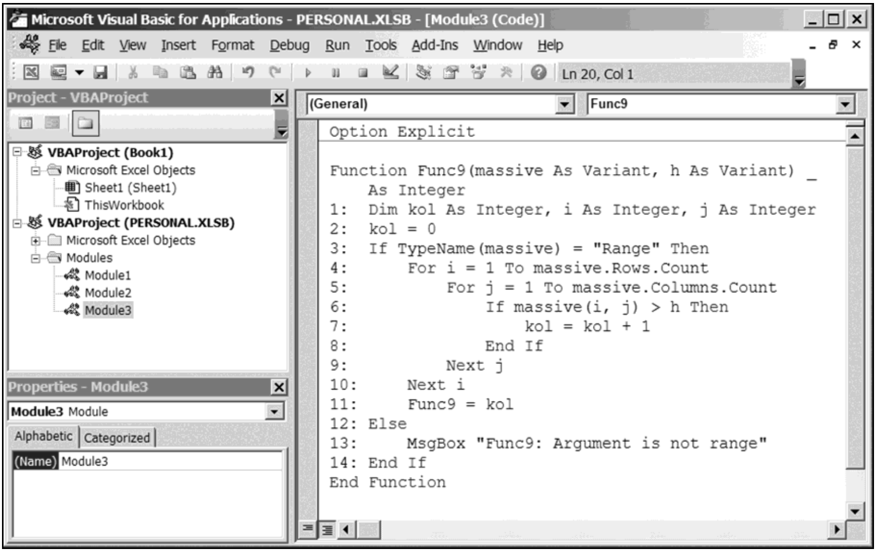
Let us create an Excel workbook and then open the VB window. Further, let us insert a module into Personal Macro Workbook, already created, and then put the following text of the user-defined function into the code window of the inserted module.

Listing 2.18

```
Function Func9(massive As Variant, h As Variant) _
    As Integer
1:  Dim k As Integer, i As Integer, j As Integer
2:  k = 0
3:  If TypeName(massive) = "Range" Then
4:      For i = 1 To massive.Rows.Count
5:          For j = 1 To massive.Columns.Count
6:              If massive(i, j) > h Then
7:                  k = k + 1
8:              End If
9:          Next j
10:     Next i
11:     Func9 = k
12: Else
13:     MsgBox "Func9: Argument is not range"
14: End If
End Function
```

The current state of the project is depicted in Fig. 2.28.

The built-in `TypeName` function (in line 3) returns its argument's data type as a string. This function is used for verifying the type of input parameter `massive`. In the correct call of the `Func9` function, the `massive` variable has the Range type.



```
Option Explicit

Function Func9(massive As Variant, h As Variant) _
    As Integer
1:  Dim kol As Integer, i As Integer, j As Integer
2:  kol = 0
3:  If TypeName(massive) = "Range" Then
4:      For i = 1 To massive.Rows.Count
5:          For j = 1 To massive.Columns.Count
6:              If massive(i, j) > h Then
7:                  kol = kol + 1
8:              End If
9:          Next j
10:     Next i
11:     Func9 = kol
12: Else
13:     MsgBox "Func9: Argument is not range"
14: End If
End Function
```

Fig. 2.28. The VB window with text of the `Func9` function in the code window

In line `4`, the quantity of rows in the `massive` range is the result of returning property `massive.Rows.Count`. In line `5`, the quantity of columns is the result of returning property `massive.Columns.Count`.

In line `6`, `massive(i, j)` is the reference to the corresponding cell of the `massive` range.

The `Func9` function is used as follows.

1. Enter values of matrix **A**, for example, into range D5:E8 on worksheet Sheet4.

2. Assign a name, for example Test9, to the D5:E8 range. For this purpose:

1) select this range, and fulfill *Formulas* > *Define Name* in area *Defined Names*;

2) in the open *New Name* window, type *Test9* in text box *Name*;

3) enter *Sheet4* into box *Scope* by means of the drop-down list (Fig. 2.29);

4) click on the *OK* button.

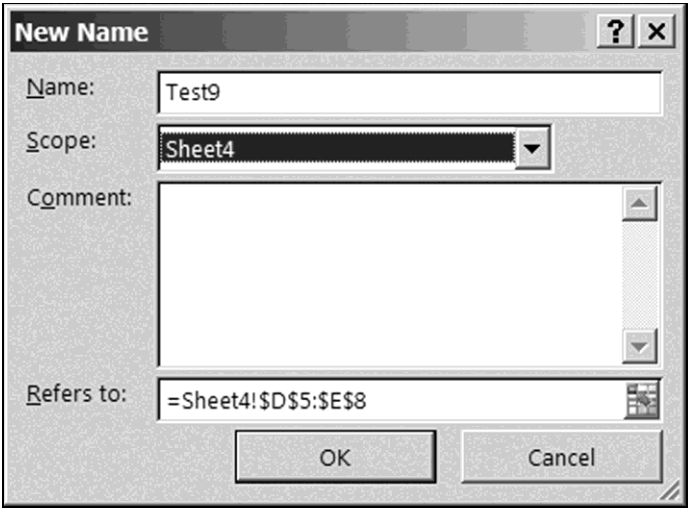
3. Enter formula

```
=PERSONAL.XLSB!Func9(Test9;2)
```

into any cell on worksheet Sheet4, for example, E11.

In the above formula, an exclamation mark means that the `Func9` function is a part of file PERSONAL.XLSB.

4. Click on the tick button of the Excel formula bar.



New Name

Name: Test9

Scope: Sheet4

Comment:

Refers to: =Sheet4!$D$5:$E$8

Fig. 2.29. The *New Name* window with automatically filled box *Refers to*

The resulting number of values in the D5:E8 cells exceeding 2 appears in the E11 cell (Fig. 2.30).

Below is the second way of using the `Func9` function, without assigning a name to the range with the **A** array.

Let us use Function Wizard as follows:

1) select a cell for the result, for example, Sheet4!E11;

2) click on the *fx* button of the Excel formula bar to start Function Wizard;

3) in the *User Defined* category of the open *Insert Function* window, highlight the line corresponding to the `Func9` function (Fig. 2.31), and click on the *OK* button;

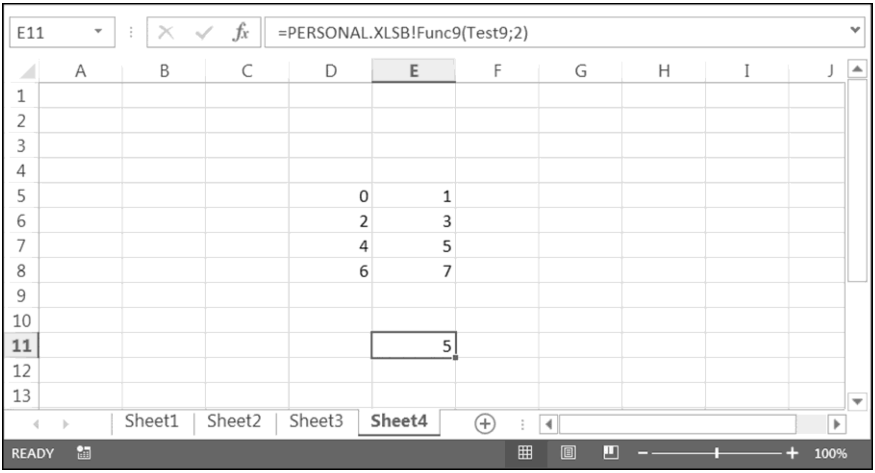
4) in the open *Function Arguments* window (Fig. 2.32), enter:

- the source range (for example, D5:E8 on the Sheet4 worksheet) into the first text box;
- the *h* value, for example 2, into the second text box;

5) click on the *OK* button.

We see the former result: five values exceed *h* = 2 (Fig. 2.30).

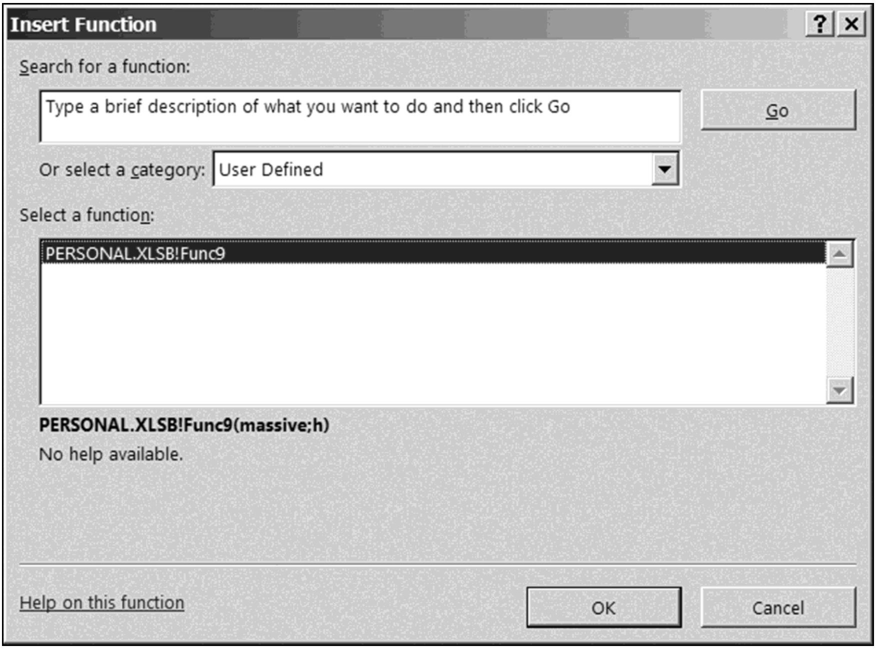
Note that cell E11 and range D5:E8 may be on one or different worksheets of the same Excel workbook.



E11 =PERSONAL.XLSB!Func9(Test9;2)

	D	E
5	0	1
6	2	3
7	4	5
8	6	7
9		
10		
11		5

Fig. 2.30. The source data and calculation result in the Excel window



Insert Function

Search for a function: Type a brief description of what you want to do and then click Go

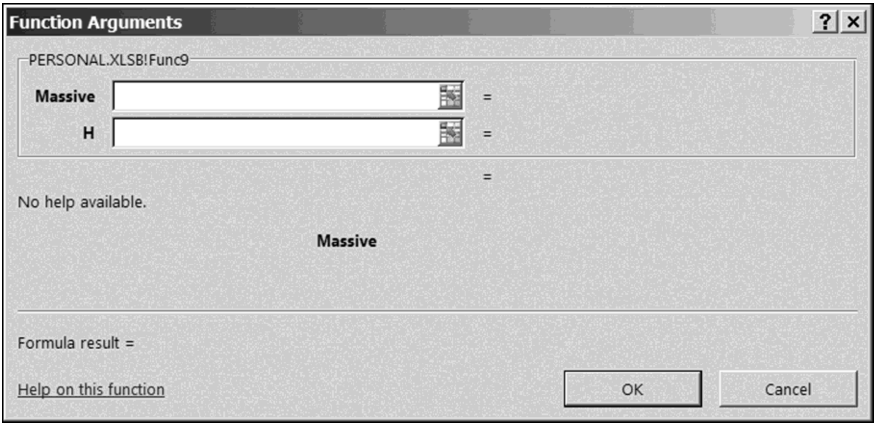
Or select a category: User Defined

Select a function: PERSONAL.XLSB!Func9

PERSONAL.XLSB!Func9(massive;h)

No help available.

Fig. 2.31. The first window of Function Wizard



No help available.

Massive

Formula result =

Fig. 2.32. The second window of Function Wizard

We advise the reader to calculate the values of function $f(x)$ from Appendix 4 and also of functions $g(x) = x\, f(x)$ and $h(x) = x^2 f(x)$ at not less than 10 points of segment $[a,\, b]$, including *a* and *b*, in the following ways:

- creating a macro by means of Excel Macro Recorder;
- programming an Excel user-defined function in VBA;
- programming a macro in VBA.

Use the Excel tools to create the graphs of functions $f(x)$, $g(x)$ and $h(x)$.

2.16. Digression. Change of Excel options

In the next four chapters of the book, languages VB and VBA will be used for programming numerical methods. Besides, we will consider the Excel procedures, which are a part of the add-ins, for solving some of our tasks.

For installing the necessary Excel add-ins, let us fulfill the following operations:

1) *File* > *Options* > *Add-Ins*;

2) enter *Excel Add-ins* into the *Manage* box by means of the drop-down list, and click on the *Go* button;

3) in the open *Add-Ins* window, turn on options *Analysis ToolPak* and *Solver Add-in*, and click on the *OK* button.

As a result, the *Data Analysis* and *Solver* commands appears in Excel Ribbon — on the *Data* tab, in the *Analysis* area.

We will need to tune Excel so that formulas are displayed in cells, instead of results of calculations according to these formulas. For this purpose, we must fulfill the following operations:

1) *File* > *Options* > *Advanced*;

2) turn on option *Show formulas in cells instead of their calculated results* located below box *Display options for this worksheet*;

3) click on the *OK* button.

In this way, we change the contents of the cells with formulas on the worksheet specified in box *Display options for this worksheet*. The contents of the cells without formulas do not change.

Upon turning off option *Show formulas in cells instead of their calculated results*, the worksheet returns to the customary form, with values in the cells.

We will also need a workbook with the module for programs, realizing numerical methods. To obtain it, let us fulfill the following:

1) open a blank workbook, and insert the Module1 module into it;

2) put text

```
Sub main()

End Sub
```

into the code window of this module;

3) in the VB window, fulfill *File* > *Save*;

4) in the open *Save As* window, choose the *c:\Users\usr* folder familiar to us (*usr* is the user name);

5) enter the workbook name, *BookNM*, into text box *File name* (*NM* is the abbreviation of "numerical methods");

6) set file type *Excel Macro-Enabled Workbook* by using the drop-down list;

7) click on the *Save* button;

8) close the VB and Excel windows.

We will enter texts of programs by name `main` into the Module1 module. Upon termination of work with the BookNM workbook, we will always save it.

Chapter 3.
Finite Difference Method for Solving Differential Equations

Two kinds of conditions on the solution of the second-order linear differential equation are reviewed, namely, the boundary and periodicity conditions. Replacing the derivatives by their finite difference analogs, we obtain the so-called finite difference schemes — systems of linear algebraic equations of special form.

We review several versions of the decomposition method [4] for solving the finite difference schemes. The simplest scheme is also solved by the Gaussian elimination method. The question of stability of these two methods is investigated in respect of not increasing the computing error during solving this scheme.

The finite difference method for solving the linear differential equation is used for solving the nonlinear differential equation by the quasilinearization method. For demonstration of the finite difference method's possibilities, we develop subroutines and programs for solving mathematical and applied problems. We use the Excel scatter diagrams for visualization of calculation results.

3.1. Finite difference analogs of derivatives for a uniform grid

Let us introduce an increasing sequence of points,

$$x_k < x_{k+1} < x_{k+2} < \ldots < x_{r-2} < x_{r-1} < x_r ,$$

on segment $a \le x \le b$, at that, $x_k = a$, $x_r = b$. In other words, segment $[a, b]$ is covered with a grid whose nodes have coordinates x_k, x_{k+1}, x_{k+2}, ..., x_{r-2}, x_{r-1}, x_r.

For simplicity, we will consider that the grid is uniform, i.e., the grid step, $x_i - x_{i-1} = h$, does not depend on i, $k+1 \le i \le r$ (later, we will consider a nonuniform grid).

In addition to this grid (Fig. 3.1), called the main grid, we will use the so-called auxiliary grid with the following nodes:

$x_{[k]} = (x_k + x_{k+1})/2$, $x_{[k+1]} = (x_{k+1} + x_{k+2})/2$, ..., $x_{[i-1]} = (x_{i-1} + x_i)/2$, $x_{[i]} = (x_i + x_{i+1})/2$, ..., $x_{[r-2]} = (x_{r-2} + x_{r-1})/2$, $x_{[r-1]} = (x_{r-1} + x_r)/2$.

As we see, numbers (indices) of the auxiliary grid nodes are in square brackets.

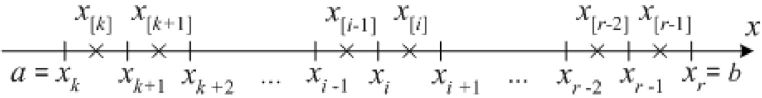


Fig. 3.1. Grids on segment $[a, b]$: main — indices without brackets; auxiliary — indices in square brackets

Let u_k, $u_{[k]}$, u_{k+1}, $u_{[k+1]}$, u_{k+2}, ..., u_{i-1}, $u_{[i-1]}$, u_i, $u_{[i]}$, u_{i+1}, ..., u_{r-2}, $u_{[r-2]}$, u_{r-1}, $u_{[r-1]}$, u_r be the $u(x)$ function values at the corresponding nodes of the main and auxiliary grids.

A function, set on a grid, is called a grid function or a function in tabular form (tabular function). An example of such function is $u(x)$ considered above.

We will come across grids and grid functions more than once.

Let us assume that function $u(x)$ is continuous in segment $[a, b]$, has continuous derivatives of $m-1$ orders, and the *m*-th derivative exists, in other words, function $u(x)$ is differentiable *m* times. At that, in a neighborhood of any interior point *x* of $[a, b]$, we can use Taylor's formula [3] in the following form:

$$u(x+\varepsilon) = u(x) + \frac{\varepsilon}{1!}\frac{du}{dx}(x) + \frac{\varepsilon^2}{2!}\frac{d^2u}{dx^2}(x) + \frac{\varepsilon^3}{3!}\frac{d^3u}{dx^3}(x) + \ldots$$

$$+\frac{\varepsilon^{m-1}}{(m-1)!}\frac{d^{m-1}u}{dx^{m-1}}(x) + O(\varepsilon^m)\,. \qquad (3.1)$$

The O notation, $O(\varepsilon^m)$, has the following sense (for natural *m*).

If ε is a small quantity, as in (3.1), $O(\varepsilon^m)$ means the quantity whose absolute value is less or equal to $C\,|\,\varepsilon^m\,|$, where *C* is a positive constant, i.e., *C* is independent of ε. In this case, $O(\varepsilon^m)$ is called the quantity of *m*-th order of smallness or the quantity of ε^m-th order.

If *E* is a large positive quantity, $O(E^m)$ means the positive quantity whose value tends to CE^m when $E \to \infty$, *C* is a positive constant.

If the grid step, *h*, is a small quantity, we have the following chain of equalities according to formula (3.1):

$$\frac{du}{dx}(x_i) = \frac{1}{2}\times[u'(x_i) + u'(x_i)] = \frac{1}{2}\times$$

$$\left[\frac{u\left(x_i+\dfrac{h}{2}\right) - u(x_i) - \dfrac{(h/2)^2}{2}\dfrac{d^2u}{dx^2}(x_i)}{\dfrac{h}{2}} + \frac{-u\left(x_i-\dfrac{h}{2}\right) + u(x_i) + \dfrac{(h/2)^2}{2}\dfrac{d^2u}{dx^2}(x_i)}{\dfrac{h}{2}}\right]$$

$$+O(h^2) = \frac{u_{[i]} - u_{[i-1]}}{h} + O(h^2)\,. \qquad (3.2)$$

Similarly, we can obtain two more expressions for the first derivative of the $u(x)$ function:

$$\frac{du}{dx}(x_{[i]}) = \frac{u_{i+1} - u_i}{h} + O(h^2)\,, \tag{3.3}$$

$$\frac{du}{dx}(x_i) = \frac{u_{i+1} - u_{i-1}}{2h} + O(h^2)\,. \tag{3.4}$$

Using formulas (3.2) and (3.3), we can also obtain an expression for the second derivative of $u(x)$ as follows:

$$\frac{d^2u}{dx^2}(x_i) = \frac{u'(x_{[i]}) - u'(x_{[i-1]})}{h} + O(h^2) =$$

$$= \frac{\dfrac{u_{i+1} - u_i}{h} - \dfrac{u_i - u_{i-1}}{h}}{h} + O(h^2) = \frac{u_{i+1} - 2u_i + u_{i-1}}{h^2} + O(h^2)\,. \tag{3.5}$$

After neglecting summands $O(h^2)$ in (3.2), (3.4) and (3.5), we have finite difference analogs of the first two derivatives of the $u(x)$ function at point x_i.

3.2. Finite difference scheme for the linear differential equation. The decomposition method

Let us consider the following second-order linear differential equation of general form on segment $[a, b]$:

$$\frac{d^2u}{dx^2} + g(x)\frac{du}{dx} + e(x)u = f(x)\,, \tag{3.6}$$

where $g(x)$, $e(x)$ and $f(x)$ are given functions, $u(x)$ is an unknown function.

For uniqueness of the solution of this equation, we use the following left and right boundary conditions:

$$A_0\,u(a) + A_1\,u'(a) = A\,, \tag{3.7}$$

$$B_0\,u(b) + B_1\,u'(b) = B\,, \tag{3.8}$$

where A and B are given parameters, $A_0 = B_0 = 1$, $A_1 = B_1 = 0$ to begin with.

Let us consider a method for solving this boundary value problem, based on replacing the derivatives by their finite difference analogs,

$$\frac{du}{dx}(x_i) = \frac{u_{i+1} - u_{i-1}}{2h}\,, \quad \frac{d^2u}{dx^2}(x_i) = \frac{u_{i+1} - 2u_i + u_{i-1}}{h^2}\,.$$

These expressions follow from (3.4) and (3.5) if we neglect the summands of second order of smallness, $O(h^2)$.

Let x_i be an internal node of the main grid (Fig. 3.1), i.e., node x_i does not coincide with points *a* and *b*. Equation (3.6) at this node looks like

$$\frac{d^2u}{dx^2}(x_i) + g(x_i)\frac{du}{dx}(x_i) + e(x_i)u(x_i) = f(x_i)\,.$$

Multiplying both sides of this equality by h^2 and substituting the finite difference analogs of the derivatives, we get the following linear algebraic equation:

$$\alpha_i u_{i-1} + \beta_i u_i + \gamma_i u_{i+1} = \delta_i\,, \tag{3.9}$$

where u_{i-1}, u_i, u_{i+1} are unknown variables, $i = k+1, k+2, ..., r-2, r-1$,

$$\alpha_i = 1 - 0.5 g_i h ,$$

$$\beta_i = e_i h^2 - 2 ,$$

$$\gamma_i = 1 + 0.5 g_i h , \tag{3.10}$$

$$\delta_i = f_i h^2 .$$

In the last four expressions, g_i, e_i and f_i are the values of the coefficients and right-hand side of equation (3.6) at node x_i: $g_i = g(x_i)$, $e_i = e(x_i)$, $f_i = f(x_i)$, $i = k+1, k+2, ..., r-2, r-1$.

Boundary conditions (3.7) and (3.8) can be written as follows:

$$\beta_k u_k + \gamma_k u_{k+1} = \delta_k , \tag{3.11}$$

$$\alpha_r u_{r-1} + \beta_r u_r = \delta_r , \tag{3.12}$$

where $\beta_k = \beta_r = -2$, $\gamma_k = \alpha_r = 0$, $\delta_k = -2A$, $\delta_r = -2B$.

The system of linear algebraic equations (3.9), (3.11) and (3.12) is called the finite difference scheme for boundary value problem (3.6) — (3.8). More precisely, this system is called the one-dimensional finite difference scheme because it corresponds to the problem with one spatial coordinate.

Using the definitions of matrix multiplication and equality (Section 1.21), we can write scheme (3.9), (3.11), (3.12) as the following matrix equation:

$$\begin{bmatrix} \beta_k & \gamma_k & 0 & 0 & ... & 0 & 0 & 0 & 0 \\ \alpha_{k+1} & \beta_{k+1} & \gamma_{k+1} & 0 & ... & 0 & 0 & 0 & 0 \\ 0 & \alpha_{k+2} & \beta_{k+2} & \gamma_{k+2} & ... & 0 & 0 & 0 & 0 \\ . & . & . & . & . & . & . & . & . \\ 0 & 0 & 0 & 0 & ... & \alpha_{r-2} & \beta_{r-2} & \gamma_{r-2} & 0 \\ 0 & 0 & 0 & 0 & ... & 0 & \alpha_{r-1} & \beta_{r-1} & \gamma_{r-1} \\ 0 & 0 & 0 & 0 & ... & 0 & 0 & \alpha_r & \beta_r \end{bmatrix} \times$$

$$\times \begin{bmatrix} u_k \\ u_{k+1} \\ u_{k+2} \\ ... \\ u_{r-2} \\ u_{r-1} \\ u_r \end{bmatrix} = \begin{bmatrix} \delta_k \\ \delta_{k+1} \\ \delta_{k+2} \\ ... \\ \delta_{r-2} \\ \delta_{r-1} \\ \delta_r \end{bmatrix} .$$

The coefficient (system) matrix, which is a part of this matrix equation, has the so-called tridiagonal form: nonzero elements are only on the main diagonal, the first “diagonal” below this and the first “diagonal” above the main diagonal.

Finite difference scheme (3.9), (3.11), (3.12) is usually solved by the decomposition method [4] whose main advantage is its efficiency in comparison with other methods for solving this system of linear algebraic equations:

- for calculating the values of the $n = r - k + 1$ unknowns (u_k, u_{k+1}, u_{k+2}, ..., u_{r-2}, u_{r-1}, u_r) by the decomposition method, $O(n)$ arithmetic operations must be performed (p. 203);
- the solution of the system of equations (3.9), (3.11) and (3.12) by the Gaussian elimination method (Sections 3.9 and 3.10) requires $O(n^3)$ arithmetic operations (p. 227).

The decomposition method includes two stages, which are called the forward and backward sweeps. To obtain formulas for the sweeps, let us connect the unknowns, u_{i-1} and u_i, through formula

$$u_{i-1} = P_i u_i + Q_i, \tag{3.13}$$

where P_i, Q_i are the auxiliary unknowns, $i = r, r-1, ..., k+1$.

After substituting the last expression into equation (3.9), we obtain

$$u_i = -\frac{\gamma_i}{\alpha_i P_i + \beta_i} u_{i+1} + \frac{\delta_i - \alpha_i Q_i}{\alpha_i P_i + \beta_i}$$

or

$$u_i = P_{i+1} u_{i+1} + Q_{i+1},$$

where P_{i+1} and Q_{i+1} are determined by the following recurrence formulas:

$$P_{i+1} = -\frac{\gamma_i}{\alpha_i P_i + \beta_i}, \tag{3.14}$$

$$Q_{i+1} = \frac{\delta_i - \alpha_i Q_i}{\alpha_i P_i + \beta_i}. \tag{3.15}$$

According to the last two formulas, if the values of P_{k+1}, Q_{k+1} are known, the use of these formulas at $i = k+1, k+2, ..., r-1$ gives the values of P_{k+2}, Q_{k+2}, P_{k+3}, Q_{k+3}, ..., P_r, Q_r.

The values of P_{k+1}, Q_{k+1} are determined by formulas

$$P_{k+1} = -\frac{\gamma_k}{\beta_k}, \tag{3.16}$$

$$Q_{k+1} = \frac{\delta_k}{\beta_k}, \tag{3.17}$$

which follow from left boundary condition (3.11).

The recurrence calculation of unknown P_i, Q_i using formulas (3.16), (3.17) and (3.14), (3.15) is called ***the forward sweep***.

Knowing the values of u_r, P_i, Q_i $(k+1 \le i \le r)$, we can calculate unknown u_{r-1}, u_{r-2}, ..., u_k using recurrence formula (3.13).

For determining the value of u_r, let us consider the following system of two linear algebraic equations with unknown u_{r-1} and u_r:

$$\alpha_r u_{r-1} + \beta_r u_r = \delta_r,$$

$$u_{r-1} - P_r u_r = Q_r.$$

The first equation is right boundary condition (3.12); the second equation is (3.13) at $i = r$. Let us solve this system for u_{r-1} and u_r by means of Cramer's rule [3].

The determinant of the system looks like

$$D = \begin{vmatrix} \alpha_r & \beta_r \\ 1 & -P_r \end{vmatrix}.$$

The following determinants are obtained from D by replacing the first and second columns by the column of the right-hand sides:

$$D_1 = \begin{vmatrix} \delta_r & \beta_r \\ Q_r & -P_r \end{vmatrix}, \quad D_2 = \begin{vmatrix} \alpha_r & \delta_r \\ 1 & Q_r \end{vmatrix}.$$

According to Cramer's rule, we have

$$u_{r-1} = \frac{D_1}{D} = \frac{\delta_r P_r + \beta_r Q_r}{\alpha_r P_r + \beta_r},$$

$$u_r = \frac{D_2}{D} = \frac{\delta_r - \alpha_r Q_r}{\alpha_r P_r + \beta_r}. \tag{3.18}$$

The recurrence calculation of unknown u_i using formulas (3.18) and (3.13) is called ***the backward sweep***.

The use of the forward and backward sweeps gives the values of unknown u_k, u_{k+1}, u_{k+2}, ..., u_{r-2}, u_{r-1}, u_r, i.e., the solution of finite difference scheme (3.9), (3.11), (3.12) for linear differential equation (3.6) with boundary conditions (3.7) and (3.8). The calculated values of u_k, u_{k+1}, u_{k+2}, ..., u_{r-2}, u_{r-1}, u_r approximate the exact solution of problem (3.6) — (3.8) at the nodes of the main grid on segment $[a, b]$.

A simple analysis of formulas (3.14) and (3.15) of the forward sweep shows that it requires $O(n)$ arithmetic operations, $n \to \infty$. A similar analysis of formula (3.13) shows that the backward sweep also requires $O(n)$ arithmetic operations. Thus, the solution of finite difference scheme (3.9), (3.11), (3.12) by the decomposition method requires $O(n)$ arithmetic operations.

3.3. Sufficient stability conditions for the decomposition method

A repeatable algorithm (computing process) is called stable if the error arising at any step (for example, the rounding-off error) does not increase during the computing process.

The decomposition method includes the forward and backward sweeps. Each of these sweeps can be unstable. Let us investigate this question.

It is easiest to write the sufficient condition of the backward sweep stability, i.e., the stability of the calculation process corresponding to recurrence formula (3.13). This condition has the following form:

$$|P_i| \le 1 \tag{3.19}$$

for all values of i from $k+1$ to r. If this condition is satisfied, then:

1) small error ε_i (which is a part of the calculated value of u_i) goes into the value of u_{i-1} without increase; $\varepsilon'_{i-1} = P_i\varepsilon_i$ is the come error of ε_i -th order;

2) because the additional error (ε''_{i-1} generated when calculating the value of u_{i-1}) may be positive or negative with probability 0.5, the total error ($\varepsilon_{i-1} = \varepsilon'_{i-1} + \varepsilon''_{i-1}$) is of ε_i -th order.

Similarly, the sufficient condition of stability of the calculation according to recurrence formula (3.15) of the forward sweep is

$$\left|\frac{\alpha_i}{\alpha_i P_i + \beta_i}\right| \le 1 \tag{3.20}$$

for all values of i from $k+1$ to $r-1$. If this condition is satisfied, then:

1) small error ε_i (which is a part of the calculated value of Q_i) goes into the value of Q_{i+1} without increase;

$$\varepsilon'_{i+1} = -\frac{\alpha_i}{\alpha_i P_i + \beta_i}\varepsilon_i$$

is the come error of ε_i -th order;

2) because the additional error (ε''_{i+1} generated when calculating the value of Q_{i+1}) may be positive or negative with probability 0.5, the total error ($\varepsilon_{i+1} = \varepsilon'_{i+1} + \varepsilon''_{i+1}$) is of ε_i -th order.

Let us show that the simultaneous satisfaction of inequalities (3.19) and (3.20) is the sufficient condition of stability of the calculation according to recurrence formula (3.14) of the forward sweep.

Let ε_i be a small error in the calculated value of P_i. Using formula (3.14) and expansion $1/(1+x) = 1 - x + x^2 - x^3 + \ldots$ from table "Important Series Expansions" [3], we have the following chain of equalities:

$$P_{i+1} + \varepsilon'_{i+1} = -\frac{\gamma_i}{\alpha_i(P_i + \varepsilon_i) + \beta_i} = -\frac{\gamma_i}{\alpha_i P_i + \beta_i} \times \frac{1}{1 + \alpha_i \varepsilon_i /(\alpha_i P_i + \beta_i)} =$$

$$= P_{i+1}\left(1 - \frac{\alpha_i}{\alpha_i P_i + \beta_i}\varepsilon_i\right) + O(\varepsilon_i^2) = P_{i+1} - P_{i+1}\frac{\alpha_i}{\alpha_i P_i + \beta_i}\varepsilon_i + O(\varepsilon_i^2).$$

By comparing the beginning and end of this chain, we obtain

$$\varepsilon'_{i+1} = -P_{i+1}\frac{\alpha_i}{\alpha_i P_i + \beta_i}\varepsilon_i . \tag{3.21}$$

If inequality (3.19), which looks like $\left|P_{i+1}\right| \le 1$, and inequality (3.20) are satisfied, then

$$\left|P_{i+1}\frac{\alpha_i}{\alpha_i P_i + \beta_i}\right| \le 1.$$

According to (3.21) and the last inequality, we can state the following:

1) small error ε_i (which is a part of the calculated value of P_i) goes into the value of P_{i+1} without increase; the come error, ε'_{i+1}, is defined by formula (3.21), i.e., ε'_{i+1} is of ε_i -th order;

2) because the additional error (ε''_{i+1} generated when calculating the value of P_{i+1}) may be positive or negative with probability 0.5, the total error ($\varepsilon_{i+1} = \varepsilon'_{i+1} + \varepsilon''_{i+1}$) is of ε_i -th order.

That is, the algorithm of the calculation according to formula (3.14) is stable.

Thus, the simultaneous satisfaction of inequalities (3.19) and (3.20) for all values of *i* is enough for stability of the forward and backward sweeps, i.e., of the decomposition method in general.

By means of conditions (3.19) and (3.20), we will obtain other stability conditions of the decomposition method, which are more convenient to use.

Let us assume that

$$0 \le P_{k+1} \le 1 \tag{3.22}$$

and inequalities

$$\gamma_i \ge \alpha_i, \quad -\beta_i \ge \alpha_i + \gamma_i, \tag{3.23}$$

$$\alpha_i > 0 \tag{3.24}$$

are simultaneously satisfied for all values of *i*. Then inequalities (3.19) and (3.20) are satisfied for all values of *i*, i.e., the decomposition method is stable.

To prove the last assertion, we transform formula (3.14) as follows:

$$P_{i+1} = -\frac{\gamma_i}{\alpha_i P_i + \beta_i} = \frac{\gamma_i}{\gamma_i + (-\beta_i - \alpha_i - \gamma_i) + \alpha_i (1 - P_i)} = \frac{\gamma_i}{\gamma_i + \Delta_i},$$

where $\Delta_i = (-\beta_i - \alpha_i - \gamma_i) + \alpha_i (1 - P_i)$.

Similarly, we obtain the following expression for a part of formula (3.15):

$$-\frac{\alpha_i}{\alpha_i P_i + \beta_i} = \frac{\alpha_i}{\gamma_i + \Delta_i}.$$

If $0 \le P_i \le 1$, inequality $1 - P_i \ge 0$ is satisfied. Then (3.23) and (3.24) result in $\Delta_i \ge 0$ and

$$0 < P_{i+1} \le 1, \tag{3.25}$$

$$0 < -\frac{\alpha_i}{\alpha_i P_i + \beta_i} \le 1. \tag{3.26}$$

According to (3.22), inequalities (3.25) and (3.26) are satisfied for $i = k+1$, $k+2$, ..., $r-1$. As the consequence of this, inequalities (3.19) and (3.20) are satisfied for all values of *i*, i.e., the decomposition method is stable.

We proved that conditions (3.22) — (3.24), as well as conditions (3.19) and (3.20), are the sufficient stability conditions for the decomposition method.

Because coefficients α_i, β_i and γ_i are defined by expressions (3.10), the consequence of the obtained stability conditions is the unconditional stability of the decomposition method for solving boundary value problem (3.6) — (3.8), for which the following conditions are satisfied:

1) $g(x) = 0$ for all values of *x* from *a* to *b*, i.e., equation (3.6) looks like

$$\frac{d^2u}{dx^2} + e(x)u = f(x);$$

2) $e(x) \le 0$ for all values of *x*;

3) left boundary condition (3.7) is such that $0 \le P_{k+1} \le 1$.

The unconditional stability means the stability for arbitrary step *h* of the grid.

3.4. Simplification of the second-order linear differential equation

Thanks to the unconditional stability of solving the last boundary value problem (formulated at the end of the previous section), the substitution into equation (3.6), which excludes the first derivative of $u(x)$, is of interest. We will show that such substitution looks like

$$u(x) = U(x)\exp\left(-0.5\int_a^x g(y)dy \right). \tag{3.27}$$

The exponential function (Appendix 3) is figured in this expression.

Let us differentiate function (3.27) twice. Using the basic rules of differentiation [3], we obtain

$$\frac{du}{dx} = \left(\frac{dU}{dx} - 0.5gU \right)\exp\left(-0.5\int_a^x g(y)dy \right), \tag{3.28}$$

$$\frac{d^2u}{dx^2} = \left[\frac{d^2U}{dx^2} - g\frac{dU}{dx} + (0.25g^2 - 0.5g')U \right]\exp\left(-0.5\int_a^x g(y)dy \right). \tag{3.29}$$

By substituting expressions (3.27) — (3.29) into equation (3.6), we obtain the following equation without the first derivative of unknown function $U(x)$:

$$\frac{d^2U}{dx^2} + E(x)U = F(x), \tag{3.30}$$

where

$$E = e(x) - 0.25g^2(x) - 0.5g'(x), \tag{3.31}$$

$$F = f(x)\exp\left(0.5\int_a^x g(y)dy \right). \tag{3.32}$$

Let expressions (3.27) and (3.28), in which $x = a$, be substituted into condition (3.7). We obtain the following left boundary condition for $U(x)$:

$$A_2 U(a) + A_1 U'(a) = A_3 , \tag{3.33}$$

where

$$A_2 = A_0 - 0.5 A_1 g(a) , \tag{3.34}$$

$$A_3 = A . \tag{3.35}$$

After substituting expressions (3.27) and (3.28), in which $x = b$, into condition (3.8), we have the following right boundary condition:

$$B_2 U(b) + B_1 U'(b) = B_3 , \tag{3.36}$$

where

$$B_2 = B_0 - 0.5 B_1 g(b) , \tag{3.37}$$

$$B_3 = B \exp\left(0.5 \int_a^b g(y) dy \right) . \tag{3.38}$$

We will use substitution (3.27) not only in this chapter, but in the next chapter too.

3.5. Program realization of the decomposition method

For solving equation (3.6) with boundary conditions (3.7) and (3.8) by the decomposition method, we insert module Module2 into the BookNM workbook (p. 194) and put the following subroutine declaration into this module:

Listing 3.1

```
Sub fb(ByVal k, ByVal r, ByVal h, ByVal A, ByVal B, _
    ByRef G() As Double, ByRef E() As Double, _
    ByRef F() As Double, ByRef U() As Double)
    Const BETAK = -2, GAMMAK = 0
    Dim DELTAK As Double
    Const ALPHAR = 0, BETAR = -2
    Dim DELTAR As Double
    Dim alpha As Double, beta As Double
    Dim gamma As Double, delta As Double
    Dim i As Integer, w As Double
    Dim P() As Double: ReDim P(k + 1 To r)
    Dim Q() As Double: ReDim Q(k + 1 To r)
    DELTAK = -2 * A
    DELTAR = -2 * B
'Forward sweep:
    P(k + 1) = -GAMMAK / BETAK
    Q(k + 1) = DELTAK / BETAK
    For i = k + 1 To r - 1
        w = 0.5 * G(i) * h
        alpha = 1 - w
        beta = E(i) * h ^ 2 - 2
        gamma = 1 + w
        delta = F(i) * h ^ 2
        w = alpha * P(i) + beta
        P(i + 1) = -gamma / w
        Q(i + 1) = (delta - alpha * Q(i)) / w
    Next i
'Backward sweep:
    U(r) = (DELTAR - ALPHAR * Q(r)) / _
    (ALPHAR * P(r) + BETAR)
    For i = r To k + 1 Step -1
        U(i - 1) = P(i) * U(i) + Q(i)
    Next i
End Sub
```

The subroutine name (`fb`) occurs from “forward-backward”: in the decomposition method, we use the sweep from left to right (in the direction of the *x* axis) and then from right to left (in the opposite direction).

If necessary, it is easy to obtain formulas of the “backward-forward” decomposition method, in which the sweep from right to left is used, and then from left to right.

The `fb` subroutine parameters have the following sense:

- `k`, `r` are numbers of the left and right boundary nodes of the main grid on segment $[a, b]$ (Fig. 3.1);
- `h` is a grid step, $h = (b - a)/(r - k)$;
- `A`, `B` are values of the solution of equation (3.6) on the left and right boundaries of segment $[a, b]$ according to boundary conditions (3.7) and (3.8);
- `G`, `E` are arrays of values of the coefficients of equation (3.6) at the nodes of the main grid;
- `F` is an array of values of the right-hand side of equation (3.6);
- `U` is an array intended for the solution values.

3.6. Examples of using the decomposition method

As an example of using the decomposition method, we will solve equation

$$\frac{d^2u}{dx^2} + x^2 \frac{du}{dx} - 3c\,xu = 0 \qquad (3.39)$$

on segment $0 \leq x \leq b$ with boundary conditions

$$u(0) = 1, \quad u(b) = 0 \qquad (3.40)$$

for $c = 10$ and $b = 1.5$.

According to Task 3 of the second chapter in book [5], equation (3.39) was obtained while studying temperature characteristics of a radial flow between parallel round disks. The radial flow is a horizontal movement of liquid, gas or plasma from the general center or to the center.

Equation (3.39) can be written in form (3.6), where

$$g(x) = x^2, \quad e(x) = -3c\,x, \quad f(x) = 0\,.$$

It is easy to verify that conditions (3.23) and (3.24) are satisfied for all values of x from 0 to b if $h < 2/b^2$ = 0.889 (that is, if $r-k \geq 2$). Besides, according to (3.16) and (3.11), $P_{k+1} = 0$, i.e., condition (3.22) is satisfied too. Thus, the sufficient stability conditions for the decomposition method are satisfied if $r-k \geq 2$.

For solving problem (3.39), (3.40), let us consider a program with the following source data table.

c	10
b	1.5
l	15
x	u

The first three rows of this table contain the values of parameters c and b and the number of steps, l = $r-k$. The bottom row contains titles of the Excel columns intended for the solution result, i.e., for the $u(x)$ function in tabular form.

Instead of text

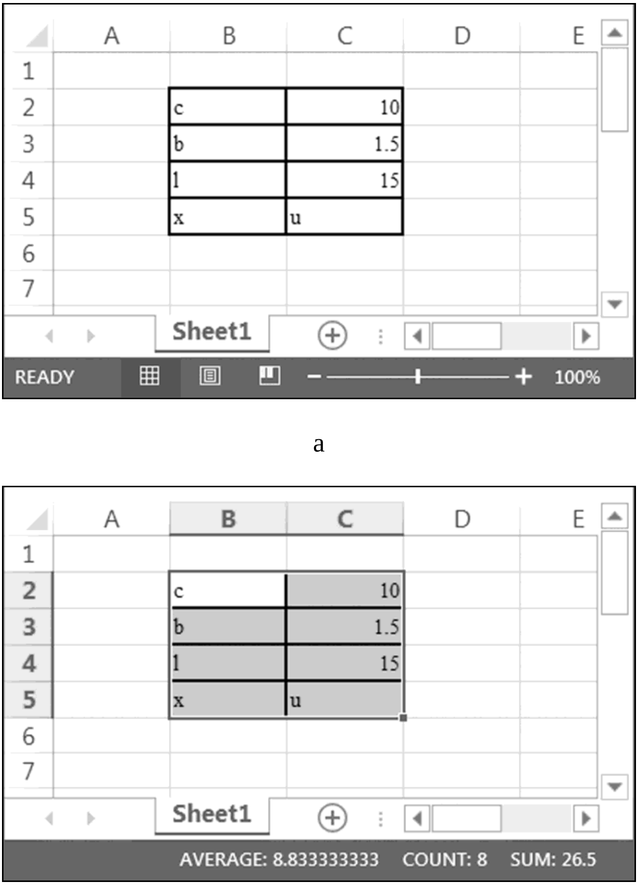
```
Sub main()

End Sub
```

we enter the following program text into Module1 of the BookNM workbook:

Listing 3.2

```
Sub main()
    Dim X() As Double
    Dim G() As Double
    Dim E() As Double
    Dim F() As Double
    Dim U() As Double
    Dim c As Double, b As Double, l As Integer
    Dim h As Double, i As Integer
    c = Selection.Cells(1, 2)
    b = Selection.Cells(2, 2)
    l = Selection.Cells(3, 2)
    h = b / l
    ReDim X(5 To 5 + l)
    ReDim G(5 To 5 + l)
    ReDim E(5 To 5 + l)
    ReDim F(5 To 5 + l)
    ReDim U(5 To 5 + l)
    For i = 5 To 5 + l
        X(i) = (i - 5) * h
1:      G(i) = X(i) ^ 2
2:      E(i) = -3 * c * X(i)
3:      F(i) = 0
    Next i
4:  Call fb(5, 5 + l, h, 1, 0, G, E, F, U)
    For i = 5 To 5 + l
        Selection.Cells(i, 1) = X(i)
5:      Selection.Cells(i, 2) = U(i)
    Next i
End Sub
```

It was mentioned above that the source data for this program are the values located in the Excel table (Fig. 3.2a). We must select this table before the program execution (Fig. 3.2b).



	A	B	C	D	E
1					
2		c	10		
3		b	1.5		
4		l	15		
5		x	u		
6					
7					

READY 100%

a

	A	B	C	D	E
1					
2		c	10		
3		b	1.5		
4		l	15		
5		x	u		
6					
7					

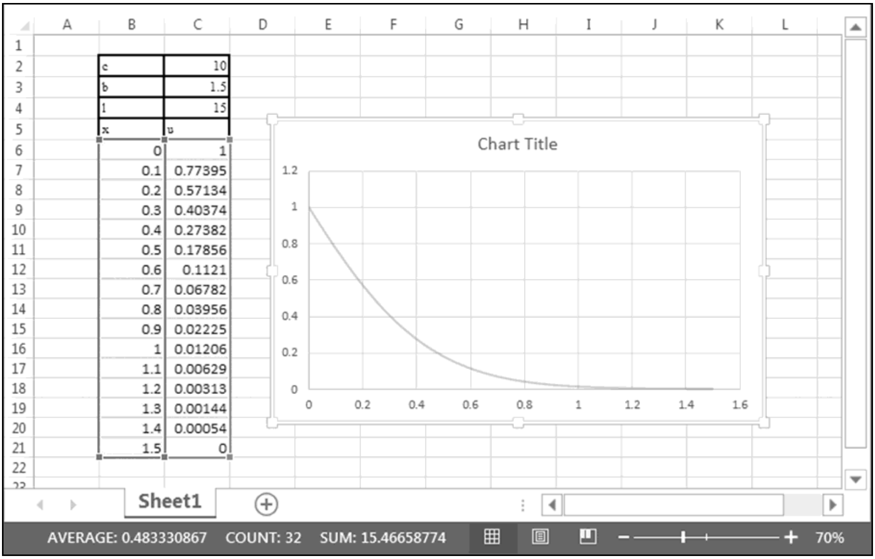
AVERAGE: 8.833333333 COUNT: 8 SUM: 26.5

b

Fig. 3.2. The Excel worksheet (a) before and
(b) after selection of the source data table

The calculated coordinates of the grid nodes and values of the solution are located in columns *x* and *u* (Fig. 3.3). For obtaining the $u(x)$ graph, located on the Excel worksheet, we must fulfill the following operations:

1) select the values of *x* and *u*, i.e., the B6:C21 range;
2) activate the *Insert* tab;
3) perform the *Insert Scatter* command in area *Charts*;
4) perform command *Scatter with Smooth Lines*.



	B	C
2	c	10
3	b	1.5
4	l	15
5	x	u
6	0	1
7	0.1	0.77395
8	0.2	0.57134
9	0.3	0.40374
10	0.4	0.27382
11	0.5	0.17856
12	0.6	0.1121
13	0.7	0.06782
14	0.8	0.03956
15	0.9	0.02225
16	1	0.01206
17	1.1	0.00629
18	1.2	0.00313
19	1.3	0.00144
20	1.4	0.00054
21	1.5	0

AVERAGE: 0.483330867 COUNT: 32 SUM: 15.46658774

Fig. 3.3. The calculated dependence, $u(x)$, and its graph

By substitution (3.27),

$$u = U(x)\exp\left(-0.5\int_0^x y^2 dy\right) = U(x)\exp\left(-\frac{x^3}{6}\right), \tag{3.41}$$

we can bring boundary value problem (3.39), (3.40) to form (3.30), (3.33), (3.36), i.e.,

$$\frac{d^2U}{dx^2} - [0.25x^4 + (3c+1)x]U = 0, \tag{3.42}$$

$$U(0) = 1, \quad U(b) = 0. \tag{3.43}$$

As we see, derivative dU/dx is absent in equation (3.42), i.e., the 1st item of the conditions on p. 207 is satisfied. The 2nd and 3rd items are also satisfied because:

- the coefficient in front of U is less or equal to zero when $x \geq 0$;
- $P_{k+1} = 0$ according to (3.16) and (3.11).

Thus, the decomposition method is unconditionally stable for solving boundary value problem (3.42), (3.43).

The following program is used as the previous one.

Listing 3.3

```
Sub main()
    Dim X() As Double
    Dim G() As Double
    Dim E() As Double
    Dim F() As Double
    Dim U() As Double
    Dim c As Double, b As Double, l As Integer
    Dim h As Double, i As Integer
    c = Selection.Cells(1, 2)
    b = Selection.Cells(2, 2)
    l = Selection.Cells(3, 2)
    h = b / l
    ReDim X(5 To 5 + l)
    ReDim G(5 To 5 + l)
    ReDim E(5 To 5 + l)
    ReDim F(5 To 5 + l)
    ReDim U(5 To 5 + l)
    For i = 5 To 5 + l
        X(i) = (i - 5) * h
1:      G(i) = 0
2:      E(i) = -0.25 * X(i) ^ 4 - (3 * c + 1) * X(i)
3:      F(i) = 0
    Next i
4:  Call fb(5, 5 + l, h, 1, 0, G, E, F, U)
    For i = 5 To 5 + l
        Selection.Cells(i, 1) = X(i)
5:      Selection.Cells(i, 2) = U(i) * _
            Exp(-X(i) ^ 3 / 6)
    Next i
End Sub
```

This program solves boundary value problem (3.42), (3.43) and then calculates the $u(x)$ dependence by means of formula (3.41). It differs from program Listing 3.2 in operators `1`, `2` and `5`. The result of using program Listing 3.3, the $u(x)$ function in tabular form, is close to the previous result depicted in Fig. 3.3.

Boundary value problem (3.42), (3.43) will be also used in Section 4.11 to demonstrate the cubic spline method for solving the second-order linear differential equation.

3.7. Examples of the computing error. Instability and loss of accuracy

The decomposition method usage can lead to an appreciable computing error. We will consider this question on an example of equation

$$\frac{d^2u}{dx^2} + c\frac{du}{dx} = c \qquad (3.44)$$

on segment $0 \le x \le b$ with boundary conditions

$$u(0) = 0, \quad u(b) = b\,. \qquad (3.45)$$

It is obvious that the solution of this boundary value problem looks like $u(x) = x$ for any value of the c constant.

Equation (3.44) can be written in form (3.6), where

$$g(x) = f(x) = c\,,$$

$$e(x) = 0\,.$$

To solve problem (3.44), (3.45) by the decomposition method, we change operators `1` — `4` of program Listing 3.2 to get the following program:

Listing 3.4

```
Sub main()
    Dim X() As Double
    Dim G() As Double
    Dim E() As Double
    Dim F() As Double
    Dim U() As Double
    Dim c As Double, b As Double, l As Integer
    Dim h As Double, i As Integer
    c = Selection.Cells(1, 2)
    b = Selection.Cells(2, 2)
    l = Selection.Cells(3, 2)
    h = b / l
    ReDim X(5 To 5 + l)
    ReDim G(5 To 5 + l)
    ReDim E(5 To 5 + l)
    ReDim F(5 To 5 + l)
    ReDim U(5 To 5 + l)
    For i = 5 To 5 + l
        X(i) = (i - 5) * h
1:      G(i) = c
2:      E(i) = 0
3:      F(i) = c
    Next i
4:  Call fb(5, 5 + l, h, 0, b, G, E, F, U)
    For i = 5 To 5 + l
        Selection.Cells(i, 1) = X(i)
5:      Selection.Cells(i, 2) = U(i)
    Next i
End Sub
```

This program is used as the programs of the previous section. In Fig. 3.4, we see the execution results for three values of c, equal to 10, 10^{17} and 10^{18}.

For problem (3.44), (3.45), it is easy to make sure that conditions (3.22) and (3.23) are satisfied for any positive values of c and h.

For $h = b / l = 0.1$, we can state the following about

$$\alpha_i = 1 - 0.5 g_i h = 1 - 0.5ch = 1 - 0.05c ,$$

condition (3.24) and the stability with respect to the computing error:

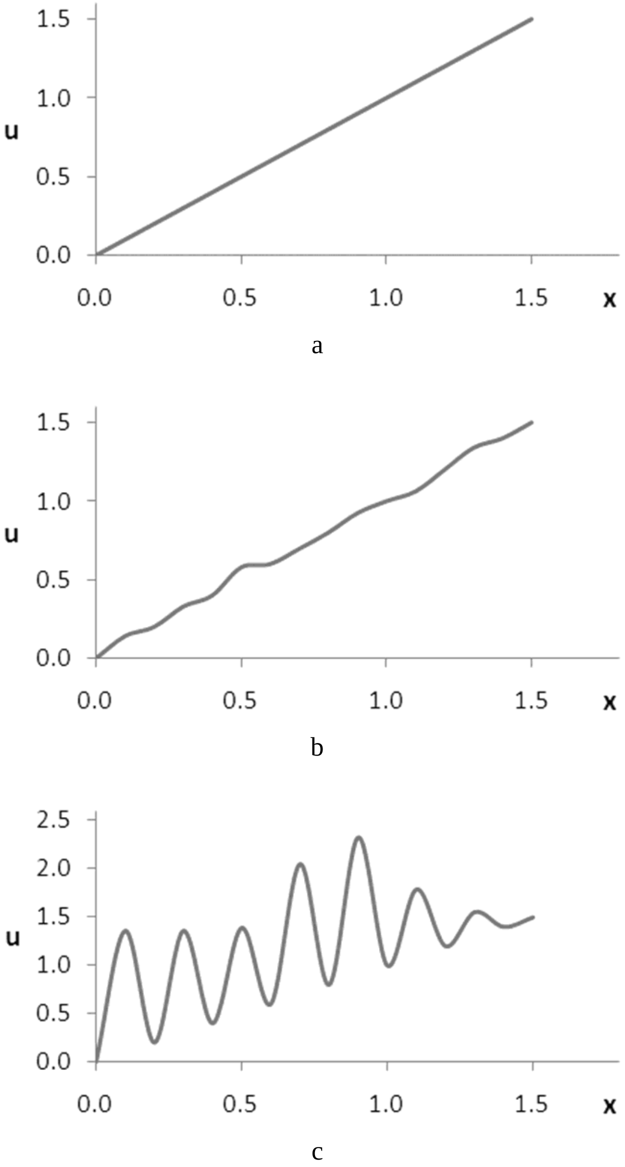
- for $c = 10$, the value of α_i is positive, i.e., condition (3.24) is satisfied, therefore, the decomposition method is stable;
- for $c = 10^{17}$ and 10^{18}, the value of α_i is negative, i.e., condition (3.24) is not satisfied, therefore, the decomposition method can be either stable or unstable.

The sign change of α_i occurs at $c = 20$.

According to Fig. 3.4a and 3.4b, an appreciable deviation of the calculated $u(x)$ dependence from the exact $u(x)$ dependence, $u(x) = x$, appears at value $c = 10^{17}$, which exceeds value $c = 20$ by several orders of magnitude. It follows from the fact that sufficient stability conditions (3.22) — (3.24) are not necessary, i.e., the process of solving the finite difference scheme for boundary value problem (3.6) — (3.8) by the decomposition method may remain stable with respect to the computing error if not all conditions from (3.22) — (3.24) are satisfied.

According to Fig. 3.4c, the computing error of the solution can exceed 100 % for large values of c.

Let us use substitution (3.27) for solving problem (3.44), (3.45).



a

b

c

Fig. 3.4. Graphic results of solving problem (3.44), (3.45) by program Listing 3.4 for b = 1.5, l = 15 and the following values of c: 10 (a), 10^{17} (b), 10^{18} (c)

Substitution

$$u = U(x)\exp\left(-0.5\int_a^x g(y)dy\right) =$$

$$= U(x)\exp\left(-0.5\int_0^x c dy\right) = U(x)\exp(-0.5cx) \qquad (3.46)$$

into (3.44), (3.45) gives the following equation and boundary conditions:

$$\frac{d^2U}{dx^2} - 0.25c^2\,U = c\exp(0.5cx)\,, \qquad (3.47)$$

$$U(0) = 0\,, \quad U(b) = b\exp(0.5cb)\,. \qquad (3.48)$$

For boundary value problem (3.47), (3.48), it is easy to make sure that all the conditions, formulated on p. 207, are satisfied when $x \geq 0$, i.e., the decomposition method is unconditionally stable.

The following program is used as the previous one.

Listing 3.5

```
Sub main()
    Dim X() As Double
    Dim G() As Double
    Dim E() As Double
    Dim F() As Double
    Dim U() As Double
    Dim c As Double, b As Double, l As Integer
    Dim h As Double, i As Integer
    c = Selection.Cells(1, 2)
    b = Selection.Cells(2, 2)
    l = Selection.Cells(3, 2)
    h = b / l
    ReDim X(5 To 5 + l)
    ReDim G(5 To 5 + l)
    ReDim E(5 To 5 + l)
    ReDim F(5 To 5 + l)
    ReDim U(5 To 5 + l)
    For i = 5 To 5 + l
        X(i) = (i - 5) * h
1:      G(i) = 0
2:      E(i) = -0.25 * c ^ 2
3:      F(i) = c * Exp(0.5 * c * X(i))
```

```
    Next i
4:  Call fb(5, 5 + l, h, 0, b * Exp(0.5 * c * b), _
    G, E, F, U)
    For i = 5 To 5 + l
        Selection.Cells(i, 1) = X(i)
5:      Selection.Cells(i, 2) = U(i) * _
            Exp(-0.5 * c * X(i))
    Next i
End Sub
```

Program Listing 3.5 solves boundary value problem (3.47), (3.48) and then calculates the $u(x)$ dependence by means of formula (3.46). This program differs from Listing 3.4 in operators 1 — 5.

Fig. 3.5 shows the results of using program Listing 3.5 at c = 10 and 20. Unlike program Listing 3.4, program Listing 3.5 was not used at $c = 10^{17}$ and 10^{18} because of exceeding 709.782712893 by the argument of the exponential function in operator 3. In other words, according to Fig. 3.6, the stop of the execution of program Listing 3.5 would occur at $c = 10^{17}$ and 10^{18} with the following information: *Run-time error '6': Overflow*.

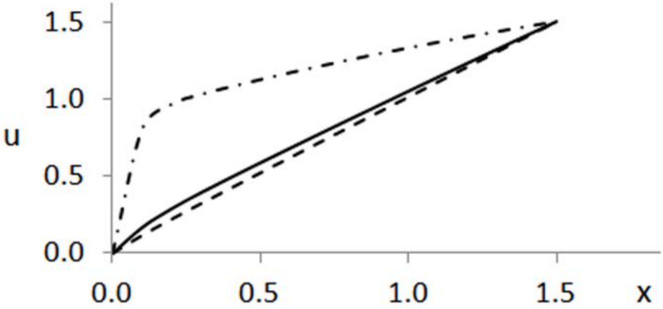
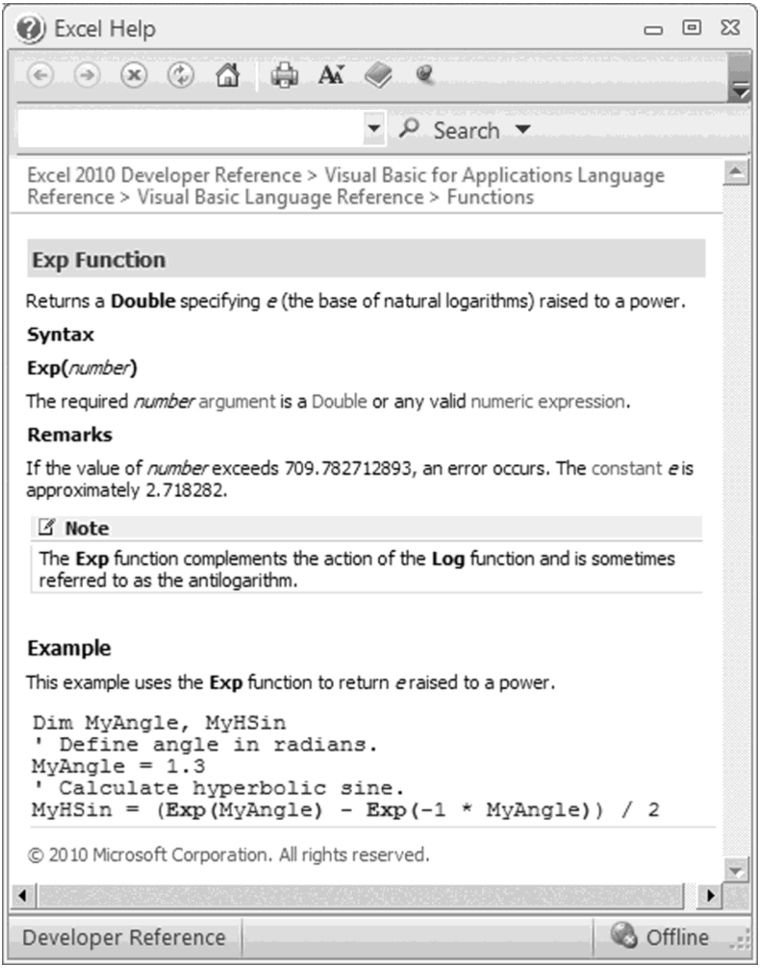


Fig. 3.5. Graphic results of using program Listing 3.5:
the continuous line — c = 10, l = 15;
the dashed line — c = 10, l = 30;
the dash-dotted line — c = 20, l = 15

According to the continuous and dashed lines in Fig. 3.5, we must reduce the number of steps on segment $[0, b]$ for improving the accuracy. According to the dash-dotted line, the accuracy may be lost even when the computing process is stable with respect to the computing error. We will speak about measures against loss of accuracy also in section "Instead of Conclusions".


Excel Help
Search
Excel 2010 Developer Reference > Visual Basic for Applications Language Reference > Visual Basic Language Reference > Functions
Developer Reference
Offline

Exp Function

Returns a **Double** specifying *e* (the base of natural logarithms) raised to a power.

Syntax

Exp(*number***)**

The required *number* argument is a Double or any valid numeric expression.

Remarks

If the value of *number* exceeds 709.782712893, an error occurs. The constant *e* is approximately 2.718282.

Note

The **Exp** function complements the action of the **Log** function and is sometimes referred to as the antilogarithm.

Example

This example uses the **Exp** function to return *e* raised to a power.

```
Dim MyAngle, MyHSin
' Define angle in radians.
MyAngle = 1.3
' Calculate hyperbolic sine.
MyHSin = (Exp(MyAngle) - Exp(-1 * MyAngle)) / 2
```

© 2010 Microsoft Corporation. All rights reserved.

Fig. 3.6. The *Excel Help* window with the Exp function description

It is natural to ask about the existence of a method for solving the finite difference scheme for boundary value problem (3.6) — (3.8), which is more stable with respect to the computing error than the decomposition method used by us. Before answering this question, we will consider several methods for solving the system of linear algebraic equations of general form.

3.8. Solving the system of linear algebraic equations by using Excel functions

We should solve the system of linear algebraic equations

$$\begin{array}{ccccccccc} a_{11}x_1 & + & a_{12}x_2 & + & \ldots & + & a_{1n}x_n & = & f_1, \\ a_{21}x_1 & + & a_{22}x_2 & + & \ldots & + & a_{2n}x_n & = & f_2, \\ . & . & . & . & . & . & . & . & . \\ a_{n1}x_1 & + & a_{n2}x_2 & + & \ldots & + & a_{nn}x_n & = & f_n \end{array} \qquad (3.49)$$

with coefficient (system) matrix

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \ldots & a_{1n} \\ a_{21} & a_{22} & \ldots & a_{2n} \\ . & . & . & . \\ a_{n1} & a_{n2} & \ldots & a_{nn} \end{bmatrix} \qquad (3.50)$$

of general form: any element a_{ij} can be either zero or nonzero.

According to the definitions of matrix multiplication and equality, given in Section 1.21, system (3.49) can be written as matrix equation

$$\mathbf{Ax} = \mathbf{f} \ , \qquad (3.51)$$

where **f** is the vector of the right-hand sides, **x** is the vector of the unknown variables:

$$\mathbf{f} = \begin{bmatrix} f_1 \\ f_2 \\ \ldots \\ f_n \end{bmatrix}, \quad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \ldots \\ x_n \end{bmatrix}.$$

System (3.49) or, which is the same, matrix equation (3.51) can be solved by means of Excel, without programming in VBA. Let us consider two methods of the solution.

1. The solution of equation (3.51) can be written as $\mathbf{x} = \mathbf{A}^{-1}\mathbf{f}$, where $\mathbf{A}^{-1}$ is the inverse **A** matrix (Section 1.21).

The $\mathbf{A}^{-1}$ matrix can be calculated by means of the built-in MINVERSE function. The subsequent multiplication of the $\mathbf{A}^{-1}$ matrix and the **f** vector can be performed by means of the built-in MMULT function. Functions MINVERSE and MMULT return an array, as well as built-in function TRANSPOSE and user-defined function TRANSPOSEA developed by us (Section 2.3).

The drawback of this method for solving matrix equation (3.51) is its inefficiency (with respect to the execution time) for large values of n because the matrix inversion is essentially the solution of the following n matrix equations:

$$\mathbf{A}^{\mathbf{T}}\mathbf{x} = \mathbf{f}^{1},\ \mathbf{A}^{\mathbf{T}}\mathbf{x} = \mathbf{f}^{2},\ \ldots,\ \mathbf{A}^{\mathbf{T}}\mathbf{x} = \mathbf{f}^{n},$$

where $\mathbf{A}^{\mathbf{T}}$ is the transposed $\mathbf{A}$ matrix,

$$\mathbf{f}^{1} = \begin{bmatrix} 1 \\ 0 \\ \cdots \\ 0 \end{bmatrix},\ \mathbf{f}^{2} = \begin{bmatrix} 0 \\ 1 \\ \cdots \\ 0 \end{bmatrix},\ \ldots,\ \mathbf{f}^{n} = \begin{bmatrix} 0 \\ 0 \\ \cdots \\ 1 \end{bmatrix}.$$

Indeed, if

$$\mathbf{x}^{1} = \begin{bmatrix} x_1^1 \\ x_2^1 \\ \cdots \\ x_n^1 \end{bmatrix},\ \mathbf{x}^{2} = \begin{bmatrix} x_1^2 \\ x_2^2 \\ \cdots \\ x_n^2 \end{bmatrix},\ \ldots,\ \mathbf{x}^{n} = \begin{bmatrix} x_1^n \\ x_2^n \\ \cdots \\ x_n^n \end{bmatrix}$$

are the solutions of the above n matrix equations and

$$\mathbf{B} = \begin{bmatrix} x_1^1 & x_2^1 & \cdots & x_n^1 \\ x_1^2 & x_2^2 & \cdots & x_n^2 \\ \cdot & \cdot & \cdot & \cdot \\ x_1^n & x_2^n & \cdots & x_n^n \end{bmatrix}, \qquad (3.52)$$

it is easy to make sure that $\mathbf{BA} = \mathbf{E}$, where **E** is the unit matrix, that is, $\mathbf{B} = \mathbf{A}^{-1}$.

2. Matrix equation (3.51) can be solved by Cramer's rule [3], which is reduced to computation of $n+1$ determinants of n-th order, D, D_1, D_2, ..., D_n. These determinants are calculated by using the MDETERM function. For large values of n, this method is even less efficient than the previous one.

Many numerical methods need the solution of system (3.49) or matrix equation (3.51). For this purpose, let us develop two subroutines: the first subroutine realizes the classical Gaussian elimination method; the second one realizes a modernization of this method.

3.9. Solving the system of linear algebraic equations by the Gaussian elimination method

In a summary, the classical Gaussian elimination method can be described as follows.

Using the first equation of the system of *n* equations (3.49), the x_1 unknown is eliminated from the remaining $n-1$ equations. Further, by means of the 2nd equation of the resulting system (of *n* equations), x_2 is eliminated from the next $n-2$ equations. By using the 3rd equation of the new system (of *n* equations), x_3 is eliminated from the next $n-3$ equations. And so on, until we get the equation determining the x_n unknown. In the last turn, the remaining $n-1$ unknowns are calculated in the reverse sequence: x_{n-1}, x_{n-2}, ..., x_1. To do this, the already calculated unknowns are substituted into the equations, starting with the penultimate equation.

In more details, the Gaussian elimination method includes two stages, which are called the forward and backward courses. ***The forward course*** consists in the transformation of matrix equation (3.51) to equation

$$\mathbf{Bx} = \mathbf{g} \tag{3.53}$$

with matrix

$$\mathbf{B} = \begin{bmatrix} 1 & b_{12} & \dots & b_{1n} \\ 0 & 1 & \dots & b_{2n} \\ . & . & . & . \\ 0 & 0 & \dots & 1 \end{bmatrix} \tag{3.54}$$

called an upper triangular matrix because nonzero elements are on the main diagonal and above it. ***The backward course*** is the solution of matrix equation (3.53) relative to the **x** vector of the unknowns.

Let us consider the forward course of the Gaussian elimination method.

We assume that the a_{11} coefficient is nonzero. In this case, when dividing both sides of the 1st equation of system (3.49) by this coefficient, we obtain equation

$$x_1 + b_{12}x_2 + \dots + b_{1n}x_n = g_1 , \tag{3.55}$$

where $b_{1j} = \dfrac{a_{1j}}{a_{11}}$, $g_1 = \dfrac{f_1}{a_{11}}$, j = 2, ..., n. The last equation is the result of the first step of the forward course.

Let us multiply both sides of (3.55) by a_{21} and subtract the resulting equation from the 2nd equation of system (3.49). We obtain an equation without x_1 .

Let us multiply both sides of (3.55) by a_{31} and subtract the resulting equation from the 3rd equation of system (3.49). We obtain one more equation without x_1 , and so on.

Let us multiply both sides of (3.55) by a_{n1} and subtract the resulting equation from the n-th equation of system (3.49).

The result is the following system of linear algebraic equations without x_1 :

$$\begin{array}{ccccccc} a_{22}^{[2]}x_2 & + & \dots & + & a_{2n}^{[2]}x_n & = & f_2^{[2]}, \\ . & . & . & . & . & . & . \\ a_{n2}^{[2]}x_2 & + & \dots & + & a_{nn}^{[2]}x_n & = & f_n^{[2]}, \end{array} \tag{3.56}$$

where $a_{ij}^{[2]} = a_{ij} - a_{i1}b_{1j}$, $f_i^{[2]} = f_i - a_{i1}g_1$, i = 2, ..., n, j = 2, ..., n. The index in square brackets is the number of the current step of the forward course.

Let us assume that the $a_{22}^{[2]}$ coefficient is nonzero. Dividing both sides of the 1st equation of system (3.56) by this coefficient, we obtain equation

$$x_2 + b_{23}x_3 + \dots + b_{2n}x_n = g_2 , \tag{3.57}$$

where $b_{2j} = \dfrac{a_{2j}^{[2]}}{a_{22}^{[2]}}$, $g_2 = \dfrac{f_2^{[2]}}{a_{22}^{[2]}}$, j = 3, ..., n. The last equation is the result of the second step of the forward course.

To eliminate x_2, we fulfill the same operations with equation (3.57) and system (3.56) that we previously fulfilled with equation (3.55) and system (3.49).

Equation

$$x_n = g_n , \tag{3.58}$$

where $g_n = f_n^{[n]} / a_{nn}^{[n]}$, is the result of the n-th step of the forward course.

Equations (3.55), (3.57) and (3.58) give the following system of linear algebraic equations:

$$\begin{array}{ccccccccc} x_1 & + & b_{12}x_2 & + & \dots & + & b_{1n}x_n & = & g_1, \\ & & x_2 & + & \dots & + & b_{2n}x_n & = & g_2, \\ & & & & . & . & . & . & . \\ & & & & & & x_n & = & g_n. \end{array} \qquad (3.59)$$

This system is the result of the forward course of the Gaussian elimination method. We can write (3.59) in the form of matrix equation (3.53) with upper triangular matrix (3.54).

The backward course consists in the determination of the unknown variables by solving system (3.59). The following expressions are used:

1) according to the ultimate equation,

$$x_n = g_n ;$$

2) according to the penultimate equation,

$$x_{n-1} = g_{n-1} - b_{n-1,n}x_n ;$$

3) according to the $(n-2)$th equation,

$$x_{n-2} = g_{n-2} - b_{n-2,n-1}x_{n-1} - b_{n-2,n}x_n ,$$

and so on.

An analysis of the forward course of the Gaussian elimination method shows that it requires $O(n^3)$ arithmetic operations, $n \to \infty$. Similarly, the backward course requires $O(n^2)$ arithmetic operations. Thus, the solution of the system of linear algebraic equations (3.49) by the Gaussian elimination method requires $O(n^3)$ arithmetic operations.

3.10. Two subroutines for solving the system of linear algebraic equations

Into Module3 of the BookNM workbook, we put the following declaration of the `gaus` subroutine, realizing the classical Gaussian elimination method.

Listing 3.6

```
Sub gaus(ByVal n, ByRef A() As Double, _
    ByRef F() As Double, ByRef X() As Double)
    Dim i As Integer, j As Integer, k As Integer
    Dim AA() As Double: ReDim AA(1 To n, 1 To n)
    Dim FF() As Double: ReDim FF(1 To n)
    Dim B() As Double: ReDim B(1 To n, 1 To n)
    Dim G() As Double: ReDim G(1 To n)
'Forward course:
    For i = 1 To n
        For j = 1 To n
            AA(i, j) = A(i, j)
        Next j
        FF(i) = F(i)
    Next i
    For k = 1 To n - 1
        For j = k + 1 To n
            B(k, j) = AA(k, j) / AA(k, k)
        Next j
        G(k) = FF(k) / AA(k, k)
        'end of step No. k
        'beginning of step No. k + 1
        For i = k + 1 To n
            For j = k + 1 To n
                AA(i, j) = AA(i, j) - AA(i, k) * _
                B(k, j)
            Next j
            FF(i) = FF(i) - AA(i, k) * G(k)
        Next i
    Next k
    G(n) = FF(n) / AA(n, n)
'Backward course:
    X(n) = G(n)
    For k = n - 1 To 1 Step -1
        X(k) = G(k)
        For j = k + 1 To n
            X(k) = X(k) - B(k, j) * X(j)
        Next j
    Next k
End Sub
```

The `gaus` subroutine parameters have the following sense:

- `n` is the number of the unknowns;
- `A` is an array corresponding to matrix (3.50) of matrix equation (3.51);
- `F`, `X` are arrays corresponding to the **f** and **x** vectors in (3.51).

Below, we will consider the program realization of the Gaussian elimination method with choice of leading coefficient.

The leading coefficient is the coefficient, by which the division is performed. Thus, at the first step of the classical Gaussian elimination method's forward course, a_{11} is the leading coefficient; at the second step, $a_{22}^{[2]}$ is the leading coefficient, and so on. Sometimes, the division is incorrect because the leading coefficient may be very small (in absolute value) or equal to zero. Therefore, the following three modernizations of the Gaussian elimination method are used in practice.

1. At each step of the forward course, permuting the equations is performed to get the maximum (in absolute value) element of the first column of the system matrix as the leading coefficient. Thus, at the first step, matrix (3.50) of system (3.49) is processed; at the second step, matrix

$$\begin{bmatrix} a_{22}^{[2]} & \ldots & a_{2n}^{[2]} \\ . & . & . \\ a_{n2}^{[2]} & \ldots & a_{nn}^{[2]} \end{bmatrix}$$

of system (3.56) is processed, and so on. In this case, we say that the leading coefficient is chosen in the column.

2. Renumbering the unknowns is performed to get the maximum element of the first row of the system matrix as the leading coefficient. In this case, we say that the leading coefficient is chosen in the row.

3. Permuting the equations and renumbering the unknowns are performed to get the maximum element of the system matrix as the leading coefficient. That is, the whole matrix is considered when choosing the leading coefficient.

Into Module4 of the BookNM workbook, we enter the following declaration of the `gauss` subroutine, which realizes the Gaussian elimination method with choice of leading coefficient in the matrix.

Listing 3.7

```
Sub gauss(ByVal n, ByRef A() As Double, _
    ByRef F() As Double, ByRef X() As Double, _
    Optional c = 0, Optional d = 0)
    Dim i As Integer, j As Integer, k As Integer
    Dim i_max As Integer, j_max As Integer
    Dim m As Integer
    Dim w As Double
    Dim AA() As Double: ReDim AA(1 To n, 1 To n)
    Dim FF() As Double: ReDim FF(1 To n)
    Dim O() As Integer: ReDim O(1 To n)
    Dim XX() As Double: ReDim XX(1 To n)
    Dim B() As Double: ReDim B(1 To n, 1 To n)
    Dim G() As Double: ReDim G(1 To n)
    For j = 1 To n
        O(j) = j
    Next j
'Forward course:
    w = 0
    For i = 1 To n
        For j = 1 To n
            AA(i, j) = A(c + i, d + j)
            If Abs(AA(i, j)) > w Then
                w = Abs(AA(i, j))
                i_max = i
                j_max = j
            End If
        Next j
        FF(i) = F(c + i)
    Next i
    For j = 1 To n
        w = AA(i_max, j)
        AA(i_max, j) = AA(1, j)
```

```
19:     AA(1, j) = w
20: Next j
21: w = FF(i_max)
22: FF(i_max) = FF(1)
23: FF(1) = w
24: For i = 1 To n
25:     w = AA(i, j_max)
26:     AA(i, j_max) = AA(i, 1)
27:     AA(i, 1) = w
28: Next i
29: m = O(j_max)
30: O(j_max) = O(1)
31: O(1) = m
    For k = 1 To n - 1
        For j = k + 1 To n
            B(k, j) = AA(k, j) / AA(k, k)
        Next j
        G(k) = FF(k) / AA(k, k)
        'end of step No. k
        'beginning of step No. k + 1
        w = 0
        For i = k + 1 To n
            For j = k + 1 To n
                AA(i, j) = AA(i, j) - AA(i, k) * _
                B(k, j)
                If Abs(AA(i, j)) > w Then
                    w = Abs(AA(i, j))
                    i_max = i
                    j_max = j
                End If
            Next j
            FF(i) = FF(i) - AA(i, k) * G(k)
        Next i
        For j = k + 1 To n
            w = AA(i_max, j)
            AA(i_max, j) = AA(k + 1, j)
            AA(k + 1, j) = w
        Next j
        w = FF(i_max)
        FF(i_max) = FF(k + 1)
        FF(k + 1) = w
        For i = k + 1 To n
```

```
            w = AA(i, j_max)
            AA(i, j_max) = AA(i, k + 1)
            AA(i, k + 1) = w
        Next i
        m = O(j_max)
        O(j_max) = O(k + 1)
        O(k + 1) = m
        For i = 1 To k     'permuting columns of
                           'calculated part of array B
            w = B(i, j_max)
            B(i, j_max) = B(i, k + 1)
            B(i, k + 1) = w
        Next i
    Next k
    G(n) = FF(n) / AA(n, n)
'Backward course:
    XX(n) = G(n)
    For k = n - 1 To 1 Step -1
        XX(k) = G(k)
        For j = k + 1 To n
            XX(k) = XX(k) - B(k, j) * XX(j)
        Next j
    Next k
32: For j = 1 To n
33:     X(d + O(j)) = XX(j)
34: Next j
End Sub
```

Among parameters of the `gauss` subroutine, we see optional parameters `c` and `d`. They are necessary to have the possibility of solving the following system of linear algebraic equations with non-traditional (shifted) numbering of the system’s elements (matrix rows and columns, right-hand sides and unknown variables):

$$
\begin{array}{ccccccccc}
a_{c+1,\,d+1}x_{d+1} & + & a_{c+1,\,d+2}x_{d+2} & + & \dots & + & a_{c+1,\,d+n}x_{d+n} & = & f_{c+1}, \\
a_{c+2,\,d+1}x_{d+1} & + & a_{c+2,\,d+2}x_{d+2} & + & \dots & + & a_{c+2,\,d+n}x_{d+n} & = & f_{c+2}, \\
. & . & . & . & . & . & . & . & . \\
a_{c+n,\,d+1}x_{d+1} & + & a_{c+n,\,d+2}x_{d+2} & + & \dots & + & a_{c+n,\,d+n}x_{d+n} & = & f_{c+n},
\end{array}
$$

where c and d are given integers. Such numbering is used in code Listing 6.15 (see operator 8 corresponding to $n = c = d = 2$).

The presence of optional parameters is one of distinctions of the `gauss` subroutine from the previous `gaus`.

Let us briefly consider the `gauss` subroutine.

Listing 3.7 includes the `O` array of original serial numbers of the unknown variables. This array is necessary for recovery of the unknowns' numbering, which changes during the solution of system (3.49). The *j*-th element of the `O` array is the original number (but not shifted by *d*) of the *j*-th unknown variable.

Operators `1` — `3` prepare array `O` for the subsequent transformation.

Operators `4` — `15`:

1) determine elements of the `AA` array corresponding to matrix (3.50) of the system of linear algebraic equations (operator `7`);

2) find the maximum (in absolute value) element of the `AA` array; its serial numbers in the vertical and horizontal directions are respectively assigned to variables `i_max` and `j_max` (operators `8` — `12`);

3) determine elements of the `FF` array corresponding to the **f** vector of the right-hand sides of the system of linear algebraic equations (operator `14`).

Operators `16` — `20` interchange the 1st and `i_max`-th rows of the `AA` array, and operators `21` — `23` interchange the 1st and `i_max`-th elements of the `FF` array. It is equivalent to permuting the equations of system (3.49).

Operators `24` — `28` interchange the 1st and `j_max`-th columns of the `AA` array, and operators `29` — `31` interchange the 1st and `j_max`-th elements of the `O` array. It is equivalent to changing the numbering of the unknowns.

Thus, because of executing operators `16` — `31`, the maximum element of the `AA` array appears in the top left corner of `AA`. Further, the divisions by this element are performed (at the beginning of the `k` cycle at `k` = `1`). The first step of the forward course terminates on it.

In the subsequent steps of the forward course:

1) the elements of arrays `AA` and `FF` are calculated;

2) during calculation of the `AA` array values, the maximum element of the `AA` array is determined; its serial numbers in the vertical and horizontal directions are determined too;

3) arrays `AA`, `FF` and `O` are transformed;

4) the columns of the already calculated part of the `B` array, corresponding to the **B** matrix of system (3.59), are permuted because of the unknowns' renumbering;

5) the divisions by the maximum element of the `AA` array are performed (at the beginning of the `k` cycle at the next value of `k`).

After the considered operators of the forward course, obvious operators of the backward course follow.

The forward and backward courses' results, contained in the `XX` array, are the calculated values of the unknowns. Operators `32 — 34` determine variables x_{d+1}, x_{d+2}, ..., x_{d+n} by recovering the original numbering.

The `gauss` subroutine usage does not guarantee the solution of the system of linear algebraic equations (3.49) because in general it may not exist if the D determinant of matrix (3.50) is equal to 0: the division by D is performed in Cramer's rule (p. 202).

The `gaus` and `gauss` subroutines allow us to invert the **A** matrix. For that, we must use formula (3.52), where $\mathbf{B} = \mathbf{A}^{-1}$.

3.11. Reduction of the computing error

Let us return to solving boundary value problem (3.6) — (3.8).

The computing error can be reduced if we use the Gaussian elimination method with choice of leading coefficient in the matrix for solving finite difference scheme (3.9), (3.11), (3.12). To verify this, let us put the following program into Module1 of the BookNM workbook instead of the program located there.

Listing 3.8

```
Sub main()
    Dim X() As Double
    Dim G() As Double
    Dim E() As Double
    Dim F() As Double
    Dim U() As Double
    Dim c As Double, b As Double, l As Integer
    Dim h As Double, i As Integer
    Dim n As Integer, j As Integer, w As Double
    Dim AA() As Double
    Dim FF() As Double
    Dim XX() As Double
    c = Selection.Cells(1, 2)
    b = Selection.Cells(2, 2)
    l = Selection.Cells(3, 2)
    h = b / l
    n = l + 1
    ReDim X(5 To 5 + l)
    ReDim G(5 To 5 + l)
    ReDim E(5 To 5 + l)
    ReDim F(5 To 5 + l)
    ReDim U(5 To 5 + l)
    ReDim AA(1 To n, 1 To n)
    ReDim FF(1 To n)
    ReDim XX(1 To n)
    For i = 5 To 5 + l
```

```
        X(i) = (i - 5) * h
1:      G(i) = c
2:      E(i) = 0
3:      F(i) = c
    Next i
    For i = 1 To n
        For j = 1 To n
            AA(i, j) = 0
        Next j
    Next i
    AA(1, 1) = 1: AA(1, 2) = 0: FF(1) = 0
    For i = 2 To n - 1
        w = 0.5 * G(i + 4) * h
        AA(i, i - 1) = 1 - w
        AA(i, i) = E(i + 4) * h ^ 2 - 2
        AA(i, i + 1) = 1 + w
        FF(i) = F(i + 4) * h ^ 2
    Next i
    AA(n, n - 1) = 0: AA(n, n) = 1: FF(n) = b
4:  Call gauss(n, AA, FF, XX)
    For i = 5 To 5 + l
        Selection.Cells(i, 1) = X(i)
        Selection.Cells(i, 2) = XX(i - 4)
    Next i
End Sub
```

This program solves the finite difference scheme corresponding to boundary value problem (3.44), (3.45). It is used as program Listing 3.4: the source data are the values in the table (Fig. 3.2a), at that, we must select this Excel table before the program execution (Fig. 3.2b).

Operator 4 in the above program is the gauss subroutine call, i.e., finite difference scheme (3.9), (3.11), (3.12) is solved by the modernized Gaussian elimination method, with choice of leading coefficient in the matrix. In this case, the computing error is not observed even for $c = 10^{308}$ (according to Appendix 1, this is almost the maximum value for the Double data type), i.e., the calculation result looks like Fig. 3.4a.

When using the classical Gaussian elimination method (that is, when replacing gauss by gaus in operator 4), the computing error is identical to that obtained when using the decomposition method. It is not surprising, because the decomposition method, actually, is an efficient version of the classical Gaussian elimination method for the system of linear algebraic equations with the tridiagonal matrix.

In practice, the Gaussian elimination method is not used for the solution of finite difference scheme (3.9), (3.11), (3.12) because of its inefficiency (with respect to the execution time) for large values of $n = r - k + 1$.

When $c \to \infty$ in problem (3.44), (3.45), the computing error of the solution can be reduced in another way, without usage of the modernized Gaussian elimination method.

This alternative method for solving problem (3.44), (3.45) at large values of c includes the following two stages:

1) changing formulation of the boundary value problem by excluding the second derivative, $u''(x)$, and one of the boundary conditions, $u(0) = 0$ or $u(b) = b$;

2) solving the resulting boundary value problem for the first-order linear differential equation,

$$\frac{du}{dx} = 1.$$

Thus, we should exercise caution in formulation of the problem and in choice of the solution method.

3.12. Solving the nonlinear differential equation by the quasilinearization method

On segment $[a, b]$, we will consider the following differential equation:

$$\frac{d^2u}{dx^2} = F\left(x,\, u,\, \frac{du}{dx}\right), \tag{3.60}$$

where $F(x, y, z)$ is a continuous nonlinear function of variables *x*, *y* and *z*, whose first partial derivatives $\frac{\partial F}{\partial y}(x, y, z)$ and $\frac{\partial F}{\partial z}(x, y, z)$ exist and continuous, and the second partial derivatives with respect to *y* and *z* exist. Because of the function nonlinearity, this equation is called a nonlinear differential equation.

As in the case of linear differential equation (3.6), the solution of nonlinear differential equation (3.60) must satisfy the left and right boundary conditions, (3.7) and (3.8).

In Section 6.10, we will solve the formulated boundary value problem by the shooting method. Solving this problem by the quasilinearization method considered below is the following iterative process: the boundary value problem for a linear differential equation is being solved when calculating the $(j+1)$th solution approximation over the known *j*-th approximation.

An initial approximation of the solution (corresponding to the zero value of j) must be given, and must satisfy boundary conditions (3.7) and (3.8).

Thus, the solution of the nonlinear problem is reduced to solving a series of linear problems. Let us obtain the linear differential equation of the quasilinearization method.

Using Taylor's formula [3] for functions of two variables, we have

$$F(x, y+\Delta y, z+\Delta z) =$$

$$= F(x, y, z) + \frac{1}{1!}\left(\Delta y\frac{\partial}{\partial y} + \Delta z\frac{\partial}{\partial z}\right)F(x, y, z) + \frac{1}{2!}\left(\Delta y\frac{\partial}{\partial y} + \Delta z\frac{\partial}{\partial z}\right)^2 F =$$

$$= F(x, y, z) + \Delta y\frac{\partial F}{\partial y}(x, y, z) + \Delta z\frac{\partial F}{\partial z}(x, y, z) + O(\Delta^2)\,, \tag{3.61}$$

where Δy and Δz are increments of the second and third arguments of function $F(x,y,z)$, $\Delta \to 0$ is the maximum of quantities $|\Delta y|$ and $|\Delta z|$.

The solution of equation (3.60) can be written in the following form:

$$u(x) = u_j(x) + v(x),$$

where $u_j(x)$ is the known *j*-th approximation of solution $u(x)$, $v(x)$ is a small quantity. By substituting this expression into equation (3.60), we obtain

$$\frac{d^2 u_j}{dx^2} + \frac{d^2 v}{dx^2} = F\left(x, u_j + v, \frac{du_j}{dx} + \frac{dv}{dx} \right).$$

Using expression (3.61) without the summand of second order of smallness, we lead the last equation to the following form:

$$\frac{d^2 u_j}{dx^2} + \frac{d^2 v}{dx^2} = F\left(x, u_j, \frac{du_j}{dx} \right) + \frac{\partial F}{\partial y}\left(x, u_j, \frac{du_j}{dx} \right) v + \frac{\partial F}{\partial z}\left(x, u_j, \frac{du_j}{dx} \right) \frac{dv}{dx}$$

or

$$\frac{d^2 v}{dx^2} + g(x)\frac{dv}{dx} + e(x)v = f(x), \tag{3.62}$$

where

$$g(x) = -\frac{\partial F}{\partial z}\left(x, u_j, \frac{du_j}{dx} \right), \tag{3.63}$$

$$e(x) = -\frac{\partial F}{\partial y}\left(x, u_j, \frac{du_j}{dx} \right), \tag{3.64}$$

$$f(x) = F\left(x, u_j, \frac{du_j}{dx} \right) - \frac{d^2 u_j}{dx^2}. \tag{3.65}$$

Let $u_0(x)$ be the initial approximation of the solution of nonlinear differential equation (3.60), satisfying boundary conditions (3.7) and (3.8):

$$u_0(a) = A, \quad u_0(b) = B. \tag{3.66}$$

In the quasilinearization method, the $(j+1)$th approximation of the $u(x)$ solution (*j* = 0, 1, 2, ...) is calculated over the known *j*-th approximation as follows:

1) the values of functions $g(x)$, $e(x)$ and $f(x)$ at the internal nodes of the main grid on segment $a \le x \le b$ (Fig. 3.1) are calculated according to formulas (3.63) — (3.65);

2) second-order linear differential equation (3.62) with zero boundary conditions $v(a) = v(b) = 0$ is solved by the decomposition method; function $v(x)$ is the result;

3) the ($j+1$)th approximation of the $u(x)$ solution is calculated according to formula

$$u_{j+1}(x) = u_j(x) + v(x).$$

The iterative process can be terminated under various conditions; we will use the following:

$$\max_{a \le x \le b} \left| f(x) \right| < \varphi, \tag{3.67}$$

where φ is a given positive constant, $f(x)$ is function (3.65) tending to zero at all points of segment $[a, b]$ when $j \to \infty$.

3.13. Solving the Shockley-Poisson equation

The quasilinearization method considered above will be used for simulation of a silicon photosensitive target. Such targets numbering between one and three are located behind the objective of a camera or movie camera. The incident light is converted to electrical signals with help of these targets.

The photosensitive target consists of cells of identical structure, and there are millions of them, as many as the number of pixels defining the maximum resolution of the camera, or several times more.

According to article [6], as a mathematical model of the cell, we can use the following nonlinear differential equation:

$$-\varepsilon \frac{d^2u}{dx^2} = q\left\{ N_0 \exp\left[-\frac{qu(x)}{k_b T} \right] - N_A(x) \right\}, \tag{3.68}$$

where $u(x)$ is the electric potential in volts, ε = 103.545·10^{-12} C / (V·m) is the dielectric permittivity of silicon, q = 1.6·10^{-19} C is the absolute value of the electron charge, k_b = 1.38·10^{-23} J / C is the Boltzmann constant, T = 300 K is the absolute room temperature, $N_A(x)$ is the difference between the acceptor and donor concentrations, $N_A(a)$ = $N_A(b)$ = N_0 is a positive value, $a \le x \le b$.

To start, let us set the following coordinates of the boundaries of segment $[a, b]$: a = –0.5·10^{-6} m, b = 0.5·10^{-6} m. The boundary conditions are as follows:

$$u(a) = u(b) = 0\,.$$

We can encounter this (or similar) equation in the physics of semiconductor devices. It is the Poisson equation whose right-hand side (the electric charge density) is a nonlinear function of electric potential $u(x)$. Because W. Shockley obtained the form of the right-hand side in [7], equation (3.68) is called the Shockley-Poisson equation.

As we know, two types of carriers of electric current exist in a semiconductor: negatively charged electrons and positively charged holes. The electron has charge –q, the hole has charge +q. Impurities of a semiconductor, which supply electrons, are called donors; impurities, which supply holes, are called acceptors.

When the donor molecule loses an electron, it becomes positively charged; the acceptor molecule losing a hole becomes negatively charged.

We can write equation (3.68) in form (3.60), where

$$F(x, y, z) = \frac{q}{\varepsilon}\left\{ N_A(x) - N_0 \exp\left[-\frac{q\,y}{k_b T} \right] \right\}.$$

Because of absence of the z variable in the right-hand side of this expression, $g(x) = 0$ in differential equation (3.62) according to (3.63). Expressions (3.64) and (3.65) take the following form:

$$e(x) = -\frac{q^2}{\varepsilon k_b T} N_0 \exp\left[-\frac{qu_j(x)}{k_b T} \right],$$

$$f(x) = \frac{q}{\varepsilon}\left\{ N_A(x) - N_0 \exp\left[-\frac{qu_j(x)}{k_b T} \right] \right\} - \frac{d^2 u_j}{dx^2}.$$

According to the previous section, the solution of the boundary value problem for equation (3.68) is reduced to the repeated solution of linear differential equation (3.62) with the above dependences, $g(x)$, $e(x)$ and $f(x)$, and with boundary conditions $v(a) = v(b) = 0$.

It was shown at the end of Section 3.3 that the process of solving such linear problem by the decomposition method is unconditionally stable.

Let us consider table Listing 3.9 with the following source data for program Listing 3.10:

- coordinates a and b of the boundaries;
- the value of φ in condition (3.67) for finishing the iterative process;
- the values of dependences $N_A(x)$ and $u_0(x)$ at the nodes of the uniform grid on segment $a \le x \le b$ (the number of grid nodes is equal to the number of values of N_A or u_0 in the table, that is, 11).

According to the table, the $N_A(x)$ dependence is symmetric to the origin of coordinates, $x = 0$. At the ends of segment $[a, b]$, the semiconductor contains acceptors whose concentration is equal to $7{\cdot}10^{20}\,\mathrm{m}^{-3}$ = $N_A(a)$ = $N_A(b)$ = N_0. At the midpoint ($x = 0$), the semiconductor contains donors whose concentration is equal to $3{\cdot}10^{22}\,\mathrm{m}^{-3}$ = $-N_A(0)$.

The initial approximation of the solution is calculated by means of Excel according to formula

$$u_0(x) = \cos\left(\pi \frac{x}{0.5 \cdot 10^{-6}}\right) + 1.$$

This $u_0(x)$ dependence satisfies boundary conditions (3.66) at $A = B = 0$.

Listing 3.9

a	-5.00E-07
b	5.00E-07
phi	1.00E-01
NA	u0
7.00E+20	0.00E+00
7.00E+20	1.91E-01
7.00E+20	6.91E-01
7.00E+20	1.31E+00
-3.00E+22	1.81E+00
-3.00E+22	2.00E+00
-3.00E+22	1.81E+00
7.00E+20	1.31E+00
7.00E+20	6.91E-01
7.00E+20	1.91E-01
7.00E+20	0.00E+00

The program below is intended for solving the boundary value problem for the Shockley-Poisson equation by the quasilinearization method.

Listing 3.10

```
Sub main()
    Dim NA() As Double
    Dim U() As Double
    Dim X() As Double
    Dim U2() As Double
    Dim V() As Double
    Dim G() As Double
    Dim E() As Double
    Dim F() As Double
    Dim m As Integer
    Dim a As Double, b As Double
    Dim phi As Double
    Dim h As Double
    Dim i As Integer, j As Integer
    Dim w1 As Double, w2 As Double
    Dim w3 As Double, max As Double
    Dim sb As String, se As String, sn As String
    Const q = 1.6E-19
    Const epsilon = 103.545E-12
    Const kb = 1.38E-23
    Const T = 300
    m = Selection.Rows.Count         'quantity of rows
    a = Selection.Cells(1, 2)
    b = Selection.Cells(2, 2)
    phi = Selection.Cells(3, 2)
    h = (b - a) / (m - 5)
    ReDim NA(5 To m)
    ReDim U(5 To m)
    ReDim X(5 To m)
    ReDim U2(5 To m)
    ReDim V(5 To m)
    ReDim G(5 To m)
    ReDim E(5 To m)
    ReDim F(5 To m)
    Selection.Cells(4, 3) = "x"
    w1 = q / epsilon: w2 = q / (kb * T)
    For i = 5 To m
        NA(i) = Selection.Cells(i, 1)
        U(i) = Selection.Cells(i, 2)
        X(i) = (i - 5) * h + a
        Selection.Cells(i, 3) = X(i)
        G(i) = 0
    Next i
    For j = 1 To 1000
        max = 0
        For i = 6 To m - 1
            w3 = NA(5) * Exp(-w2 * U(i))
            E(i) = -w1 * w2 * w3
            U2(i) = (U(i + 1) - 2 * U(i) + _
            U(i - 1)) / h ^ 2       'second derivative
            F(i) = w1 * (NA(i) - w3) - U2(i)
            If Abs(F(i)) > max Then max = Abs(F(i))
        Next i
        Call fb(5, m, h, 0, 0, G, E, F, V)
        For i = 5 To m
            U(i) = U(i) + V(i)
        Next i
        If max < phi Then Exit For
    Next j
    Selection.Cells(4, 4) = "u"
    For i = 5 To m
        Selection.Cells(i, 4) = U(i)
    Next i
1:  sb = Selection.Cells(5, 3).Address
2:  se = Selection.Cells(m, 4).Address
3:  sn = ActiveSheet.Name
4:  Range(sb & ":" & se).Select
5:  Selection.NumberFormat = "0.0E+00"
6:  Charts.Add
7:  ActiveChart.ChartType = xlXYScatterSmoothNoMarkers
8:  ActiveChart.SetSourceData Source:= _
        Sheets(sn).Range(sb & ":" & se), PlotBy:= _
        xlColumns
9:  ActiveChart.Location Where:= xlLocationAsObject, _
        Name:=sn
10: ActiveChart.Axes(xlValue).MajorGridlines.Select
11: Selection.Delete
12: ActiveChart.Legend.Select
13: Selection.Delete
End Sub
```

We enter this program into Module1 of the BookNM workbook.

Data of Listing 3.9 are contained in text file Listing_3_09.txt, which is on the enclosed CD. To copy this data into range B2:C16 on the Sheet2 worksheet of the BookNM workbook, we fulfill the following operations, which are close to operations described on p. 26:

1) open the Listing_3_09.txt file with Notepad, for example, by double click on the pictogram of this file in Windows Explorer;

2) in the Notepad window opened, highlight the table text and copy it into Windows Clipboard, for example, by pressing *Ctrl* + *C*;

3) on the Sheet2 worksheet of the BookNM workbook, select the B2 cell by clicking on it;

4) paste the Windows Clipboard contents into the B2:C16 range, for example, by pressing *Ctrl* + *V*;

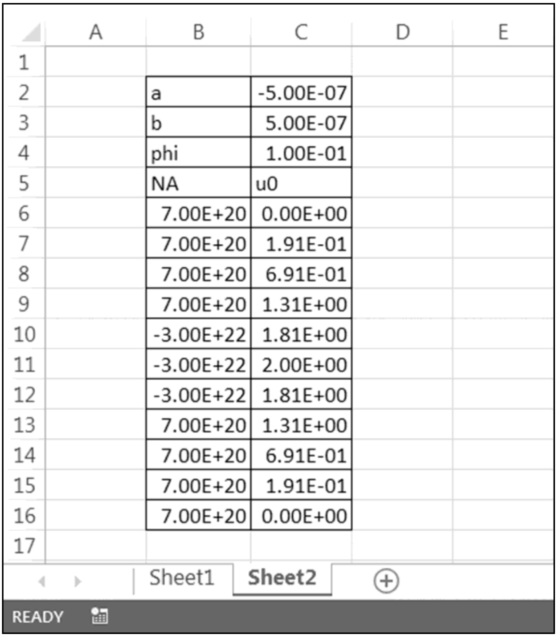
5) close the Notepad window with the Listing_3_09.txt file.

Using the standard features of Excel, the resulting table can be decorated as in Fig. 3.7.

Before the program execution, we must select the Excel table depicted in Fig. 3.7. The execution results are as follows:

- the coordinate and solution values located in the *x* and *u* columns, respectively (Fig. 3.8);
- the $u(x)$ graph on the Excel worksheet.

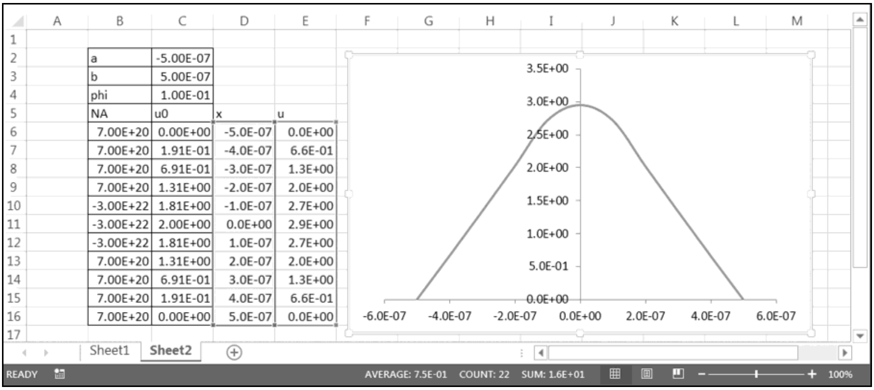
Operators `1` — `13`, intended for constructing the $u(x)$ graph, were programmed by means of Excel Macro Recorder (Sections 2.4 and 2.5). Let us consider the appointment of these operators.



	A	B	C	D	E
1					
2		a	-5.00E-07		
3		b	5.00E-07		
4		phi	1.00E-01		
5		NA	u0		
6		7.00E+20	0.00E+00		
7		7.00E+20	1.91E-01		
8		7.00E+20	6.91E-01		
9		7.00E+20	1.31E+00		
10		-3.00E+22	1.81E+00		
11		-3.00E+22	2.00E+00		
12		-3.00E+22	1.81E+00		
13		7.00E+20	1.31E+00		
14		7.00E+20	6.91E-01		
15		7.00E+20	1.91E-01		
16		7.00E+20	0.00E+00		
17					

Fig. 3.7. The Excel table with the source data

Operator `1` assigns the address of the first cell of the *x* column (that is, string `"D6"`) to the `sb` variable; operator `2` assigns the address of the last cell of the *u* column (that is, string `"E16"`) to the `se` variable; operator `3` assigns the name of the active Excel worksheet to the `sn` variable. Operator `4` selects the *x* and *u* columns (that is, range D6:E16); operator `5` assigns the necessary numerical format to the selected cells. Operators `6` — `9` construct the graph; they correspond to the 2nd, 3rd and 4th operations on p. 214. Operators `10` and `11` delete the gridlines from the graph area; operators `12` and `13` delete the legend.



a	-5.00E-07		
b	5.00E-07		
phi	1.00E-01		
NA	u0	x	u
7.00E+20	0.00E+00	-5.0E-07	0.0E+00
7.00E+20	1.91E-01	-4.0E-07	6.6E-01
7.00E+20	6.91E-01	-3.0E-07	1.3E+00
7.00E+20	1.31E+00	-2.0E-07	2.0E+00
-3.00E+22	1.81E+00	-1.0E-07	2.7E+00
-3.00E+22	2.00E+00	0.0E+00	2.9E+00
-3.00E+22	1.81E+00	1.0E-07	2.7E+00
7.00E+20	1.31E+00	2.0E-07	2.0E+00
7.00E+20	6.91E-01	3.0E-07	1.3E+00
7.00E+20	1.91E-01	4.0E-07	6.6E-01
7.00E+20	0.00E+00	5.0E-07	0.0E+00

Fig. 3.8. The program execution results

The calculated spatial distribution of the electric potential, $u(x)$, is not realistic. The fact is that the derivative,

$$u'(x) = \frac{du}{dx},$$

must equal zero at the ends of segment $[a, b]$ according to the physics of semiconductor devices, i.e., $u'(a) = u'(b) = 0$ must be, but we do not see it in Fig. 3.8.

To obtain realistic distribution $u(x)$, we add areas with length of $9.5{\cdot}10^{-6}$ m and $N_A = N_0 = 7{\cdot}10^{20}$ m^{-3} to segment $[a, b]$ on the left- and right-hand sides. We leave the grid step, h, unchanged, at that, the number of steps increases 20-fold.

The left and right boundaries of new segment $[a, b]$ have the following coordinates: a = -10^{-5} m, b = 10^{-5} m. The initial approximation of the solution of equation (3.68), satisfying boundary conditions (3.66) at $A = B = 0$, is calculated according to formula

$$u_0(x) = \cos\left(\pi \frac{x}{10^{-5}}\right) + 1 \qquad (3.69)$$

by means of Excel.

The $u(x)$ graph, constructed by program Listing 3.10, is given in Fig. 3.9. We see that $u'(a) = u'(b) = 0$, in other words, function $u(x)$ is flat at the ends of segment $[a, b]$.

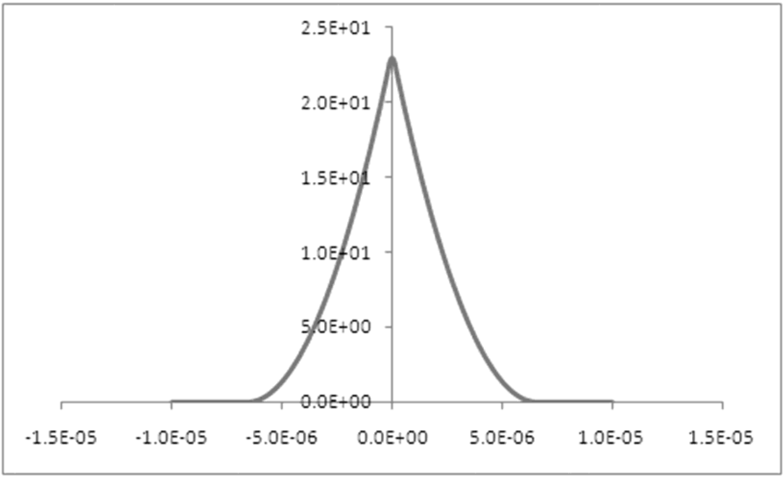


Fig. 3.9. The realistic spatial distribution of the electric potential: the horizontal coordinate, *x*, is in meters; the vertical coordinate, *u*, is in volts

3.14. Finite difference analogs of derivatives for a nonuniform grid

The execution time for solving the boundary value problem can be reduced by replacing the uniform grid on $[a, b]$ with a nonuniform grid whose step, $x_i - x_{i-1} = h_i$, depends on i. At the transition to a nonuniform grid, expressions (3.4) and (3.5) for the first and second derivatives of the $u(x)$ function at node x_i become more complicated.

To obtain new expressions for the derivatives, we introduce axis z parallel to the x axis (Fig. 3.10). If the origin of coordinates ($z = 0$) is at the x_i node of the grid, the x_{i-1} node has coordinate $z = -h_i$, and the x_{i+1} node has coordinate $z = h_{i+1}$.

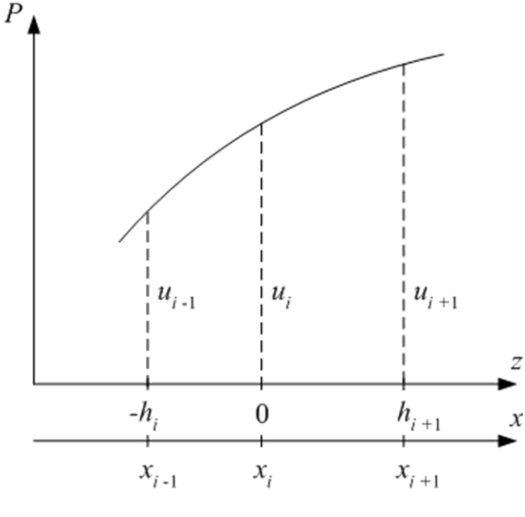


Fig. 3.10. The parabola passing through points (x_{i-1}, u_{i-1}), (x_i, u_i) and (x_{i+1}, u_{i+1})

Let us consider the following second-degree polynomial:

$$P(z) = \alpha z^2 + \beta z + u_i , \tag{3.70}$$

where u_i is the $u(x)$ function value at the x_i node; $P(0) = u_i$.

Coefficients α and β are determined by equations $P(-h_i) = u_{i-1}$ and $P(h_{i+1}) = u_{i+1}$, which can be written as follows:

$$h_i^2\alpha - h_i\beta = u_{i-1} - u_i ,$$

$$h_{i+1}^2\alpha + h_{i+1}\beta = u_{i+1} - u_i .$$

Solving this system of linear algebraic equations by Cramer's rule [3], we obtain

$$\alpha = \frac{D_1}{D} , \quad \beta = \frac{D_2}{D} , \tag{3.71}$$

where

$$\begin{gathered} D = h_i h_{i+1}(h_i + h_{i+1}) , \\ D_1 = (u_{i+1} - u_i)h_i + (u_{i-1} - u_i)h_{i+1} , \\ D_2 = (u_{i+1} - u_i)\, h_i^2 - (u_{i-1} - u_i)h_{i+1}^2 . \end{gathered} \tag{3.72}$$

Differentiating polynomial (3.70) twice, we obtain

$$\frac{dP}{dz}(z) = 2\alpha\, z + \beta \;\; , \tag{3.73}$$

$$\frac{d^2P}{dz^2}(z) = 2\alpha .$$

From here, expressions for the derivatives at $z = 0$ follow:

$$\frac{dP}{dz}(0) = \beta = \frac{D_2}{D} = \frac{(u_{i+1} - u_i)h_i^2 - (u_{i-1} - u_i)h_{i+1}^2}{h_i h_{i+1}(h_i + h_{i+1})} , \tag{3.74}$$

$$\frac{d^2P}{dz^2}(0) = 2\alpha = 2\frac{D_1}{D} = 2\frac{(u_{i+1} - u_i)h_i + (u_{i-1} - u_i)h_{i+1}}{h_i h_{i+1}(h_i + h_{i+1})} . \tag{3.75}$$

We have the finite difference analogs of the first and second derivatives of the $u(x)$ function at the x_i node.

Using expression (3.1), we can show the following:

$$\frac{du}{dx}(x_i) = \frac{(u_{i+1} - u_i)h_i^2 - (u_{i-1} - u_i)h_{i+1}^2}{h_i h_{i+1}(h_i + h_{i+1})} - \frac{h_i h_{i+1}}{6}\frac{d^3u}{dx^3}(x_i) + \ldots,$$

$$\frac{d^2u}{dx^2}(x_i) = 2\frac{(u_{i+1} - u_i)h_i + (u_{i-1} - u_i)h_{i+1}}{h_i h_{i+1}(h_i + h_{i+1})} + \frac{h_i - h_{i+1}}{3}\frac{d^3u}{dx^3}(x_i) + \ldots$$

or

$$\frac{du}{dx}(x_i) = \frac{(u_{i+1} - u_i)h_i^2 - (u_{i-1} - u_i)h_{i+1}^2}{h_i h_{i+1}(h_i + h_{i+1})} + O(h_{max}^2)\,, \tag{3.76}$$

$$\frac{d^2u}{dx^2}(x_i) = 2\frac{(u_{i+1} - u_i)h_i + (u_{i-1} - u_i)h_{i+1}}{h_i h_{i+1}(h_i + h_{i+1})} + O(h_{max})\,, \tag{3.77}$$

where $h_{max} = \max\limits_{k+1 \le i \le r} \{h_i\}$ is the maximum step of the grid, $h_{max} \to 0$.

The resulting expressions, (3.76) and (3.77), are similar to expressions (3.4) and (3.5). Naturally, (3.76) and (3.77) become (3.4) and (3.5), respectively, for the constant step ($h_i = h_{i+1} = h$).

Expressions (3.5) and (3.77) include summands $O(h^2)$ and $O(h_{max})$, respectively. That is, at the transition from a uniform grid to a nonuniform grid, the error of the finite difference analog of the second derivative changes from the 2nd order of smallness to the 1st order. It means deterioration of the finite difference approximation accuracy of the second derivative at the transition from a uniform grid to a nonuniform grid.

3.15. The decomposition method for a nonuniform grid

Substituting expressions (3.74) and (3.75) into linear differential equation (3.6) instead of $u'(x)$ and $u''(x)$, respectively, we obtain linear algebraic equation (3.9),

$$\alpha_i u_{i-1} + \beta_i u_i + \gamma_i u_{i+1} = \delta_i \,,$$

where

$$\alpha_i = \frac{h_{i+1}(2 - g_i h_{i+1})}{h_i + h_{i+1}}\,,$$

$$\beta_i = e_i h_i h_{i+1} - g_i (h_i - h_{i+1}) - 2\,,$$

$$\gamma_i = \frac{h_i (2 + g_i h_i)}{h_i + h_{i+1}}\,, \tag{3.78}$$

$$\delta_i = f_i h_i h_{i+1}\,.$$

Equation (3.9) for $i = k+1,\ k+2,\ ...,\ r-2,\ r-1$ and boundary conditions (3.11) and (3.12) still form the system of linear algebraic equations with the tridiagonal matrix, which can be solved by the decomposition method as follows:

1) at first, the forward sweep is performed, i.e., unknown P_{k+1}, Q_{k+1}, P_{k+2}, Q_{k+2}, ..., P_r, Q_r are calculated according to formulas (3.16), (3.17) and (3.14), (3.15);

2) then the backward sweep is performed, i.e., unknown u_r, u_{r-1}, ..., u_k are calculated according to formulas (3.18) and (3.13).

Let us put the following declaration of the subroutine, realizing the decomposition method for differential equation (3.6) with boundary conditions (3.11) and (3.12) for a nonuniform grid, into Module5 of the BookNM workbook.

Listing 3.11

```
Sub foba(ByVal k, ByVal r, ByRef X() As Double, _
    ByRef G() As Double, ByRef E() As Double, _
    ByRef F() As Double, _
    ByVal GAMMAK, ByVal DELTAK, _
    ByVal ALPHAR, ByVal DELTAR, _
    ByRef U() As Double)
    Const BETAK = -2, BETAR = -2
    Dim alpha As Double, beta As Double
    Dim gamma As Double, delta As Double
    Dim i As Integer, w As Double
    Dim H() As Double: ReDim H(k + 1 To r)
    Dim P() As Double: ReDim P(k + 1 To r)
    Dim Q() As Double: ReDim Q(k + 1 To r)
    For i = k + 1 To r
        H(i) = X(i) - X(i - 1)
    Next i
'Forward sweep:
    P(k + 1) = -GAMMAK / BETAK
    Q(k + 1) = DELTAK / BETAK
    For i = k + 1 To r - 1
        w = H(i) + H(i + 1)
        alpha = H(i + 1) * (2 - G(i) * H(i + 1)) / w
        beta = E(i) * H(i) * H(i + 1) - _
        G(i) * (H(i) - H(i + 1)) - 2
        gamma = H(i) * (2 + G(i) * H(i)) / w
        delta = F(i) * H(i) * H(i + 1)
        w = alpha * P(i) + beta
        P(i + 1) = -gamma / w
        Q(i + 1) = (delta - alpha * Q(i)) / w
    Next i
'Backward sweep:
    U(r) = (DELTAR - ALPHAR * Q(r)) / _
    (ALPHAR * P(r) + BETAR)
    For i = r To k + 1 Step -1
        U(i - 1) = P(i) * U(i) + Q(i)
    Next i
End Sub
```

The `foba` subroutine parameters have the following sense:

- `k`, `r` are numbers of the left and right boundary nodes of the grid;
- `X` is an array of grid nodes;
- `G`, `E` are arrays of values of the coefficients of equation (3.6) at the grid nodes;

- `F` is an array of values of the right-hand side of equation (3.6);
- `GAMMAK`, `DELTAK` are values of parameters γ_k and δ_k in left boundary condition (3.11), where $\beta_k = -2$;
- `ALPHAR`, `DELTAR` are values of parameters α_r and δ_r in right boundary condition (3.12), where $\beta_r = -2$;
- `U` is an array intended for the solution values.

Here and below, we consider only the main grid on segment $[a, b]$ with node coordinates x_k, x_{k+1}, x_{k+2}, ..., x_{r-2}, x_{r-1}, x_r (Fig. 3.1).

3.16. Solving the Shockley-Poisson equation on a nonuniform grid

Let us consider the following source data table.

Listing 3.12

maxtime	1	
maxiter	1000	
phi	1.00E-01	
NA	u0	x
7.00E+20	0.00E+00	-1.0E-05
7.00E+20	4.89E-02	-9.0E-06
7.00E+20	1.91E-01	-8.0E-06
7.00E+20	4.12E-01	-7.0E-06
7.00E+20	6.91E-01	-6.0E-06
7.00E+20	1.00E+00	-5.0E-06
7.00E+20	1.31E+00	-4.0E-06
7.00E+20	1.59E+00	-3.0E-06
7.00E+20	1.81E+00	-2.0E-06
7.00E+20	1.95E+00	-1.0E-06
7.00E+20	1.98E+00	-6.0E-07
7.00E+20	1.99E+00	-4.0E-07
7.00E+20	2.00E+00	-3.0E-07
7.00E+20	2.00E+00	-2.0E-07
-3.00E+22	2.00E+00	-1.0E-07
-3.00E+22	2.00E+00	0.0E+00
-3.00E+22	2.00E+00	1.0E-07
7.00E+20	2.00E+00	2.0E-07
7.00E+20	2.00E+00	3.0E-07
7.00E+20	1.99E+00	4.0E-07
7.00E+20	1.98E+00	6.0E-07
7.00E+20	1.95E+00	1.0E-06
7.00E+20	1.81E+00	2.0E-06
7.00E+20	1.59E+00	3.0E-06
7.00E+20	1.31E+00	4.0E-06
7.00E+20	1.00E+00	5.0E-06
7.00E+20	6.91E-01	6.0E-06
7.00E+20	4.12E-01	7.0E-06
7.00E+20	1.91E-01	8.0E-06
7.00E+20	4.89E-02	9.0E-06
7.00E+20	0.00E+00	1.0E-05

Table Listing 3.12 contains:

- *maxtime*, the limiting execution time in seconds;
- *maxiter*, the limiting number of the quasilinearization method iterations; it must be less than the maximum value of the `Integer` data type, that is, 32767 (Appendix 1);
- the value of φ in condition (3.67) for finishing the iterative process of the quasilinearization method;
- the values of spatial coordinate *x*;
- the values of dependence $N_A(x)$;
- the values of dependence $u_0(x)$, which are calculated according to formula (3.69) by using Excel.

If *maxtime* is greater than 59 or less than 0, then the execution time is not limited.

The program below is intended for solving the boundary value problem for the Shockley-Poisson equation on the nonuniform grid defined by the right column of table Listing 3.12.

Listing 3.13

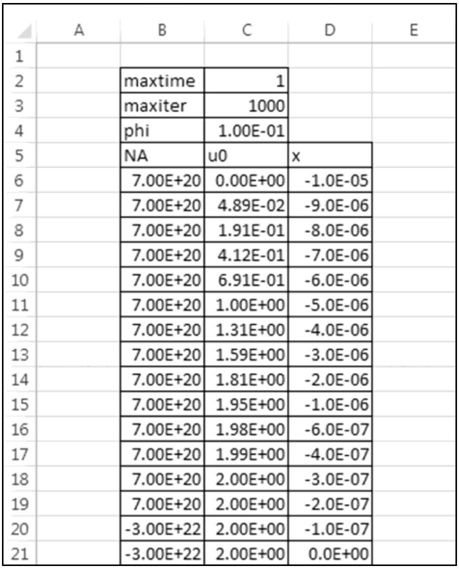
```
Sub main()
    Dim NA() As Double
    Dim U() As Double
    Dim X() As Double
    Dim H() As Double
    Dim U2() As Double
    Dim V() As Double
    Dim G() As Double
    Dim E() As Double
    Dim F() As Double
    Dim m As Integer
    Dim maxtime As Integer
    Dim maxiter As Integer
    Dim phi As Double
    Dim i As Integer
    Dim j As Integer
    Dim w1 As Double
    Dim w2 As Double
    Dim w3 As Double
    Dim max As Double
    Dim sb As String
    Dim se As String
    Dim sn As String
    Dim tm As Date
    Const q = 1.6E-19
    Const epsilon = 103.545E-12
    Const kb = 1.38E-23
    Const T = 300
    m = Selection.Rows.Count         'quantity of rows
    maxtime = Selection.Cells(1, 2)
    maxiter = Selection.Cells(2, 2)
    phi = Selection.Cells(3, 2)
    ReDim NA(5 To m)
    ReDim U(5 To m)
    ReDim X(5 To m)
    ReDim H(6 To m)
    ReDim U2(5 To m)
    ReDim V(5 To m)
    ReDim G(5 To m)
    ReDim E(5 To m)
    ReDim F(5 To m)
    w1 = q / epsilon
    w2 = q / (kb * T)
    For i = 5 To m
        NA(i) = Selection.Cells(i, 1)
        U(i) = Selection.Cells(i, 2)
        X(i) = Selection.Cells(i, 3)
        G(i) = 0
    Next i
    For i = 6 To m
        H(i) = X(i) - X(i - 1)
    Next i
    If maxtime >= 0 And maxtime < 60 Then
        tm = Now + TimeValue("00:00:" & CStr(maxtime))
    End If
    For j = 1 To maxiter
        max = 0
        For i = 6 To m - 1
            w3 = NA(5) * Exp(-w2 * U(i))
            E(i) = -w1 * w2 * w3
            U2(i) = 2 * ((U(i + 1) - U(i)) * H(i) + _
            (U(i - 1) - U(i)) * H(i + 1)) / _
            (H(i) * H(i + 1) * (H(i) + H(i + 1)))
            F(i) = w1 * (NA(i) - w3) - U2(i)
```

```
            If Abs(F(i)) > max Then max = Abs(F(i))
        Next i
0:      Call foba(5, m, X, G, E, F, 0, 0, 0, 0, V)
        For i = 5 To m
            U(i) = U(i) + V(i)
        Next i
        If max < phi Then Exit For
        If maxtime >= 0 And maxtime < 60 And _
        Now > tm Then Exit For
    Next j
    Selection.Cells(4, 4) = "u"
    For i = 5 To m
        Selection.Cells(i, 4) = U(i)
    Next i
1:  sb = Selection.Cells(5, 3).Address
2:  se = Selection.Cells(m, 4).Address
3:  sn = ActiveSheet.Name
4:  Range(sb & ":" & se).Select
5:  Selection.NumberFormat = "0.0E+00"
6:  Charts.Add
7:  ActiveChart.ChartType = xlXYScatterSmoothNoMarkers
8:  ActiveChart.SetSourceData Source:= _
        Sheets(sn).Range(sb & ":" & se), PlotBy:= _
        xlColumns
9:  ActiveChart.Location Where:= xlLocationAsObject, _
        Name:=sn
10: ActiveChart.Axes(xlValue).MajorGridlines.Select
11: Selection.Delete
12: ActiveChart.Legend.Select
13: Selection.Delete
End Sub
```

The source data for this program are the values of table Listing 3.12 (Fig. 3.11). Before the program execution, we have to select this Excel table (range B2:D36, Fig. 3.12).

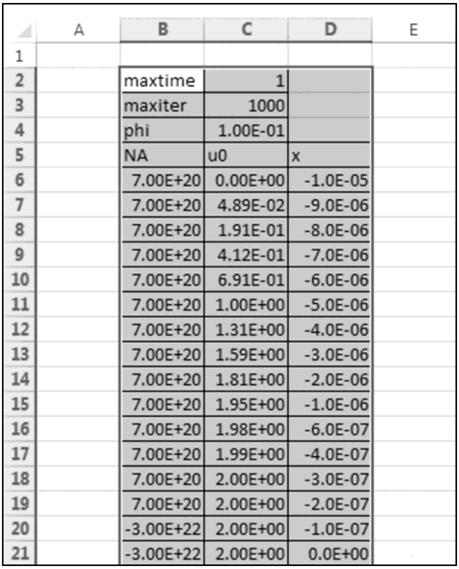
The execution results are the *u* solution values, which are located near the corresponding values of the *x* coordinate, and the $u(x)$ graph on the Excel worksheet (Fig. 3.13).

The $u(x)$ graph is constructed automatically when executing operators `1` — `13`. The same operators are present in program Listing 3.10.



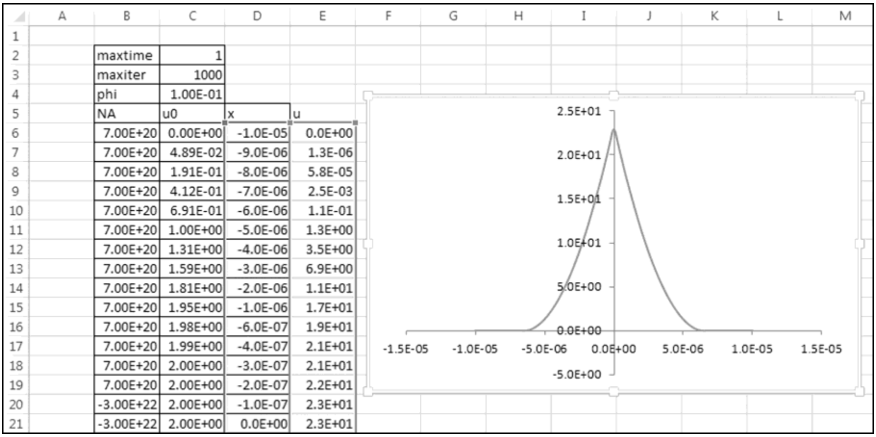
	A	B	C	D	E
1					
2		maxtime	1		
3		maxiter	1000		
4		phi	1.00E-01		
5		NA	u0	x	
6		7.00E+20	0.00E+00	-1.0E-05	
7		7.00E+20	4.89E-02	-9.0E-06	
8		7.00E+20	1.91E-01	-8.0E-06	
9		7.00E+20	4.12E-01	-7.0E-06	
10		7.00E+20	6.91E-01	-6.0E-06	
11		7.00E+20	1.00E+00	-5.0E-06	
12		7.00E+20	1.31E+00	-4.0E-06	
13		7.00E+20	1.59E+00	-3.0E-06	
14		7.00E+20	1.81E+00	-2.0E-06	
15		7.00E+20	1.95E+00	-1.0E-06	
16		7.00E+20	1.98E+00	-6.0E-07	
17		7.00E+20	1.99E+00	-4.0E-07	
18		7.00E+20	2.00E+00	-3.0E-07	
19		7.00E+20	2.00E+00	-2.0E-07	
20		-3.00E+22	2.00E+00	-1.0E-07	
21		-3.00E+22	2.00E+00	0.0E+00	

Fig. 3.11. The Excel table with the source data



	A	B	C	D	E
1					
2		maxtime	1		
3		maxiter	1000		
4		phi	1.00E-01		
5		NA	u0	x	
6		7.00E+20	0.00E+00	-1.0E-05	
7		7.00E+20	4.89E-02	-9.0E-06	
8		7.00E+20	1.91E-01	-8.0E-06	
9		7.00E+20	4.12E-01	-7.0E-06	
10		7.00E+20	6.91E-01	-6.0E-06	
11		7.00E+20	1.00E+00	-5.0E-06	
12		7.00E+20	1.31E+00	-4.0E-06	
13		7.00E+20	1.59E+00	-3.0E-06	
14		7.00E+20	1.81E+00	-2.0E-06	
15		7.00E+20	1.95E+00	-1.0E-06	
16		7.00E+20	1.98E+00	-6.0E-07	
17		7.00E+20	1.99E+00	-4.0E-07	
18		7.00E+20	2.00E+00	-3.0E-07	
19		7.00E+20	2.00E+00	-2.0E-07	
20		-3.00E+22	2.00E+00	-1.0E-07	
21		-3.00E+22	2.00E+00	0.0E+00	

Fig. 3.12. The worksheet before the program execution



	A	B	C	D	E
1					
2		maxtime	1		
3		maxiter	1000		
4		phi	1.00E-01		
5		NA	u0	x	u
6		7.00E+20	0.00E+00	-1.0E-05	0.0E+00
7		7.00E+20	4.89E-02	-9.0E-06	1.3E-06
8		7.00E+20	1.91E-01	-8.0E-06	5.8E-05
9		7.00E+20	4.12E-01	-7.0E-06	2.5E-03
10		7.00E+20	6.91E-01	-6.0E-06	1.1E-01
11		7.00E+20	1.00E+00	-5.0E-06	1.3E+00
12		7.00E+20	1.31E+00	-4.0E-06	3.5E+00
13		7.00E+20	1.59E+00	-3.0E-06	6.9E+00
14		7.00E+20	1.81E+00	-2.0E-06	1.1E+01
15		7.00E+20	1.95E+00	-1.0E-06	1.7E+01
16		7.00E+20	1.98E+00	-6.0E-07	1.9E+01
17		7.00E+20	1.99E+00	-4.0E-07	2.1E+01
18		7.00E+20	2.00E+00	-3.0E-07	2.1E+01
19		7.00E+20	2.00E+00	-2.0E-07	2.2E+01
20		-3.00E+22	2.00E+00	-1.0E-07	2.3E+01
21		-3.00E+22	2.00E+00	0.0E+00	2.3E+01

Fig. 3.13. The results of the program execution: the horizontal coordinate of the graph, *x*, is in meters, the vertical coordinate, *u*, is in volts

3.17. Use of solution symmetry

According to table Listing 3.12, the $N_A(x)$ dependence is symmetric to the origin of coordinates: $N_A(-x) = N_A(x)$. Therefore, under symmetric boundary conditions, in particular $u(-10^{-5}) = u(10^{-5}) = 0$, the solution of the Shockley-Poisson equation is also symmetric: $u(-x) = u(x)$. We can use the symmetry for further reducing the execution time.

By setting $b = 0$, we reduce the length of segment $[a, b]$ to half. The right boundary condition becomes $u'(b) = 0$. We leave the left boundary and condition unchanged: $a = -10^{-5}$, $u(a) = 0$.

Let us correct the `foba` subroutine to use it for solving differential equation (3.6) with left boundary condition (3.11) at given $u'(b)$.

We consider that $i = r-1$ in Fig. 3.10. Thus, the right node ($z = h_r$) coincides with the right boundary of segment $[a, b]$, that is, with point *b*.

Let us return to expression (3.73) for the first derivative of second-degree polynomial (3.70). If $z = h_r$, then

$$\frac{dP}{dz}(h_r) = 2\alpha h_r + \beta \ .$$

Using expressions (3.71) and (3.72) at $i = r-1$, we have

$$\frac{dP}{dz}(h_r) = \frac{h_r^2 u_{r-2} + (-h_{r-1}^2 - 2h_{r-1}h_r - h_r^2)u_{r-1} + (h_{r-1}^2 + 2h_{r-1}h_r)u_r}{h_{r-1}h_r(h_{r-1} + h_r)} .$$

Equating the last expression and $u'(b)$, we obtain equation

$$\frac{h_r^2 u_{r-2} + (-h_{r-1}^2 - 2h_{r-1}h_r - h_r^2)u_{r-1} + (h_{r-1}^2 + 2h_{r-1}h_r)u_r}{h_{r-1}h_r(h_{r-1} + h_r)} = u'(b)$$

or

$$\alpha' u_{r-2} + \beta' u_{r-1} + \gamma' u_r = \delta' , \tag{3.79}$$

where

$$\alpha' = h_r^2 ,$$

$$\beta' = -h_{r-1}^2 - 2h_{r-1}h_r - h_r^2 ,$$

$$\gamma' = h_{r-1}^2 + 2h_{r-1}h_r , \tag{3.80}$$

$$\delta' = h_{r-1}h_r(h_{r-1} + h_r)u'(b) .$$

According to formula (3.13) for the backward sweep, we have

$$u_{r-1} = P_r u_r + Q_r ,$$

$$u_{r-2} = P_{r-1}u_{r-1} + Q_{r-1} = P_{r-1}(P_r u_r + Q_r) + Q_{r-1} .$$

Substituting these expressions into equation (3.79), we have

$$\alpha' [P_{r-1}(P_r u_r + Q_r) + Q_{r-1}] + \beta' [P_r u_r + Q_r] + \gamma' u_r = \delta' .$$

The solution of this equation follows:

$$u_r = \frac{\delta' - \alpha'(P_{r-1}Q_r + Q_{r-1}) - \beta' Q_r}{\alpha' P_{r-1}P_r + \beta' P_r + \gamma'} , \tag{3.81}$$

where α', β', γ' and δ' are defined by formulas (3.80).

Let us put the following declaration of the subroutine, which realizes the decomposition method for differential equation (3.6) with $u'(b)$ given, into Module6 of the BookNM workbook.

Listing 3.14

```
Sub forbac(ByVal k, ByVal r, ByRef X() As Double, _
    ByRef G() As Double, ByRef E() As Double, _
    ByRef F() As Double, _
    ByVal GAMMAK, ByVal DELTAK, _
    ByVal U1B, ByRef U() As Double)
    Const BETAK = -2
    Dim alpha As Double, beta As Double
    Dim gamma As Double, delta As Double
    Dim i As Integer, w As Double
    Dim H() As Double: ReDim H(k + 1 To r)
    Dim P() As Double: ReDim P(k + 1 To r)
    Dim Q() As Double: ReDim Q(k + 1 To r)
    For i = k + 1 To r
```

```
        H(i) = X(i) - X(i - 1)
    Next i
'Forward sweep:
    P(k + 1) = -GAMMAK / BETAK
    Q(k + 1) = DELTAK / BETAK
    For i = k + 1 To r - 1
        w = H(i) + H(i + 1)
        alpha = H(i + 1) * (2 - G(i) * H(i + 1)) / w
        beta = E(i) * H(i) * H(i + 1) - _
        G(i) * (H(i) - H(i + 1)) - 2
        gamma = H(i) * (2 + G(i) * H(i)) / w
        delta = F(i) * H(i) * H(i + 1)
        w = alpha * P(i) + beta
        P(i + 1) = -gamma / w
        Q(i + 1) = (delta - alpha * Q(i)) / w
    Next i
'Backward sweep:
    alpha = H(r) ^ 2
    beta = -H(r - 1) ^ 2 - 2 * H(r - 1) * H(r) - _
    H(r) ^ 2
    gamma = H(r - 1) ^ 2 + 2 * H(r - 1) * H(r)
    delta = H(r - 1) * H(r) * (H(r - 1) + H(r)) * U1B
    U(r) = (delta - alpha * (P(r - 1) * Q(r) + _
    Q(r - 1)) - beta * Q(r)) / _
    (alpha * P(r - 1) * P(r) + beta * P(r) + gamma)
    For i = r To k + 1 Step -1
        U(i - 1) = P(i) * U(i) + Q(i)
    Next i
End Sub
```

Formula (3.81) is used to start the backward sweep in the `forbac` subroutine.

The subroutine parameters have the following sense:

- `k`, `r` are numbers of the left and right boundary nodes of the grid;
- `X` is an array of grid nodes;
- `G`, `E` are arrays of values of the coefficients of equation (3.6) at the grid nodes;
- `F` is an array of values of the right-hand side of equation (3.6);
- `GAMMAK`, `DELTAK` are values of parameters γ_k and δ_k in left boundary condition (3.11), where $\beta_k = -2$;
- `U1B` is a value of $u'(b)$;

- `U` is an array intended for the solution values.

The program, which solves the boundary value problem for the Shockley-Poisson equation by means of subroutine `forbac`, differs from Listing 3.13 of the previous section only in the following operators:

```
0:  Call forbac(5, m, X, G, E, F, 0, 0, 0, V)
5:  Selection.NumberFormat = "0.000E+00"
```

The source data for this program are the values located in table Listing 3.15 (Fig. 3.14). We must select this Excel table (range B2:D21) before the program execution.

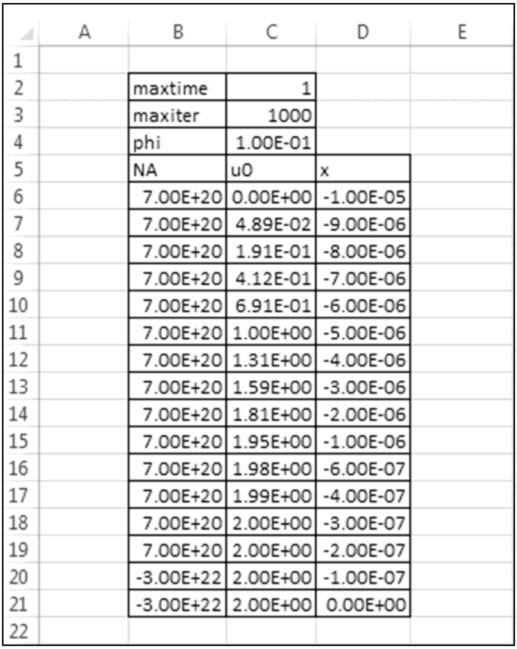
Listing 3.15

maxtime	1	
maxiter	1000	
phi	1.00E-01	
NA	u0	x
7.00E+20	0.00E+00	-1.00E-05
7.00E+20	4.89E-02	-9.00E-06
7.00E+20	1.91E-01	-8.00E-06
7.00E+20	4.12E-01	-7.00E-06
7.00E+20	6.91E-01	-6.00E-06
7.00E+20	1.00E+00	-5.00E-06
7.00E+20	1.31E+00	-4.00E-06
7.00E+20	1.59E+00	-3.00E-06
7.00E+20	1.81E+00	-2.00E-06
7.00E+20	1.95E+00	-1.00E-06
7.00E+20	1.98E+00	-6.00E-07
7.00E+20	1.99E+00	-4.00E-07
7.00E+20	2.00E+00	-3.00E-07
7.00E+20	2.00E+00	-2.00E-07
-3.00E+22	2.00E+00	-1.00E-07
-3.00E+22	2.00E+00	0.00E+00

The results of the program execution are the *u* solution values and the $u(x)$ graph for negative values of *x* (Fig. 3.15).

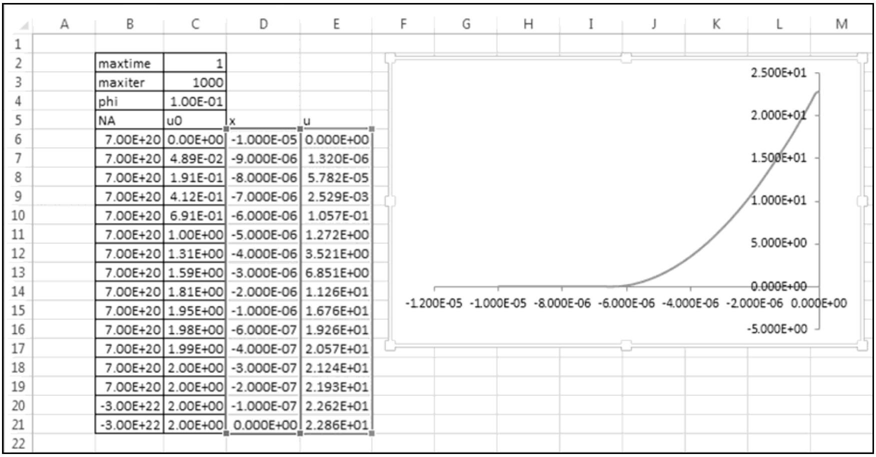
According to the $u(x)$ graph, semiconductor layer -6 μm ≤ *x* ≤ 6 μm, whose plane is perpendicular to the *x* axis, is the potential well for signal electrons: this layer collects electrons knocked out by photons (light particles) from semiconductor molecules. The light falls on the layer plane.

In Section 4.7, the mathematical modeling of the silicon photosensitive target will be continued, and we will use the resulting $u(x)$ dependence depicted in Fig. 3.15.



	A	B	C	D	E
1					
2		maxtime	1		
3		maxiter	1000		
4		phi	1.00E-01		
5		NA	u0	x	
6		7.00E+20	0.00E+00	-1.00E-05	
7		7.00E+20	4.89E-02	-9.00E-06	
8		7.00E+20	1.91E-01	-8.00E-06	
9		7.00E+20	4.12E-01	-7.00E-06	
10		7.00E+20	6.91E-01	-6.00E-06	
11		7.00E+20	1.00E+00	-5.00E-06	
12		7.00E+20	1.31E+00	-4.00E-06	
13		7.00E+20	1.59E+00	-3.00E-06	
14		7.00E+20	1.81E+00	-2.00E-06	
15		7.00E+20	1.95E+00	-1.00E-06	
16		7.00E+20	1.98E+00	-6.00E-07	
17		7.00E+20	1.99E+00	-4.00E-07	
18		7.00E+20	2.00E+00	-3.00E-07	
19		7.00E+20	2.00E+00	-2.00E-07	
20		-3.00E+22	2.00E+00	-1.00E-07	
21		-3.00E+22	2.00E+00	0.00E+00	
22					

Fig. 3.14. The Excel table with the source data



	A	B	C	D	E
1					
2		maxtime	1		
3		maxiter	1000		
4		phi	1.00E-01		
5		NA	u0	x	u
6		7.00E+20	0.00E+00	-1.000E-05	0.000E+00
7		7.00E+20	4.89E-02	-9.000E-06	1.320E-06
8		7.00E+20	1.91E-01	-8.000E-06	5.782E-05
9		7.00E+20	4.12E-01	-7.000E-06	2.529E-03
10		7.00E+20	6.91E-01	-6.000E-06	1.057E-01
11		7.00E+20	1.00E+00	-5.000E-06	1.272E+00
12		7.00E+20	1.31E+00	-4.000E-06	3.521E+00
13		7.00E+20	1.59E+00	-3.000E-06	6.851E+00
14		7.00E+20	1.81E+00	-2.000E-06	1.126E+01
15		7.00E+20	1.95E+00	-1.000E-06	1.676E+01
16		7.00E+20	1.98E+00	-6.000E-07	1.926E+01
17		7.00E+20	1.99E+00	-4.000E-07	2.057E+01
18		7.00E+20	2.00E+00	-3.000E-07	2.124E+01
19		7.00E+20	2.00E+00	-2.000E-07	2.193E+01
20		-3.00E+22	2.00E+00	-1.000E-07	2.262E+01
21		-3.00E+22	2.00E+00	0.000E+00	2.286E+01
22					

Fig. 3.15. The results of the program execution: the horizontal coordinate of the graph, *x*, is in meters, the vertical coordinate, *u*, is in volts

We advise the reader to develop the `forbacs` subroutine (similar to `forbac`), which realizes the decomposition method for differential equation (3.6) on segment $[a, b]$ with $u'(a)$ given. The `forbacs` subroutine must have the following parameters:

- `k`, `r` are numbers of the left and right boundary nodes of the grid on segment $[a, b]$;
- `X` is an array of grid nodes;
- `G`, `E` are arrays of values of the coefficients of equation (3.6) at the grid nodes;
- `F` is an array of values of the right-hand side of equation (3.6);
- `U1A` is a value of $u'(a)$;
- `ALPHAR`, `DELTAR` are values of parameters α_r and δ_r in right boundary condition (3.12), where $\beta_r = -2$;
- `U` is an array intended for the solution values.

We advise to put the `forbacs` subroutine declaration into Module6 of the BookNM workbook below the `forbac` subroutine declaration developed above.

In addition, ***we advise the reader*** to use the `forbacs` subroutine to create a picture similar to Fig. 3.15, but for $x \geq 0$.

3.18. The cyclic decomposition method

Let us consider linear differential equation (3.6) on the whole x axis, $-\infty < x < \infty$, assuming that the coefficients and right-hand side of the equation are periodic functions with period $\Pi = b - a$:

$$g(x+\Pi) = g(x), \quad e(x+\Pi) = e(x), \quad f(x+\Pi) = f(x).$$

In this case, the equation solution is also periodic:

$$u(x+\Pi) = u(x) \text{ for } -\infty < x < \infty .$$

We supplement the grid on segment $a \le x \le b$ (Fig. 3.1) by nodes outside this segment in such manner that $x_{i+r-k} = x_i + \Pi$ for $-\infty < i < \infty$.

Let us replace boundary conditions (3.11) and (3.12) by the following periodicity condition for the solution: $u_{i+r-k} = u_i$, $-\infty < i < \infty$. As a result, we have the system of linear algebraic equations

$$\alpha_k u_{r-1} + \beta_k u_k + \gamma_k u_{k+1} = \delta_k , \tag{3.82}$$

$$\alpha_i u_{i-1} + \beta_i u_i + \gamma_i u_{i+1} = \delta_i , \tag{3.83}$$

$$i = k+1, k+2, ..., r-2,$$

$$\alpha_{r-1} u_{r-2} + \beta_{r-1} u_{r-1} + \gamma_{r-1} u_k = \delta_{r-1} . \tag{3.84}$$

The coefficients and right-hand sides are determined as follows:

- the values of α_i, β_i, γ_i, δ_i are calculated according to formulas (3.78) at $k+1 \le i \le r-1$;
- the values of α_k, β_k, γ_k, δ_k are calculated according to formulas

$$\alpha_k = \frac{h_{k+1}(2 - g_k h_{k+1})}{h_r + h_{k+1}},$$

$$\beta_k = e_k h_r h_{k+1} - g_k (h_r - h_{k+1}) - 2 ,$$

$$\gamma_k = \frac{h_r (2 + g_k h_r)}{h_r + h_{k+1}}, \tag{3.85}$$

$$\delta_k = f_k h_r h_{k+1} .$$

The formulated system of $r-k$ equations (3.82) — (3.84) with the $r-k$ unknowns (u_k, u_{k+1}, u_{k+2}, ..., u_{r-2}, u_{r-1}) is the finite difference scheme for linear differential equation (3.6) with the solution periodicity condition. We can write this scheme as the following matrix equation:

$$\begin{bmatrix} \beta_k & \gamma_k & 0 & 0 & \ldots & 0 & 0 & \alpha_k \\ \alpha_{k+1} & \beta_{k+1} & \gamma_{k+1} & 0 & \ldots & 0 & 0 & 0 \\ 0 & \alpha_{k+2} & \beta_{k+2} & \gamma_{k+2} & \ldots & 0 & 0 & 0 \\ . & . & . & . & . & . & . & . \\ 0 & 0 & 0 & 0 & \ldots & \alpha_{r-2} & \beta_{r-2} & \gamma_{r-2} \\ \gamma_{r-1} & 0 & 0 & 0 & \ldots & 0 & \alpha_{r-1} & \beta_{r-1} \end{bmatrix} \times$$

$$\times \begin{bmatrix} u_k \\ u_{k+1} \\ u_{k+2} \\ \ldots \\ u_{r-2} \\ u_{r-1} \end{bmatrix} = \begin{bmatrix} \delta_k \\ \delta_{k+1} \\ \delta_{k+2} \\ \ldots \\ \delta_{r-2} \\ \delta_{r-1} \end{bmatrix}.$$

The matrix of this equation has the so-called cyclic tridiagonal form.

The system of equations (3.82) — (3.84), i.e., the above matrix equation, is usually solved by the cyclic decomposition method. Let us consider this algorithm.

Let $y(x)$, $z(x)$ and $u(x)$ be grid functions defined on grid $x_k < x_{k+1} < x_{k+2} < \ldots < x_{r-2} < x_{r-1} < x_r$ (Fig. 3.1), and:

- variables y_i satisfy the following system of linear algebraic equations (3.9) with zero boundary conditions:

$$\alpha_i y_{i-1} + \beta_i y_i + \gamma_i y_{i+1} = \delta_i, \tag{3.86}$$
$$i = k+1, k+2, \ldots, r-2, r-1,$$

$$y_k = y_r = 0; \tag{3.87}$$

- variables z_i satisfy the following system of equations (3.9) with zero right-hand side and unit boundary conditions:

$$\alpha_i z_{i-1} + \beta_i z_i + \gamma_i z_{i+1} = 0, \tag{3.88}$$

$$i = k+1, k+2, ..., r-2, r-1,$$

$$z_k = z_r = 1; \tag{3.89}$$

- variables u_i are the following linear combination of y_i and z_i:

$$u_i = y_i + u_k z_i, \tag{3.90}$$

$$i = k, k+1, k+2, ..., r-2, r-1, r.$$

By substituting expression (3.90) into equations (3.83) and (3.84), we can easily verify that grid function (3.90), $u(x)$, satisfies these equations at an arbitrary value of u_k. Let us find the value of u_k, at which grid function $u(x)$ satisfies equation (3.82). For this purpose, we will consider the sweep formulas for solution of systems (3.86), (3.87) and (3.88), (3.89).

Let formulas (3.18) and (3.13) of the backward sweep look like

$$y_r = 0, \quad z_r = 1, \tag{3.91}$$

$$y_{i-1} = P_i y_i + Q_i, \quad z_{i-1} = P_i z_i + S_i, \tag{3.92}$$

where $i = r, r-1, ..., k+1$. In this case, formulas (3.16), (3.17) and (3.14), (3.15) of the forward sweep become

$$P_{k+1} = 0, \quad Q_{k+1} = 0, \quad S_{k+1} = 1, \tag{3.93}$$

$$P_{i+1} = -\frac{\gamma_i}{\alpha_i P_i + \beta_i}, \quad Q_{i+1} = \frac{\delta_i - \alpha_i Q_i}{\alpha_i P_i + \beta_i}, \quad S_{i+1} = \frac{-\alpha_i S_i}{\alpha_i P_i + \beta_i}, \tag{3.94}$$

where $i = k+1, k+2, ..., r-1$.

By substituting expression (3.90) at $i = r-1$ and $i = k+1$,

$$u_{r-1} = y_{r-1} + u_k z_{r-1},$$

$$u_{k+1} = y_{k+1} + u_k z_{k+1},$$

into equation (3.82), we obtain

$$\alpha_k (y_{r-1} + u_k z_{r-1}) + \beta_k u_k + \gamma_k (y_{k+1} + u_k z_{k+1}) = \delta_k$$

or

$$u_k = \frac{\delta_k - \alpha_k y_{r-1} - \gamma_k y_{k+1}}{\beta_k + \alpha_k z_{r-1} + \gamma_k z_{k+1}}. \tag{3.95}$$

At this value of u_k, the linear combination of grid functions $y(x)$ and $z(x)$, defined by formula (3.90), satisfies not only equations (3.83) and (3.84), but also equation (3.82).

According to the cyclic decomposition method, the system of linear algebraic equations (3.82) — (3.84) is solved as follows:

1) the forward sweep is performed according to formulas (3.93) and (3.94), $i = k+1, k+2, ..., r-1$;

2) the backward sweep is performed according to formulas (3.91) and (3.92), $i = r, r-1, ..., k+2$;

3) the value of u_k is calculated according to formulas (3.85) and (3.95);

4) the values of u_i $(k+1 \le i \le r-1)$ are calculated according to (3.90).

3.19. Program realization of the cyclic decomposition method

Let us consider a subroutine for solving linear differential equation (3.6) under the following periodicity condition: $u(x+\mathit{\Pi}) = u(x)$, where $\mathit{\Pi} = b - a$ is the period, $-\infty < x < \infty$. The coefficients and right-hand side of equation (3.6) are periodic functions: $g(x+\mathit{\Pi}) = g(x)$, $e(x+\mathit{\Pi}) = e(x)$, $f(x+\mathit{\Pi}) = f(x)$.

For program realization of the cyclic decomposition method, we put the following subroutine declaration into Module7 of the BookNM workbook.

Listing 3.16

```
Sub forwback(ByVal k, ByVal r, ByRef X() As Double, _
    ByRef G() As Double, ByRef E() As Double, _
    ByRef F() As Double, ByRef U() As Double)
    Dim alpha As Double, beta As Double
    Dim gamma As Double, delta As Double
    Dim i As Integer, w As Double
    Dim H() As Double: ReDim H(k + 1 To r)
    Dim P() As Double: ReDim P(k + 1 To r)
    Dim Q() As Double: ReDim Q(k + 1 To r)
    Dim S() As Double: ReDim S(k + 1 To r)
    Dim Y() As Double: ReDim Y(k + 1 To r)
    Dim Z() As Double: ReDim Z(k + 1 To r)
    For i = k + 1 To r
        H(i) = X(i) - X(i - 1)
    Next i
'Forward sweep:
    P(k + 1) = 0
    Q(k + 1) = 0
    S(k + 1) = 1
    For i = k + 1 To r - 1
        w = H(i) + H(i + 1)
        alpha = H(i + 1) * (2 - G(i) * H(i + 1)) / w
        beta = E(i) * H(i) * H(i + 1) - _
        G(i) * (H(i) - H(i + 1)) - 2
        gamma = H(i) * (2 + G(i) * H(i)) / w
        delta = F(i) * H(i) * H(i + 1)
        w = alpha * P(i) + beta
        P(i + 1) = -gamma / w
        Q(i + 1) = (delta - alpha * Q(i)) / w
        S(i + 1) = -alpha * S(i) / w
    Next i
'Backward sweep:
    Y(r) = 0
    Z(r) = 1
    For i = r To k + 2 Step -1
        Y(i - 1) = P(i) * Y(i) + Q(i)
        Z(i - 1) = P(i) * Z(i) + S(i)
    Next i
'Calculation of solution:
    w = H(r) + H(k + 1)
    alpha = H(k + 1) * (2 - G(k) * H(k + 1)) / w
    beta = E(k) * H(r) * H(k + 1) - _
    G(k) * (H(r) - H(k + 1)) - 2
    gamma = H(r) * (2 + G(k) * H(r)) / w
    delta = F(k) * H(r) * H(k + 1)
    U(k) = (delta - alpha * Y(r - 1) - _
    gamma * Y(k + 1)) / (beta + alpha * Z(r - 1) + _
    gamma * Z(k + 1))             'calculation of U(k)
    For i = k + 1 To r - 1
        U(i) = Y(i) + U(k) * Z(i) 'calculation of U(i)
    Next i
    U(r) = U(k)
End Sub
```

The `forwback` subroutine parameters have the following sense:

- `k`, `r` are numbers of the left and right boundary nodes of the grid on segment $[a, b]$;
- `X` is an array of grid nodes;
- `G`, `E` are arrays of values of the coefficients of equation (3.6) at the grid nodes;
- `F` is an array of values of the right-hand side of equation (3.6);
- `U` is an array intended for the solution values.

Elements `U(k)` and `U(r)` are equal in the resulting `U` array.

3.20. Solving the oscillation equation

We will use the cyclic decomposition method for mathematical modeling of the oscillating motion of a bar. For this, we will be guided by the theory of the fifth chapter in book [8].

Let a bar with mass M be attached to the free end of a spring (Fig. 3.16). The bar's position is defined by horizontal coordinate u; the origin of coordinates ($u=0$) corresponds to the equilibrium state, in which the spring is not strained.

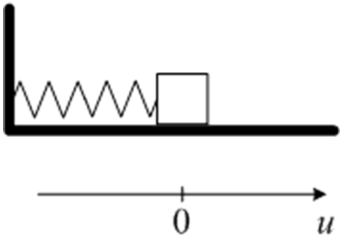


Fig. 3.16. A bar sliding along a horizontal surface

If the bar is deviated to the left or right from the equilibrium state, the spring resilience force (directed opposite to the displacement) acts on the bar. The force value is defined by formula

$$F_r = -Ku \; .$$

Positive coefficient K is called the elastic constant of the spring. As is customary in physics, we consider that the direction of the force vector coincides with the u axis for positive values of F_r (that is, the force is directed from left to right) and the force vector is directed opposite to the u axis for negative values of F_r (that is, the force is directed from right to left).

Let us deviate the bar from the equilibrium state and then release it. The bar starts to make an oscillatory motion, which is described by the following equation of Newton's second law:

$$M\frac{d^2u}{dt^2} = -Ku \; ,$$

where d^2u/dt^2 is the bar acceleration.

If the initial deviation of the bar (from the equilibrium state) is positive and equal to A, the solution of the last equation looks like

$$u(t) = A\cos(\omega_0 t) ,$$

where A is the oscillation amplitude, $\omega_0 = \sqrt{K/M}$ is the cyclic frequency of the oscillatory motion. The cosine argument, $\omega_0 t$, is in radians.

Cyclic frequency ω_0 (in radians per second) is related with oscillation frequency f_0 (in hertzs) and period T_0 (in seconds) as follows:

$$\omega_0 = 2\pi\, f_0 = 2\pi / T_0 .$$

In the presence of friction between the bar and horizontal surface, we observe the oscillation damping. If the bar velocity is small, the friction force is proportional to the bar mass, *M*, and to the first power of the bar velocity, du/dt. The friction force is directed opposite to the velocity vector.

Taking into account the friction, we write the equation of Newton's second law as follows:

$$M\frac{d^2u}{dt^2} = -K\,u - LM\frac{du}{dt} ,$$

where $L \geq 0$ is the so-called coefficient of the oscillation damping.

In order to exclude the oscillation damping, we introduce force $F(t)$ with period *T*. The equation of Newton's second law becomes

$$M\frac{d^2u}{dt^2} = -Ku - LM\frac{du}{dt} + F(t) . \tag{3.96}$$

It is obvious that after a while the oscillation of the bar becomes periodic with period *T*.

Let us consider that time *t* changes from a negative value, t_0, to infinity. Let t_1 be a negative value exceeding t_0, such that:

- $t_0 \leq t < t_1$ is an area of establishing the oscillation, in which the $u(t)$ solution is an aperiodic function;
- $t \geq t_1$ is an area of the established oscillation, in which the $u(t)$ solution is a periodic function with period *T*.

Because the periodic oscillation of the bar is interesting for us, we consider equation (3.96) on segment $[0, T]$ with the solution periodicity condition.

Equation (3.96) can be written in form (3.6),

$$\frac{d^2u}{dt^2} + g\frac{du}{dt} + eu = f(t) \,, \tag{3.97}$$

where $g = L$, $e = \omega_0^2 = K / M$, $f(t) = F(t) / M$.

Let us consider the following source data table.

Listing 3.17

M	0.001
K	800
L	100
F	t
0.00E+00	0.00E+00
0.00E+00	2.00E-03
0.00E+00	4.00E-03
0.00E+00	6.00E-03
0.00E+00	8.00E-03
0.00E+00	1.00E-02
0.00E+00	1.20E-02
0.00E+00	1.40E-02
0.00E+00	1.60E-02
0.00E+00	1.80E-02
0.00E+00	2.00E-02
0.00E+00	2.20E-02
0.00E+00	2.40E-02
0.00E+00	2.60E-02
5.00E+00	2.80E-02
1.00E+01	3.00E-02
5.00E+00	3.20E-02
0.00E+00	3.40E-02
0.00E+00	3.60E-02
0.00E+00	3.80E-02
0.00E+00	4.00E-02
0.00E+00	4.20E-02
0.00E+00	4.40E-02
0.00E+00	4.60E-02
0.00E+00	4.80E-02
0.00E+00	5.00E-02

In this table:

- M is the bar mass in kilograms;
- K is the elastic constant of the spring, in N / m;
- L is the coefficient of the oscillation damping, in 1 / s;
- t are the values of time for one period, in seconds;
- F are the $F(t)$ function values in newtons.

According to table Listing 3.17:

1) every 0.05 seconds, a positive force acts on the bar, for example, it gets a kick from left to right;
2) the maximum value of the force equals 10 N;
3) the duration of the force action equals 0.008 s.

The program for solving equation (3.97) under the periodicity condition is given below.

Listing 3.18

```
Sub main()
    Dim T() As Double
    Dim G() As Double
    Dim E() As Double
    Dim F() As Double
    Dim U() As Double
    Dim m As Integer
    Dim MM As Double
    Dim KK As Double
    Dim LL As Double
    Dim i As Integer
    Dim sb As String, se As String
    Dim sn As String
    m = Selection.Rows.Count         'quantity of rows
    MM = Selection.Cells(1, 2)
    KK = Selection.Cells(2, 2)
    LL = Selection.Cells(3, 2)
    ReDim T(5 To m)
    ReDim G(5 To m)
    ReDim E(5 To m)
    ReDim F(5 To m)
    ReDim U(5 To m)
    For i = 5 To m
        T(i) = Selection.Cells(i, 2)
        G(i) = LL
        E(i) = KK / MM
        F(i) = Selection.Cells(i, 1) / MM
    Next i
0:  Call forwback(5, m, T, G, E, F, U)
    Selection.Cells(4, 3) = "u"
    For i = 5 To m
        Selection.Cells(i, 3) = U(i)
    Next i
1:  sb = Selection.Cells(5, 2).Address
2:  se = Selection.Cells(m, 3).Address
3:  Range(sb & ":" & se).Select
4:  sn = ActiveSheet.Name
5:  Selection.NumberFormat = "0.00E+00"
6:  Charts.Add
7:  ActiveChart.ChartType = xlXYScatterSmoothNoMarkers
8:  ActiveChart.SetSourceData Source:= _
        Sheets(sn).Range(sb & ":" & se), PlotBy:= _
        xlColumns
9:  ActiveChart.Location Where:= xlLocationAsObject, _
        Name:=sn
10: ActiveChart.Axes(xlValue).MajorGridlines.Select
11: Selection.Delete
12: ActiveChart.Legend.Select
13: Selection.Delete
14: With ActiveChart
15:     .Axes(xlCategory, xlPrimary).HasTitle = True
16:     .Axes(xlCategory, _
        xlPrimary).AxisTitle.Characters.Text = "t, s"
17:     .Axes(xlValue, xlPrimary).HasTitle = True
18:     .Axes(xlValue, _
        xlPrimary).AxisTitle.Characters.Text = "u, m"
19: End With
20: ActiveChart.Axes(xlCategory).AxisTitle.Select
21: Selection.AutoScaleFont = True
22: With Selection.Font
23:     .FontStyle = "regular"
24:     .Size = 12
25: End With
26: ActiveChart.Axes(xlValue).AxisTitle.Select
27: Selection.AutoScaleFont = True
28: With Selection.Font
29:     .FontStyle = "regular"
30:     .Size = 12
```

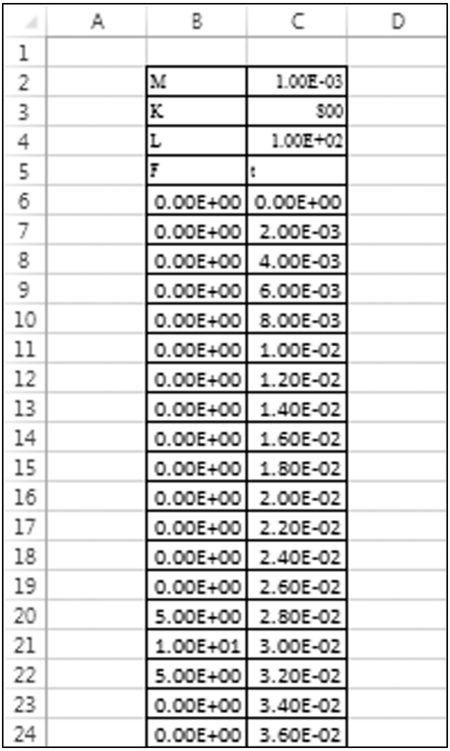
```
31: End With
32: ActiveChart.ChartArea.Select
End Sub
```

In this program, operator `0` is the call of the `forwback` subroutine, realizing the cyclic decomposition method.

The source data for the program are the values located in table Listing 3.17 (Fig. 3.17). We must select this Excel table (range B2:C31) before the program execution. The execution results are the solution values, located in the *u* column (near the *t* column, Fig. 3.18), and the $u(t)$ graph on the Excel worksheet.

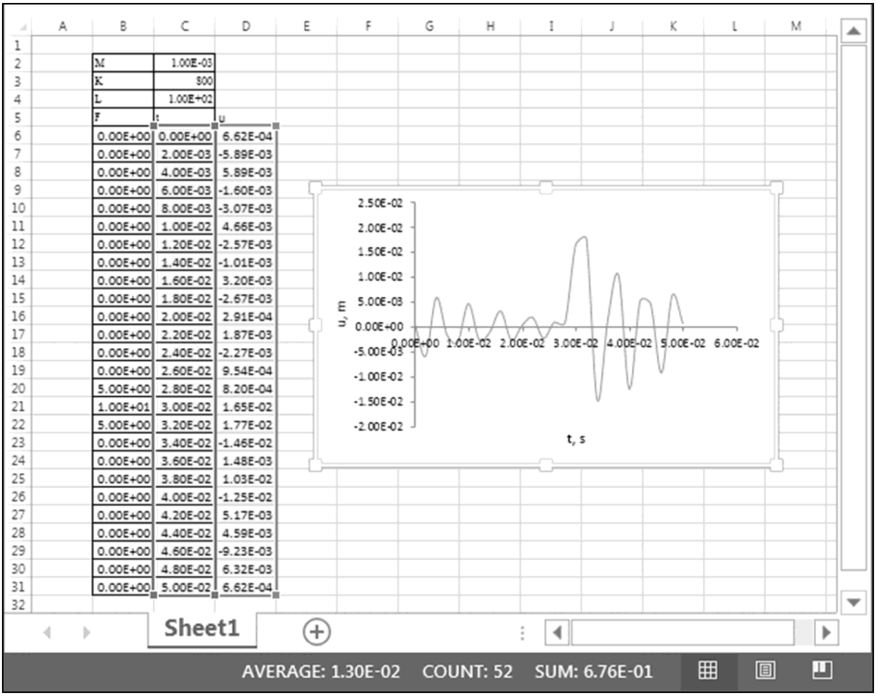
The $u(t)$ graph is created automatically:

- operators `1` — `13` of program Listing 3.18 construct the graph and delete the gridlines and legend;
- operators `14` — `32` superscribe the axes.



	A	B	C	D
1				
2		M	1.00E-03	
3		K	800	
4		L	1.00E+02	
5		F	t	
6		0.00E+00	0.00E+00	
7		0.00E+00	2.00E-03	
8		0.00E+00	4.00E-03	
9		0.00E+00	6.00E-03	
10		0.00E+00	8.00E-03	
11		0.00E+00	1.00E-02	
12		0.00E+00	1.20E-02	
13		0.00E+00	1.40E-02	
14		0.00E+00	1.60E-02	
15		0.00E+00	1.80E-02	
16		0.00E+00	2.00E-02	
17		0.00E+00	2.20E-02	
18		0.00E+00	2.40E-02	
19		0.00E+00	2.60E-02	
20		5.00E+00	2.80E-02	
21		1.00E+01	3.00E-02	
22		5.00E+00	3.20E-02	
23		0.00E+00	3.40E-02	
24		0.00E+00	3.60E-02	

Fig. 3.17. The Excel table with the source data



	A	B	C	D
1				
2		M	1.00E-03	
3		K	800	
4		L	1.00E+02	
5		F	t	u
6		0.00E+00	0.00E+00	6.62E-04
7		0.00E+00	2.00E-03	-5.89E-03
8		0.00E+00	4.00E-03	5.89E-03
9		0.00E+00	6.00E-03	-1.60E-03
10		0.00E+00	8.00E-03	-3.07E-03
11		0.00E+00	1.00E-02	4.66E-03
12		0.00E+00	1.20E-02	-2.57E-03
13		0.00E+00	1.40E-02	-1.01E-03
14		0.00E+00	1.60E-02	3.20E-03
15		0.00E+00	1.80E-02	-2.67E-03
16		0.00E+00	2.00E-02	2.91E-04
17		0.00E+00	2.20E-02	1.87E-03
18		0.00E+00	2.40E-02	-2.27E-03
19		0.00E+00	2.60E-02	9.54E-04
20		5.00E+00	2.80E-02	8.20E-04
21		1.00E+01	3.00E-02	1.65E-02
22		5.00E+00	3.20E-02	1.77E-02
23		0.00E+00	3.40E-02	-1.46E-02
24		0.00E+00	3.60E-02	1.48E-03
25		0.00E+00	3.80E-02	1.03E-02
26		0.00E+00	4.00E-02	-1.25E-02
27		0.00E+00	4.20E-02	5.17E-03
28		0.00E+00	4.40E-02	4.59E-03
29		0.00E+00	4.60E-02	-9.23E-03
30		0.00E+00	4.80E-02	6.32E-03
31		0.00E+00	5.00E-02	6.62E-04
32				

AVERAGE: 1.30E-02 COUNT: 52 SUM: 6.76E-01

Fig. 3.18. The program execution results,
including the graph of periodic dependence
$u(t)$ with frequency $1/T = 20\,\text{Hz}$

The reviewed operators (from `1` to `32`) were programmed by means of Excel Macro Recorder. In Section 4.8, we will consider the graph creation subroutine based on these operators.

In Section 5.12, we will use periodic dependence $u(t)$ depicted above in Fig. 3.18.

We advise the reader to develop a program, similar to Listing 3.18, for calculating a periodic time dependence of the current, $i(t)$, in the electrical circuit shown in Fig. 3.19. The electromotive force (in volts) of the generator is the following periodic function of time: $v(t+\kappa\Pi) = f(t)$, where t is time in seconds, $a \le t \le b$, $\Pi = b - a$ is the period, κ is an integer. Function $f(t)$ and the values of a and b are given in Appendix 4.

The process of solving the suggested task must include the following two stages:

1) on segment $[a, b]$, periodic dependence of the capacitor charge, $q(t)$, is calculated by solving equation

$$L\frac{d^2q}{dt^2}+R\frac{dq}{dt}+\frac{1}{C}q=v(t)\,;$$

2) the current is calculated by differentiation of the capacitor charge:

$$i(t)=\frac{dq}{dt}\,.$$

Dependences $v(t)$ and $i(t)$ will be figured in the task on p. 414.

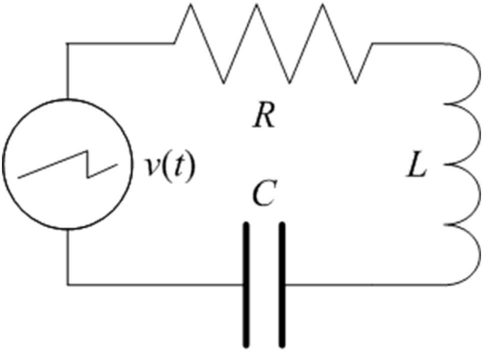


Fig. 3.19. The electrical circuit with the following parameters: electric resistance R = 100 Ω, inductance L = 0.2 H, capacitance C = 3·10^{-4} F

Chapter 4.
Cubic Spline

Let us begin with the origin of term “spline”.

Long ago, engineers needed to draw smooth curves through given points. To do so, they used long elastic wooden strips. Such a strip, called a spline, was fixed (nailed up to a drawing board) at the given points. As a result, the strip was bent to provide the smooth curve.

In mathematics, the function describing the bending of an elastic strip is called a third-degree (cubic) spline [9]. In this chapter, this mathematical construction is used for interpolation, differentiation and integration of the grid (tabular) function and also for solving the nonlinear algebraic and linear differential equations. Besides, we consider two classical methods for solving the nonlinear algebraic equation, namely, the bisection and secant methods.

For demonstration of the spline possibilities, we solve applied problems concerning the field-effect transistor, silicon photosensitive target and geophysical cable. The locally one-dimensional scheme [4] is considered for solving the heat equation of the last applied problem with two spatial coordinates.

In addition to user-defined procedures, realizing the numerical methods, a subroutine for automatic creation of graphs is developed.

4.1. Definition of cubic spline. Spline moments

Let an increasing sequence of points on segment $[a, b]$ be given as follows: $a = x_k < x_{k+1} < x_{k+2} < ... < x_{r-2} < x_{r-1} < x_r = b$. In other words, as in the previous chapter of the book, segment $[a, b]$ is covered with a grid whose nodes have coordinates x_k, x_{k+1}, x_{k+2}, ..., x_{r-2}, x_{r-1}, x_r. Segments $[x_{i-1}, x_i]$ are called elementary segments, $k + 1 \le i \le r$.

Let $f(x)$ be a grid function, and f_k, f_{k+1}, f_{k+2}, ..., f_{r-2}, f_{r-1}, f_r are given values of $f(x)$ at points x_k, x_{k+1}, x_{k+2}, ..., x_{r-2}, x_{r-1}, x_r, respectively.

A cubic spline (or third-degree spline, Fig. 4.1) is function $S(x)$, which satisfies the following conditions:

1) on each elementary segment $x_{i-1} \le x \le x_i$ ($k + 1 \le i \le r$), the spline coincides with a third-degree polynomial (generally, the polynomials are different on different elementary segments);

2) at the grid nodes, the spline has the corresponding grid function values: $S(x_i) = f_i$;

3) the spline has a continuous first derivative, i.e., the spline is smooth;

4) the spline has a continuous second derivative;

5) on the boundaries of segment $[a, b]$, the spline satisfies additional conditions (we will consider these boundary conditions below, closer to the end of the section).

According to above item (2), the $S(x)$ graph passes through points (x_k, f_k), (x_{k+1}, f_{k+1}), (x_{k+2}, f_{k+2}), . . . , (x_{r-2}, f_{r-2}), (x_{r-1}, f_{r-1}), (x_r, f_r). According to items (3) and (4), the jumps of the first and second derivatives of $S(x)$ are absent at the interior grid nodes, i.e., at x_{k+1}, x_{k+2}, ..., x_{r-2}, x_{r-1}.

The values of the second derivative, $S''(x)$, at the grid nodes are called moments of the cubic spline:

$$\frac{d^2S}{dx^2}(x_i) = M_i,$$

where M_i is the spline moment, $k \le i \le r$.

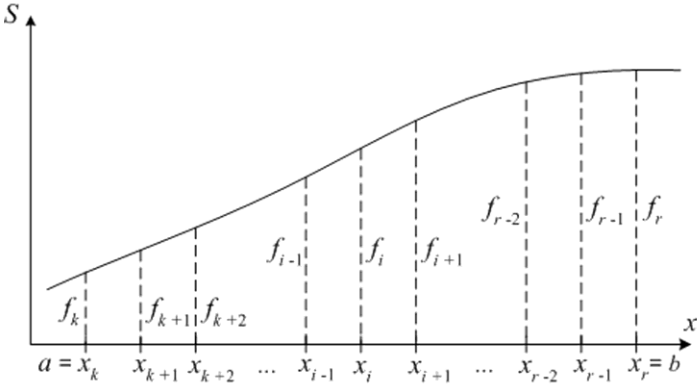


Fig. 4.1. The cubic spline graph

Let the spline moments, M_i ($k \le i \le r$), be given (later we will know how to calculate them). In this case, items (1) and (4) of the spline definition give the following expression for the second derivative on elementary segment $[x_{i-1}, x_i]$:

$$S''(x) = M_{i-1}\frac{x_i - x}{h_i} + M_i\frac{x - x_{i-1}}{h_i}, \tag{4.1}$$

where $h_i = x_i - x_{i-1}$ is the elementary segment's length or the grid step, $k+1 \le i \le r$.

Below, we will obtain expressions for the spline and its first derivative.

Let us integrate expression (4.1):

$$S'(x) = -M_{i-1}\frac{(x_i - x)^2}{2h_i} + M_i\frac{(x - x_{i-1})^2}{2h_i} + C_1, \tag{4.2}$$

where C_1 is the integration constant.

Integrating expression (4.2), we have

$$S(x) = M_{i-1}\frac{(x_i - x)^3}{6h_i} + M_i\frac{(x - x_{i-1})^3}{6h_i} + C_1 x + C_2, \tag{4.3}$$

where C_2 is the integration constant.

Integration constants C_1 and C_2 will be determined by means of item (2) of the spline definition, according to which $S(x_i) = f_i$ and $S(x_{i-1}) = f_{i-1}$ or

$$x_i C_1 + C_2 = f_i - M_i\frac{(x_i - x_{i-1})^3}{6h_i},$$

$$x_{i-1} C_1 + C_2 = f_{i-1} - M_{i-1}\frac{(x_i - x_{i-1})^3}{6h_i}.$$

The solution of this system of two linear algebraic equations (with unknown C_1 and C_2) has the following form:

$$C_1 = \frac{f_i - f_{i-1}}{h_i} - \frac{M_i - M_{i-1}}{6}h_i,$$

$$C_2 = -f_i\frac{x_{i-1}}{h_i} + f_{i-1}\frac{x_i}{h_i} + M_i\frac{h_i}{6}x_{i-1} - M_{i-1}\frac{h_i}{6}x_i.$$

By substituting the last two expressions for C_1 and C_2 into (4.3) and (4.2), we obtain the following expressions for the spline and its derivative on segment $[x_{i-1}, x_i]$:

$$S(x) = M_{i-1}\frac{(x_i - x)^3}{6h_i} + M_i\frac{(x - x_{i-1})^3}{6h_i} + \left(f_{i-1} - \frac{M_{i-1}h_i^2}{6} \right)\frac{x_i - x}{h_i} +$$
$$+\left(f_i - \frac{M_i h_i^2}{6} \right)\frac{x - x_{i-1}}{h_i}, \tag{4.4}$$

$$S'(x) = -M_{i-1}\frac{(x_i - x)^2}{2h_i} + M_i\frac{(x - x_{i-1})^2}{2h_i} + \frac{f_i - f_{i-1}}{h_i} -$$
$$-\frac{M_i - M_{i-1}}{6}h_i. \tag{4.5}$$

Expressions (4.1), (4.4) and (4.5) include moments M_{i-1} and M_i.

Below, we will obtain the system of linear algebraic equations with the tridiagonal coefficient matrix, which allows us to calculate moments M_k, M_{k+1}, M_{k+2}, ..., M_{r-2}, M_{r-1}, M_r.

The expression for the spline derivative at point x_i on the left,

$$S'(x_i - 0) = M_i \frac{(x_i - x_{i-1})^2}{2h_i} + \frac{f_i - f_{i-1}}{h_i} - \frac{M_i - M_{i-1}}{6} h_i , \tag{4.6}$$

follows from (4.5).

Expression (4.5) for elementary segment $x_i \le x \le x_{i+1}$ looks like

$$S'(x) = -M_i \frac{(x_{i+1} - x)^2}{2h_{i+1}} + M_{i+1} \frac{(x - x_i)^2}{2h_{i+1}} + \frac{f_{i+1} - f_i}{h_{i+1}} - \\ - \frac{M_{i+1} - M_i}{6} h_{i+1} . \tag{4.7}$$

The expression for the spline derivative at point x_i on the right,

$$S'(x_i + 0) = -M_i \frac{(x_{i+1} - x_i)^2}{2h_{i+1}} + \frac{f_{i+1} - f_i}{h_{i+1}} - \frac{M_{i+1} - M_i}{6} h_{i+1}, \tag{4.8}$$

follows from (4.7).

According to item (3) of the spline definition, the left and right derivatives are equal:

$$S'(x_i - 0) = S'(x_i + 0) .$$

By means of expressions (4.6) and (4.8), the last equality can be written as follows:

$$\frac{h_i}{6} M_{i-1} + \frac{h_i + h_{i+1}}{3} M_i + \frac{h_{i+1}}{6} M_{i+1} = \frac{f_{i+1} - f_i}{h_{i+1}} - \frac{f_i - f_{i-1}}{h_i} ,$$

where $i = k+1, k+2, \ldots, r-2, r-1$ are the numbers of the interior grid nodes, or

$$\alpha_i M_{i-1} + 2M_i + \gamma_i M_{i+1} = \delta_i , \tag{4.9}$$

where

$$\alpha_i = \frac{h_i}{h_i + h_{i+1}} ,$$

$$\gamma_i = \frac{h_{i+1}}{h_i + h_{i+1}} = 1 - \alpha_i , \tag{4.10}$$

$$\delta_i = 6 \frac{\dfrac{f_{i+1} - f_i}{h_{i+1}} - \dfrac{f_i - f_{i-1}}{h_i}}{h_i + h_{i+1}} .$$

As the boundary conditions in item (5) of the spline definition, we will use the following linear equations connecting the moments at the ends of segment $[a, b]$:

$$2M_k + \gamma_k M_{k+1} = \delta_k , \tag{4.11}$$

$$\alpha_r M_{r-1} + 2M_r = \delta_r , \tag{4.12}$$

where γ_k, δ_k, α_r, δ_r are given parameters, *k* and *r* are numbers of the left and right grid nodes; $x_k = a$ and $x_r = b$.

If a given value of the function derivative on the left boundary, $f_k' = f'(a)$, is the condition in item (5), then (4.8) at $i = k$ leads to the following expressions for the parameters of equation (4.11):

$$\gamma_k = 1, \quad \delta_k = \frac{6}{h_{k+1}} \left(\frac{f_{k+1} - f_k}{h_{k+1}} - f_k' \right). \tag{4.13}$$

If a given value of the function derivative on the right boundary, $f_r' = f'(b)$, is the condition in item (5), then (4.6) at $i = r$ leads to the following expressions for the parameters of equation (4.12):

$$\alpha_r = 1, \quad \delta_r = \frac{6}{h_r} \left(f_r' - \frac{f_r - f_{r-1}}{h_r} \right). \tag{4.14}$$

A given value of the second derivative on the left boundary ($M_k = f_k''$) leads to the following expressions for the parameters of equation (4.11):

$$\gamma_k = 0, \quad \delta_k = 2 f_k'' . \tag{4.15}$$

A given value of the second derivative on the right boundary ($M_r = f_r''$) leads to the following expressions for the parameters of equation (4.12):

$$\alpha_r = 0, \quad \delta_r = 2 f_r'' . \tag{4.16}$$

The constancy of the second derivative at the left end of segment $[a, b]$ ($M_k = M_{k+1}$) leads to

$$\gamma_k = -2\,, \quad \delta_k = 0 \tag{4.17}$$

in equation (4.11).

The constancy of the second derivative at the right end of segment $[a,\,b]$ ($M_r = M_{r-1}$) leads to

$$\alpha_r = -2\,, \quad \delta_r = 0 \tag{4.18}$$

in equation (4.12).

Expressions (4.13) — (4.18), as well as equations (4.11) and (4.12), may be called spline boundary conditions.

Moments M_k, M_{k+1}, M_{k+2}, ..., M_{r-2}, M_{r-1}, M_r are determined by solving the system of linear algebraic equations (4.9), (4.11) and (4.12). In this case, the decomposition method (Section 3.2) can be used because forms (3.9), (3.11) and (3.12) are available for equations (4.9), (4.11) and (4.12).

After calculating the moments, the values of the cubic spline and its first and second derivatives at any point x of segment $[a,\,b]$ can be calculated according to formulas (4.4), (4.5) and (4.1), respectively.

The error of interpolating the $f(x)$ function (and its derivatives) by the $S(x)$ spline (and by its derivatives) is determined by the following expression:

$$f^{(n)}(x) - S^{(n)}(x) = O(h_{max}^{4-n})\,, \tag{4.19}$$

where $h_{max} = \max\limits_{k+1 \le i \le r} \{h_i\}$ is the maximum grid step ($h_{max} \to 0$), n = 0, 1, 2, 3 is the derivative order, $f^{(0)}(x) = f(x)$, $S^{(0)}(x) = S(x)$. We considered the sense of the O notation used here in Section 3.1.

4.2. Spline interpolation

Into Module8 of the BookNM workbook, we enter the following declaration of the subroutine, which realizes the decomposition method for solving the system of linear algebraic equations (4.9), (4.11) and (4.12), i.e., for calculating the spline moments.

Listing 4.1

```
Sub mos(ByVal k, ByVal r, ByRef X() As Double, _
    ByRef F() As Double, _
    ByVal GAMMAK, ByVal DELTAK, _
    ByVal ALPHAR, ByVal DELTAR, _
    ByRef M() As Double)
    Dim alpha As Double
    Dim gamma As Double, delta As Double
    Dim i As Integer, w As Double
    Dim H() As Double: ReDim H(k + 1 To r)
    Dim P() As Double: ReDim P(k + 1 To r)
    Dim Q() As Double: ReDim Q(k + 1 To r)
    For i = k + 1 To r
        H(i) = X(i) - X(i - 1)
    Next i
'Forward sweep:
    P(k + 1) = -GAMMAK / 2
    Q(k + 1) = DELTAK / 2
    For i = k + 1 To r - 1
        w = H(i) + H(i + 1)
        alpha = H(i) / w
        gamma = 1 - alpha
        delta = 6 * ((F(i + 1) - F(i)) / H(i + 1) - _
        (F(i) - F(i - 1)) / H(i)) / w
        w = alpha * P(i) + 2
        P(i + 1) = -gamma / w
        Q(i + 1) = (delta - alpha * Q(i)) / w
    Next i
```

```
'Backward sweep:
    M(r) = (DELTAR - ALPHAR * Q(r)) / _
     (ALPHAR * P(r) + 2)
    For i = r To k + 1 Step -1
        M(i - 1) = P(i) * M(i) + Q(i)
    Next i
End Sub
```

The subroutine name (`mos`) occurs from “moments of spline”. The parameters have the following sense:

- `k`, `r` are numbers of the left and right boundary nodes of the grid on segment $[a, b]$;
- `X` is an array of grid nodes;
- `F` is an array of the $f(x)$ function values at the grid nodes;
- `GAMMAK`, `DELTAK` correspond to γ_k and δ_k in left boundary condition (4.11);
- `ALPHAR`, `DELTAR` correspond to α_r and δ_r in right boundary condition (4.12);
- `M` is an array intended for the spline moments.

The `mos` subroutine is based on the `foba` subroutine (Section 3.15).

In practice, not only the spline with boundary conditions (4.11) and (4.12) is used, but also the periodic spline defined as follows.

Let $f(x)$ be a periodic grid function with period $\varPi = b - a$:

$$f(x_{i+r-k}) = f(x_i),$$

where $x_{i+r-k} = x_i + b - a$, $-\infty < i < \infty$.

The periodic third-degree spline is function $S(x)$ defined on the whole axis, $-\infty < x < \infty$, for which:

1) the first four conditions of the cubic spline definition (p. 282) are satisfied on segment $[a, b]$;

2) $S'(x_{i+r-k}) = S'(x_i)$ and $S''(x_{i+r-k}) = S''(x_i)$ for $-\infty < i < \infty$.

We will use the periodic third-degree spline in Section 5.11, at that, we will not calculate the moments of this spline. If the reader needs a subroutine for calculating the moments of the periodic spline, its development is not a difficult task: the `forwback` subroutine (Section 3.19), realizing the cyclic decomposition method, should be the basis for the new subroutine.

Let us consider a subroutine of spline interpolation intended for calculating values of the cubic spline (periodic or with the boundary conditions) and its first and second derivatives at given point χ of segment $[a, b]$. This subroutine is named `si` from “spline interpolation”.

Into Module9 of the BookNM workbook, we enter the following declaration of the `si` subroutine:

Listing 4.2

```
Sub si(ByVal k, ByVal r, ByRef X() As Double, _
    ByRef F() As Double, ByRef M() As Double, _
    ByVal chi, ByRef s, _
    Optional s1 As Variant, Optional s2 As Variant)
    Dim i As Integer
    Dim h As Double, hh As Double
    Dim h1 As Double, h1h1 As Double
    Dim h2 As Double, h2h2 As Double
'Searching elementary segment containing chi:
    For i = k + 1 To r
        If X(i) > chi Then Exit For
    Next i
    If i > r Then i = r
'Calculating value of cubic spline at point chi:
    h = X(i) - X(i - 1): hh = h * h
    h1 = chi - X(i - 1): h1h1 = h1 * h1
    h2 = X(i) - chi: h2h2 = h2 * h2
    s = (M(i - 1) * h2h2 * h2 + M(i) * h1h1 * h1) / _
    (6 * h) + _
    ((F(i - 1) - M(i - 1) * hh / 6) * h2 + _
    (F(i) - M(i) * hh / 6) * h1) / h
'Calculating spline's first derivative at point chi:
    If Not IsMissing(s1) Then
        s1 = (-M(i - 1) * h2h2 + M(i) * h1h1) / _
        (2 * h) + _
        (F(i) - F(i - 1)) / h - _
        (M(i) - M(i - 1)) / 6 * h
    End If
'Calculating spline's second derivative at point chi:
    If Not IsMissing(s2) Then
        s2 = (M(i - 1) * h2 + M(i) * h1) / h
    End If
End Sub
```

This subroutine has 9 parameters, and the last two parameters, `s1` and `s2`, are optional. The parameters have the following sense:

- `k`, `r` are numbers of the left and right boundary nodes of the grid on segment $[a, b]$;
- `X` is an array of grid nodes;
- `F` is an array of the $f(x)$ function values at the grid nodes;
- `M` is an array of the spline moments, for example, determined by the `mos` subroutine execution;
- `chi` is given point χ on $[a, b]$;
- `s` is a variable (memory cell) intended for the spline's value at the χ point;
- `s1`, `s2` are variables respectively intended for the spline's first and second derivatives at the χ point.

In the `si` subroutine, cycle `For…Next` is used to find elementary segment $[x_{i-1}, x_i]$ containing the χ point. After finding this segment, the spline's value is calculated according to formula (4.4), and, if needed, the spline's first and second derivatives are calculated according to (4.5) and (4.1), respectively.

4.3. Use of cubic spline for processing transistor electrical characteristics

Transistors are the base elements of modern radio electronics. Two types of transistors exist — bipolar and field-effect transistors. Both those and other such elements are three-electrode devices based on semiconductors.

Let us consider a concrete example of using the cubic spline construction for processing electrical characteristics of a field-effect transistor, and such, in which the main carriers of electric current are electrons rather than holes.

The field-effect transistor electrodes are called source, drain and gate. Without delving into the device physics, we note the following regarding the electrodes:

- the source injects electrons into the semiconductor, the drain collects these electrons, the gate regulates the electron flow;
- by varying the electric potential difference between the gate and the source, U_{gs}, we change the drain current, I_d.

The major electrical characteristics of the field-effect transistor are the output current-voltage characteristics (OCVC) representing dependences of the drain current, I_d, on the potential difference between the drain and the source, U_{ds}, for various values of U_{gs}. Fig. 4.2 shows the OCVC calculated by means of mathematical model [10] for electron-hole plasma in the transistor. The simulation based on this model is quite time-consuming: the calculation of the I_d value (for given U_{gs} and U_{ds}) can take hours on a personal computer.

In the bottom right corner of this book's cover, we see the two-dimensional distribution of electron concentration in the transistor for U_{gs} = -3 V and U_{ds} = 14 V. This picture, as well as the OCVC, is from article [10].

Listing 4.3 with tabular representation of the OCVC is given below. In this table, as well as in Fig. 4.2, the values of potential differences are in volts, the current is in milliamperes. We see that 18 cells of the table are empty. It is because the I_d value was not calculated for some values of U_{gs} and U_{ds} due to economic reasons.

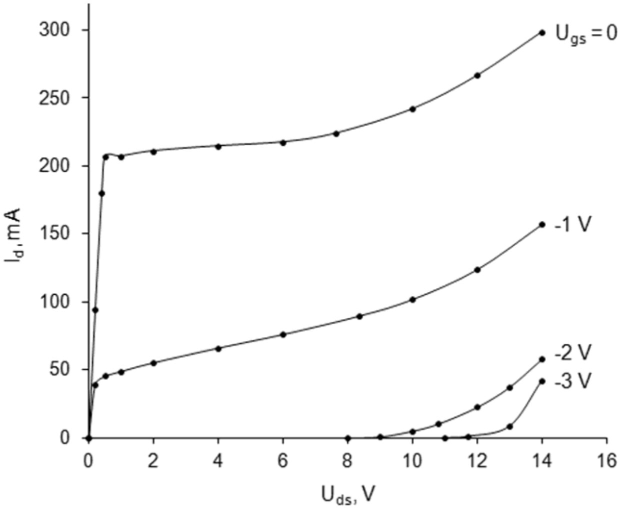


Fig. 4.2. The output current-voltage characteristics of the modern transistor with 1 mm gate width

Listing 4.3

The original table of the transistor characteristics

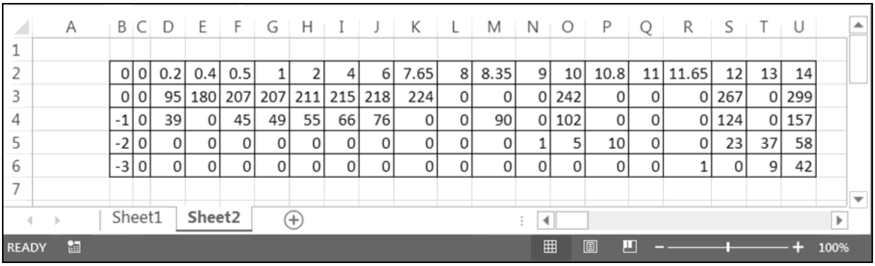
U_{ds} \ U_{gs}	0	-1	-2	-3
0	0	0	0	0
0.2	94.5065	39.3361	0	0
0.4	179.9835		0	0
0.5	206.7334	45.3559	0	0
1	207.3824	49.0988	0	0
2	211.1779	55.4772	0	0
4	214.8481	66.276	0	0
6	217.707	76.2717	0	0
7.65	224.2229		0	0
8			0	0
8.35		89.793	0	0
9			0.809	0
10	242.0198	102.0164	5.1056	0
10.8			10.369	0
11				0
11.65				1.0724
12	266.5736	123.9969	22.674	
13			37.3677	8.6525
14	298.5628	157.1723	58.0258	42.0182

To build electrical circuits based on the above transistor, we must know the parameters of its equivalent circuit. To determine these parameters by method [11], the empty cells of the OCVC table should be filled beforehand. The program given below allows doing it by means of the spline interpolation.

Listing 4.4

```
Sub main()
    Dim X() As Double
    Dim F() As Double
    Dim MOM() As Double
    Dim m As Integer, n As Integer
    Dim g As Integer, d As Integer
    Dim k As Integer, r As Integer
    Dim s As Double
    m = Selection.Rows.Count       'quantity of rows
    n = Selection.Columns.Count    'quantity of columns
    ReDim X(2 To n)
    ReDim F(2 To n)
    ReDim MOM(2 To n)
    For g = 2 To m                 'setting row number
'Formation of arrays X and F for row No. g:
        For r = 2 To n
            X(r) = Selection.Cells(1, r)
            F(r) = Selection.Cells(g, r)
            If F(r) <> 0 Then Exit For
        Next r
        k = r
        For d = k + 1 To n
            If Selection.Cells(g, d) <> 0 Then
                r = r + 1
                X(r) = Selection.Cells(1, d)
                F(r) = Selection.Cells(g, d)
            End If
        Next d
'Calculating array MOM of moments for row No. g:
        Call mos(2, r, X, F, 0, 0, 0, 0, MOM)
'Filling all cells of row No. g:
        For d = 2 To n
            Call si(2, r, X, F, MOM, _
            Selection.Cells(1, d), s)
            Selection.Cells(g, d) = s
        Next d
    Next g
End Sub
```

We enter this program into Module1 of the BookNM workbook. The source data are the values given in table Listing 4.3. The program uses this table in the transposed form (Fig. 4.3).



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
1																					
2		0	0	0.2	0.4	0.5	1	2	4	6	7.65	8	8.35	9	10	10.8	11	11.65	12	13	14
3		0	0	95	180	207	207	211	215	218	224	0	0	0	242	0	0	0	267	0	299
4		-1	0	39	0	45	49	55	66	76	0	0	90	0	102	0	0	0	124	0	157
5		-2	0	0	0	0	0	0	0	0	0	0	0	1	5	10	0	0	23	37	58
6		-3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	9	42
7																					

Fig. 4.3. The transposed table of the transistor characteristics: the number format without decimals is set for range C3:U6

For transposing a matrix, Excel includes the TRANSPOSE function (p. 141), which interprets the contents of empty cells as zero.

After using the TRANSPOSE function, we fulfill the following operations:

1) select the table shown in Fig. 4.3, which is the transposition result;

2) copy this table into Windows Clipboard, for example, by clicking on the *Copy* button in the *Clipboard* area of the *Home* tab;

3) click on the *Paste* arrow;

4) in the *Paste Values* area of the open window, click on the left icon.

The table, depicted in Fig. 4.3, visually does not change. We must select this table of the transistor characteristics (range B2:U6) before running program Listing 4.4.

When executing the program, the `g` cycle parameter accepts the values of 2, 3, 4 and 5. Respectively, the cells of the 2nd, 3rd, 4th and 5th rows of the selected table are being filled.

Let us consider arrays `X` and `F` at a fixed value of `g`. Before calling the `mos` subroutine, arrays `X` and `F` contain the following values.

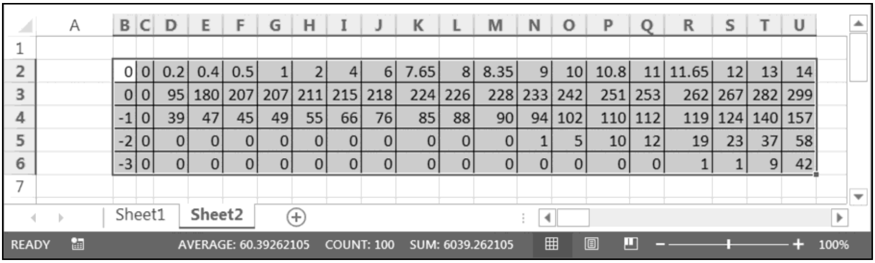
The `F` array begins with zero, i.e., `F(2) = 0`. The next elements of the `F` array are the values of the `g`-th row (of the selected table), but not all: the zeros, which are between the nonzero values, are eliminated. It is these zeros that filled the empty cells of table Listing 4.3 upon its transposition by the TRANSPOSE function.

The `X` array contains the values of the 1st row (of the selected table), which are located above the `g`-th row's values included in the `F` array.

For example:

- for fixed `g = 2`, array `F` contains the values of `F(2) = 0`, `F(3) = 95`, `F(4) = 180`, `F(5) = 207`, `F(6) = 207`, `F(7) = 211`, `F(8) = 215`, `F(9) = 218`, `F(10) = 224`, `F(11) = 242`, `F(12) = 267`, `F(13) = 299`, and array `X` contains the values of `X(2) = 0`, `X(3) = 0.2`, `X(4) = 0.4`, `X(5) = 0.5`, `X(6) = 1`, `X(7) = 2`, `X(8) = 4`, `X(9) = 6`, `X(10) = 7.65`, `X(11) = 10`, `X(12) = 12`, `X(13) = 14` ($k = 2$, $r = 13$);
- for fixed `g = 4`, array `F` contains the values of `F(2) = 0`, `F(3) = 0`, `F(4) = 0`, `F(5) = 0`, `F(6) = 0`, `F(7) = 0`, `F(8) = 0`, `F(9) = 0`, `F(10) = 0`, `F(11) = 0`, `F(12) = 0`, `F(13) = 1`, `F(14) = 5`, `F(15) = 10`, `F(16) = 23`, `F(17) = 37`, `F(18) = 58`, and array `X` contains the values of `X(2) = 0`, `X(3) = 0.2`, `X(4) = 0.4`, `X(5) = 0.5`, `X(6) = 1`, `X(7) = 2`, `X(8) = 4`, `X(9) = 6`, `X(10) = 7.65`, `X(11) = 8`, `X(12) = 8.35`, `X(13) = 9`, `X(14) = 10`, `X(15) = 10.8`, `X(16) = 12`, `X(17) = 13`, `X(18) = 14` ($k = 2$, $r = 18$).

For every value of `g` (2, 3, 4 and 5), the `MOM` array of the spline moments, corresponding to arrays `X` and `F`, is calculated by means of the `mos` subroutine. In the call of this subroutine, boundary conditions (4.15) and (4.16) are used, where $f''_2 = f''_{r(g)} = 0$. The values, being calculated when executing the `si` subroutine, fill the cells of the `g`-th row of the table. Thus, zero or nonzero values fill the cells with zeros of Fig. 4.3. Fig. 4.4 shows the program execution result.



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
1																					
2		0	0	0.2	0.4	0.5	1	2	4	6	7.65	8	8.35	9	10	10.8	11	11.65	12	13	14
3		0	0	95	180	207	207	211	215	218	224	226	228	233	242	251	253	262	267	282	299
4		-1	0	39	47	45	49	55	66	76	85	88	90	94	102	110	112	119	124	140	157
5		-2	0	0	0	0	0	0	0	0	0	0	0	1	5	10	12	19	23	37	58
6		-3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	9	42
7																					

Fig. 4.4. The transposed table of the transistor characteristics after the execution

After transposing the last table (Fig. 4.4) by means of the TRANSPOSE function, we have table Listing 4.5, which is similar to Listing 4.3, but without empty cells. Table Listing 4.5 can be used for calculating parameters of the transistor equivalent circuit by method [11].

Listing 4.5

The completed table of the transistor characteristics

U_{ds} U_{gs}	0	-1	-2	-3
0	0	0	0	0
0.2	94.5065	39.3361	0	0
0.4	179.9835	47.336	0	0
0.5	206.7334	45.3559	0	0
1	207.3824	49.0988	0	0
2	211.1779	55.4772	0	0
4	214.8481	66.276	0	0
6	217.707	76.2717	0	0
7.65	224.2229	85.364	0	0
8	226.068	87.537	0	0
8.35	228.206	89.793	0	0
9	232.931	94.227	0.809	0
10	242.0198	102.0164	5.1056	0
10.8	250.805	109.515	10.369	0
11	253.214	111.626	12.04	0
11.65	261.644	119.291	18.554	1.0724
12	266.5736	123.9969	22.674	1.102
13	282	139.645	37.3677	8.6525
14	298.5628	157.1723	58.0258	42.0182

4.4. Spline integration

According to the rules of integration and basic integrals [3], the integral of polynomial (4.4) over segment $[x_{i-1}, x_i]$ equals

$$\int_{x_{i-1}}^{x_i} S(x)dx = \frac{f_{i-1} + f_i}{2} h_i - \frac{M_{i-1} + M_i}{24} h_i^3 .$$

Therefore, the integral of the spline over segment $[a, b]$ equals

$$\int_a^b S(x)dx = \sum_{i=k+1}^{r} \frac{f_{i-1} + f_i}{2} h_i - \sum_{i=k+1}^{r} \frac{M_{i-1} + M_i}{24} h_i^3 . \qquad (4.20)$$

Formulas (4.19) and (4.20) give the following estimation of the spline integration error:

$$\int_a^b f(x)dx - \int_a^b S(x)dx = O(h_{max}^4) ,$$

where $h_{max} \to 0$ is the maximum grid step.

Let us enter the following declaration of function `ios` (from “integral of spline”) into Module10 of the BookNM workbook.

Listing 4.6

```
Function ios(ByVal k, ByVal r, ByRef X() As Double, _
    ByRef F() As Double, ByRef M() As Double)
    Dim i As Integer
    Dim h As Double
    ios = 0
    For i = k + 1 To r
        h = X(i) - X(i - 1)
        ios = ios + (F(i - 1) + F(i)) / 2 * h - _
            (M(i - 1) + M(i)) / 24 * h ^ 3
    Next i
End Function
```

The `ios` function returns (into the program) the value of the integral of spline $S(x)$ over segment $[a, b]$. Parameters `k`, `r`, `X`, `F` and `M` have the same sense as the corresponding parameters of the `si` subroutine (p. 291). We will use the `ios` function for solving the following task.

Table Listing 4.7 below is taken from Task 3.1 in book [8]. It contains the mass of a vertically falling plastic foam ball (with radius equal to one inch) and the results of measurement of its coordinate at different instants of time.

Listing 4.7

Experimental dependence of coordinate of a falling ball versus time

Ball mass M in kilograms	0.000254
Time t in seconds	Coordinate y in meters
-0.132	0
0	0.075
0.1	0.26
0.2	0.525
0.3	0.87
0.4	1.27
0.5	1.73
0.6	2.23
0.7	2.77
0.8	3.35

Because of the air resistance, the ball movement differs from the movement of a material point with the same mass, M = 0.000254 kg. To estimate this difference, we have to calculate the ball velocity at moment t = 0.8 s and compare it with the corresponding velocity of the material point.

For solving this task, we use the laws of mechanics as follows:

- according to the work-energy theorem, the change in kinetic energy of the ball is equal to the work done by the forces acting on the ball, i.e., by the gravitational and air resistance forces;
- according to Newton’s second law, the resultant of the forces, acting on the ball, is equal to the product of its mass, M, and acceleration, $\frac{d^2y}{dt^2}$, which is a function of the y coordinate.

Thus, the change in kinetic energy of the ball (which will be calculated by a program) is equal to

$$W = M \int_a^b y_2(y)dy, \tag{4.21}$$

where a = 0, b = 3.35,

$$y_2(y) = \frac{d^2y}{dt^2}. \tag{4.22}$$

Let us consider the following program for calculating the change in kinetic energy of the ball.

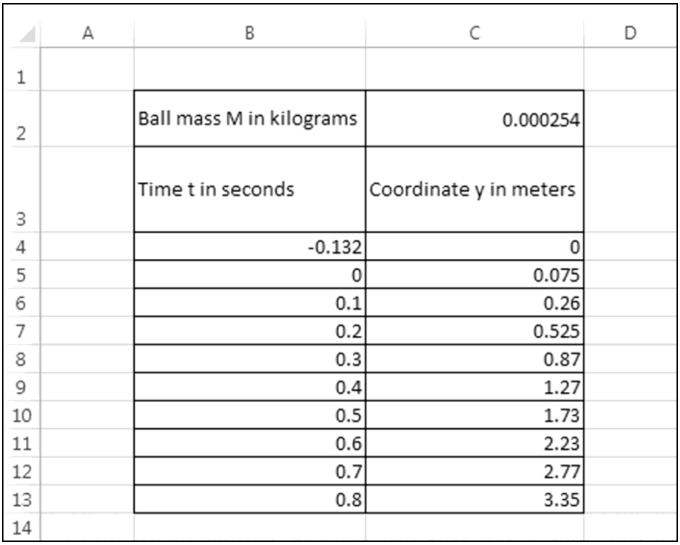
Listing 4.8

```
Sub main()
    Dim T() As Double
    Dim Y() As Double
    Dim Y2() As Double
    Dim MOM() As Double
    Dim m As Integer
    Dim i As Integer
    Dim W As Double
    m = Selection.Rows.Count         'quantity of rows
    ReDim T(3 To m)
    ReDim Y(3 To m)
    ReDim Y2(3 To m)
    ReDim MOM(3 To m)
'Inputting dependence of coordinate Y versus time T:
    For i = 3 To m
        T(i) = Selection.Cells(i, 1)
        Y(i) = Selection.Cells(i, 2)
    Next i
'Calculating spline moments, i.e., acceleration Y2:
1:  Call mos(3, m, T, Y, -2, 0, -2, 0, Y2)
'Calculating and outputting change in kinetic energy:
2:  Call mos(3, m, Y, Y2, -2, 0, -2, 0, MOM)
3:  W = Selection.Cells(1, 2) * ios(3, m, Y, Y2, MOM)
4:  MsgBox "W =" & Str(Round(W, 6)) & " J"
End Sub
```

The source data for this program are the values given in table Listing 4.7 (Fig. 4.5). We have to select this Excel table before running the program.

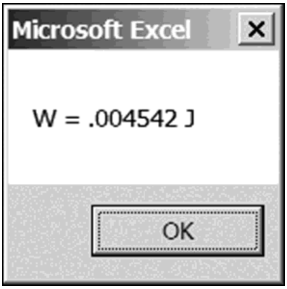
The program contains two calls of the `mos` subroutine, operators `1` and `2`. In these calls, we use boundary conditions (4.17) and (4.18) for the constancy of the spline moments at the left and right ends of segment $[a, b]$: $M_3 = M_4$ and

$M_m = M_{m-1}$ (m = 12). Operator `1` calculates the $y_2(y)$ function values according to formula (4.22). Operator `2` calculates array `MOM` of the spline moments, which are used for integrating the $y_2(y)$ function over $[a, b]$. Operator `3`, containing the call of the `ios` function, calculates the change in kinetic energy of the ball according to formula (4.21). When executing operator `4`, the calculated value is rounded up to six decimal places and displayed in the standard window (Fig. 4.6). To finish the program execution, we must click on the *OK* button in this window.



	A	B	C	D
1				
2		Ball mass M in kilograms	0.000254	
3		Time t in seconds	Coordinate y in meters	
4		-0.132	0	
5		0	0.075	
6		0.1	0.26	
7		0.2	0.525	
8		0.3	0.87	
9		0.4	1.27	
10		0.5	1.73	
11		0.6	2.23	
12		0.7	2.77	
13		0.8	3.35	
14				

Fig. 4.5. The Excel table containing the experimental data



W = .004542 J

Fig. 4.6. The execution result

According to the program execution result (Fig. 4.6), the change in kinetic energy of the ball is equal to

$$W = 0.004542\ \mathrm{J}\,.$$

Because the material point's acceleration is equal to the free fall acceleration, g = 0.981 m / s^2, the change in its kinetic energy on segment $[a, b]$ is equal to the following value:

$$Mg(b-a) = 0.008347\ \mathrm{J}\,.$$

Because of the zero value of the ball velocity at moment t = -0.132 s, the ball's kinetic energy at t = 0.8 s is equal to W:

$$W = \frac{M\,V^2}{2} = 0.004542\ \mathrm{J}\,.$$

This gives the following value of the ball velocity at moment t = 0.8 s:

$$V = \sqrt{\frac{2W}{M}} = 5.980\ \mathrm{m/s}\,.$$

The corresponding value for the material point is

$$V = \sqrt{\frac{2Mg(b-a)}{M}} = 8.107\ \mathrm{m/s}\ .$$

Thus, the air resistance plays an important role in the fall of the plastic foam ball.

To consolidate the material of this and the previous sections, ***we advise the reader*** to write a program for calculating the average value of function $f(x)$ on segment $[a, b]$. Formula

$$f_{average} = \frac{1}{b-a}\int_a^b f(x)dx$$

and the cubic spline on uniform grid $a = x_0 < x_1 < x_2 < ... < x_{n-2} < x_{n-1} < x_n = b$ must be used. The $f(x)$ function is from Appendix 4; segment $[a, b]$ is this function's domain.

4.5. Iterative methods for solving the nonlinear algebraic equation

Let nonlinear function $f(x)$ and segment $[u, v]$ be given with the following properties:

- function $f(x)$ is continuous and monotonous on $[u, v]$;
- the function values on the left and right boundaries of segment $[u, v]$ have different signs.

We will consider nonlinear algebraic equation

$$f(x) = 0. \tag{4.23}$$

It is obvious that this equation has one and only one solution on $[u, v]$. We have to find this solution.

A segment containing a unique solution of equation (4.23), for example $[u, v]$, is called the uncertainty segment.

Equation (4.23) can be solved by using the Solver add-in for Excel. Such usage of Solver will be considered on an example of

$$f(x) = x - \cos\, x - 1.5. \tag{4.24}$$

For this function, segment [0.5, 2.5] is the uncertainty segment (this is easily verified).

We enter an initial approximation of the solution, for example 2, into cell G1. Into cell F1, we enter formula

```
=G1-COS(G1)-1.5
```

corresponding to mathematical formula (4.24). Fig. 4.7 shows the worksheet after clicking on the tick button of the Excel formula bar.

Let us fulfill the following operations:

1) *Data* > *Solver* in area *Analysis*;

2) in the *Solver Parameters* window opened, enter $F$1 into text box *Set Objective*;

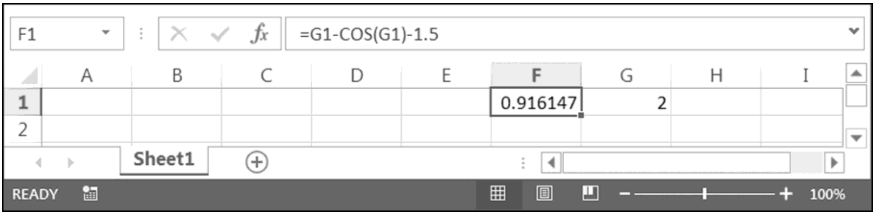
3) turn on option *Value Of*, and put zero into the corresponding text box;

4) enter $G$1 into text box *By Changing Variable Cells*;

5) enter *GRG Nonlinear* into box *Select a Solving Method* by means of the drop-down list (Fig. 4.8);

6) click on the *Solve* button;

7) in the *Solver Results* window opened (Fig. 4.9), click on *OK* to finish solving equation (4.23), (4.24).

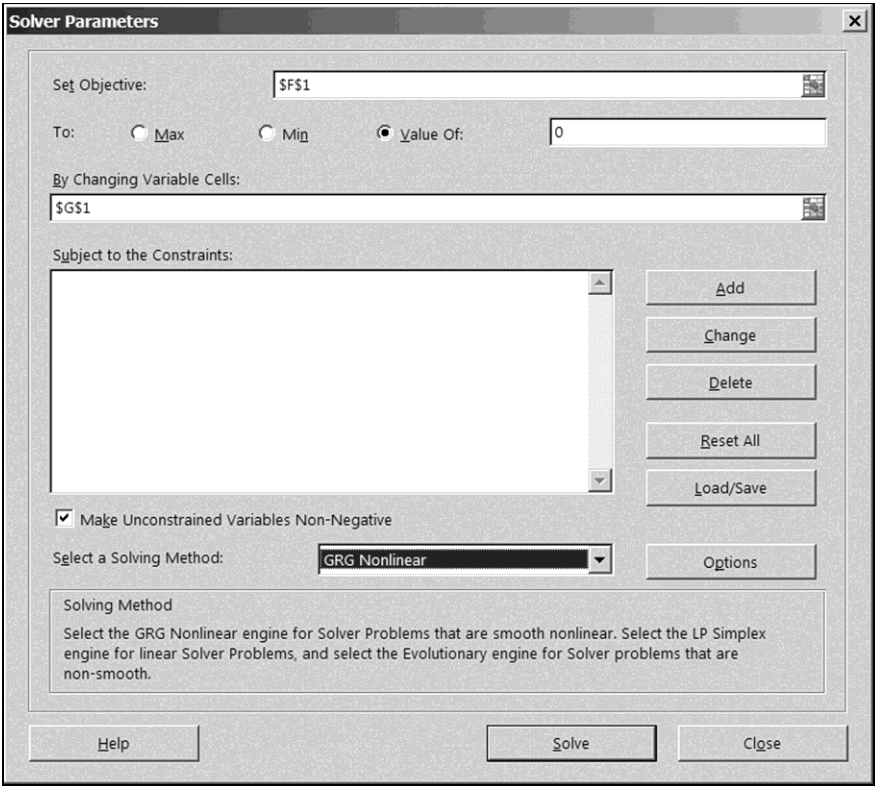


F1

=G1-COS(G1)-1.5

	A	B	C	D	E	F	G	H	I
1						0.916147	2		
2									

Fig. 4.7. The Excel worksheet before start of solving equation (4.23), (4.24)



Solver Parameters

Set Objective: $F$1

To: ○ Max ○ Min ● Value Of: 0

By Changing Variable Cells:
$G$1

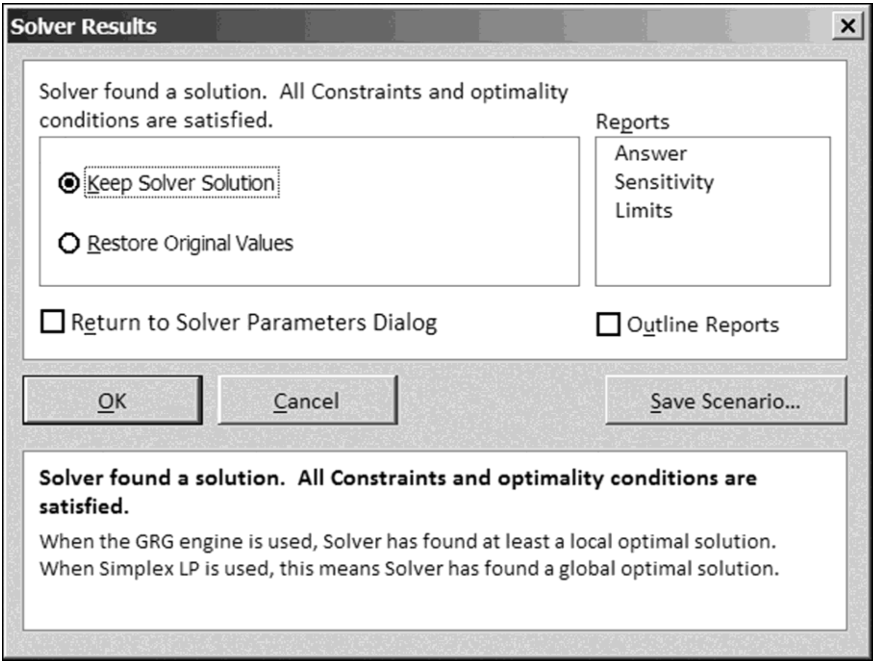
Subject to the Constraints:

☑ Make Unconstrained Variables Non-Negative

Select a Solving Method: GRG Nonlinear

Solving Method
Select the GRG Nonlinear engine for Solver Problems that are smooth nonlinear. Select the LP Simplex engine for linear Solver Problems, and select the Evolutionary engine for Solver problems that are non-smooth.

Fig. 4.8. The *Solver Parameters* window before start of solving the equation



Solver Results

Solver found a solution. All Constraints and optimality conditions are satisfied.

● Keep Solver Solution

○ Restore Original Values

□ Return to Solver Parameters Dialog

□ Outline Reports

Solver found a solution. All Constraints and optimality conditions are satisfied.

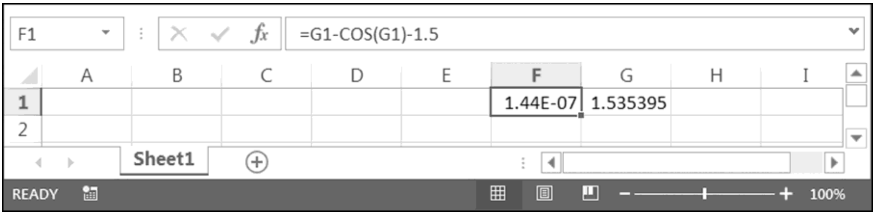
When the GRG engine is used, Solver has found at least a local optimal solution. When Simplex LP is used, this means Solver has found a global optimal solution.

Fig. 4.9

The following solution results are given in Fig. 4.10:

- cell G1 contains the result of solving equation (4.23), (4.24): $x^* = 1.535395$;
- cell F1 contains 1.44E-07.

The last value, $f(1.535395) = 1.44 \cdot 10^{-7}$, is the so-called residual right-hand side of equation (4.23), (4.24). It characterizes the accuracy of solving this equation.



F1 =G1-COS(G1)-1.5

	A	B	C	D	E	F	G	H	I
1						1.44E-07	1.535395		
2									

Fig. 4.10. The Excel worksheet upon termination of solving the equation

If the accuracy of the solution is unsatisfactory, we must:

1) open the *Options* window by clicking on the *Options* button in the *Solver Parameters* window depicted in Fig. 4.8;

2) change settings in the *Options* window;

3) click on the *OK* button for returning to the *Solver Parameters* window.

The limitation of the Solver add-in is obvious: it “must know” how to calculate the value of $f(x)$ at an arbitrary value of *x*. Therefore, we cannot use this procedure if $f(x)$ is a tabular (grid) function.

Below, we will develop a program for solving equation (4.23), in which $f(x)$ is a tabular function. Let us consider ***the bisection method***, which will be used in this program.

Let f_u and f_v be the values of $f(x)$ on the corresponding boundaries of segment $[u,v]$, and these values have different signs. The bisection method includes the following steps.

1. The midpoint of segment $[u,v]$ and the function value in this midpoint are calculated according to formulas

$$w=(u+v)/2 \tag{4.25}$$

and $f_w = f(w)$.

2. If the signs of f_w and f_u are the same, the left boundary of $[u,v]$ is shifted to the right: assignments $u=w$ and $f_u = f_w$ are performed. Further, the 4th item is fulfilled.

3. If the signs of f_w and f_v are the same, the right boundary of $[u,v]$ is shifted to the left: assignments $v=w$ and $f_v = f_w$ are performed.

4. The previous three items are repeated (that is, the iteration is repeated) until condition

$$v-u<\varepsilon \tag{4.26}$$

is satisfied, where ε is a given positive constant.

5. When condition (4.26) is satisfied, the iterative process of solving equation (4.23) terminates. The *w* value is considered as the solution of this equation.

If the residual right-hand side of equation (4.23), $f_w = f(w)$, is not small enough, we have to reduce the value of ε in condition (4.26).

If the uncertainty segment is not known, it can be found by means of the step-by-step movement along the *x* axis in the direction of decrease or increase of the $f(x)$ function as long as f_u and f_v have identical signs, i.e., until satisfaction of condition $f_u f_v < 0$. In Section 1.13, we already talked about the movement along the *x* axis when considering various cycles. In Section 6.2, the step-by-step movement will be used to find the segment that contains the minimum point of a nonlinear function of one variable.

The limitation of the bisection method is the same as for the Solver add-in: the program realization of the method “must know” how to calculate the $f(x)$ value at an arbitrary value of *x*. Therefore, the program realization of the bisection method for tabular function $f(x)$ will use the spline interpolation to determine the function value at any *x* value, i.e., for the function continuation.

We will solve the following task by the bisection method.

Time moments *t* (further, *x*) and corresponding values of coordinate *y* of the vertically falling plastic foam ball are given in table Listing 4.7 from the previous section. We are interested in the *T* moment of time when the ball has a given coordinate, for example $Y = 1$ m.

The desired value of *T* is the solution of equation (4.23), where

$$f(x) = y(x) - Y\,. \tag{4.27}$$

Because $y(x)$ is a grid function (defined by table Listing 4.7), $f(x)$ determined by (4.27) is a grid function too.

According to table Listing 4.7:

- at $u = 0.3$ and $v = 0.4$, segment $[u, v]$ is the uncertainty segment;
- $f_u = y(0.3) - 1 = -0.13$, $f_v = y(0.4) - 1 = 0.27$.

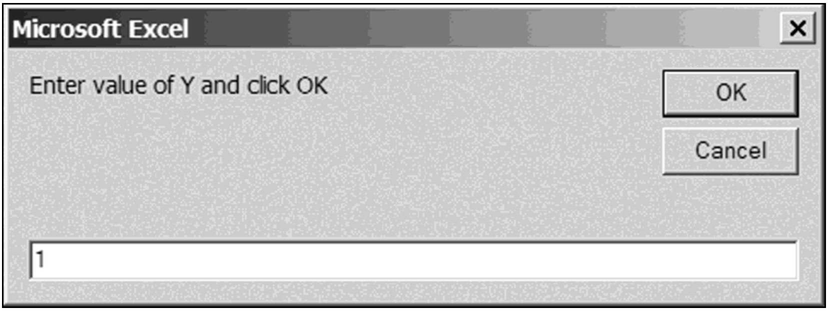
To solve equation (4.23), (4.27) by the bisection method, we will use the cubic spline to determine the function value for any *x* value (for example, $x = 0.35$), i.e., for making the $f(x)$ function continuous.

The following code is intended for determining moment *T* when the ball has a given *Y* coordinate.

Listing 4.9

```
Dim ns As Long                              'counter of calls
Dim m As Integer
Dim X() As Double
Dim F() As Double
Dim MOM() As Double

Sub main()
    Dim s As String, Y As Double
    Dim i As Integer
    Dim u As Double, fu As Double
    Dim v As Double, fv As Double
    Dim w As Double, fw As Double
    Dim epsilon As Double
1:  s = InputBox("Enter value of Y and click OK")
2:  Y = Val(s)
    m = Selection.Rows.Count         'quantity of rows
    ReDim X(3 To m)
    ReDim F(3 To m)
    ReDim MOM(3 To m)
    For i = 3 To m
        X(i) = Selection.Cells(i, 1)
        F(i) = Selection.Cells(i, 2) - Y
        If F(i) = 0 Then
3:          MsgBox "T =" & Str(Round(X(i), 6))
            End
        End If
    Next i
    If F(3) * F(m) > 0 Then
        MsgBox "On boundaries of segment [" & _
        Cstr(X(3)) & ", " & Cstr(X(m)) & "], " & _
        vbCrLf & "function f(x) has identical signs"
        End
    End If
'Calculating spline moments:
    Call mos(3, m, X, F, -2, 0, -2, 0, MOM)
'Searching uncertainty segment:
    For i = 4 To m
4:      If F(i - 1) * F(i) < 0 Then Exit For
    Next i
5:  epsilon = (X(i) - X(i - 1)) * 1E-6
'Solving equation:
    u = X(i - 1): fu = F(i - 1)
    v = X(i): fv = F(i)
    ns = 0
6:  Call segment(u, fu, v, fv, w, fw, u, fu, v, fv)
7:  If v - u >= epsilon GoTo 6
8:  MsgBox "T =" & Str(Round(w, 6)) & vbCrLf & _
        CStr(ns) & " iterations"
End Sub

Sub segment(ByVal u As Double, ByVal fu As Double, _
    ByVal v As Double, ByVal fv As Double, _
    ByRef w As Double, ByRef fw As Double, _
    ByRef uu As Double, ByRef fuu As Double, _
    ByRef vv As Double, ByRef fvv As Double)
    ns = ns + 1
    If Sgn(fu) = 0 Then
        uu = u: fuu = fu
        vv = u: fvv = fu
        Exit Sub
    End If
    If Sgn(fv) = 0 Then
        uu = v: fuu = fv
        vv = v: fvv = fv
        Exit Sub
    End If
9:  w = (u + v) / 2                'bisection method
    Call si(3, m, X, F, MOM, w, fw)
    If Sgn(fw) = Sgn(fu) Then
        uu = w: fuu = fw
        Exit Sub
    End If
    If Sgn(fw) = Sgn(fv) Then
        vv = w: fvv = fw
    End If
End Sub
```

The source data are given in table Listing 4.7 (Fig. 4.5). We must select this Excel table before the code execution. A value of *Y* must be entered into the text box of the standard window (see operators `1` and `2` and Fig. 4.11). Uncertainty segment $[x_{i-1}, x_i]$ is determined by means of operator `4`.


Enter value of Y and click OK

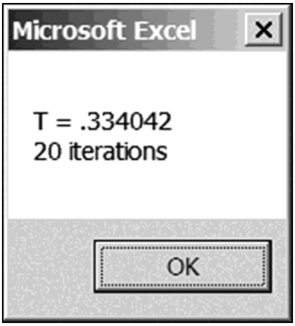
1

Fig. 4.11. The window with the coordinate of the falling plastic foam ball

Further, equation (4.23), (4.27) is solved by the bisection method (see operators `6` and `7`). In condition (4.26) for finishing the iterative process, the value of ε is equal to a millionth part of the initial uncertainty segment's length (see operator `5`).

In the iterative process, the `segment` subroutine is used for reducing the uncertainty segment length (see operator `6`). This subroutine is declared below the `main` program.

When executing operator `8`, the solution of equation (4.23), (4.27) is rounded up to six decimal places and displayed in the standard window (Fig. 4.12): T = 0.334042. The number of iterations is also displayed in this window; it equals 20. We have to click on the *OK* button for finishing the execution of code Listing 4.9.



```
T = .334042
20 iterations
```

Fig. 4.12. The window for the bisection method with the calculated moment of time and the number of iterations

Let us use ***the secant method*** (instead of the bisection method) in the iterative process of solving equation (4.23), (4.27). For that, formula (4.25) must be replaced with the following:

$$w = v - \frac{(v-u)\,f_v}{f_v - f_u}\,. \tag{4.28}$$

For obtaining the last formula, let us consider the geometric interpretation of the secant method depicted in Fig. 4.13. According to this figure, we can determine a new position ($x = w$) of the uncertainty segment boundary (left or right) as follows:

1) run the secant line through points (u, f_u) and (v, f_v);
2) denote the coordinate of the crosspoint of this line with the x axis by w.

According to Fig. 4.13, the slope of the secant line, passing through points (u, f_u) and (v, f_v), is equal to

$$\frac{-f_u}{w-u}=\frac{f_v}{v-w}.$$

This ratio leads to formula (4.28).

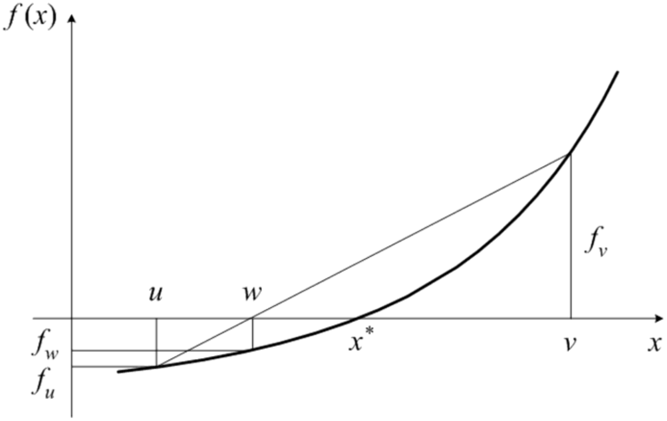


Fig. 4.13. The geometric interpretation of the secant method

For replacing the iterative method in code Listing 4.9, we have to replace operator `9` with the following:

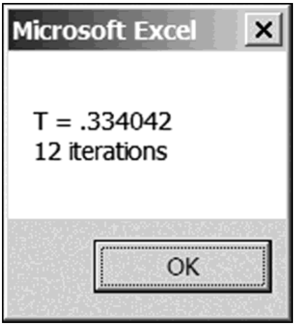
```
9:  w = v -(v - u) * fv / (fv - fu)      'secant method
```

Fig. 4.14 shows the result of executing the new version of code Listing 4.9 for $Y = 1$. As we see, the number of iterations is reduced to 12 (from 20).

Let us consider the question of the convergence of the above iterative processes defined by formulas (4.25) and (4.28). Starting with the bisection method, we use the following designations:

- $\varepsilon^{j+1} = w - x^*$;
- ε^j is $u - x^*$ or $v - x^*$ to get the same sign of ε^j as the sign of ε^{j+1} defined above.

Here, j is the iteration number, x^* is the exact solution of the equation, that is, $f(x^*) = 0$.



T = .334042
12 iterations

Fig. 4.14. The window for the secant method with the calculated moment of time and the number of iterations

The x^* location near any boundary of the initial uncertainty segment is the worst situation for the convergence. In this case,

$$\varepsilon^{j+1} = C\varepsilon^{j}, \tag{4.29}$$

where C is a constant close to 0.5 on the left. At an arbitrary location of x^*, double inequality $0 \le C < 0.5$ is true for the bisection method.

Formula (4.29) allows us to estimate the rate of the convergence of the iterative process. According to this formula, quantity ε^{j+1} is proportional to the first power of quantity ε^{j}, that is, the bisection method's iterative process converges linearly.

Similar consideration of the secant method also gives formula (4.29) for estimating the rate of the convergence of the corresponding iterative process, but with inequality $0 \le C < 1$.

In Section 5.5, we will consider another version of the secant method whose iterative process has quadratic convergence according to formula (5.26) or, in the simplest form,

$$\varepsilon^{j+1} = C\varepsilon^{j}\varepsilon^{j-1}, \tag{4.30}$$

where C is a constant, ε^{j+1}, ε^{j}, ε^{j-1} are small quantities, j is the iteration number.

We advise the reader to write a program for solving equation $f(x) = 0$ on segment $[a, b]$ by the bisection and secant methods. In this equation, $f(x)$ is a function from Appendix 4; segment $[a, b]$ is this function's domain. The user-defined form with the CheckBox element (for choosing the method) must be the user interface of the program. Segment $[a, b]$ must be the initial uncertainty segment.

In the next section, we will develop a noniterative method for solving the nonlinear algebraic equation with grid function $f(x)$. In Section 5.5, we will return to iterative methods for solving the nonlinear algebraic equation.

4.6. Noniterative method for solving the nonlinear algebraic equation

In the previous section, we considered two iterative methods for solving equation (4.23), (4.27), at that, we used the spline interpolation of grid function (4.27) in the program realization of these methods. However, equation (4.23), (4.27) can be solved by means of the cubic spline construction without any iterations.

The fact is that, having calculated the moments of the spline, which corresponds to grid function $f(x)$ on segment [-0.132, 0.8], we obtain the representation of $f(x)$ in the form of third-degree polynomial $S(x)$ on the uncertainty segment, for example $[u, v]$ = [0.3, 0.4]. To obtain the value of *T*, it is enough to solve cubic equation $S(x) = 0$, which can be written in canonical form

$$ax^3 + bx^2 + cx + d = 0. \tag{4.31}$$

According to handbook [3], equation (4.23), (4.27) is solved in the following three stages if $a \neq 0$.

1. By substituting

$$x = z - \frac{b}{3a}, \tag{4.32}$$

equation (4.31) is transformed to

$$z^3 + 3p\,z + 2q = 0, \tag{4.33}$$

where $3p = \dfrac{3ac - b^2}{3a^2}$, $2q = \dfrac{2b^3}{27a^3} - \dfrac{bc}{3a^2} + \dfrac{d}{a}$.

2. The real roots of equation (4.33) are calculated, one or three (two of which may coincide), and then the corresponding roots of equation (4.31) are calculated.

For solving equation (4.33), we use the method represented in the table of Appendix 5. For subsequent calculations of the real roots of equation (4.31), we use formula (4.32).

3. The unique solution of equation (4.23), (4.27) is determined.

The declaration of the `table` subroutine, intended for calculating the real roots of equation (4.31), follows:

Listing 4.10

```
Sub table(ByVal a, ByVal b, ByVal c, ByVal d, _
    ByRef x() As Double)
    Dim b3a As Double, p As Double, q As Double
    Dim r As Double, gamma As Double, phi As Double
    Const beta = 1 / 3, pi = 3.141592654
    x(1) = 1E+308
    x(2) = 1E+308
    x(3) = 1E+308
    If 27 * a ^ 3 = 0 Then Exit Sub         'if a = 0
    b3a = b / (3 * a)
    p = (3 * a * c - b ^ 2) / (3 * a ^ 2) / 3
    q = (2 * b ^ 3 / (27 * a ^ 3) - b * c / _
        (3 * a ^ 2) + d / a) / 2
    r = Sgn(q) * Sqr(Abs(p))
    If r ^ 3 = 0 Then                       'if p = 0
        x(1) = -Sgn(q) * Abs(2 * q) ^ beta - b3a
        Exit Sub
    End If
    If q = 0 Then                           'if q = 0
        x(1) = - b3a
        If p < 0 Then
            x(2) = Sqr(-3 * p) - b3a
            x(3) = -Sqr(-3 * p) - b3a
        End If
        Exit Sub
    End If
    gamma = q / r ^ 3                       'gamma > 0
    If p < 0 Then
        If gamma <= 1 Then
            phi = Atn(Sqr(1 - gamma ^ 2) / gamma)
            x(1) = -2 * r * Cos(phi / 3) - b3a
            x(2) = 2 * r * Cos((pi - phi) / 3) - b3a
            x(3) = 2 * r * Cos((pi + phi) / 3) - b3a
        Else
            phi = Log(gamma + Sqr(gamma ^ 2 - 1))
            x(1) = -2 * r * _
            (Exp(phi / 3) + Exp(-phi / 3)) / 2 - b3a
        End If
    Else
        phi = Log(gamma + Sqr(gamma ^ 2 + 1))
        x(1) = -2 * r * _
        (Exp(phi / 3) - Exp(-phi / 3)) / 2 - b3a
    End If
End Sub
```

The subroutine name (`table`) is due to the fact that the subroutine algorithm is based on the table in Appendix 5. Let us enter declaration Listing 4.10 into Module11 of the BookNM workbook.

The `table` subroutine parameters have the following sense:

- `a`, `b`, `c`, `d` are coefficients *a*, *b* and *c* and constant term *d* of cubic equation (4.31);
- `x` is an array for real solutions (one or three; free elements of the `x` array are assumed to be equal to 10^{308}).

The following program is intended for determining the *T* moment when the falling plastic foam ball (from the two previous sections) has a given *Y* coordinate.

Listing 4.11

```
Sub main()
    Dim s As String, Y As Double
    Dim m As Integer
    Dim X() As Double
    Dim F() As Double
    Dim MOM() As Double
    Dim i As Integer, h As Double
    Dim a1 As Double, a2 As Double
    Dim b1 As Double, b2 As Double
    Dim a As Double, b As Double
    Dim c As Double, d As Double
    Dim T As Double
    Dim xxx(1 To 3) As Double, k As Integer
1:  s = InputBox("Enter value of Y and click OK")
2:  Y = Val(s)
    m = Selection.Rows.Count         'quantity of rows
    ReDim X(3 To m)
    ReDim F(3 To m)
    ReDim MOM(3 To m)
    For i = 3 To m
        X(i) = Selection.Cells(i, 1)
        F(i) = Selection.Cells(i, 2) - Y
        If F(i) = 0 Then
3:          MsgBox "T =" & Str(Round(X(i), 6))
            End
        End If
    Next i
    If F(3) * F(m) > 0 Then
        MsgBox "On boundaries of segment [" & _
        CStr(X(3)) & ", " & CStr(X(m)) & "], " & _
        vbCrLf & "function f(x) has identical signs"
        End
    End If
'Calculating spline moments:
    Call mos(3, m, X, F, -2, 0, -2, 0, MOM)
'Searching uncertainty segment:
    For i = 4 To m
4:      If F(i - 1) * F(i) < 0 Then Exit For
    Next i
'Forming cubic equation:
    h = X(i) - X(i - 1)
    a1 = MOM(i - 1) / (6 * h)
    a2 = MOM(i) / (6 * h)
    b1 = (F(i - 1) - MOM(i - 1) * h ^ 2 / 6) / h
    b2 = (F(i) - MOM(i) * h ^ 2 / 6) / h
5:  a = a2 - a1
6:  b = 3 * a1 * X(i) - 3 * a2 * X(i - 1)
7:  c = 3 * a2 * X(i - 1) ^ 2 - 3 * a1 * X(i) ^ 2 + _
    b2 - b1
8:  d = a1 * X(i) ^ 3 - a2 * X(i - 1) ^ 3 + _
    b1 * X(i) - b2 * X(i - 1)
'Solving linear equation:
    If 27 * a ^ 3 = 0 And 2 * b = 0 Then
9:      T = -d / c
10:     MsgBox "T =" & Str(Round(T, 6)) & _
        vbCrLf & "- root of linear equation"
        End
    End If
'Solving quadratic equation:
    If 27 * a ^ 3 = 0 Then
11:     T = (-c + Sqr(c ^ 2 - 4 * b * d)) / (2 * b)
        If X(i - 1) <= T And T <= X(i) Then
12:         MsgBox "T =" & Str(Round(T, 6)) & _
            vbCrLf & "- root of quadratic equation"
            End
        End If
13:     T = (-c - Sqr(c ^ 2 - 4 * b * d)) / (2 * b)
        If X(i - 1) <= T And T <= X(i) Then
14:         MsgBox "T =" & Str(Round(T, 6)) & _
            vbCrLf & "- root of quadratic equation"
            End
        End If
    End If
'Solving cubic or linear equation:
15: Call table(a, b, c, d, xxx)
    If xxx(2) = 1E+308 Then
16:     MsgBox "T =" & Str(Round(xxx(1), 6)) & _
        vbCrLf & "- root of cubic equation"
        End
    End If
    If X(i - 1) <= xxx(1) And xxx(1) <= X(i) _
    And X(i - 1) <= xxx(2) And xxx(2) <= X(i) _
    And X(i - 1) <= xxx(3) And xxx(3) <= X(i) Then
17:     T = (F(i) * X(i - 1) - F(i - 1) * X(i)) / _
        (F(i) - F(i - 1))
18:     MsgBox "T =" & Str(Round(T, 6)) & _
        vbCrLf & "- root of linear equation"
        End
    End If
    For k = 1 To 3
        If X(i - 1) <= xxx(k) And xxx(k) <= X(i) Then
19:         MsgBox "T =" & Str(Round(xxx(k), 6)) & _
            vbCrLf & "- root of cubic equation"
            End
        End If
    Next k
End Sub
```

The source data are given in table Listing 4.7 (Fig. 4.5). We must select this Excel table before the program execution. The value of *Y* must be entered into the text box of the standard window (see operators 1 and 2 and Fig. 4.11). Uncertainty segment $[x_{i-1}, x_i]$ is determined by means of operator 4.

Operators 5 — 8 calculate the coefficients and constant term of cubic equation (4.31) according to the following formulas:

$$a = a_2 - a_1,$$

$$b = 3a_1 x_i - 3a_2 x_{i-1},$$

$$c = 3a_2 x_{i-1}^2 - 3a_1 x_i^2 + b_2 - b_1,$$

$$d = a_1 x_i^3 - a_2 x_{i-1}^3 + b_1 x_i - b_2 x_{i-1},$$

where

$$a_1 = \frac{M_{i-1}}{6h_i}, \quad a_2 = \frac{M_i}{6h_i},$$

$$b_1 = \left(f_{i-1} - \frac{M_{i-1} h_i^2}{6} \right) \frac{1}{h_i}, \quad b_2 = \left(f_i - \frac{M_i h_i^2}{6} \right) \frac{1}{h_i}.$$

For obtaining these formulas, we must transform expression (4.4) to the following form:

$$S(x) = ax^3 + bx^2 + cx + d.$$

After determining *a*, *b*, *c* and *d*, the approximate solution of equation (4.23), (4.27) is calculated by solving one of three algebraic equations — a linear, quadratic or cubic equation.

1. The linear equation is solved if the resulting values of coefficients *a* and *b* are equal to zero. Operator `9` calculates a root of equation $cx + d = 0$ according to formula $x = -d/c$.

2. The quadratic equation is solved at the zero value of *a* ($b \neq 0$). Two roots of equation $bx^2 + cx + d = 0$ are calculated by operators `11` and `13` according to the well known formula [3],

$$x_{\pm} = \left(-c \pm \sqrt{c^2 - 4bd} \right) / (2b).$$

The solution of equation (4.23), (4.27) is the root (x_+ or x_-) belonging to uncertainty segment $[x_{i-1}, x_i]$.

3. The cubic equation is solved when $a \neq 0$. One or three real roots of equation $ax^3 + bx^2 + cx + d = 0$ are the result of calling the `table` subroutine (operator `15`).

In the ordinary situation, uncertainty segment $[x_{i-1}, x_i]$ contains only one out of one or three real roots of equation $ax^3 + bx^2 + cx + d = 0$, and this root is the solution of equation (4.23), (4.27). However, an extraordinary situation is possible when the number of real roots of equation $ax^3 + bx^2 + cx + d = 0$ is equal to three, and all these roots belong to uncertainty segment $[x_{i-1}, x_i]$. In this case, operator `17` calculates the solution of equation (4.23), (4.27) according to the following formula:

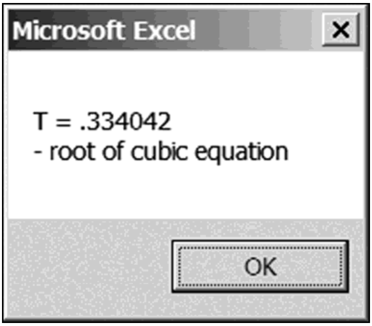
$$x = \frac{f_i x_{i-1} - f_{i-1} x_i}{f_i - f_{i-1}}.$$

This formula is obtained by solving equation $L(x) = 0$, in which

$$L(x) = f_{i-1} \frac{x_i - x}{x_i - x_{i-1}} + f_i \frac{x - x_{i-1}}{x_i - x_{i-1}} \tag{4.34}$$

is the linear function that equals f_{i-1} and f_i on the corresponding boundaries of segment $[x_{i-1}, x_i]$.

As a result of executing operator `3`, `10`, `12`, `14`, `16`, `18` or `19`, the solution of equation (4.23), (4.27) is rounded up to six decimal places and displayed in the standard window (Fig. 4.15). After clicking on the *OK* button, the program execution is terminated.



T = .334042
- root of cubic equation

Fig. 4.15. The window for the noniterative method
with the calculated moment of time and
the type of the solved algebraic equation

We advise the reader to write a program, similar to Listing 4.11, for solving equation $f(x) = 0$ on segment $[a, b]$ by the noniterative method. In this equation, $f(x)$ is a function from Appendix 4; segment $[a, b]$ is this function's domain. Uniform grid $a = x_0 < x_1 < x_2 < ... < x_{n-2} < x_{n-1} < x_n = b$ must be used.

4.7. Calculating the charge storage capacity

Having studied the cubic spline construction, we will continue the modeling of the silicon photosensitive target (Section 3.13).

It was mentioned on p. 264 that semiconductor layer -6 µm ≤ x ≤ 6 µm, perpendicular to the x axis, is the potential well for signal electrons generated by light falling on the layer plane. In this regard, an important electrical parameter is the charge storage capacity of the target cell.

The charge storage capacity, Q_{max}, is the signal electrons' maximum charge (in absolute value), which can be localized in the considered layer of unit area (meter × meter).

The value of Q_{max} is calculated in the following five stages:

1) the moments of cubic spline $S(x)$, determined by the tabulated electric charge density,

$$f(x) = q\left\{ N_0 \exp\left[-\frac{qu(x)}{k_b T} \right] - N_A(x) \right\},$$

are calculated, where $u(x)$ is the solution of Shockley-Poisson equation (3.68), $a \le x \le 0$;

2) the coordinate, $x = e$, at which spline $S(x)$ changes its sign from negative (at $x < e$) to positive (at $x > e$), is calculated;

3) the second derivative, w_2, of spline $S(x)$ at point $x = e$ is calculated (w_2 is used in the next item when integrating the cubic spline over segment $e \le x \le 0$);

4) the positive charge, Q_1, localized in the semiconductor layer, $a \le x \le 0$, is calculated by integrating $S(x)$ over segment $e \le x \le 0$:

$$Q_1 = \int_e^0 S(x)\,dx; \tag{4.35}$$

5) the calculated value of Q_1 is multiplied by 2 because the right-hand side of the semiconductor layer ($x > 0$) exists in addition to the reviewed left-hand side of the layer ($x < 0$):

$$Q_{max} = 2Q_1 . \tag{4.36}$$

Let us consider the program for calculating the charge storage capacity, Q_{max}.

The source data table below is a part of the table represented in Fig. 3.15.

Listing 4.12

NA	u0	x	u
7.00E+20	0.00E+00	-1.00E-05	0.000E+00
7.00E+20	4.89E-02	-9.00E-06	1.320E-06
7.00E+20	1.91E-01	-8.00E-06	5.782E-05
7.00E+20	4.12E-01	-7.00E-06	2.529E-03
7.00E+20	6.91E-01	-6.00E-06	1.057E-01
7.00E+20	1.00E+00	-5.00E-06	1.272E+00
7.00E+20	1.31E+00	-4.00E-06	3.521E+00
7.00E+20	1.59E+00	-3.00E-06	6.851E+00
7.00E+20	1.81E+00	-2.00E-06	1.126E+01
7.00E+20	1.95E+00	-1.00E-06	1.676E+01
7.00E+20	1.98E+00	-6.00E-07	1.926E+01
7.00E+20	1.99E+00	-4.00E-07	2.057E+01
7.00E+20	2.00E+00	-3.00E-07	2.124E+01
7.00E+20	2.00E+00	-2.00E-07	2.193E+01
-3.00E+22	2.00E+00	-1.00E-07	2.262E+01
-3.00E+22	2.00E+00	0.00E+00	2.286E+01

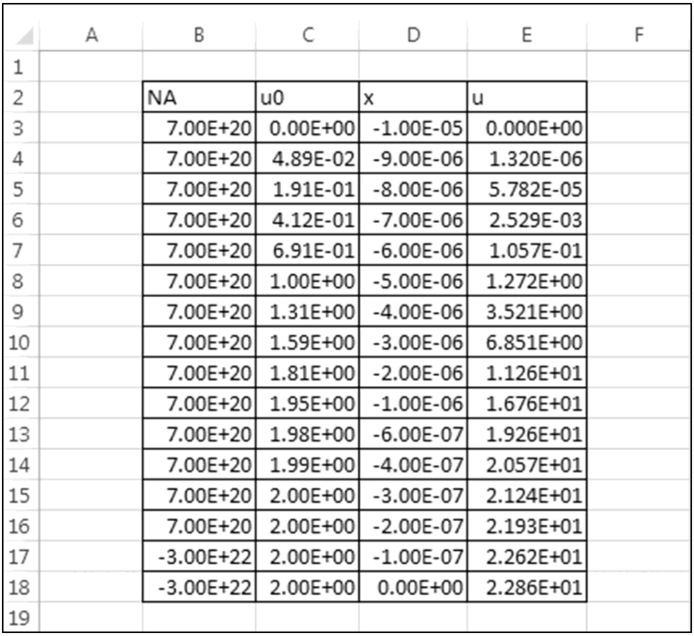
The program follows:

Listing 4.13

```
Sub main()
    Dim m As Integer
    Dim X() As Double
    Dim NA() As Double
    Dim U() As Double
    Dim F() As Double
    Dim MOM() As Double
    Const q = 1.6E-19
    Const kb = 1.38E-23
    Const T = 300
    Dim w As Double, w1 As Double, w2 As Double
    Dim i As Integer, h As Double
    Dim a1 As Double, a2 As Double
    Dim b1 As Double, b2 As Double
    Dim a As Double, b As Double
    Dim c As Double, d As Double
    Dim e As Double
    Dim xxx(1 To 3) As Double, k As Integer
    m = Selection.Rows.Count         'quantity of rows
    ReDim X(2 To m)
    ReDim NA(2 To m)
    ReDim U(2 To m)
    ReDim F(2 To m)
    ReDim MOM(2 To m)
    w = q / (kb * T)
    For i = 2 To m
        NA(i) = Selection.Cells(i, 1)
        X(i) = Selection.Cells(i, 3)
        U(i) = Selection.Cells(i, 4)
    Next i
    For i = 2 To m
        F(i) = q * (NA(2) * Exp(-w * U(i)) - NA(i))
    Next i
'Calculating spline moments:
    w = 6 * (F(3) - F(2)) / (X(3) - X(2)) ^ 2
    w1 = 6 * (F(m - 1) - F(m)) / (X(m) - X(m - 1)) ^ 2
0:  Call mos(2, m, X, F, 1, w, 1, w1, MOM)
'Searching uncertainty segment:
    For i = 3 To m
4:      If F(i - 1) * F(i) < 0 Then Exit For
    Next i
'Forming cubic equation:
    h = X(i) - X(i - 1)
    a1 = MOM(i - 1) / (6 * h)
    a2 = MOM(i) / (6 * h)
    b1 = (F(i - 1) - MOM(i - 1) * h ^ 2 / 6) / h
    b2 = (F(i) - MOM(i) * h ^ 2 / 6) / h
5:  a = a2 - a1
6:  b = 3 * a1 * X(i) - 3 * a2 * X(i - 1)
7:  c = 3 * a2 * X(i - 1) ^ 2 - 3 * a1 * X(i) ^ 2 + _
    b2 - b1
```

```
8:  d = a1 * X(i) ^ 3 - a2 * X(i - 1) ^ 3 + _
    b1 * X(i) - b2 * X(i - 1)
'Solving linear equation:
    If 27 * a ^ 3 = 0 And 2 * b = 0 Then
9:      e = -d / c
        GoTo 21
    End If
'Solving quadratic equation:
    If 27 * a ^ 3 = 0 Then
11:     e = (-c + Sqr(c ^ 2 - 4 * b * d)) / (2 * b)
        If X(i - 1) <= e And e <= X(i) Then GoTo 21
13:     e = (-c - Sqr(c ^ 2 - 4 * b * d)) / (2 * b)
        If X(i - 1) <= e And e <= X(i) Then GoTo 21
    End If
'Solving cubic or linear equation:
15: Call table(a, b, c, d, xxx)
    If xxx(2) = 1E+308 Then
16:     e = xxx(1)
        GoTo 21
    End If
    If X(i - 1) <= xxx(1) And xxx(1) <= X(i) _
    And X(i - 1) <= xxx(2) And xxx(2) <= X(i) _
    And X(i - 1) <= xxx(3) And xxx(3) <= X(i) Then
17:     e = (F(i) * X(i - 1) - F(i - 1) * X(i)) / _
        (F(i) - F(i - 1))
        GoTo 21
    End If
    For k = 1 To 3
        If X(i - 1) <= xxx(k) And xxx(k) <= X(i) Then
19:         e = xxx(k)
            GoTo 21
        End If
    Next k
'Calculating charge storage capacity:
21: Call si(2, m, X, F, MOM, e, w, w1, w2)
    X(i - 1) = e
    F(i - 1) = 0
    MOM(i - 1) = w2
22: w1 = ios(i - 1, m, X, F, MOM)    'calculating Q1
23: w = Round(2 * w1, 4)             'calculating Qmax
24: MsgBox "Qmax = " & CStr(w) & " C/m^2"
End Sub
```

The source data are the values given in table Listing 4.12 (Fig. 4.16). We must select this table before running the program.



NA	u0	x	u
7.00E+20	0.00E+00	-1.00E-05	0.000E+00
7.00E+20	4.89E-02	-9.00E-06	1.320E-06
7.00E+20	1.91E-01	-8.00E-06	5.782E-05
7.00E+20	4.12E-01	-7.00E-06	2.529E-03
7.00E+20	6.91E-01	-6.00E-06	1.057E-01
7.00E+20	1.00E+00	-5.00E-06	1.272E+00
7.00E+20	1.31E+00	-4.00E-06	3.521E+00
7.00E+20	1.59E+00	-3.00E-06	6.851E+00
7.00E+20	1.81E+00	-2.00E-06	1.126E+01
7.00E+20	1.95E+00	-1.00E-06	1.676E+01
7.00E+20	1.98E+00	-6.00E-07	1.926E+01
7.00E+20	1.99E+00	-4.00E-07	2.057E+01
7.00E+20	2.00E+00	-3.00E-07	2.124E+01
7.00E+20	2.00E+00	-2.00E-07	2.193E+01
-3.00E+22	2.00E+00	-1.00E-07	2.262E+01
-3.00E+22	2.00E+00	0.00E+00	2.286E+01

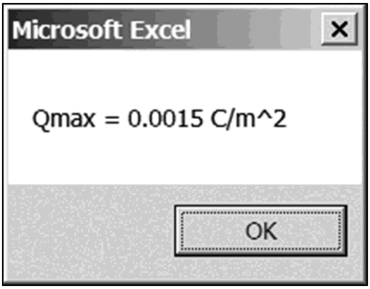
Fig. 4.16. The Excel table with the source data

The program contains the call of the `mos` subroutine (operator `0`) intended for calculating the spline moments. In this call, conditions (4.13) and (4.14) are used, corresponding to the zero value of the derivative on the left and right boundaries of segment $[a, 0]$: $f_2' = f'(a) = 0$, $f_m' = f'(0) = 0$ $(m = 17)$.

The value of `e` is the result of solving equation $S(x) = 0$ by the noniterative method given in the previous section. Regarding the part that solves this equation, program Listing 4.13 is similar to program Listing 4.11. The solution of the cubic equation is equal to $e = -1.967 \cdot 10^{-7}$ (we observed this value by using the program debugger of Visual Basic Environment).

Operator `21` calculates the w_2 value of the second derivative of the $S(x)$ spline at point $x = e$ by calling the `si` subroutine intended for the spline interpolation. Further, the w_2 value is used (as the moment) when integrating $S(x)$ over segment $[e, 0]$.

Operator `22` calculates charge Q_1 according to formula (4.35) by calling the `ios` function intended for integrating the cubic spline. Operator `23` calculates the charge storage capacity, Q_{max}, according to formula (4.36) and rounds the result up to four decimal places. When executing operator `24`, the rounded charge storage capacity is displayed in the standard window (Fig. 4.17).



Qmax = 0.0015 C/m^2

Fig. 4.17. The window with the program execution result

The calculated value of Q_{max} is of interest for designing both the photosensitive target and elements of extraction of signal electrons stored in the target's cells: the considered device belongs to the class of charge transfer devices.

4.8. Subroutine for automatic creation of graphs

Further, we will use the graph creation subroutine whose declaration given below must be put into Module12 of the BookNM workbook.

Listing 4.14

```
Sub graph(ByVal sb, ByVal se, ByVal sx, ByVal sy)
    Dim wbOldSelection As Range
    Set wbOldSelection = Selection
    Range(sb & ":" & se).Select
    Dim sn As String
    sn = ActiveSheet.Name
    Selection.NumberFormat = "0.00E+00"
    Charts.Add
    ActiveChart.ChartType = xlXYScatterSmoothNoMarkers
    ActiveChart.SetSourceData Source:= _
        Sheets(sn).Range(sb & ":" & se), PlotBy:= _
        xlColumns
    ActiveChart.Location Where:=xlLocationAsObject, _
        Name:=sn
    ActiveChart.Axes(xlValue).MajorGridlines.Select
    Selection.Delete
    ActiveChart.Legend.Select
    Selection.Delete
    With ActiveChart
        .Axes(xlCategory, xlPrimary).HasTitle = True
        .Axes(xlCategory, _
        xlPrimary).AxisTitle.Characters.Text = sx
        .Axes(xlValue, xlPrimary).HasTitle = True
        .Axes(xlValue, _
        xlPrimary).AxisTitle.Characters.Text = sy
    End With
    ActiveChart.Axes(xlCategory).AxisTitle.Select
    Selection.AutoScaleFont = True
    With Selection.Font
```

```
        .FontStyle = "regular"
        .Size = 12
    End With
    ActiveChart.Axes(xlValue).AxisTitle.Select
    Selection.AutoScaleFont = True
    With Selection.Font
        .FontStyle = "regular"
        .Size = 12
    End With
    wbOldSelection.Select
End Sub
```

The parameters of the graph subroutine are the following four strings:

- sb, se — the strings, which define the Excel range containing the argument and function values: sb is the address of the top left cell of the range; se is the address of the bottom right cell;
- sx, sy — the names of the horizontal and vertical axes, respectively.

In the parameter names:

- letter “s” corresponds to word “string”;
- “b” corresponds to word “beginning”;
- “e” corresponds to word “end”;
- “x” means the horizontal axis;
- “y” means the vertical axis.

The basis of the graph subroutine are operators 1 — 32 of program Listing 3.18. However, we see something new in the above subroutine, namely, the selection recovery (pp. 173 and 174):

- in the subroutine beginning, operator

```
Set wbOldSelection = Selection
```

assigns the selected range to the wbOldSelection variable of the Range type;

- in the subroutine end, operator

```
wbOldSelection.Select
```

selects the wbOldSelection range.

Because of these two operators, the source data table is selected in Fig. 6.37: the graph subroutine execution (see operator 13 in Listing 6.15) does not change the selection. This picture can be considered as an example of the subroutine work.

4.9. Cubic spline usage for solving the second-order linear differential equation

In Chapter 3, for solving the boundary value problem for the second-order linear differential equation, we considered a method based on the solution approximation by the second-degree polynomial (Section 3.14). Below, for solving the same problem, we will consider a method based on the solution approximation by the cubic spline.

As shown in Section 3.4, equation (3.6) can be written in form (3.30),

$$\frac{d^2U}{dx^2} + E(x)U = F(x), \tag{4.37}$$

where $E(x)$ and $F(x)$ are given functions, $U(x)$ is an unknown function. Conditions (3.7) and (3.8) on the boundaries of segment $[a, b]$ take form (3.33) and (3.36),

$$A_2 U(a) + A_1 U'(a) = A_3, \tag{4.38}$$

$$B_2 U(b) + B_1 U'(b) = B_3, \tag{4.39}$$

where A_1, A_2, A_3, B_1 , B_2 , B_3 are given parameters.

We will develop a cubic spline method for solving the formulated boundary value problem, (4.37) — (4.39), on the following grid familiar to us: $a = x_k < x_{k+1} < x_{k+2} < ... < x_{r-2} < x_{r-1} < x_r = b$.

Let U_k, U_{k+1}, U_{k+2}, ..., U_{r-2}, U_{r-1}, U_r be the solution values at points x_k, x_{k+1}, x_{k+2}, ..., x_{r-2}, x_{r-1}, x_r, respectively. The $S(x)$ cubic spline is considered whose graph passes through points (x_k, U_k), (x_{k+1}, U_{k+1}), $(x_{k+2}, U_{k+2}), \dots, (x_{r-2}, U_{r-2}), (x_{r-1}, U_{r-1}), (x_r, U_r)$.

Because the spline moments are the values of $S''(x)$ at the grid nodes, equation (4.37) gives the following expression for the moment at the *i*-th node:

$$M_i = F_i - E_i U_i, \tag{4.40}$$

where F_i and E_i are the values of functions $E(x)$ and $F(x)$ at the i-th node, $k \le i \le r$.

After replacing f_i with U_i in expression (4.10) for δ_i, we have

$$\delta_i = 6\frac{\dfrac{U_{i+1} - U_i}{h_{i+1}} - \dfrac{U_i - U_{i-1}}{h_i}}{h_i + h_{i+1}}. \tag{4.41}$$

By substituting expressions (4.40) and (4.41) into (4.9), we obtain the following equality:

$$\left[\alpha_i E_{i-1} + \frac{6}{h_i(h_i + h_{i+1})}\right]U_{i-1} + \left[2E_i - \frac{6}{h_{i+1}(h_i + h_{i+1})} - \frac{6}{h_i(h_i + h_{i+1})}\right]U_i$$

$$+\left[\gamma_i E_{i+1} + \frac{6}{h_{i+1}(h_i + h_{i+1})}\right]U_{i+1} = \alpha_i F_{i-1} + 2F_i + \gamma_i F_{i+1}.$$

Substituting expressions (4.10) for α_i and γ_i, we get

$$\left[\frac{h_i}{h_i + h_{i+1}}E_{i-1} + \frac{6}{h_i(h_i + h_{i+1})}\right]U_{i-1} +$$

$$+\left[2E_i - \frac{6}{h_{i+1}(h_i + h_{i+1})} - \frac{6}{h_i(h_i + h_{i+1})}\right]U_i +$$

$$+\left[\frac{h_{i+1}}{h_i + h_{i+1}}E_{i+1} + \frac{6}{h_{i+1}(h_i + h_{i+1})}\right]U_{i+1} =$$

$$= \frac{h_i}{h_i + h_{i+1}}F_{i-1} + 2F_i + \frac{h_{i+1}}{h_i + h_{i+1}}F_{i+1}.$$

Let us multiply both sides of the last equality by $h_i h_{i+1}/3$. The following linear algebraic equation is the result:

$$\alpha_i U_{i-1} + \beta_i U_i + \gamma_i U_{i+1} = \delta_i, \tag{4.42}$$

where U_{i-1}, U_i, U_{i+1} are the unknowns, $i = k+1, k+2, ..., r-2, r-1$,

$$\alpha_i = \frac{h_{i+1}(E_{i-1}h_i^2 + 6)}{3(h_i + h_{i+1})},$$

$$\beta_i = \frac{2}{3} E_i h_i h_{i+1} - 2 ,$$

$$\gamma_i = \frac{h_i (E_{i+1} h_{i+1}^2 + 6)}{3(h_i + h_{i+1})} , \tag{4.43}$$

$$\delta_i = \frac{F_{i-1} h_i^2 h_{i+1}}{3(h_i + h_{i+1})} + \frac{2}{3} F_i h_i h_{i+1} + \frac{F_{i+1} h_{i+1}^2 h_i}{3(h_i + h_{i+1})} .$$

We write the boundary conditions in the following form similar to (3.11) and (3.12):

$$2U_k + \gamma_k U_{k+1} = \delta_k , \tag{4.44}$$

$$\alpha_r U_{r-1} + 2U_r = \delta_r , \tag{4.45}$$

where γ_k , δ_k , α_r , δ_r are given parameters.

To obtain formulas for calculating the values of γ_k and δ_k , we set $i = k$, $x_{k+1} - x_k = h_{k+1}$, $f_{k+1} = U_{k+1}$, $f_k = U_k$ in expression (4.8):

$$S'(x_k + 0) = -M_k \frac{h_{k+1}}{2} + \frac{U_{k+1} - U_k}{h_{k+1}} - \frac{M_{k+1} - M_k}{6} h_{k+1} .$$

Let us multiply both sides of the last equality by A_1 and substitute expressions

$$M_k = F_k - E_k U_k ,$$

$$M_{k+1} = F_{k+1} - E_{k+1} U_{k+1} ,$$

corresponding to (4.40) at $i = k$ and $i = k+1$. The following equality is the result:

$$A_1 S'(x_k + 0) = -A_1 (F_k - E_k U_k) \frac{h_{k+1}}{2} + A_1 \frac{U_{k+1} - U_k}{h_{k+1}} -$$

$$- A_1 \frac{F_{k+1} - E_{k+1} U_{k+1} - F_k + E_k U_k}{6} h_{k+1} . \tag{4.46}$$

Because $U_k = U(a)$ and $S'(x_k + 0) = U'(a)$, expression

$$A_1 S'(x_k + 0) = A_3 - A_2 U_k \tag{4.47}$$

follows from (4.38).

Equating the right-hand sides of (4.46) and (4.47), we have

$$A_3 - A_2U_k = -A_1(F_k - E_kU_k)\frac{h_{k+1}}{2} + A_1\frac{U_{k+1} - U_k}{h_{k+1}} -$$
$$-A_1\frac{F_{k+1} - E_{k+1}U_{k+1} - F_k + E_kU_k}{6}h_{k+1}.$$

This equation can be written in form (4.44), where

$$\gamma_k = \frac{(E_{k+1}h_{k+1}^2 + 6)A_1}{(E_kh_{k+1}^2 - 3)A_1 + 3h_{k+1}A_2}, \qquad (4.48)$$

$$\delta_k = \frac{[(2F_k + F_{k+1})h_{k+1}A_1 + 6A_3]h_{k+1}}{(E_kh_{k+1}^2 - 3)A_1 + 3h_{k+1}A_2}. \qquad (4.49)$$

To obtain formulas for calculating the values of α_r and δ_r in (4.45), we set $i = r$, $x_r - x_{r-1} = h_r$, $f_r = U_r$, $f_{r-1} = U_{r-1}$ in (4.6):

$$S'(x_r - 0) = M_r\frac{h_r}{2} + \frac{U_r - U_{r-1}}{h_r} - \frac{M_r - M_{r-1}}{6}h_r.$$

Let us multiply both sides of the last equality by B_1 and substitute expressions

$$M_r = F_r - E_rU_r,$$
$$M_{r-1} = F_{r-1} - E_{r-1}U_{r-1},$$

corresponding to (4.40) at $i = r$ and $i = r-1$. The following equality is the result:

$$B_1S'(x_r - 0) = B_1(F_r - E_rU_r)\frac{h_r}{2} + B_1\frac{U_r - U_{r-1}}{h_r} -$$
$$-B_1\frac{F_r - E_rU_r - F_{r-1} + E_{r-1}U_{r-1}}{6}h_r. \qquad (4.50)$$

Because $U_r = U(b)$ and $S'(x_r - 0) = U'(b)$, expression

$$B_1S'(x_r - 0) = B_3 - B_2U_r \qquad (4.51)$$

follows from (4.39).

Equating the right-hand sides of (4.50) and (4.51), we have

$$B_3 - B_2U_r = B_1(F_r - E_rU_r)\frac{h_r}{2} + B_1\frac{U_r - U_{r-1}}{h_r} -$$

$$-B_1\frac{F_r-E_rU_r-F_{r-1}+E_{r-1}U_{r-1}}{6}h_r\,. \tag{4.52}$$

This equation can be written in form (4.45), where

$$\alpha_r=\frac{(E_{r-1}h_r^2+6)B_1}{(E_rh_r^2-3)B_1-3h_rB_2}\,, \tag{4.53}$$

$$\delta_r=\frac{[(2F_r+F_{r-1})h_rB_1-6B_3]h_r}{(E_rh_r^2-3)B_1-3h_rB_2}\,. \tag{4.54}$$

The system of linear algebraic equations (4.42), (4.44) and (4.45) is called the cubic spline scheme for boundary value problem (4.37) — (4.39).

The values of unknown U_k, U_{k+1}, U_{k+2}, ..., U_{r-2}, U_{r-1}, U_r are determined by solving the formulated cubic spline scheme. In this case, the decomposition method (Section 3.2) can be used because forms (3.9), (3.11) and (3.12) are respectively available for equations (4.42), (4.44) and (4.45). Formulas (4.48), (4.49) and $\beta_k=2$ are used in formulas (3.16) and (3.17) to start the forward sweep. Formulas (4.53), (4.54) and $\beta_r=2$ are used in formula (3.18) to start the backward sweep.

4.10. Program realization of the cubic spline method for solving the linear differential equation

Let us develop a subroutine for solving differential equation (4.37) under conditions (4.38) and (4.39) on the left and right boundaries of segment $[a, b]$.

The values of parameters γ_k and δ_k in equation (4.44), corresponding to the left boundary condition, are calculated according to formulas (4.48) and (4.49). These values and $\beta_k = 2$ are used in formulas (3.16) and (3.17) for starting the forward sweep of the decomposition method. Further, the calculation is performed according to recurrence formulas (3.14) and (3.15).

The values of parameters α_r and δ_r in equation (4.45), corresponding to the right boundary condition, are calculated according to formulas (4.53) and (4.54). These values and $\beta_r = 2$ are used in formula (3.18) for starting the backward sweep. Further, the calculation is performed according to recurrence formula (3.13).

The `fobas` subroutine for solving the boundary value problem for differential equation (4.37) has the following form:

Listing 4.15

```
Sub fobas(ByVal k, ByVal r, ByRef X() As Double, _
    ByRef E() As Double, ByRef F() As Double, _
    ByVal A1, ByVal A2, ByVal A3, _
    ByVal B1, ByVal B2, ByVal B3, _
    ByRef U() As Double)
    Dim alpha As Double, beta As Double
    Dim gamma As Double, delta As Double
    Dim i As Integer, w As Double
    Const c As Double = 2 / 3
    Dim h() As Double: ReDim h(k + 1 To r)
    Dim P() As Double: ReDim P(k + 1 To r)
    Dim Q() As Double: ReDim Q(k + 1 To r)
    For i = k + 1 To r
        h(i) = X(i) - X(i - 1)
    Next i
```

```
'Forward sweep:
    w = (E(k) * h(k + 1) ^ 2 - 3) * A1 + _
    3 * h(k + 1) * A2
    gamma = (E(k + 1) * h(k + 1) ^ 2 + 6) * A1 / w
    delta = ((2 * F(k) + F(k + 1)) * _
    h(k + 1) * A1 + 6 * A3) * h(k + 1) / w
    P(k + 1) = -gamma / 2
    Q(k + 1) = delta / 2
    For i = k + 1 To r - 1
        w = 3 * (h(i) + h(i + 1))
        alpha = h(i + 1) * _
        (E(i - 1) * h(i) ^ 2 + 6) / w
        beta = c * E(i) * h(i) * h(i + 1) - 2
        gamma = h(i) * _
        (E(i + 1) * h(i + 1) ^ 2 + 6) / w
        delta = F(i - 1) * h(i) ^ 2 * h(i + 1) / w + _
        c * F(i) * h(i) * h(i + 1) + _
        F(i + 1) * h(i + 1) ^ 2 * h(i) / w
        w = alpha * P(i) + beta
        P(i + 1) = -gamma / w
        Q(i + 1) = (delta - alpha * Q(i)) / w
    Next i
'Backward sweep:
    w = (E(r) * h(r) ^ 2 - 3) * B1 - 3 * h(r) * B2
    alpha = (E(r - 1) * h(r) ^ 2 + 6) * B1 / w
    delta = ((2 * F(r) + F(r - 1)) * _
    h(r) * B1 - 6 * B3) * h(r) / w
    U(r) = (delta - alpha * Q(r)) / _
    (alpha * P(r) + 2)
    For i = r To k + 1 Step -1
        U(i - 1) = P(i) * U(i) + Q(i)
    Next i
End Sub
```

We enter the above declaration into Module13 of the BookNM workbook. The set of parameters of `fobas` is close to the set of parameters of the `foba` subroutine in Section 3.15:

- `k`, `r` are numbers of the left and right boundary nodes of the grid;
- `X` is an array of grid nodes;
- `E` is an array of values of the coefficient of equation (4.37) at the grid nodes;

- `F` is an array of values of the right-hand side of equation (4.37);
- `A1`, `A2`, `A3` are values of the corresponding parameters in left boundary condition (4.38);
- `B1`, `B2`, `B3` are values of the corresponding parameters in right boundary condition (4.39);
- `U` is an array intended for the solution values.

The developed `fobas` subroutine will be used in the remaining sections of this chapter for solving two concrete boundary value problems.

4.11. Solving the linear differential equation by the cubic spline method

To demonstrate the use of the cubic spline method described above, we will solve equation (3.42) with boundary conditions (3.43). This boundary value problem, concerning temperature characteristics of the radial flow between parallel round disks, can be written in form (4.37) — (4.39), where

$$E(x) = -0.25x^4 - (3c+1)x\,,$$

$$F(x) = 0\,,$$

$$A_1 = B_1 = B_3 = 0\,,$$

$$A_2 = A_3 = B_2 = 1\,.$$

As in Section 3.6, we consider that $c = 10$ and $a \le x \le b$, where $a = 0$, $b = 1.5$.

The program below is intended for solving the formulated problem by using the fobas subroutine from the previous section.

Listing 4.16

```
Sub main()
    Dim X() As Double
    Dim E() As Double
    Dim F() As Double
    Dim U() As Double
    Dim c As Double, b As Double, l As Integer
    Dim h As Double, i As Integer
    Dim sb As String, se As String
    c = Selection.Cells(1, 2)
    b = Selection.Cells(2, 2)
    l = Selection.Cells(3, 2)
    h = b / l
    ReDim X(5 To 5 + l)
    ReDim E(5 To 5 + l)
    ReDim F(5 To 5 + l)
    ReDim U(5 To 5 + l)
    For i = 5 To 5 + l
        X(i) = (i - 5) * h
        E(i) = -0.25 * X(i) ^ 4 - (3 * c + 1) * X(i)
        F(i) = 0
    Next i
0:  Call fobas(5, 5 + l, X, E, F, 0, 1, 1, 0, 1, 0, U)
    Selection.Cells(4, 3) = "x"
    Selection.Cells(4, 4) = "U"
    For i = 5 To 5 + l
        Selection.Cells(i, 3) = X(i)
        Selection.Cells(i, 4) = U(i)
    Next i
    sb = Selection.Cells(5, 3).Address
    se = Selection.Cells(5 + l, 4).Address
1:  Call graph(sb, se, "x", "U")
    For i = 5 To 5 + l
        Selection.Cells(i, 1) = X(i)
2:      U(i) = U(i) * Exp(-X(i) ^ 3 / 6)
        Selection.Cells(i, 2) = U(i)
    Next i
    sb = Selection.Cells(5, 1).Address
    se = Selection.Cells(5 + l, 2).Address
3:  Call graph(sb, se, "x", "u")
    Range("P33").Select
End Sub
```

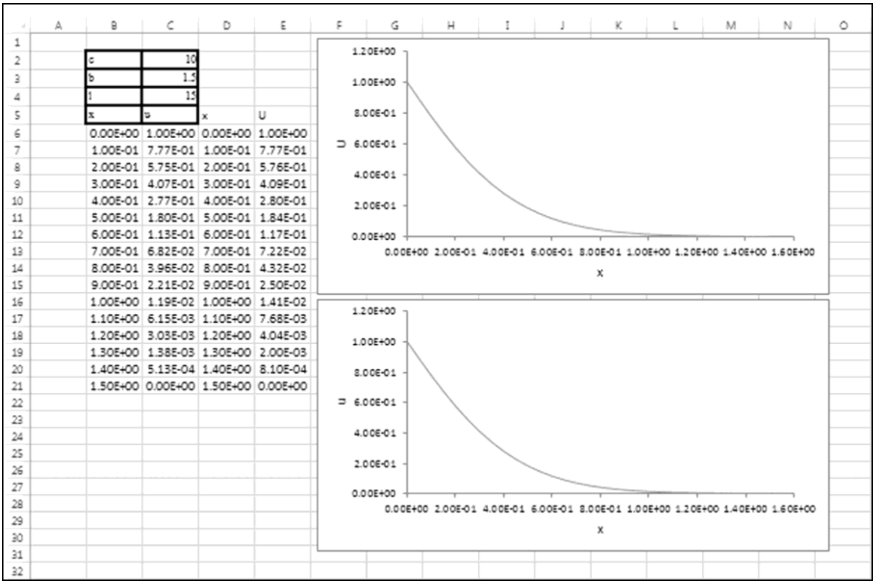
This `main` program is used similarly to programs Listing 3.2 and Listing 3.3 of Section 3.6:

- the initial data are the values located in the table (Fig. 3.2a);
- we must select this Excel table before the program execution (Fig. 3.2b).

Let us consider several operators of program Listing 4.16.

Operator `0` is the call of the `fobas` subroutine for solving boundary value problem (3.42), (3.43) by the cubic spline method. The calculated values of the solution, $U(x)$, are put into the U column (Fig. 4.18). The $U(x)$ graph on the Excel worksheet is created automatically when executing the `graph` subroutine from Section 4.8; operator `1` is the subroutine call.

Operator `2` calculates solution $u(x)$ of boundary value problem (3.39), (3.40) by using formula (3.41). The calculated values of $u(x)$ are put into the u column (Fig. 4.18). The $u(x)$ graph on the Excel worksheet is the result of the second call of the `graph` subroutine (operator `3`). This $u(x)$ dependence is close to $u(x)$ depicted in Fig. 3.3.



	A	B	C	D	E
1					
2		c	10		
3		b	1.5		
4		l	15		
5		x	u	x	U
6		0.00E+00	1.00E+00	0.00E+00	1.00E+00
7		1.00E-01	7.77E-01	1.00E-01	7.77E-01
8		2.00E-01	5.75E-01	2.00E-01	5.76E-01
9		3.00E-01	4.07E-01	3.00E-01	4.09E-01
10		4.00E-01	2.77E-01	4.00E-01	2.80E-01
11		5.00E-01	1.80E-01	5.00E-01	1.84E-01
12		6.00E-01	1.13E-01	6.00E-01	1.17E-01
13		7.00E-01	6.82E-02	7.00E-01	7.22E-02
14		8.00E-01	3.96E-02	8.00E-01	4.32E-02
15		9.00E-01	2.21E-02	9.00E-01	2.50E-02
16		1.00E+00	1.19E-02	1.00E+00	1.41E-02
17		1.10E+00	6.15E-03	1.10E+00	7.68E-03
18		1.20E+00	3.03E-03	1.20E+00	4.04E-03
19		1.30E+00	1.38E-03	1.30E+00	2.00E-03
20		1.40E+00	5.13E-04	1.40E+00	8.10E-04
21		1.50E+00	0.00E+00	1.50E+00	0.00E+00

U
1.20E+00
1.00E+00
8.00E-01
6.00E-01
4.00E-01
2.00E-01
0.00E+00
0.00E+00 2.00E-01 4.00E-01 6.00E-01 8.00E-01 1.00E+00 1.20E+00 1.40E+00 1.60E+00
x
u
1.20E+00
1.00E+00
8.00E-01
6.00E-01
4.00E-01
2.00E-01
0.00E+00
0.00E+00 2.00E-01 4.00E-01 6.00E-01 8.00E-01 1.00E+00 1.20E+00 1.40E+00 1.60E+00
x

Fig. 4.18. The solutions of problems (3.42), (3.43) and (3.39), (3.40) and the corresponding graphs (after their displacement by the mouse)

Note that one of the `fobas` parameters is the `X` array of grid nodes, i.e., this subroutine is usable for both uniform and nonuniform grids. In this section, we used a uniform grid; in the next section, a nonuniform grid will be used.

4.12. Modeling of heating of a geophysical cable. Locally one-dimensional scheme

Cables for geophysical works are intended for repeated descent of instruments (fastened to the end of the cable) into a borehole and for electrical connection of these instruments with terrestrial equipment. By measuring the cable length, the depth of the strata bedding can be determined. Single- and multi-strand cables are used in practice.

Let us consider the cross section of an armored single-strand cable for geophysical works (Fig. 4.19).

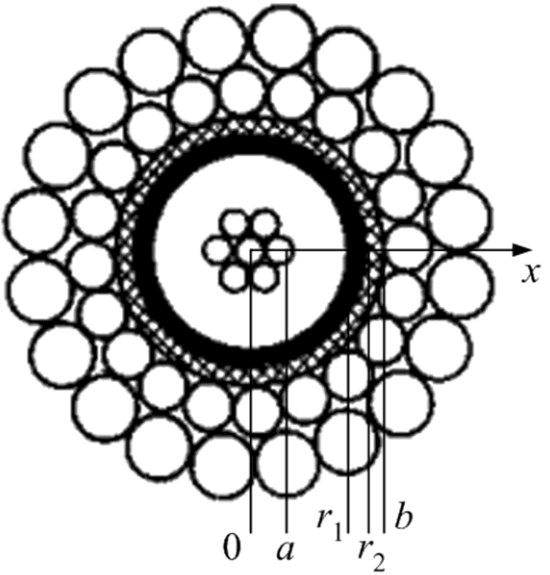


Fig. 4.19. Cross section of an armored single-strand cable: 0 is the cross section center, a, r_1, r_2, b are special points of the x radius

According to website http://www.proelectro.ru/lib/kabel/128.html, the cable for temperatures up to 180 °C has the following design. The central conductor consists of seven copper wires, each of diameter 0.35 mm. This conductor is covered with a 1.35 mm polymer layer for electrical insulation. A cotton layer (which is an electrical insulator too) and double-layer armor (two zinced steel wires) cover the polymer.

We will consider that the conductor is a copper wire of radius 0.5 mm ($0 \le x \le a$ in Fig. 4.20) and the insulator has the following three layers:

- the 1.35 mm homogeneous polymer adjoins the central wire ($a \le x \le r_1$);
- the 0.35 mm homogeneous cotton adjoins the armor ($r_2 \le x \le b$);
- the 0.3 mm non-homogeneous material is placed between these two layers ($r_1 < x < r_2$).

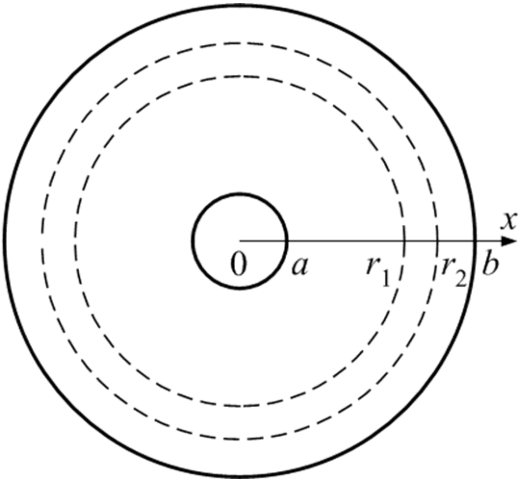


Fig. 4.20. The circular ring, $a \le x \le b$, corresponding to an axially-symmetric insulator

We should calculate the overheating of the conductor and insulator (in relation to the armor) caused by constant electric current I_0 in the conductor. At high electric current, this overheating added to the armor (borehole) temperature damages the cable. The problem will be solved under the assumption that physical parameters of the conductor and insulator are temperature-independent.

The temperature distribution in the insulator, $u(x)$ at $a \le x \le b$, is described by the following heat equation in cylindrical coordinates:

$$cd\frac{\partial u}{\partial t} = \frac{1}{x}\frac{\partial}{\partial x}\left(\lambda(x)x\frac{\partial u}{\partial x}\right), \qquad (4.55)$$

where *t* is time, *x* is the radial coordinate, $\lambda(x)$ is the thermal conductivity of the insulator, $c(x)$ is the specific heat capacity, $d(x)$ is the density of the insulator.

Because $I_0 = const$, the temperature distribution does not depend on time, $\partial u / \partial t = 0$, and equation (4.55) becomes

$$\frac{d}{d\,x}\left(\lambda(x) x \frac{du}{dx} \right) = 0 . \tag{4.56}$$

The boundary conditions are as follows:

$$\frac{du}{dx}(a) = T'(a) , \tag{4.57}$$

$$u(b) = T(b) , \tag{4.58}$$

where *a*, *b* are the inner and outer radii of the ring corresponding to the complicated insulator (Fig. 4.20), $T'(a)$ is the derivative of temperature with respect to *x* on the inner boundary of the ring, $T(b)$ is the temperature on the outer boundary.

To use the `fobas` subroutine from Section 4.10, we will transform boundary value problem (4.56) — (4.58). For that, we use the following designation:

$$\lambda(x) x = \mu(x) . \tag{4.59}$$

Equation (4.56) takes form

$$\frac{d}{d\,x}\left(\mu(x) \frac{du}{dx} \right) = 0 .$$

By differentiating the left-hand side of the last equation as the product of two functions, $\mu(x)$ and du/dx, we can write this equation as follows:

$$\mu \frac{d^2u}{dx^2} + \frac{d\mu}{dx}\frac{du}{dx} = 0 .$$

Dividing both sides by $\mu(x)$, we have the following equation:

$$\frac{d^2u}{dx^2} + g(x)\frac{du}{dx} = 0 , \tag{4.60}$$

where

$$g(x) = \frac{d\mu / dx}{\mu} = \frac{d \ln \mu}{dx} . \tag{4.61}$$

Excluding the first derivative from equation (4.60) by substitution (3.27),

$$u(x) = U(x)\sqrt{\frac{\mu(a)}{\mu(x)}} , \tag{4.62}$$

we write equation (4.60) in form (4.37),

$$\frac{d^2U}{dx^2} + E(x)U = F(x)\,,$$

where

$$E = -0.25g^2(x) - 0.5g'(x)\,,$$
$$F = 0$$

according to (3.31) and (3.32). Using formula (4.61), we have

$$E = -0.25\left(\frac{d\ln\mu}{dx}\right)^2 - 0.5\frac{d^2\ln\mu}{dx^2}\,. \tag{4.63}$$

Boundary conditions (4.57) and (4.58) take forms (4.38) and (4.39),

$$A_2\,U(a) + A_1U'(a) = A_3\,,$$
$$B_2\,U(b) + B_1U'(b) = B_3\,,$$

where

$$A_1 = 1\,,$$
$$A_2 = -0.5\frac{d\ln\mu}{dx}(a)\,,$$
$$A_3 = T'(a)\,, \tag{4.64}$$
$$B_1 = 0\,,$$
$$B_2 = 1\,,$$
$$B_3 = T(b)\sqrt{\frac{\mu(b)}{\mu(a)}}$$

according to (3.34), (3.35), (3.37) and (3.38).

We need a formula that relates the electric current, I_0, with the derivative of temperature (with respect to *x*) on the inner boundary of the ring, $T'(a)$.

To relate I_0 with $T'(a)$, let us look at a piece of wire of length *h*. Geometrically, this piece of wire is a right circular cylinder [3] of height *h* with the base whose radius is equal to *a*.

According to the heat equation in integral form, the power of electric current is equal to the heat flow through the lateral surface of the cylinder:

$$I_0^2R = -\lambda(a)T'(a)M\,, \tag{4.65}$$

where *R* is the electric resistance of the piece of wire, *M* is the lateral area of the cylinder. Because

$$R = \frac{\rho}{\pi a^2} h , \quad M = 2\pi a h ,$$

where ρ is the resistivity of copper, equality (4.65) gives

$$T'(a) = -\frac{I_0^2 \rho}{2\pi^2 a^3 \lambda(a)} . \tag{4.66}$$

Let us develop a program for calculation of overheating of the conductor and insulator (in relation to the armor) caused by the I_0 current. This program will use the source data table given below.

Listing 4.17

a	5.00E-04
I0	25
rho	1.78E-08
r1	1.85E-03
l1	10
lambda1	0.25
r2	2.15E-03
l2	5
b	2.50E-03
l3	5
lambda3	0.05

In Listing 4.17, we see:

- *a* — the radius of the copper wire or the inner radius of the homogeneous polymer (Fig. 4.20), in meters;
- I_0 — the electric current in the copper wire, in amperes;
- ρ — the resistivity of copper, in Ω·m;
- r_1 — the outer radius of the homogeneous polymer, in meters;
- l_1 — the number of steps on segment $a \le x \le r_1$;
- λ_1 — the thermal conductivity of the homogeneous polymer, W / (m·K);
- r_2 — the outer radius of the non-homogeneous material, in meters;
- l_2 — the number of steps on segment $r_1 \le x \le r_2$;

- b — the outer radius of the homogeneous cotton, in meters;
- l_3 — the number of steps on segment $r_2 \le x \le b$;
- λ_3 — the thermal conductivity of the homogeneous cotton, W / (m·K).

The $u(x)$ dependence should be the main result of the program execution. The program follows:

Listing 4.18

```
Sub main()
    Dim x() As Double
    Dim lambda() As Double
    Dim mu() As Double
    Dim ln_mu() As Double
    Dim ln_mu1() As Double
    Dim ln_mu2() As Double
    Dim E() As Double
    Dim F() As Double
    Dim U() As Double
    Dim a As Double, I0 As Double, rho As Double
    Dim r1 As Double, l1 As Integer, lambda1 As Double
    Dim r2 As Double, l2 As Integer
    Dim b As Double, l3 As Integer, lambda3 As Double
    Dim h1 As Double, h2 As Double, h3 As Double
    Dim l As Integer, i As Integer
    Dim T1a As Double, Tb As Double
    Dim A1 As Double, A2 As Double, A3 As Double
    Dim B1 As Double, B2 As Double, B3 As Double
    Const pi As Double = 3.141592654
    Dim w As Double
    Dim sb As String, se As String
    a = Selection.Cells(1, 2)
    I0 = Selection.Cells(2, 2)
    rho = Selection.Cells(3, 2)
    r1 = Selection.Cells(4, 2)
    l1 = Selection.Cells(5, 2)
    lambda1 = Selection.Cells(6, 2)
    r2 = Selection.Cells(7, 2)
    l2 = Selection.Cells(8, 2)
    b = Selection.Cells(9, 2)
    l3 = Selection.Cells(10, 2)
    lambda3 = Selection.Cells(11, 2)
    h1 = (r1 - a) / l1
    h2 = (r2 - r1) / l2
    h3 = (b - r2) / l3
    l = l1 + l2 + l3
    ReDim x(13 To 13 + l)
    ReDim lambda(13 To 13 + l)
    ReDim mu(13 To 13 + l)
    ReDim ln_mu(13 To 13 + l)
    ReDim ln_mu1(13 To 13 + l)
    ReDim ln_mu2(13 To 13 + l)
    ReDim E(13 To 13 + l)
    ReDim F(13 To 13 + l)
    ReDim U(13 To 13 + l)
'Definition of arrays x and lambda:
    x(13) = a
    lambda(13) = lambda1
    For i = 14 To 13 + l1
        x(i) = x(i - 1) + h1
        lambda(i) = lambda1
    Next i
    For i = 14 + l1 To 13 + l1 + l2
        x(i) = x(i - 1) + h2
        lambda(i) = 0.5 * (lambda1 - lambda3) * _
        Cos(pi * (x(i) - r1) / (r2 - r1)) + _
        0.5 * (lambda1 + lambda3)
    Next i
    For i = 14 + l1 + l2 To 13 + l
        x(i) = x(i - 1) + h3
        lambda(i) = lambda3
    Next i
    Selection.Cells(12, 1) = "x"
    Selection.Cells(12, 2) = "lambda"
    For i = 13 To 13 + l
        Selection.Cells(i, 1) = x(i)
        Selection.Cells(i, 2) = lambda(i)
    Next i
    sb = Selection.Cells(13, 1).Address
    se = Selection.Cells(13 + l, 2).Address
1:  Call graph(sb, se, "x", "lambda")
'Definition of arrays mu, ln_mu, ln_mu1 and ln_mu2:
    For i = 13 To 13 + l
        mu(i) = lambda(i) * x(i)
        ln_mu(i) = Log(mu(i))
    Next i
2:  Call mos(13, 13 + l, x, ln_mu, _
    0, -2 / a ^ 2, 0, -2 / b ^ 2, ln_mu2)
    For i = 13 To 13 + l
3:      Call si(13, 13 + l, x, ln_mu, ln_mu2, _
        x(i), w, ln_mu1(i))
    Next i
'Definition and solution of boundary value problem:
    For i = 13 To 13 + l
        E(i) = -0.25 * ln_mu1(i) ^ 2 - 0.5 * ln_mu2(i)
        F(i) = 0
    Next i
    T1a = -I0 ^ 2 * rho / _
        (2 * pi ^ 2 * a ^ 3 * lambda1)
    Tb = 0
    A1 = 1
    A2 = -0.5 * ln_mu1(13)
    A3 = T1a
    B1 = 0
    B2 = 1
    B3 = Tb * Sqr(mu(13 + l) / mu(13))
4:  Call fobas(13, 13 + l, x, E, F, A1, A2, A3, _
    B1, B2, B3, U)
    Selection.Cells(12, 3) = "x"
    Selection.Cells(12, 4) = "u"
    For i = 13 To 13 + l
        Selection.Cells(i, 3) = x(i)
5:      U(i) = U(i) * Sqr(mu(13) / mu(i))
        Selection.Cells(i, 4) = U(i)
    Next i
    sb = Selection.Cells(13, 3).Address
    se = Selection.Cells(13 + l, 4).Address
6:  Call graph(sb, se, "x", "u")
    Range("O36").Select
End Sub
```

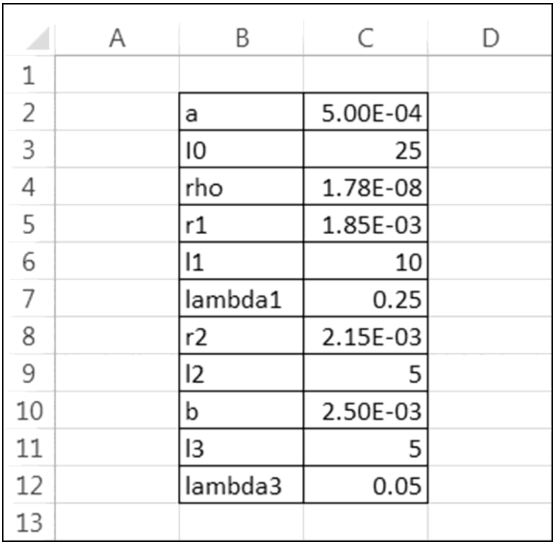
The source data for this program are the values located in the Excel table depicted in Fig. 4.21. We must select this table before the program execution.

Let us consider the main stages of the program execution.

After inputting the source data, the $\lambda(x)$ function is defined in domain $[a, b]$ as follows:

$$\lambda(x)=\begin{cases}\lambda_1 \text{ if } a\le x\le r_1\\ \dfrac{\lambda_1-\lambda_3}{2}\cos\left(\pi\dfrac{x-r_1}{r_2-r_1}\right)+\dfrac{\lambda_1+\lambda_3}{2} \text{ if } r_1<x\le r_2\\ \lambda_3 \text{ if } r_2<x\le b\end{cases}$$

Operator `1` displays the $\lambda(x)$ graph on the Excel worksheet (Fig. 4.22).



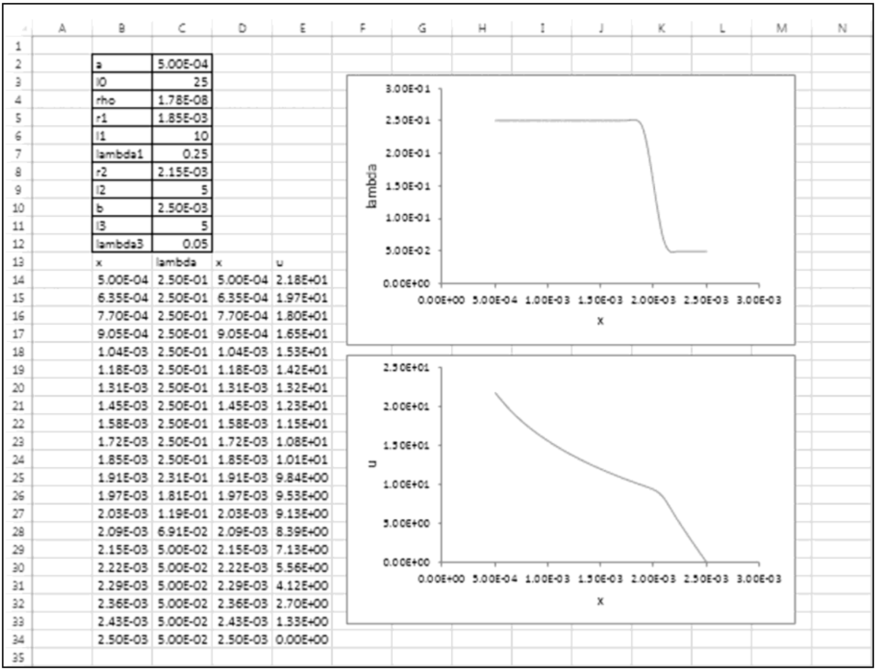
	A	B	C	D
1				
2		a	5.00E-04	
3		I0	25	
4		rho	1.78E-08	
5		r1	1.85E-03	
6		I1	10	
7		lambda1	0.25	
8		r2	2.15E-03	
9		I2	5	
10		b	2.50E-03	
11		I3	5	
12		lambda3	0.05	
13				

Fig. 4.21. The source data table

Further, function $\mu(x)$ is determined according to formula (4.59), and $\ln\mu(x)$, $\dfrac{d\ln\mu}{dx}$, $\dfrac{d^2\ln\mu}{dx^2}$ are determined too. Function $\dfrac{d^2\ln\mu}{dx^2}$ is the result of calling the `mos` subroutine (operator `2`) with parameters defined by formulas (4.15) and (4.16), in which

$$f_k''=\frac{d^2\ln\mu}{dx^2}(a)=-\frac{1}{a^2},\quad f_r''=\frac{d^2\ln\mu}{dx^2}(b)=-\frac{1}{b^2}.$$

Function $\dfrac{d\ln\mu}{dx}$ is the result of calling the `si` subroutine (operator `3`).



	A	B	C	D	E
1					
2		a	5.00E-04		
3		I0	25		
4		rho	1.78E-08		
5		r1	1.85E-03		
6		l1	10		
7		lambda1	0.25		
8		r2	2.15E-03		
9		l2	5		
10		b	2.50E-03		
11		l3	5		
12		lambda3	0.05		
13		x	lambda	x	u
14		5.00E-04	2.50E-01	5.00E-04	2.18E+01
15		6.35E-04	2.50E-01	6.35E-04	1.97E+01
16		7.70E-04	2.50E-01	7.70E-04	1.80E+01
17		9.05E-04	2.50E-01	9.05E-04	1.65E+01
18		1.04E-03	2.50E-01	1.04E-03	1.53E+01
19		1.18E-03	2.50E-01	1.18E-03	1.42E+01
20		1.31E-03	2.50E-01	1.31E-03	1.32E+01
21		1.45E-03	2.50E-01	1.45E-03	1.23E+01
22		1.58E-03	2.50E-01	1.58E-03	1.15E+01
23		1.72E-03	2.50E-01	1.72E-03	1.08E+01
24		1.85E-03	2.50E-01	1.85E-03	1.01E+01
25		1.91E-03	2.31E-01	1.91E-03	9.84E+00
26		1.97E-03	1.81E-01	1.97E-03	9.53E+00
27		2.03E-03	1.19E-01	2.03E-03	9.13E+00
28		2.09E-03	6.91E-02	2.09E-03	8.39E+00
29		2.15E-03	5.00E-02	2.15E-03	7.13E+00
30		2.22E-03	5.00E-02	2.22E-03	5.56E+00
31		2.29E-03	5.00E-02	2.29E-03	4.12E+00
32		2.36E-03	5.00E-02	2.36E-03	2.70E+00
33		2.43E-03	5.00E-02	2.43E-03	1.33E+00
34		2.50E-03	5.00E-02	2.50E-03	0.00E+00
35					

F
G
H
I
J
K
L
M
N
lambda
3.00E-01
2.50E-01
2.00E-01
1.50E-01
1.00E-01
5.00E-02
0.00E+00
0.00E+00 5.00E-04 1.00E-03 1.50E-03 2.00E-03 2.50E-03 3.00E-03
x
u
2.50E+01
2.00E+01
1.50E+01
1.00E+01
5.00E+00
0.00E+00
0.00E+00 5.00E-04 1.00E-03 1.50E-03 2.00E-03 2.50E-03 3.00E-03
x

Fig. 4.22. The program execution results: $\lambda(x)$ is the thermal conductivity change; $u(x)$ is the overheating change

Thus, we defined boundary value problem (4.37) — (4.39) by using formulas (4.63), (4.64) and (4.66), $F = 0$. Operator `4` is the call of the `fobas` subroutine for solving this problem by the cubic spline method. Function $U(x)$ is the result.

Operator `5` calculates solution $u(x)$ of boundary value problem (4.56) — (4.58) by using formula (4.62). The $u(x)$ graph, also located on the Excel worksheet (Fig. 4.22), is the result of the second call of the `graph` subroutine (operator `6`). The $u(x)$ dependence shows the wire and insulator overheating.

According to Fig. 4.22, the wire overheating is equal to $\Delta T = T_a - T_b$ = 21.8 K for I_0 = 25 A. The insulator around the wire has the same overheating. For I_0 = 50 A, the program execution gives $\Delta T(50)$ = 87.3 K.

It was mentioned above that the considered geophysical cable was designed for temperatures up to 180 °C. Therefore, for given current I_0, the cable can be used in boreholes whose temperature does not exceed $180 - \Delta T(I_0)$ degrees Celsius, where $\Delta T(I_0)$ is the result of the program execution for I_0.

We could use the finite difference method (Chapter 3) in the above program concerning the geophysical cable. However, the cubic spline method is more accurate because we use the nonuniform grid. Let us focus on this.

As it was stated at the end of Section 3.14, at the transition from a uniform grid to a nonuniform grid, the error of the finite difference analog of the second derivative changes from the 2nd order of smallness to the 1st order. According to formula (4.19), the second derivative of the difference between the function and the corresponding cubic spline has the 2nd order of smallness for both uniform and nonuniform grids. Thus, the cubic spline method is more accurate than the finite difference method in the case of a nonuniform grid.

Let us return to (4.55). This form of the heat equation is caused by the axial symmetry of the problem under consideration (Fig. 4.20). In the absence of this symmetry (Fig. 4.23), the heat equation has the following form:

$$cd\frac{\partial u}{\partial t} = \frac{\partial}{\partial x_1}\left(\lambda\frac{\partial u}{\partial x_1}\right) + \frac{\partial}{\partial x_2}\left(\lambda\frac{\partial u}{\partial x_2}\right), \tag{4.67}$$

where x_1 and x_2 are the Cartesian coordinates, $\lambda(x_1,\ x_2)$ is the thermal conductivity of the insulator, $c(x_1,\ x_2)$ is the specific heat capacity, $d(x_1,\ x_2)$ is the density. We must solve the initial value problem for (4.67), i.e., calculate the temperature distribution, $u(t,\ x_1,\ x_2)$, at $t > 0$ when $u(0,\ x_1,\ x_2)$ is given.

For solving this problem, we introduce a uniform grid on the t axis, that is, we consider moments $t = k\tau$, where τ is the time step, k = 1, 2, 3, ... Besides, we cover the equation's domain (Fig. 4.23) by a two-dimensional spatial grid.

According to the locally one-dimensional scheme (LOS) [4], to calculate the values of $u(t,\ x_1,\ x_2)$ over the values of $u(t-\tau,\ x_1,\ x_2)$, we must successively solve the boundary value problems for the following differential equations:

$$cd\frac{\tilde{u}(x_1, x_{2,\,j}) - u(t-\tau, x_1, x_{2,\,j})}{\tau} = \frac{d}{d\,x_1}\left(\lambda\frac{d\tilde{u}(x_1, x_{2,\,j})}{dx_1}\right), \tag{4.68}$$

$$cd\frac{u(t, x_{1,\,i}, x_2) - \tilde{u}(x_{1,\,i}, x_2)}{\tau} = \frac{d}{d\,x_2}\left(\lambda\frac{du(t, x_{1,\,i}, x_2)}{dx_2}\right), \tag{4.69}$$

where $\tilde{u}(x_1,\ x_2)$ is an auxiliary function of two variables, $x_{1,\,i}$ and $x_{2,\,j}$ are the coordinates of the gridlines (Fig. 4.23), indices i and j are the gridline numbers.

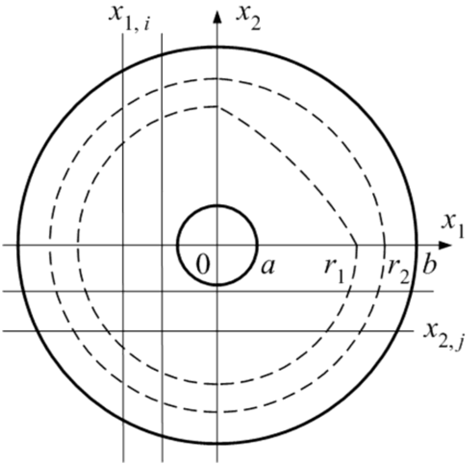


Fig. 4.23. Cross section of an axially-asymmetric insulator; ring $a \le \sqrt{x_1^2 + x_2^2} \le b$ is the equation's domain

More precisely, the calculation of the temperature distribution at the t moment, $u(t,\ x_1,\ x_2)$, over the known distribution at the previous $t-\tau$ moment, $u(t-\tau,\ x_1,\ x_2)$, includes the following two stages:

1) calculation of two-dimensional distribution $\tilde{u}(x_1,\ x_2)$ by solving the boundary value problems for differential equation (4.68) at fixed values of j (x_1 is the independent variable);

2) calculation of two-dimensional distribution $u(t,\ x_1,\ x_2)$ by solving the boundary value problems for differential equation (4.69) at fixed values of i (x_2 is the independent variable).

To solve the boundary value problems for equations (4.68) and (4.69) of the LOS, the following two methods are used: the finite difference method in [4] and the cubic spline method in [6]. In other words, the locally one-dimensional finite difference scheme and the locally one-dimensional cubic spline scheme are respectively developed in [4] and [6].

The LOS can be used for solving various problems with $n \ge 2$ spatial coordinates: the problem solved in [6] has little resemblance to the above one.

In addition to the cubic spline, more simple quadratic and linear splines are used for the solution of applied problems. Below, we will consider these splines.

Chapter 5.
Quadratic and Linear Splines

In addition to the third-degree (cubic) spline of the previous chapter, we consider the simpler second-degree (quadratic) and first-degree (linear) splines. The quadratic spline is used for solving the initial value problem (of Cauchy) for the system of differential equations. The linear spline is used in the least-squares method.

While solving the initial value problem, the system of nonlinear algebraic equations should be solved. Therefore, we also consider the Newton method. For solving a single nonlinear algebraic equation, in addition to the Newton (tangent) method, we consider two Newton-like methods, namely, the secant and Steffensen methods (the consideration of the secant method was started in Section 4.5).

For demonstration of the splines possibilities, programs are developed for the following purposes:

- to simulate the piano mechanism linking a key with hammer;
- to determine the dependence of production results versus factors (the so-called production function) by the least-squares method;
- to solve the sound insulation problem.

The last problem also demonstrates possibilities of user-defined subroutines for the forward and backward discrete Fourier transforms of a periodic tabular function. These subroutines (Section 5.11) are based on the corresponding Algol 60 procedures developed by A. L. Zakharov, scientific worker of Pulsar R&D Manufacturing Company, Moscow, in the end of the 1970s.

5.1. Definition of quadratic spline. Spline slopes

As in Section 4.1, let us consider segment $[a, b]$ covered with grid $a = x_k < x_{k+1} < x_{k+2} < ... < x_{r-2} < x_{r-1} < x_r = b$. Values f_k, f_{k+1}, f_{k+2}, ..., f_{r-2}, f_{r-1}, f_r of grid function $f(x)$ are given.

A quadratic spline (or second-degree spline, Fig. 5.1) is function $P(x)$, which satisfies the following conditions:

1) on each elementary segment $x_{i-1} \le x \le x_i$ $(k + 1 \le i \le r)$, the spline coincides with a second-degree polynomial (generally, the polynomials are different on different elementary segments);

2) at the grid nodes, the spline has the grid function values: $P(x_i) = f_i$;

3) the spline has a continuous derivative, i.e., the spline is smooth;

4) on the left boundary of segment $[a, b]$, the spline satisfies an additional condition (the boundary condition may be formulated on the right boundary).

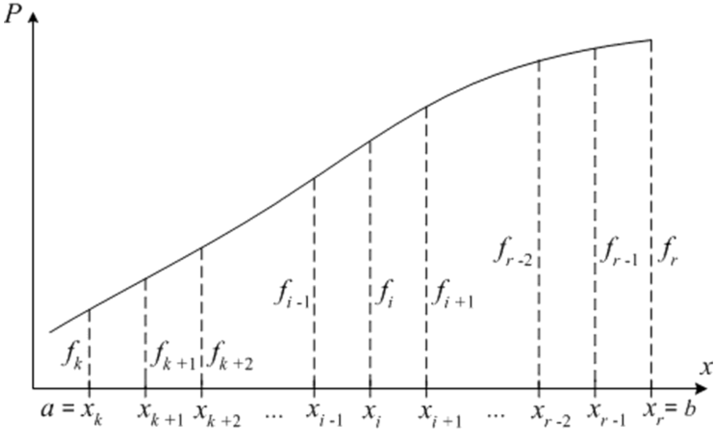


Fig. 5.1. The quadratic spline graph

According to item (2), the $P(x)$ graph passes through points (x_k, f_k), (x_{k+1}, f_{k+1}), (x_{k+2}, f_{k+2}), …, (x_{r-2}, f_{r-2}), (x_{r-1}, f_{r-1}), (x_r, f_r). According to item (3), the jumps of derivative $P'(x)$ are absent at the interior grid nodes, i.e., at points x_{k+1}, x_{k+2}, ..., x_{r-2}, x_{r-1}.

The values of $P'(x)$ at the grid nodes are called slopes of the quadratic spline: $P'(x_i) = Q_i$ $(k \le i \le r)$.

Let the spline slopes, Q_i $(k \le i \le r)$, be given. In this case, items (1) and (3) of the quadratic spline definition lead to the following expression for the derivative on elementary segment $[x_{i-1}, x_i]$:

$$P'(x) = Q_{i-1}\frac{x_i - x}{h_i} + Q_i\frac{x - x_{i-1}}{h_i}, \tag{5.1}$$

where $h_i = x_i - x_{i-1}$ is the elementary segment's length or the grid step, $k + 1 \le i \le r$.

By integrating (5.1), we obtain the following expression:

$$P(x) = -Q_{i-1}\frac{(x_i - x)^2}{2h_i} + Q_i\frac{(x - x_{i-1})^2}{2h_i} + C, \tag{5.2}$$

where C is the integration constant.

Assuming that slope Q_{i-1} is known, we will determine integration constant C and slope Q_i. The resulting expression for C will be substituted into expression (5.2) for $P(x)$.

According to item (2) of the quadratic spline definition, we have the following equalities:

$$P(x_{i-1}) = f_{i-1},$$

$$P(x_i) = f_i.$$

Using expression (5.2), we get the system of two linear algebraic equations with unknown C and Q_i:

$$C = f_{i-1} + Q_{i-1}\frac{h_i}{2}, \tag{5.3}$$

$$Q_i\frac{h_i}{2} + C = f_i. \tag{5.4}$$

Substituting expression (5.3) into equation (5.4), we have

$$Q_i = 2\frac{f_i - f_{i-1}}{h_i} - Q_{i-1}. \tag{5.5}$$

By substituting expression (5.3) into (5.2), we obtain the following expression for the spline on segment $[x_{i-1}, x_i]$:

$$P(x) = Q_{i-1}\left[\frac{h_i}{2} - \frac{(x_i - x)^2}{2h_i}\right] + Q_i\frac{(x - x_{i-1})^2}{2h_i} + f_{i-1}. \tag{5.6}$$

According to item (4) of the quadratic spline definition, we define the slope on the left boundary of segment $[a, b]$ as follows:

$$Q_k = f'(a),$$

where $f'(a)$ is a given value of the function derivative on the left boundary.

Using recurrence formula (5.5) successively at $i = k+1, k+2, ..., r$, we can calculate all slopes of the quadratic spline. Further, we can:

- interpolate grid function $f(x)$ by means of formula (5.6);
- calculate the $f'(x)$ value at arbitrary point x of segment $[a, b]$ by using formula (5.1).

The error of interpolating the $f(x)$ function (and its derivatives) by the $P(x)$ spline (and by its derivatives) is determined by the following expression:

$$f^{(n)}(x) - P^{(n)}(x) = O(h_{max}^{3-n}),$$

where $h_{max} = \max\limits_{k+1 \le i \le r} \{h_i\}$ is the maximum grid step ($h_{max} \to 0$), $n = 0, 1, 2$ is the derivative order, $f^{(0)}(x) = f(x)$, $P^{(0)}(x) = P(x)$. The sense of the used O notation is explained in Section 3.1.

The above mathematical construction of quadratic spline will be used for solving the initial value problem for the system of differential equations.

5.2. Method for solving the initial value problem for the system of differential equations

Let the system of two differential equations

$$\frac{du}{dx} = E(x, u, v)\,, \tag{5.7}$$

$$\frac{dv}{dx} = F(x, u, v) \tag{5.8}$$

be given on segment $[a, b]$. Generally, E and F are nonlinear functions of three variables.

Let the values of unknown functions $u(x)$ and $v(x)$ be given on the left boundary:

$$u(a) = A\,,$$
$$v(a) = Q\,,$$

where A and Q are parameters.

The conditions on the left boundary are called the initial conditions.

For solving this initial value (Cauchy) problem on segment $[a, b]$, the grid is constructed (Fig. 5.1) and the $u(x)$ and $v(x)$ functions are considered as quadratic splines. In this case, according to equations (5.7) and (5.8), the values of functions E and F at the grid nodes are the slopes of these splines.

With regard to splines $u(x)$ and $v(x)$, expression (5.5) for the slope takes the following forms:

$$E(x_i, u_i, v_i) = 2\frac{u_i - u_{i-1}}{h_i} - E(x_{i-1}, u_{i-1}, v_{i-1})\,,$$

$$F(x_i, u_i, v_i) = 2\frac{v_i - v_{i-1}}{h_i} - F(x_{i-1}, u_{i-1}, v_{i-1})$$

or

$$u_i - \frac{h_i}{2} E(x_i, u_i, v_i) = \alpha\,, \tag{5.9}$$

$$v_i - \frac{h_i}{2} F(x_i,\, u_i,\, v_i) = \beta\,, \tag{5.10}$$

where

$$\alpha = u_{i\,-\,1} + \frac{h_i}{2} E(x_{i\,-\,1},\, u_{i\,-\,1},\, v_{i\,-\,1})\,,$$

$$\beta = v_{i\,-\,1} + \frac{h_i}{2} F(x_{i\,-\,1},\, u_{i\,-\,1},\, v_{i\,-\,1})\,.$$

Equations (5.9) and (5.10) form the system of algebraic equations with respect to u_i and v_i (at known $u_{i\,-\,1}$ and $v_{i\,-\,1}$).

If $u_k = u(a) = A$ and $v_k = v(a) = Q$ are given, unknown functions $u(x)$ and $v(x)$ can be calculated at all grid nodes on segment $[a,\, b]$ by solving the system of algebraic equations (5.9) and (5.10) successively at $i = k+1,\, k+2,\, ...,\, r$.

As an example, we will solve the system of differential equations

$$\frac{du}{dx} = v\,, \tag{5.11}$$

$$\frac{dv}{dx} = -\frac{v^2}{2-u} \tag{5.12}$$

on segment $[0,\, 1]$. The initial conditions look like

$$u(0) = A\,,$$

$$v(0) = Q\,.$$

This initial value problem was solved in the fifth chapter of book [5] while modeling a catalytic converter.

The catalytic converter is a device intended for lowering toxicity of waste gases in the car exhaust system.

Let us write system (5.11), (5.12) in form (5.7), (5.8), where

$$E = v\,,$$

$$F = -\frac{v^2}{2-u}\,.$$

We see that E is a linear function of variable *v*, F is a nonlinear function of variables *u* and *v*.

Algebraic equations (5.9) and (5.10) take the following form:

$$u_i - \frac{h_i}{2} v_i = \alpha\,, \tag{5.13}$$

$$v_i + \frac{h_i}{2} \frac{v_i^2}{2 - u_i} = \beta\,, \tag{5.14}$$

where

$$\alpha = u_{i-1} + \frac{h_i}{2} v_{i-1}\,,$$

$$\beta = v_{i-1} - \frac{h_i}{2} \frac{v_{i-1}^2}{2 - u_{i-1}}\,.$$

Let us solve the system of algebraic equations (5.13) and (5.14) for unknown u_i and v_i.

Equation (5.13) can be written as

$$u_i = \alpha + \frac{h_i}{2} v_i\,. \tag{5.15}$$

Multiplying both sides of equation (5.14) by $2 - u_i$ and substituting expression (5.15) for u_i, we obtain

$$v_i = \frac{(2-\alpha)\beta}{2 - \alpha + \frac{h_i}{2}\beta}\,. \tag{5.16}$$

Formulas (5.16) and (5.15) allow us to calculate the values of v_i and u_i successively at $i = k+1, k+2, ..., r$.

5.3. Program for solving the initial value problem

Let us develop a program for solving the initial value problem for the system of differential equations (5.11) and (5.12). Number $l = r - k$ of steps on segment $[0, 1]$ and the values of A and Q are given in the table below.

l	10
A	0.25
Q	-0.1

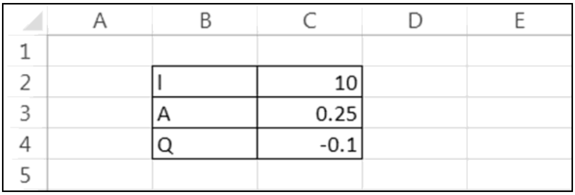
As a result of the program execution, dependences $u(x)$ and $v(x)$ should appear under the original table.

The program has the following form:

Listing 5.1

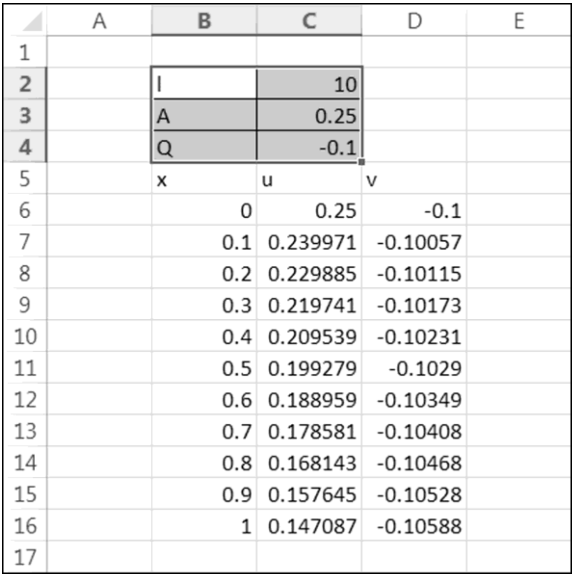
```
Sub main()
    Dim l As Integer
    Dim u As Double, v As Double
    Dim h As Double, h2 As Double
    Dim i As Integer
    Dim alpha As Double, beta As Double
    l = Selection.Cells(1, 2)
    u = Selection.Cells(2, 2)                'value of A
    v = Selection.Cells(3, 2)                'value of Q
    h = 1 / l: h2 = h / 2
    Selection.Cells(4, 1) = "x"
    Selection.Cells(4, 2) = "u"
    Selection.Cells(4, 3) = "v"
    Selection.Cells(5, 1) = 0
    Selection.Cells(5, 2) = u
    Selection.Cells(5, 3) = v
    For i = 6 To 5 + l          'movement along axis x
        Selection.Cells(i, 1) = (i - 5) * h
        alpha = u + h2 * v
        beta = v - h2 * v ^ 2 / (2 - u)
        v = (2 - alpha) * beta / (2 - alpha + _
        h2 * beta)
        u = alpha + h2 * v
        Selection.Cells(i, 2) = u
        Selection.Cells(i, 3) = v
    Next i
End Sub
```

The program uses the values of the above table (Fig. 5.2). We must select this Excel table before the program execution. The coordinates of the grid nodes and the corresponding values of the $u(x)$ and $v(x)$ dependences are placed in columns *x*, *u*, *v* (Fig. 5.3).



	A	B	C	D	E
1					
2		l	10		
3		A	0.25		
4		Q	-0.1		
5					

Fig. 5.2. The Excel table with the source data



	A	B	C	D	E
1					
2		l	10		
3		A	0.25		
4		Q	-0.1		
5		x	u	v	
6		0	0.25	-0.1	
7		0.1	0.239971	-0.10057	
8		0.2	0.229885	-0.10115	
9		0.3	0.219741	-0.10173	
10		0.4	0.209539	-0.10231	
11		0.5	0.199279	-0.1029	
12		0.6	0.188959	-0.10349	
13		0.7	0.178581	-0.10408	
14		0.8	0.168143	-0.10468	
15		0.9	0.157645	-0.10528	
16		1	0.147087	-0.10588	
17					

Fig. 5.3. The program execution results

We advise the reader to write a program (similar to Listing 5.1) for solving the Bernoulli differential equation [3],

$$\frac{du}{dx} = f(x)u^2,$$

on segment $[a, b]$ under initial condition $u(a) = 1$. In this equation, $f(x)$ is a function from Appendix 4; segment $[a, b]$ is this function's domain.

5.4. Solving the system of nonlinear algebraic equations by the Newton method

In Section 5.6, we will need to solve the system of nonlinear algebraic equations

$$\begin{aligned} f_1(x_1,\ x_2,\ \ldots,\ x_n) &= \alpha_1\,, \\ f_2(x_1,\ x_2,\ \ldots,\ x_n) &= \alpha_2\,, \\ &\cdot\ \cdot\ \cdot\ \cdot\ \cdot\ \cdot \\ f_n(x_1,\ x_2,\ \ldots,\ x_n) &= \alpha_n\,, \end{aligned} \tag{5.17}$$

where functions f_1, f_2, ..., f_n are twice differentiable with respect to arguments x_1, x_2, ..., x_n, right-hand sides α_1, α_2, ..., α_n are given constants (we can consider them equal to zero). The solution process is iterative.

To understand the iteration content, we write solution $(x_1^*,\ x_2^*,\ \ldots,\ x_n^*) = \mathbf{x}^*$ of system (5.17) in form

$$x_1^* = x_1^j + z_1\,,\ x_2^* = x_2^j + z_2\,,\ \ldots,\ x_n^* = x_n^j + z_n\,,$$

where $(x_1^j,\ x_2^j,\ \ldots,\ x_n^j) = \mathbf{x}^j$ is the j-th approximation of the $\mathbf{x}^*$ solution, z_1, z_2, ..., z_n are small quantities.

By substituting expressions for x_1^*, x_2^*, ..., x_n^* into (5.17), we obtain the following system of nonlinear algebraic equations with respect to z_1, z_2, ..., z_n:

$$\begin{aligned} f_1(x_1^j + z_1,\ x_2^j + z_2,\ \ldots,\ x_n^j + z_n) &= \alpha_1\,, \\ f_2(x_1^j + z_1,\ x_2^j + z_2,\ \ldots,\ x_n^j + z_n) &= \alpha_2\,, \\ \cdot\ \cdot\ \ \ \cdot\ \cdot\ \cdot\ \cdot\ \cdot\ \cdot\ \cdot\ \cdot \\ f_n(x_1^j + z_1,\ x_2^j + z_2,\ \ldots,\ x_n^j + z_n) &= \alpha_n\,. \end{aligned}$$

Using Taylor's formula [3] for functions of n variables in a neighborhood of point $\mathbf{x}^j$, we have

$$f_i(x_1^j + z_1,\ x_2^j + z_2,\ ...,\ x_n^j + z_n) = f_i(x_1^j,\ x_2^j,\ ...,\ x_n^j) +$$
$$+\frac{1}{1!}\left(z_1 \frac{\partial}{\partial x_1} + z_2 \frac{\partial}{\partial x_2} + ... + z_n \frac{\partial}{\partial x_n} \right) f_i(x_1^j,\ x_2^j,\ ...,\ x_n^j) + O(z_{max}^2),$$

$1 \le i \le n$. At that, the last algebraic equations with respect to $z_1,\ z_2,\ ...,\ z_n$ take the following linear form:

$$z_1 \frac{\partial f_1}{\partial x_1}(\mathbf{x}^j) + z_2 \frac{\partial f_1}{\partial x_2}(\mathbf{x}^j) + ... + z_n \frac{\partial f_1}{\partial x_n}(\mathbf{x}^j) = \alpha_1 - f_1(\mathbf{x}^j) + O(z_{max}^2),$$
$$z_1 \frac{\partial f_2}{\partial x_1}(\mathbf{x}^j) + z_2 \frac{\partial f_2}{\partial x_2}(\mathbf{x}^j) + ... + z_n \frac{\partial f_2}{\partial x_n}(\mathbf{x}^j) = \alpha_2 - f_2(\mathbf{x}^j) + O(z_{max}^2),$$
$$. \quad . \quad . \quad . \quad . \quad . \quad . \quad . \quad . \quad . \quad . \quad . \quad . \quad . \quad . \quad . \quad . \quad . \quad . \quad .$$
$$z_1 \frac{\partial f_n}{\partial x_1}(\mathbf{x}^j) + z_2 \frac{\partial f_n}{\partial x_2}(\mathbf{x}^j) + ... + z_n \frac{\partial f_n}{\partial x_n}(\mathbf{x}^j) = \alpha_n - f_n(\mathbf{x}^j) + O(z_{max}^2),$$

where z_{max} is the maximum of small quantities $|z_1|,\ |z_2|,\ ...,\ |z_n|$,

$$z_{max} = \max_{1 \le i \le n} \{|z_i|\}.$$

In the Newton method, the $(j+1)$th approximation of the solution (j = 0, 1, 2, ...) is calculated over the known j-th approximation as follows:

1) matrix

$$\begin{bmatrix} a_{11} & a_{12} & ... & a_{1n} \\ a_{21} & a_{22} & ... & a_{2n} \\ . & . & . & . \\ a_{n1} & a_{n2} & ... & a_{nn} \end{bmatrix} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1}(\mathbf{x}^j) & \frac{\partial f_1}{\partial x_2}(\mathbf{x}^j) & ... & \frac{\partial f_1}{\partial x_n}(\mathbf{x}^j) \\ \frac{\partial f_2}{\partial x_1}(\mathbf{x}^j) & \frac{\partial f_2}{\partial x_2}(\mathbf{x}^j) & ... & \frac{\partial f_2}{\partial x_n}(\mathbf{x}^j) \\ . & . & . & . \\ \frac{\partial f_n}{\partial x_1}(\mathbf{x}^j) & \frac{\partial f_n}{\partial x_2}(\mathbf{x}^j) & ... & \frac{\partial f_n}{\partial x_n}(\mathbf{x}^j) \end{bmatrix} \qquad (5.18)$$

of the partial derivatives of functions $f_1,\ f_2,\ ...,\ f_n$ is calculated;

2) vector

$$\begin{bmatrix} b_1 \\ b_2 \\ \dots \\ b_n \end{bmatrix} = \begin{bmatrix} \alpha_1 - f_1(\mathbf{x}^j) \\ \alpha_2 - f_2(\mathbf{x}^j) \\ . \quad . \quad . \quad . \\ \alpha_n - f_n(\mathbf{x}^j) \end{bmatrix} \tag{5.19}$$

is calculated;

3) the system of linear algebraic equations

$$\begin{array}{ccccccccc} a_{11}z_1 & + & a_{12}z_2 & + & \dots & + & a_{1n}z_n & = & b_1, \\ a_{21}z_1 & + & a_{22}z_2 & + & \dots & + & a_{2n}z_n & = & b_2, \\ . & . & . & . & . & . & . & . & . \\ a_{n1}z_1 & + & a_{n2}z_2 & + & \dots & + & a_{nn}z_n & = & b_n \end{array} \tag{5.20}$$

is solved, for example, by the Gaussian elimination method considered in Sections 3.9 and 3.10;

4) the $(j+1)$th approximation of the solution is calculated according to formulas

$$\begin{aligned} x_1^{j+1} &= x_1^j + z_1, \\ x_2^{j+1} &= x_2^j + z_2, \\ . \quad . \quad &. \quad . \quad . \\ x_n^{j+1} &= x_n^j + z_n. \end{aligned} \tag{5.21}$$

Various conditions can be applied for finishing the iterative process; we will use the following:

$$z_{max} < \zeta, \tag{5.22}$$

where ζ is a given positive constant,

$$z_{max} = \max_{1 \le i \le n} \{|z_i|\}.$$

Matrix (5.18) of the partial derivatives of functions f_1, f_2, ..., f_n is called ***the Jacobian matrix*** of these functions.

5.5. Newton and Newton-like methods for solving the single nonlinear algebraic equation

It is obvious that the Newton iterative process of the previous section can be used for solving not only the system of nonlinear algebraic equations, but also for solving one equation, which can be considered as a system containing one equation with respect to x_1:

$$f_1(x_1) = \alpha_1 .$$

In this case, the Newton method has a simple geometric interpretation.

The geometric interpretation will be considered on an example of equation (4.23),

$$f(x) = 0 ,$$

satisfying the following conditions in the $f(x)$ function's domain:

- the $f(x)$ function is continuous and monotonous;
- the derivative, $f'(x)$, is continuous, i.e., the $f(x)$ function is smooth;
- $f'(x)$ is different from 0;
- the equation solution, x^*, exists.

In addition, we suppose that the solution's initial approximation (x^0, which must be given) is located in the $f(x)$ function's domain.

The Newton iterative process, defined by items (1) — (4) of the previous section, can be written as follows:

$$x^{j+1} = x^j - \frac{f(x^j)}{f'(x^j)} , \tag{5.23}$$

where x^j, x^{j+1} are the *j*-th and ($j+1$)th approximations of the equation solution, *j* = 0, 1, 2, ... Initial approximation x^0 is given.

Fig. 5.4 gives the geometric interpretation of the process of solving equation $f(x) = 0$. According to this figure, the ($j+1$)th approximation of the x^* solution can be determined as follows:

1) restore the perpendicular to the *x* axis from point x^j;

2) run the tangent line through the point of intersection of the perpendicular with the $f(x)$ graph;

3) consider the coordinate of the point of intersection of the tangent line with the *x* axis as the ($j+1$)th approximation of the equation solution, x^{j+1}.

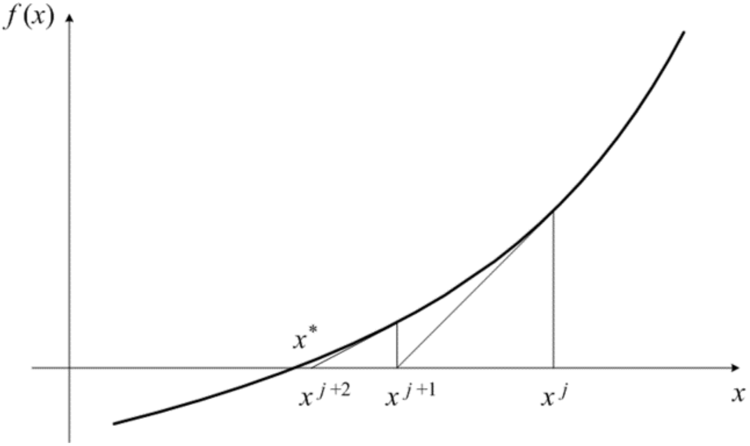


Fig. 5.4. The geometric interpretation of the Newton method

This algorithm follows from formula (5.23) and the following property of the tangent line passing through the point with coordinates x^j and $f(x^j)$: the slope of this tangent line is equal to

$$f'(x^j) = \frac{f(x^j)}{x^j - x^{j+1}}.$$

Due to the above geometric interpretation of the Newton method, it is also called ***the tangent method***.

In Section 4.5, equation (4.23), (4.24) of form $x - \cos x - 1.5 = 0$ was solved by using the Solver add-in for Excel. Below is a code for solving this equation by the tangent method.

Listing 5.2

```
Sub main()
    Dim j_lim As Integer
    Dim zeta As Double
    Dim x As Double
    Dim f As Double
    Dim a As Double
    Dim j As Integer
    Dim z As Double
1:  j_lim = Range("I1").Value
2:  zeta = Range("H1").Value
3:  x = Range("G1").Value
    For j = 1 To j_lim              'Newton iterations
4:      Call f_function(x, f)
5:      Call fx_jacobian(x, a)
6:      z = -f / a
7:      x = x + z
8:      If Abs(z) < zeta Then Exit For
    Next j
9:  Range("G2").Value = x
10: Range("F2").Value = f
11: Range("E2").Value = a
12: Range("I2").Value = j
End Sub

Sub f_function(ByVal x As Double, _
    ByRef f As Double)
    f = x - Cos(x) - 1.5
End Sub

Sub fx_jacobian(ByVal x As Double, _
    ByRef fx As Double)
    fx = 1 + Sin(x)
End Sub
```

In this code:

1) `j_lim` is the limiting number of iterations; its value is taken from cell I1 of the active worksheet (operator `1`);

2) `zeta` is constant ζ in condition (5.22) of finishing the Newton iterative process; its value is taken from cell H1 (operator `2`);

3) `x` is the current approximation of the equation solution; the initial value of `x` (the initial approximation of the solution) is taken from cell G1 (operator `3`).

After inputting the values of `j_lim`, `zeta` and `x`, the `j` cycle of the Newton iterations is performed. In this cycle:

1) operator `4` calculates value `f` of the function according to formula $f(x) = x - \cos\ x - 1.5$ by calling the `f_function` subroutine;

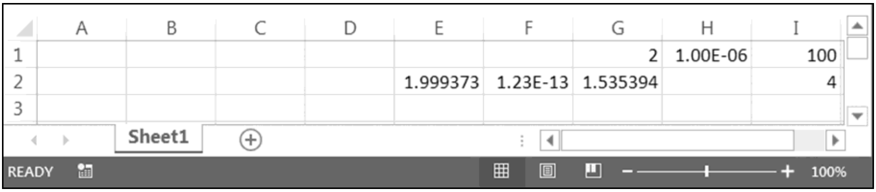
2) operator `5` calculates the derivative's value, `a`, according to formula $f'(x) = 1 + \sin\ x$ by calling the `fx_jacobian` subroutine;

3) operators `6` and `7` calculate the next approximation of the equation solution according to formula (5.23);

4) operator `8` checks condition (5.22), $|z| < \zeta$, for finishing the Newton iterations.

When finishing the iterative process (after leaving the `j` cycle), the calculated values of `x`, `f` and `a` are respectively put into cells G2, F2 and E2 (operators `9`, `10` and `11`). Operator `12` puts the number of the Newton iterations into cell I2.

Fig. 5.5 shows the Excel worksheet upon termination of the code execution. Cell G2 contains the following value of the solution of equation (4.23), (4.24): $x^* = 1.535394$. It practically coincides with the x^* value calculated by means of Excel in Section 4.5.



	A	B	C	D	E	F	G	H	I
1							2	1.00E-06	100
2					1.999373	1.23E-13	1.535394		4
3									

Fig. 5.5. The worksheet upon termination of solving the equation

Let us consider the question of the convergence of the Newton iterative process defined by formula (5.23). For that, we use the following designation (similar to the designations on p. 311): $x^j - x^* = \varepsilon^j$, where x^* is the exact solution of the equation, i.e., $f(x^*) = 0$. Let ε^j be a small quantity.

Substituting expressions $x^j = x^* + \varepsilon^j$ and $x^{j+1} = x^* + \varepsilon^{j+1}$ into formula (5.23), we have

$$\varepsilon^{j+1} = \varepsilon^j - \frac{f(x^* + \varepsilon^j)}{f'(x^* + \varepsilon^j)}.$$

Let us use expressions similar to (3.1) for $f(x^* + \varepsilon^j)$ and $f'(x^* + \varepsilon^j)$ and take into account $f(x^*) = 0$. At that, we have the following chain of equalities:

$$\varepsilon^{j+1} = \varepsilon^j - \frac{f(x^*) + \varepsilon^j f'(x^*) + 0.5(\varepsilon^j)^2 f''(x^*) + O[(\varepsilon^j)^3]}{f'(x^*) + \varepsilon^j f''(x^*) + O[(\varepsilon^j)^2]} =$$

$$= \varepsilon^j - \frac{\varepsilon^j f'(x^*) + 0.5(\varepsilon^j)^2 f''(x^*) + O[(\varepsilon^j)^3]}{f'(x^*) + \varepsilon^j f''(x^*) + O[(\varepsilon^j)^2]} =$$

$$= \varepsilon^j \frac{0.5\varepsilon^j f''(x^*) + O[(\varepsilon^j)^2]}{f'(x^*) + \varepsilon^j f''(x^*) + O[(\varepsilon^j)^2]} = (\varepsilon^j)^2 \frac{0.5 f''(x^*) + O(\varepsilon^j)}{f'(x^*) + O(\varepsilon^j)} .$$

Neglecting $O(\varepsilon^j)$ in the numerator and denominator, we have the following formula:

$$\varepsilon^{j+1} = 0.5 \frac{f''(x^*)}{f'(x^*)} (\varepsilon^j)^2 . \tag{5.24}$$

Because $f'(x^*) \neq 0$, the last formula shows that the Newton iterative process has quadratic convergence.

The secant method (Section 4.5), defined by formula (4.28) in form

$$x^{j+1} = x^j - \frac{(x^j - x^{j-1}) f(x^j)}{f(x^j) - f(x^{j-1})} , \tag{5.25}$$

can be considered as a Newton-like method. The reason is that formula (5.25) takes form (5.23) of the Newton (tangent) method at $x^j - x^{j-1} \to 0$ because

$$\frac{f(x^j) - f(x^{j-1})}{x^j - x^{j-1}} \approx f'(x^j) .$$

As we see, the secant method in form (5.25) needs two initial approximations of the equation solution, x^0 and x^1.

Fig. 5.6 gives the geometric interpretation of the secant method. According to this figure, the ($j+1$)th approximation of the x^* solution over the ($j-1$)th and *j*-th approximations can be calculated as follows:

1) restore the perpendiculars to the *x* axis from points x^{j-1} and x^j of the *x* axis;

2) run the secant line through points (x^{j-1}, f_u) and (x^j, f_v) of intersection of the perpendiculars with the $f(x)$ graph;

3) consider the coordinate of the point of intersection of the secant line with the *x* axis as the ($j+1$)th approximation of the equation solution, x^{j+1}.

Formula, intended for estimating the rate of the convergence of the secant method's iterative process, is close to formula (5.24):

$$\varepsilon^{j+1} = 0.5\frac{f''(x^*)}{f'(x^*)}\varepsilon^j\varepsilon^{j-1}. \tag{5.26}$$

The derivation of this formula is similar to the above derivation of (5.24). Formula (4.30) is the simplified form of (5.26).

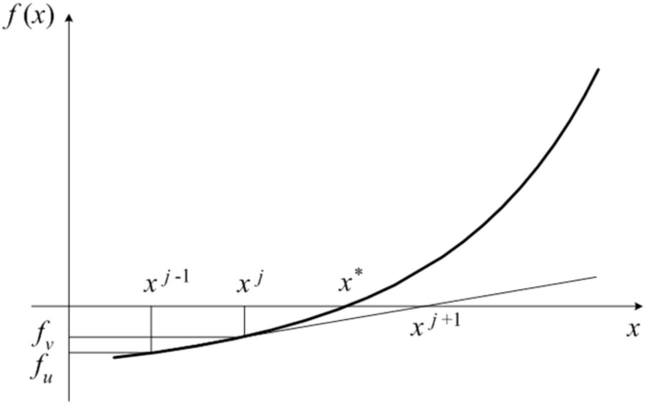


Fig. 5.6. The geometric interpretation of the secant method:

$$f_u = f(x^{j-1}), \quad f_v = f(x^j)$$

In Section 4.5, we considered another version of the secant method, which has slow (linear) convergence according to formula (4.29). However, $f'(x^*) \neq 0$ is not the necessary condition for the convergence.

The Steffensen method, defined by formula

$$x^{j+1} = x^j - \frac{f^2(x^j)}{f[x^j + f(x^j)] - f(x^j)},$$

can also be considered as a Newton-like method. The reason is that the last formula takes form (5.23) of the Newton method because

$$f[x^j + f(x^j)] - f(x^j) \approx f'(x^j)f(x^j)$$

at $f(x^j) \to 0$.

Unlike the previous Newton-like method, the Steffensen method needs only one initial approximation of the equation solution, x^0.

Fig. 5.7 shows the possibility of cycling the Newton iterative process without convergence to solution x^* if not all conditions, formulated at the beginning of this section, are satisfied (set $j = 0$ in Fig. 5.7). If the Newton method is used for solving the system of nonlinear algebraic equations (5.17), the cycling is also possible.

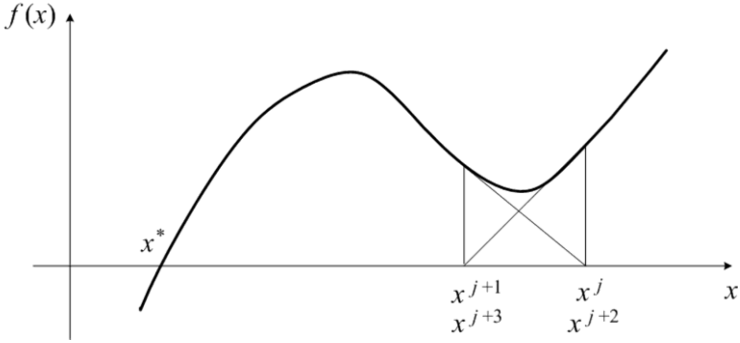


Fig. 5.7. Cycling the iterative process of the Newton (tangent) method

The Newton method, described in the previous section, may be considered as the tangent method generalization to the system of nonlinear algebraic equations (5.17). The secant and Steffensen methods also have the generalizations.

In Appendix 6, the Excel circular reference is used for realization of the tangent method for solving nonlinear equation $x - \cos x - 1.5 = 0$. In a similar way, we can realize the bisection, secant and Steffensen methods in Excel without programming in VBA.

We advise the reader to write a program for solving equation $f(x) = 0$ on segment $[a, b]$ by the Steffensen method. In this equation, $f(x)$ is a function from Appendix 4; segment $[a, b]$ is this function's domain. The user-defined form (as the user interface of the program) must include the CheckBox element for choice of the solution's initial approximation:

- $x^0 = a$ in the absence of the check mark;
- $x^0 = b$ in the presence of the check mark.

If point $x^j + f(x^j)$ appears outside segment $[a, b]$ during the solution, the secant method with $x^{j+1} = 0.5(a + b)$ must be used for continuation of solving equation $f(x) = 0$.

5.6. Modeling of the piano mechanism linking a key with hammer

In Section 5.2, we considered a method for solving the initial value problem for the system of two first-order differential equations, (5.7) and (5.8). However, the solution method does not change if the number of the system equations is greater than two, as in the initial value problem below.

Article [12] considers a series of mathematical models of the piano mechanism linking a key with hammer. We will use the model of medium complexity, in which the key and hammer motion is described by the following system of second-order differential equations:

$$\frac{d^2u_1}{dt^2} = E_2(u_1,\ u_3)\ , \tag{5.27}$$

$$\frac{d^2u_3}{dt^2} = E_4\left(u_1,\ u_3, \frac{du_3}{dt}\right), \tag{5.28}$$

where

$$E_2(u_1,\ u_3) = \frac{f - k(u_1 - u_3)}{m_1}\ , \tag{5.29}$$

$$E_4\left(u_1,\ u_3, \frac{du_3}{dt}\right) = \frac{q\left(\dfrac{du_3}{dt}\right)^2 + k(u_1 - u_3)}{p - qu_3}\ . \tag{5.30}$$

In this model, u_1 is the key's displacement downward, u_3 is the hammer's displacement forward, f is the force acting on the key, k is the elastic constant of the spring, m_1 is the key's mass, p and q are given constants, $p - qu_3 = m_3$ is the hammer's mass depending on its displacement (more exactly, m_3 is the hammer's effective mass).

We have to develop a code for solving this system of differential equations on the time segment, $0 \le t \le b$, with the following zero initial conditions:

$$u_1(0) = \frac{du_1}{dt}(0) = u_3(0) = \frac{du_3}{dt}(0) = 0 .$$

Let us introduce the key and hammer velocities, u_2 and u_4 respectively, as follows:

$$\frac{du_1}{dt} = u_2 ,$$

$$\frac{du_3}{dt} = u_4 .$$

The system of equations (5.27) and (5.28) takes the following form:

$$\begin{gathered} \frac{du_1}{dt} = E_1(u_2) , \\ \frac{du_2}{dt} = E_2(u_1,\ u_3) , \\ \frac{du_3}{dt} = E_3(u_4) , \\ \frac{du_4}{dt} = E_4(u_1,\ u_3,\ u_4) , \end{gathered} \tag{5.31}$$

where

$$\begin{gathered} E_1 = u_2 , \\ E_2 = \frac{f - k(u_1 - u_3)}{m_1} , \\ E_3 = u_4 , \\ E_4 = \frac{qu_4^2 + k(u_1 - u_3)}{p - qu_3} . \end{gathered} \tag{5.32}$$

The initial conditions become

$$u_1(0) = u_2(0) = u_3(0) = u_4(0) = 0 . \tag{5.33}$$

As in Section 5.2, we construct the grid on segment $0 \le t \le b$ for solving the formulated initial value problem and consider functions $u_1(t)$, $u_2(t)$, $u_3(t)$, $u_4(t)$ at the grid nodes. The grid step is assumed constant: $\tau = b/l$, where l is a given number of time steps.

Let $t = i\tau$ be the coordinate of a node, which is not the initial time moment, i.e., $1 \le i \le l$. To calculate the values of $u_1(t)$, $u_2(t)$, $u_3(t)$, $u_4(t)$ over known values of $u_1(t-\tau)$, $u_2(t-\tau)$, $u_3(t-\tau)$, $u_4(t-\tau)$, we must solve the following system of nonlinear algebraic equations:

$$u_1(t) - \frac{\tau}{2} E_1[u_2(t)] = u_1(t-\tau) + \frac{\tau}{2} E_1[u_2(t-\tau)],$$

$$u_2(t) - \frac{\tau}{2} E_2[u_1(t), u_3(t)] = u_2(t-\tau) + \frac{\tau}{2} E_2[u_1(t-\tau), u_3(t-\tau)],$$

$$u_3(t) - \frac{\tau}{2} E_3[u_4(t)] = u_3(t-\tau) + \frac{\tau}{2} E_3[u_4(t-\tau)],$$

$$u_4(t) - \frac{\tau}{2} E_4[u_1(t), u_3(t), u_4(t)] =$$

$$= u_4(t-\tau) + \frac{\tau}{2} E_4[u_1(t-\tau), u_3(t-\tau), u_4(t-\tau)].$$

After designations

$$\begin{gathered} u_1(t-\tau) + \frac{\tau}{2} E_1[u_2(t-\tau)] = \alpha_1, \\ u_2(t-\tau) + \frac{\tau}{2} E_2[u_1(t-\tau), u_3(t-\tau)] = \alpha_2, \\ u_3(t-\tau) + \frac{\tau}{2} E_3[u_4(t-\tau)] = \alpha_3, \\ u_4(t-\tau) + \frac{\tau}{2} E_4[u_1(t-\tau), u_3(t-\tau), u_4(t-\tau)] = \alpha_4, \end{gathered} \tag{5.34}$$

we can write this system of nonlinear algebraic equations in form (5.17) at $n = 4$:

$$\begin{gathered} f_1(x_1,\ x_2,\ x_3,\ x_4) = \alpha_1, \\ f_2(x_1,\ x_2,\ x_3,\ x_4) = \alpha_2, \\ f_3(x_1,\ x_2,\ x_3,\ x_4) = \alpha_3, \\ f_4(x_1,\ x_2,\ x_3,\ x_4) = \alpha_4, \end{gathered} \tag{5.35}$$

where $x_1 = u_1(t)$, $x_2 = u_2(t)$, $x_3 = u_3(t)$, $x_4 = u_4(t)$.

According to expressions (5.32), we have

$$f_1 = x_1 - \frac{\tau}{2} E_1(x_2) = x_1 - \frac{\tau}{2} x_2 ,$$

$$f_2 = x_2 - \frac{\tau}{2} E_2(x_1,\ x_3) = x_2 - \frac{\tau}{2} \frac{f - k(x_1 - x_3)}{m_1} ,$$

$$f_3 = x_3 - \frac{\tau}{2} E_3(x_4) = x_3 - \frac{\tau}{2} x_4 , \tag{5.36}$$

$$f_4 = x_4 - \frac{\tau}{2} E_4(x_1,\ x_3,\ x_4) = x_4 - \frac{\tau}{2} \frac{q x_4^2 + k(x_1 - x_3)}{p - q x_3} .$$

For solving the system of nonlinear algebraic equations (5.35) by the Newton method, we need expressions for the partial derivatives of functions f_1, f_2, f_3, f_4 with respect to arguments x_1, x_2, x_3, x_4, i.e., for the elements of Jacobian matrix (5.18). By using the basic rules of differentiation [3], we obtain

$$\frac{\partial f_1}{\partial x_1} = 1 , \quad \frac{\partial f_1}{\partial x_2} = -\frac{\tau}{2} , \quad \frac{\partial f_1}{\partial x_3} = \frac{\partial f_1}{\partial x_4} = 0 ,$$

$$\frac{\partial f_2}{\partial x_1} = \frac{\tau k}{2 m_1} , \quad \frac{\partial f_2}{\partial x_2} = 1 , \quad \frac{\partial f_2}{\partial x_3} = -\frac{\tau k}{2 m_1} , \quad \frac{\partial f_2}{\partial x_4} = 0 ,$$

$$\frac{\partial f_3}{\partial x_1} = \frac{\partial f_3}{\partial x_2} = 0 , \quad \frac{\partial f_3}{\partial x_3} = 1 , \quad \frac{\partial f_3}{\partial x_4} = -\frac{\tau}{2} , \tag{5.37}$$

$$\frac{\partial f_4}{\partial x_1} = -\frac{\tau}{2} \frac{k}{p - q x_3} , \quad \frac{\partial f_4}{\partial x_2} = 0 ,$$

$$\frac{\partial f_4}{\partial x_3} = \frac{\tau}{2} \frac{k(p - q x_3) - [q x_4^2 + k(x_1 - x_3)] q}{(p - q x_3)^2} , \quad \frac{\partial f_4}{\partial x_4} = 1 - \tau \frac{q x_4}{p - q x_3} .$$

Let us develop a code for solving the system of differential equations (5.31) with initial conditions (5.33). In table Listing 5.3 with the source data for the required code:

- l is the number of time steps;
- τ is the time step in seconds;
- f is the force acting on the key, in newtons;
- m_1 is the key's mass in kilograms;

- *p*, *q*, *k* are constants in expressions (5.29) and (5.30); their dimensions are as follows: $[p] = \text{kg}$, $[q] = \text{kg}/\text{m}$, $[k] = \text{N}/\text{m}$;
- ζ is a positive constant in condition (5.22) for finishing the Newton iterative process.

Listing 5.3

l	30
tau	1.00E-03
f	10
m1	0.074
p	0.406
q	18.3
k	1.16E+04
zeta	1.00E-09

As results of the execution, we must have the values of time *t* and the corresponding values of the key's and hammer's displacements and velocities, i.e., we must have u_1, $\dfrac{du_1}{dt} = u_2$, u_3, $\dfrac{du_3}{dt} = u_4$ as functions of time *t*.

Below are the `main` program and the `e_functions` and `fx_jacobian` subroutines for solving the initial value problem.

Listing 5.4

```
Dim tau As Double
Dim f As Double, m1 As Double
Dim p As Double, q As Double
Dim k As Double
Dim tau2 As Double

Sub main()
    Dim l As Integer
    Dim zeta As Double
    Dim u(1 To 4) As Double
    Dim x(1 To 4) As Double
    Dim z(1 To 4) As Double
    Dim e(1 To 4) As Double
    Dim a(1 To 4, 1 To 4) As Double
    Dim b(1 To 4) As Double
    Dim alpha(1 To 4) As Double
    Dim m As Integer, i As Integer, j As Integer
    Dim max As Double
    Dim sb As String, se As String
    l = Selection.Cells(1, 2)
    tau = Selection.Cells(2, 2)
    f = Selection.Cells(3, 2)
    m1 = Selection.Cells(4, 2)
    p = Selection.Cells(5, 2)
    q = Selection.Cells(6, 2)
    k = Selection.Cells(7, 2)
    zeta = Selection.Cells(8, 2)
    tau2 = tau / 2
    For m = 1 To 4
1:      u(m) = 0                      'values at t = 0
    Next m
    Selection.Cells(9, 1) = "t"
    Selection.Cells(9, 2) = "u1"
    Selection.Cells(9, 3) = "u2"
    Selection.Cells(9, 4) = "u3"
    Selection.Cells(9, 5) = "t"
    Selection.Cells(9, 6) = "u4"
    Selection.Cells(9, 7) = "j max"
    Selection.Cells(10, 1) = 0
    Selection.Cells(10, 2) = u(1)
    Selection.Cells(10, 3) = u(2)
    Selection.Cells(10, 4) = u(3)
    Selection.Cells(10, 5) = 0
    Selection.Cells(10, 6) = u(4)
    For i = 11 To 10 + l     'movement along time axis
2:      Call e_functions(u, e)
        For m = 1 To 4
3:          alpha(m) = u(m) + tau2 * e(m)
4:          x(m) = u(m) + tau * e(m)
        Next m
        For j = 1 To 1000           'Newton iterations
5:          Call fx_jacobian(x, a)
6:          Call e_functions(x, e)
            For m = 1 To 4
7:              b(m) = alpha(m) - (x(m) - tau2 * e(m))
            Next m
8:          Call gauss(4, a, b, z)
            For m = 1 To 4
9:              x(m) = x(m) + z(m)
            Next m
            max = 0
            For m = 1 To 4
                If Abs(z(m)) > max Then _
                max = Abs(z(m))
            Next m
10:         If max < zeta Then Exit For
        Next j
        For m = 1 To 4
11:         u(m) = x(m)
        Next m
        Selection.Cells(i, 1) = (i - 10) * tau
        Selection.Cells(i, 2) = u(1)
        Selection.Cells(i, 3) = u(2)
        Selection.Cells(i, 4) = u(3)
        Selection.Cells(i, 5) = (i - 10) * tau
        Selection.Cells(i, 6) = u(4)
        Selection.Cells(i, 7) = j
    Next i
    sb = Selection.Cells(10, 1).Address
    se = Selection.Cells(10 + l, 2).Address
12: Call graph(sb, se, "t, s", "u1, m")
    sb = Selection.Cells(10, 5).Address
    se = Selection.Cells(10 + l, 6).Address
13: Call graph(sb, se, "t, s", "u4, m/s")
    Range("O36").Select
End Sub

Sub e_functions(ByRef x() As Double, _
    ByRef e() As Double)
    e(1) = x(2)
    e(2) = (f - k * (x(1) - x(3))) / m1
    e(3) = x(4)
    e(4) = (q * x(4) ^ 2 + k * (x(1) - x(3))) / _
    (p - q * x(3))
End Sub

Sub fx_jacobian(ByRef x() As Double, _
    ByRef fx() As Double)
    Dim m3 As Double
    fx(1, 1) = 1
    fx(1, 2) = -tau2
    fx(1, 3) = 0: fx(1, 4) = 0
    fx(2, 1) = tau2 * k / m1
    fx(2, 2) = 1
    fx(2, 3) = -tau2 * k / m1
    fx(2, 4) = 0
    fx(3, 1) = 0: fx(3, 2) = 0
    fx(3, 3) = 1
    fx(3, 4) = -tau2
    m3 = p - q * x(3)
    fx(4, 1) = -tau2 * k / m3
    fx(4, 2) = 0
    fx(4, 3) = tau2 * (k * m3 - _
    (q * x(4) ^ 2 + k * (x(1) - x(3))) * q) / m3 ^ 2
    fx(4, 4) = 1 - tau * q * x(4) / m3
End Sub
```

The u array contains values u_1, u_2, u_3, u_4 of the solution. Operator 1 sets the solution's values at $t = 0$ according to (5.33). The x array contains the solution approximations.

By means of the i cycle, the movement along the time axis is performed with step τ. The u array, used in the call of the e_functions subroutine (operator 2), contains values u_1, u_2, u_3, u_4 of the solution at the $t - \tau$ moment of time. This subroutine calculates corresponding values E_1, E_2, E_3, E_4 of the e array according to formulas (5.32). Operator 3 calculates values α_1, α_2, α_3, α_4 of the alpha array according to formulas (5.34). Operator 4 calculates the x array of the initial approximation of the solution at the t moment according to formulas

$$x_1 = u_1(t-\tau) + \tau\, E_1[u_2(t-\tau)],$$

$$x_2 = u_2(t-\tau) + \tau\, E_2[u_1(t-\tau), u_3(t-\tau)],$$

$$x_3 = u_3(t-\tau) + \tau\, E_3[u_4(t-\tau)],$$

$$x_4 = u_4(t-\tau) + \tau\, E_4[u_1(t-\tau), u_3(t-\tau), u_4(t-\tau)].$$

Further (inside the i cycle being under consideration), the j cycle of the Newton iterations is performed. In the j cycle:

1) by calling the fx_jacobian subroutine (operator 5), the a array is calculated according to formulas (5.18) and (5.37);

2) after calling the e_functions subroutine (operator 6), the b array is calculated according to formulas (5.19) and (5.36) by means of operator 7;

3) by calling the gauss subroutine (operator 8), the system of linear algebraic equations (5.20), defined by arrays a and b, is solved;

4) by means of operator 9, the next approximation of the solution at the *t* moment is calculated according to formulas (5.21);

5) operator 10 checks if condition (5.22) is satisfied for termination of the Newton iterations.

When terminating the iterative process (after leaving the j cycle), values u_1, u_2, u_3, u_4 of the solution at the *t* moment (which are in the x array) are assigned to the u array's elements by means of operator 11. These values (together with the value of *t* and the number of the Newton iterations, j_{max}) are put into cells of Excel.

At the end of the main program (after leaving the i cycle), the graphs of calculated dependences $u_1(t)$ and $u_4(t)$ are constructed automatically by means of the graph subroutine. Operators 12 and 13 are the calls of this subroutine.

The declarations of e_functions and fx_jacobian are located below the main program (see Listing 5.4).

The e_functions subroutine calculates values E_1, E_2, E_3, E_4 according to formulas (5.32). The parameters of this subroutine have the following sense:

- x is an array of values u_1, u_2, u_3, u_4;
- e is a one-dimensional array intended for the required values.

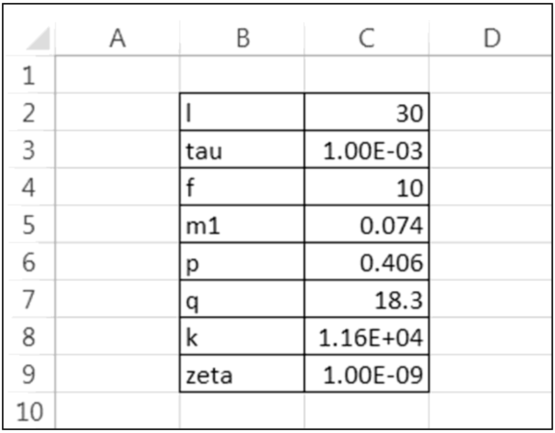
The fx_jacobian subroutine calculates the Jacobian matrix of functions f_1, f_2, f_3, f_4 according to formulas (5.18) and (5.37). The parameters have the following sense:

- x is an array of arguments x_1, x_2, x_3, x_4;
- fx is a two-dimensional array intended for the required values of the partial derivatives.

The source data for code Listing 5.4 are the values given in table Listing 5.3 considered above. The corresponding Excel table, depicted in Fig. 5.8, must be selected before the code execution.

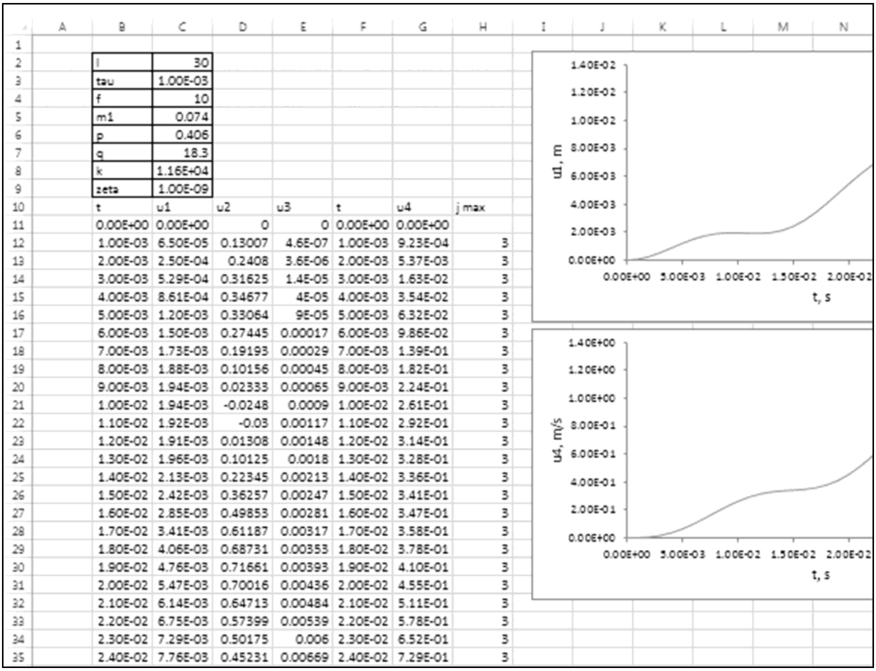
Upon termination of the execution, the values of *t*, u_1, u_2, u_3, u_4 and j_{max} are located in the corresponding columns on the Excel worksheet (Fig. 5.9). Besides, the $u_1(t)$ and $u_4(t)$ graphs are located on this worksheet; Fig. 5.10 shows them completely.

The graphs, depicted in Fig. 5.10, are results of the mentioned calls of the graph subroutine.



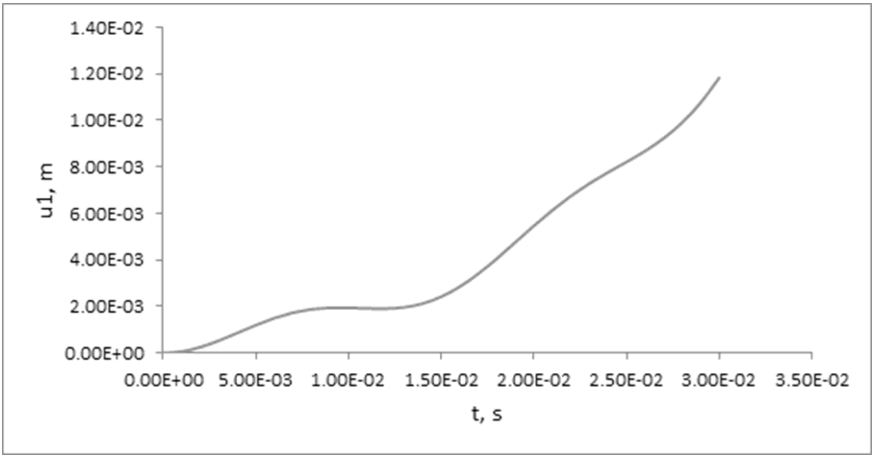
	A	B	C	D
1				
2		l	30	
3		tau	1.00E-03	
4		f	10	
5		m1	0.074	
6		p	0.406	
7		q	18.3	
8		k	1.16E+04	
9		zeta	1.00E-09	
10				

Fig. 5.8. The Excel table with the source data

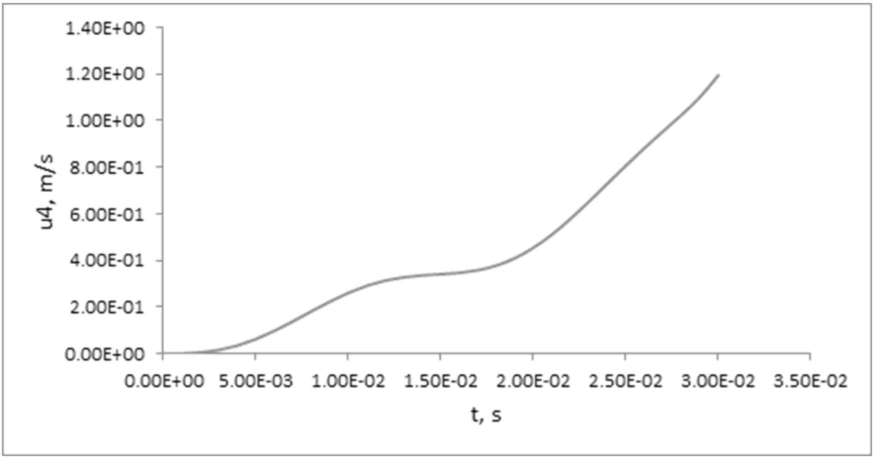


	A	B	C	D	E	F	G	H
1								
2		l	30					
3		tau	1.00E-03					
4		f	10					
5		m1	0.074					
6		p	0.406					
7		q	18.3					
8		k	1.16E+04					
9		zeta	1.00E-09					
10		t	u1	u2	u3	t	u4	j max
11		0.00E+00	0.00E+00	0	0	0.00E+00	0.00E+00	
12		1.00E-03	6.50E-05	0.13007	4.6E-07	1.00E-03	9.23E-04	3
13		2.00E-03	2.50E-04	0.2408	3.6E-06	2.00E-03	5.37E-03	3
14		3.00E-03	5.29E-04	0.31625	1.4E-05	3.00E-03	1.63E-02	3
15		4.00E-03	8.61E-04	0.34677	4E-05	4.00E-03	3.54E-02	3
16		5.00E-03	1.20E-03	0.33064	9E-05	5.00E-03	6.32E-02	3
17		6.00E-03	1.50E-03	0.27445	0.00017	6.00E-03	9.86E-02	3
18		7.00E-03	1.73E-03	0.19193	0.00029	7.00E-03	1.39E-01	3
19		8.00E-03	1.88E-03	0.10156	0.00045	8.00E-03	1.82E-01	3
20		9.00E-03	1.94E-03	0.02333	0.00065	9.00E-03	2.24E-01	3
21		1.00E-02	1.94E-03	-0.0248	0.0009	1.00E-02	2.61E-01	3
22		1.10E-02	1.92E-03	-0.03	0.00117	1.10E-02	2.92E-01	3
23		1.20E-02	1.91E-03	0.01308	0.00148	1.20E-02	3.14E-01	3
24		1.30E-02	1.96E-03	0.10125	0.0018	1.30E-02	3.28E-01	3
25		1.40E-02	2.13E-03	0.22345	0.00213	1.40E-02	3.36E-01	3
26		1.50E-02	2.42E-03	0.36257	0.00247	1.50E-02	3.41E-01	3
27		1.60E-02	2.85E-03	0.49853	0.00281	1.60E-02	3.47E-01	3
28		1.70E-02	3.41E-03	0.61187	0.00317	1.70E-02	3.58E-01	3
29		1.80E-02	4.06E-03	0.68731	0.00353	1.80E-02	3.78E-01	3
30		1.90E-02	4.76E-03	0.71661	0.00393	1.90E-02	4.10E-01	3
31		2.00E-02	5.47E-03	0.70016	0.00436	2.00E-02	4.55E-01	3
32		2.10E-02	6.14E-03	0.64713	0.00484	2.10E-02	5.11E-01	3
33		2.20E-02	6.75E-03	0.57399	0.00539	2.20E-02	5.78E-01	3
34		2.30E-02	7.29E-03	0.50175	0.006	2.30E-02	6.52E-01	3
35		2.40E-02	7.76E-03	0.45231	0.00669	2.40E-02	7.29E-01	3

Fig. 5.9. The code execution results



a



b

Fig. 5.10. The automatically created graphs: a — $u_1(t)$; b — $u_4(t)$

5.7. Definition of linear spline

The simplest spline, linear, is defined as follows.

Let values f_k, f_{k+1}, f_{k+2}, ..., f_{r-2}, f_{r-1}, f_r of grid function $f(x)$ be given at the nodes of grid $x_k < x_{k+1} < x_{k+2} < ... < x_{r-2} < x_{r-1} < x_r$. A linear spline (or first-degree spline, Fig. 5.11) is function $L(x)$, which satisfies the following conditions:

1) on each elementary segment $x_{i-1} \le x \le x_i$ $(k+1 \le i \le r)$, the spline coincides with a first-degree polynomial (generally, the polynomials are different on different elementary segments);

2) at the grid nodes, the spline has the corresponding grid function values: $L(x_i) = f_i$.

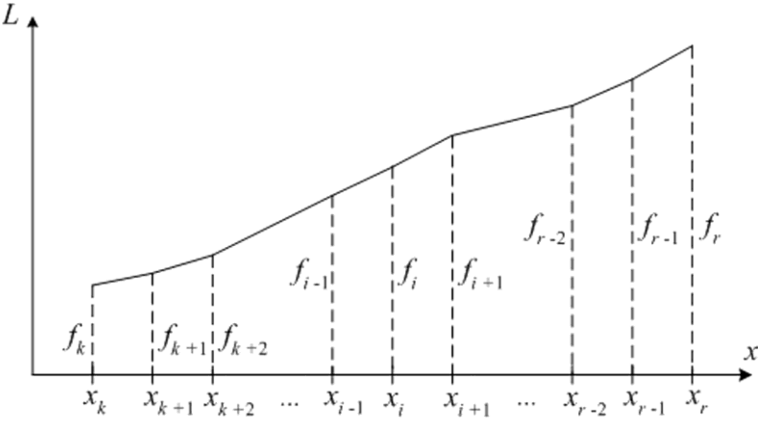


Fig. 5.11. The linear spline graph

According to condition (2), the $L(x)$ graph passes through points (x_k, f_k), (x_{k+1}, f_{k+1}), (x_{k+2}, f_{k+2}), . . . , (x_{r-2}, f_{r-2}), (x_{r-1}, f_{r-1}), (x_r, f_r).

The definition of linear spline leads to the following expression for the $L(x)$ function on elementary segment $[x_{i-1}, x_i]$:

$$L(x) = f_{i-1}\frac{x_i - x}{h_i} + f_i\frac{x - x_{i-1}}{h_i}, \tag{5.38}$$

where $h_i = x_i - x_{i-1}$ is the elementary segment's length or the grid step, $k + 1 \le i \le r$. Thus, the linear spline is the linearly interpolated tabular function.

Formula (5.38) was already used in programs Listing 4.11 and Listing 4.13: see formula (4.34) on p. 320.

The error of interpolating the $f(x)$ function (and its derivative) by the $L(x)$ spline (and by its derivative) is determined by the following expression:

$$f^{(n)}(x) - L^{(n)}(x) = O(h_{max}^{2-n}), \tag{5.39}$$

where $h_{max} = \max\limits_{k+1 \le i \le r}\{h_i\}$ is the maximum grid step ($h_{max} \to 0$), $n = 0, 1$ is the derivative order, $f^{(0)}(x) = f(x)$, $L^{(0)}(x) = L(x)$.

It is obvious that the mathematical constructions of the linear and cubic splines are similar. However, the cubic interpolation is much more exact than the linear interpolation: the interpolation errors differ by two orders of h_{max}. To make sure, we have to compare expressions (4.19) and (5.39).

The linear spline can be used for interpolation, differentiation and integration of the grid (tabular) function. For this purpose, we can use the `si` subroutine and the `ios` function, developed for the cubic spline construction, but all elements of the `M` array (containing the spline moments) must be zero in the calls of these user-defined procedures.

When the moments are nullified, formula (4.20), realized in the `ios` function, takes the following form:

$$\int_a^b f(x)dx \approx \int_a^b L(x)dx = \sum_{i=k+1}^{r}\frac{f_{i-1} + f_i}{2}h_i\ . \tag{5.40}$$

It is obvious that summand $\frac{f_{i-1} + f_i}{2}h_i$ is equal to the area of a trapezium with height $h_i = x_i - x_{i-1}$ and bases f_{i-1} and f_i. Therefore, integration formula (5.40) is widely known as ***the trapezoidal method***.

Formula (5.39) gives the following estimation of the error of numerical integration by the trapezoidal method:

$$\int_a^b f(x)dx - \int_a^b L(x)dx = \int_a^b f(x)dx - \sum_{i=k+1}^{r} \frac{f_{i-1}+f_i}{2} h_i = O(h_{max}^2),$$

where $h_{max} \to 0$ is the maximum grid step.

The linear spline can be used in the noniterative method for solving the nonlinear algebraic equation (instead of the cubic spline, Section 4.6). This leads to a simplification of the program, but also to a deterioration in accuracy of the equation solution.

We advise the reader to write a program for calculating the values of function

$$u(x) = \left[1 - \int_a^x f(y)dy \right]^{-1} \tag{5.41}$$

at the nodes of a uniform grid on segment $[a, b]$. In this formula, $f(x)$ is a function from Appendix 4; segment $[a, b]$ is the $f(x)$ function's domain. The integration must be performed by the user-defined `ios` function, as the program realization of formula (5.40).

Function (5.41) is the analytical solution of the Bernoulli differential equation [3],

$$\frac{du}{dx} = f(x)u^2,$$

on segment $[a, b]$ under initial condition $u(a) = 1$. On p. 361, we spoke about the numerical solution of this initial value problem.

We advise the reader to write a program, similar to Listing 4.11, for solving equation $f(x) = 0$ on segment $[a, b]$ by the noniterative method based on the linear spline. In this equation, $f(x)$ is a function from Appendix 4; segment $[a, b]$ is this function's domain. Uniform grid $a = x_0 < x_1 < x_2 < ... < x_{n-2} < x_{n-1} < x_n = b$ must be used.

In the next section, the linear spline will be used in the least-squares method. Besides, this mathematical construction appears in the sound insulation problem.

5.8. The least-squares method

The linear spline can be used in the least-squares method. We will consider this question on an example of the following table from Task 8.1 in book [13].

The source data

The land plot number (j)	The land quality mark (x_j)	The wheat productivity (u_j), centner per hectare
1	30	23.5
2	35	23.7
3	35	24.0
4	38	26.7
5	29	24.3
6	40	28.8
7	45	33.5
8	37	27.6
9	35	23.0
10	40	29.4
11	50	30.5
12	52	35.0

The above table gives the values characterizing the land quality and wheat productivity for each of $\nu = 12$ land plots. The least-squares method allows us to establish the functional dependence of the wheat productivity, *u*, on the land quality mark, *x*.

Let $F(x)$ be an unknown functional dependence (of a given type) defined up to parameters c_1, c_2, ..., c_n. More precisely, this dependence looks like $F(c_1, c_2, ..., c_n, x)$, where c_1, c_2, ..., c_n and *x* are the function parameters and argument, respectively.

According to the least-squares method under consideration, $F(x)$ is the required functional dependence of productivity u on mark x if the parameters of $F(x)$ are equal to the coordinates of the minimum point of function

$$G(c_1, c_2, ..., c_n) = \sum_{j=1}^{\nu} [F(x_j) - u_j]^2 \qquad (5.42)$$

on c_1, c_2, ..., c_n.

Let us assume that:

1) $F(x)$ is linear spline $L(x)$ at a grid on the *x* axis;

2) the *n* number of the grid nodes and their location are given: $z_1 < ... < z_\kappa < ... < z_n$ are the coordinates of the grid nodes on the *x* axis;

3) parameters c_1, c_2, ..., c_n of $F(x)$ are values L_1, ..., L_κ, ..., L_n of the $L(x)$ spline at grid nodes z_1, ..., z_κ, ..., z_n, respectively.

According to the least-squares method, we have to find the minimum point of function

$$G(L_1, ..., L_\kappa, ..., L_n) = \sum_{j=1}^{\nu} [L(x_j) - u_j]^2 \qquad (5.43)$$

on L_1, ..., L_κ, ..., L_n.

In Section 6.12, we will minimize this function iteratively (by subroutines `mini` and `minim`). Here, we will consider a noniterative method to find the required minimum point, which is based on the concept of fundamental spline.

According to the necessary condition for an extreme value [3], the minimum points of function (5.43) belong to the set of points that are solutions of the system of equations

$$\begin{aligned} &\frac{\partial G}{\partial L_1} = 0, \\ &. \quad . \quad . \\ &\frac{\partial G}{\partial L_i} = 0, \\ &. \quad . \quad . \\ &\frac{\partial G}{\partial L_n} = 0. \end{aligned} \qquad (5.44)$$

Therefore, for finding the minimum point of function (5.43), we have to solve the last system and then to analyse the solution results.

The solution of system (5.44) begins with the following concept of fundamental spline.

The fundamental spline, $\lambda_\kappa(x)$, is a spline equaling to 1 at the z_κ node of grid $z_1 < ... < z_\kappa < ... < z_n$ and 0 at all other $n-1$ nodes. Because $1 \le \kappa \le n$, we have *n* fundamental splines $\lambda_1(x)$, ..., $\lambda_\kappa(x)$, ..., $\lambda_n(x)$.

If the fundamental splines are linear, then (according to the fundamental spline definition) an arbitrary linear spline can be written as follows:

$$L(x) = L_1\lambda_1(x) + ... + L_\kappa\lambda_\kappa(x) + ... + L_n\lambda_n(x) =$$

$$= L_1\lambda_1(x) + ... + L_i\lambda_i(x) + ... + L_n\lambda_n(x).$$

Using the last expression and formula (5.43), we have the following chain of equalities:

$$0.5\frac{\partial G}{\partial L_i} = \sum_{j=1}^{\nu}\{[L(x_j) - u_j]\partial L(x_j)/\partial L_i\} =$$

$$= \sum_{j=1}^{\nu}\{\lambda_i(x_j)[L(x_j) - u_j]\} =$$

$$= \sum_{j=1}^{\nu}\{\lambda_i(x_j)[L_1\lambda_1(x_j) + ... + L_\kappa\lambda_\kappa(x_j) + ... + L_n\lambda_n(x_j) - u_j]\} =$$

$$= L_1\sum_{j=1}^{\nu}[\lambda_i(x_j)\lambda_1(x_j)] + ... + L_\kappa\sum_{j=1}^{\nu}[\lambda_i(x_j)\lambda_\kappa(x_j)] + ... +$$

$$+ L_n\sum_{j=1}^{\nu}[\lambda_i(x_j)\lambda_n(x_j)] - \sum_{j=1}^{\nu}[\lambda_i(x_j)u_j] =$$

$$= L_1\sum_{j=1}^{\nu}(\lambda_{ij}\lambda_{1j}) + ... + L_\kappa\sum_{j=1}^{\nu}(\lambda_{ij}\lambda_{\kappa j}) + ... + L_n\sum_{j=1}^{\nu}(\lambda_{ij}\lambda_{nj}) -$$

$$- \sum_{j=1}^{\nu}(\lambda_{ij}u_j), \qquad (5.45)$$

where

$$\lambda_{ij} = \lambda_i(x_j), \qquad (5.46)$$

$$1 \le i \le n, \quad 1 \le j \le \nu,$$

i.e., coefficients λ_{ij} form the **A** matrix of size *n* × *v*:

$$\mathbf{A} = \begin{bmatrix} \lambda_{11} & \lambda_{12} & \dots & \lambda_{1\nu} \\ \lambda_{21} & \lambda_{22} & \dots & \lambda_{2\nu} \\ . & . & . & . \\ \lambda_{n1} & \lambda_{n2} & \dots & \lambda_{n\nu} \end{bmatrix}. \tag{5.47}$$

Due to (5.45), equations (5.44) become linear algebraic equations

$$\begin{array}{ccccccccccc} q_{11}L_1 & + & \dots & + & q_{1\kappa}L_\kappa & + & \dots & + & q_{1n}L_n & = & r_1, \\ . & . & . & . & . & . & . & . & . & . & . \\ q_{i1}L_1 & + & \dots & + & q_{i\kappa}L_\kappa & + & \dots & + & q_{in}L_n & = & r_i, \\ . & . & . & . & . & . & . & . & . & . & . \\ q_{n1}L_1 & + & \dots & + & q_{n\kappa}L_\kappa & + & \dots & + & q_{nn}L_n & = & r_n, \end{array} \tag{5.48}$$

where

$$q_{i\kappa} = \sum_{j=1}^{\nu} (\lambda_{ij}\lambda_{\kappa j}), \quad r_i = \sum_{j=1}^{\nu} (\lambda_{ij}u_j). \tag{5.49}$$

The system of linear algebraic equations (5.48) can be written as matrix equation

$$\mathbf{QL} = \mathbf{R}, \tag{5.50}$$

where **Q** is the system matrix, **R** is the vector of the right-hand sides, **L** is the vector of the unknown variables:

$$\mathbf{Q} = \begin{bmatrix} q_{11} & \dots & q_{1\kappa} & \dots & q_{1n} \\ . & . & . & . & . \\ q_{i1} & \dots & q_{i\kappa} & \dots & q_{in} \\ . & . & . & . & . \\ q_{n1} & \dots & q_{n\kappa} & \dots & q_{nn} \end{bmatrix}, \quad \mathbf{R} = \begin{bmatrix} r_1 \\ \dots \\ r_i \\ \dots \\ r_n \end{bmatrix}, \quad \mathbf{L} = \begin{bmatrix} L_1 \\ \dots \\ L_\kappa \\ \dots \\ L_n \end{bmatrix}. \tag{5.51}$$

The system of linear algebraic equations (5.48) can be solved for unknown L_1, ..., L_κ, ..., L_n (the linear spline values at the grid nodes) by the Gaussian elimination method, i.e., by means of subroutine `gaus` or `gauss`.

5.9. Program to determine the dependence of the wheat productivity on the land quality

Let us develop a program for computing values L_1, ..., L_κ, ..., L_n of the linear spline at the nodes of grid $z_1 < ... < z_\kappa < ... < z_n$ on the *x* axis, which is based on the theoretical material of the previous section.

In the source data table given below:

- *n* is the number of the grid nodes;
- the *Z* column contains the *x* coordinates of the grid nodes;
- the remaining two columns (*Mark* and *Productivity*) are the same as in table “The source data” of the previous section.

Listing 5.5

n	3	
Mark	Productivity	Z
30	23.5	25
35	23.7	45
35	24.0	55
38	26.7	
29	24.3	
40	28.8	
45	33.5	
37	27.6	
35	23.0	
40	29.4	
50	30.5	
52	35.0	

The program for solving this task follows:

Listing 5.6

```
Sub main()
    Dim X() As Double
    Dim U() As Double
    Dim Z() As Double
```

5.9. Program to determine the dependence of the wheat productivity on the land quality

```
    Dim lambda() As Double
    Dim MOM() As Double
    Dim A() As Double
    Dim Q() As Double
    Dim R() As Double
    Dim L() As Double
    Dim m As Integer
    Dim n As Integer
    Dim j As Integer
    Dim i As Integer, k As Integer
    m = Selection.Rows.Count         'quantity of rows
    n = Selection.Cells(1, 2)        'number of nodes
    ReDim X(3 To m)
    ReDim U(3 To m)
    ReDim Z(1 To n)
    ReDim lambda(1 To n)
    ReDim MOM(1 To n)
    ReDim A(1 To n, 3 To m)
    ReDim Q(1 To n, 1 To n)
    ReDim R(1 To n)
    ReDim L(1 To n)
    For j = 3 To m
        X(j) = Selection.Cells(j, 1)
        U(j) = Selection.Cells(j, 2)
    Next j
'Preparations for spline interpolation:
    For i = 1 To n
        Z(i) = Selection.Cells(2 + i, 3)
0:      MOM(i) = 0
    Next i
'Forming matrix A:
    For i = 1 To n
        For k = 1 To n              'forming i-th spline
            If k = i Then
                lambda(k) = 1
            Else
                lambda(k) = 0
            End If
        Next k
        For j = 3 To m
                    'calculating values of i-th spline
1:          Call si(1, n, Z, lambda, MOM, X(j), _
            A(i, j))
        Next j
    Next i
'Forming matrix Q and vector R:
    For i = 1 To n
        For k = 1 To n
            Q(i, k) = 0
            For j = 3 To m
                Q(i, k) = Q(i, k) + A(i, j) * A(k, j)
            Next j
        Next k
        R(i) = 0
        For j = 3 To m
            R(i) = R(i) + A(i, j) * U(j)
        Next j
    Next i
'Solving system of linear algebraic equations:
2:  Call gauss(n, Q, R, L)
'Outputting results:
    Selection.Cells(2, 4) = "L"
    For i = 1 To n
        Selection.Cells(2 + i, 4) = L(i)
    Next i
End Sub
```

The **A** matrix (array `A`) is formed according to formulas (5.46) and (5.47) by means of the `si` subroutine used for the interpolation (operator `1`). Because the fundamental splines, $\lambda_1(x)$, ..., $\lambda_i(x)$, ..., $\lambda_n(x)$, are linear (not cubic), the `MOM` array of the moments is nulled by operator `0`.

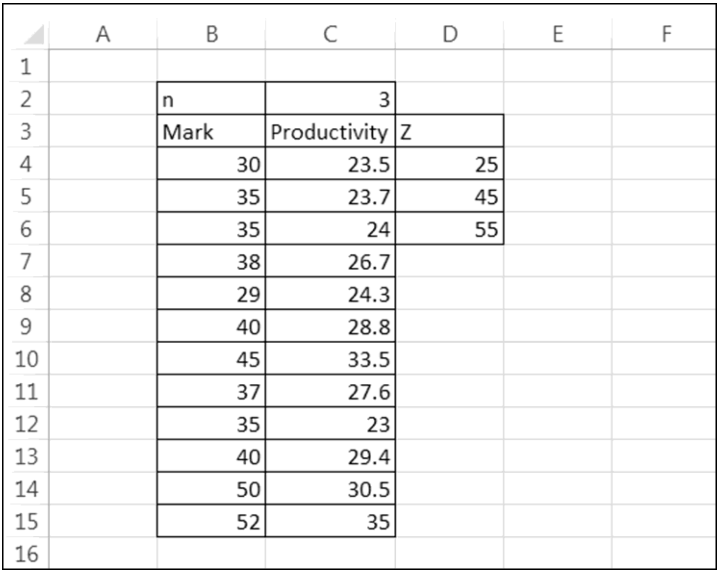
Matrix **Q** and vector **R** are formed according to formulas (5.49) and (5.51). Matrix equation (5.50) is solved by calling the `gauss` subroutine (operator `2`). We can use the simpler subroutine by replacing the subroutine name with `gaus` in operator `2`.

The remaining operators of the program put the **L** solution of matrix equation (5.50) into cells of Excel.

The source data for program Listing 5.6 are given in table Listing 5.5 (Fig. 5.12). Before the program execution, we must select this Excel table (range B2:D15).

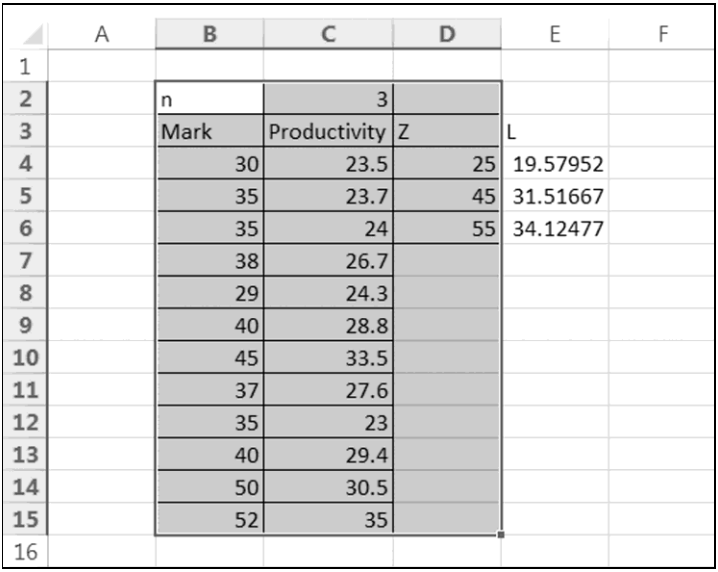
The execution results are the linear spline values at the grid nodes, which are in the *L* column located near the *Z* column (Fig. 5.13). Fig. 5.14 shows the experimental points and the calculated line.

5.9. Program to determine the dependence of the wheat productivity on the land quality



	A	B	C	D	E	F
1						
2		n	3			
3		Mark	Productivity	Z		
4		30	23.5	25		
5		35	23.7	45		
6		35	24	55		
7		38	26.7			
8		29	24.3			
9		40	28.8			
10		45	33.5			
11		37	27.6			
12		35	23			
13		40	29.4			
14		50	30.5			
15		52	35			
16						

Fig. 5.12. The Excel table with the source data



	A	B	C	D	E	F
1						
2		n	3			
3		Mark	Productivity	Z	L	
4		30	23.5	25	19.57952	
5		35	23.7	45	31.51667	
6		35	24	55	34.12477	
7		38	26.7			
8		29	24.3			
9		40	28.8			
10		45	33.5			
11		37	27.6			
12		35	23			
13		40	29.4			
14		50	30.5			
15		52	35			
16						

Fig. 5.13. The program execution results

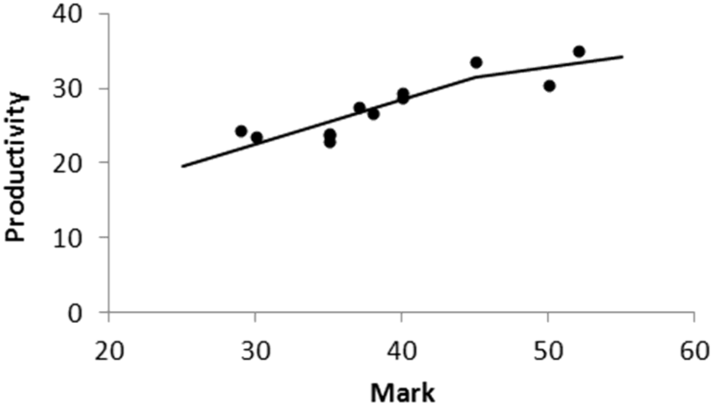


Fig. 5.14. Experimental points and the fractured line of functional dependence of the wheat productivity (in units of centner per hectare) on the land quality mark

According to Fig. 5.14, we really determined the minimum point of function (5.43).

The value of spline $L(x)$ at point *x*, located between nodes z_{i-1} and z_i (i = 2, 3, ..., $n-1$, n), can be calculated by means of the Excel formula corresponding to mathematical formula (5.38) of the following form:

$$L(x) = \frac{(z_i - x)L_{i-1} + (x - z_{i-1})L_i}{z_i - z_{i-1}}$$

If $x < z_1$, then

$$L(x) = \frac{(z_2 - x)L_1 + (x - z_1)L_2}{z_2 - z_1} .$$

If $x > z_n$, then

$$L(x) = \frac{(z_n - x)L_{n-1} + (x - z_{n-1})L_n}{z_n - z_{n-1}} .$$

Note that the cubic spline can be used instead of the linear spline in the least-squares method. In this case, naturally, the fundamental splines are cubic.

The functional dependence of the wheat productivity on the land quality mark, determined by the least-squares method, describes the dependence of production results on factors. Such dependence is called the production function; it is the basis for many economic calculations.

5.10. The forward and backward Fourier transforms of a periodic function

Let $f(t)$ be a real-valued periodic function of time t with given period T, which describes an oscillation with frequency $1/T$. This function can be represented as the Fourier series [3]:

$$f(t) = \frac{a_0}{2} + \sum_{k=1}^{\infty} [a_k \cos(k\omega t) + b_k \sin(k\omega t)], \tag{5.52}$$

where $\omega = 2\pi/T$ is the cyclic frequency, a_0, a_k, b_k are the Fourier coefficients:

$$a_k = \frac{2}{T}\int_0^T f(t)\cos(k\omega t)dt, \tag{5.53}$$

$$k = 0, 1, 2, 3, ...,$$

$$b_k = \frac{2}{T}\int_0^T f(t)\sin(k\omega t)dt, \tag{5.54}$$

$$k = 1, 2, 3, ...$$

Let us recall several terms related to periodic function

$$f_k(t) = A_k \sin(k\omega t + \varphi_k), \tag{5.55}$$

where k is a natural number.

The oscillation, described by this formula, is called a harmonic oscillation with cyclic frequency $k\omega$. Quantities A_k, $k\omega t + \varphi_k$, φ_k are respectively the amplitude, phase and initial phase of the harmonic oscillation.

Using this terminology, we can say that (5.52) — (5.54) is the decomposition of the oscillation, described by the $f(t)$ function, into harmonic oscillations:

$$f(t) = \frac{a_0}{2} + \sum_{k=1}^{\infty} f_k(t) = \frac{a_0}{2} + \sum_{k=1}^{\infty} A_k \sin(k\omega t + \varphi_k). \tag{5.56}$$

The Fourier coefficients, amplitude and initial phase are related by the following formulas:

$$a_k = A_k \sin \varphi_k \,,$$
$$b_k = A_k \cos \varphi_k \,, \tag{5.57}$$
$$A_k = \sqrt{a_k^2 + b_k^2} \,,$$

where $k = 1, 2, 3, ...$

The calculation of the coefficients of the function's expansion into the Fourier series is called the analysis or forward Fourier transform. The analysis is defined by formulas (5.53) and (5.54).

The calculation of the $f(t)$ function's values, which correspond to the known coefficients, a_0, a_k and b_k (k = 1, 2, 3, ...), is called the synthesis or backward Fourier transform. The synthesis is defined by formula (5.52).

Along with the Fourier transform theory, which considers the $f(t)$ function known at all points of period $[0, T]$, there is a theory, which considers the $f(t)$ function known only at the nodes of a uniform grid on $[0, T]$.

Let n be a given number of equal steps or elementary segments $[t_{j-1}, t_j]$, j = 1, 2, …, n, on period $[0, T]$ and $f(t)$ be a grid function given at points $t_0 = 0$, $t_1 = T/n$, $t_2 = 2T/n$, ..., $t_{n-1} = (n-1)T/n$, $t_n = T$. We respectively denote the $f(t)$ function's values at these points as f_0, f_1, f_2, ..., f_{n-1}, f_n, at that, $f_0 = f_n$ because of the $f(t)$ periodicity. The calculation of the Fourier coefficients of grid function $f(t)$ is called the forward discrete Fourier transform. The calculation of values f_0, f_1, f_2, ..., f_{n-1}, f_n, which correspond to the Fourier coefficients, is called the backward discrete Fourier transform.

The Data Analysis add-in for Excel includes the Fourier Analysis procedure for performing the discrete Fourier transform. We will consider the use of this procedure on an example of function

$$f(t) = 1 + \sin(\omega t) + \cos(2\omega t)\,. \tag{5.58}$$

Into cells A1:A8 on an Excel worksheet, we put this function's values at the first $n = 8$ points (ωt = 0, π/4, π/2, 3π/4, π, 5π/4, 3π/2, 7π/4) of the period with 9 equidistant points (0, π/4, π/2, 3π/4, π, 5π/4, 3π/2, 7π/4, 2π), i.e., we enter numbers 2, 1.707107, 1, 1.707107, 2, 0.292893, -1, 0.292893 (Fig. 5.15).

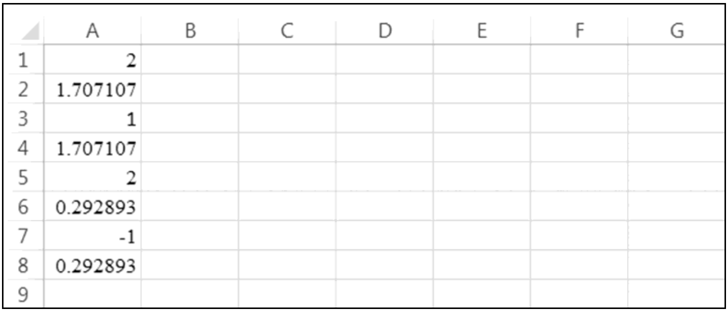
Let us fulfill the following operations:

1) *Data* > *Data Analysis* > *Fourier Analysis* > *OK*;

2) in the *Fourier Analysis* window opened, enter \$A\$1:\$A\$8 into text box *Input Range*;

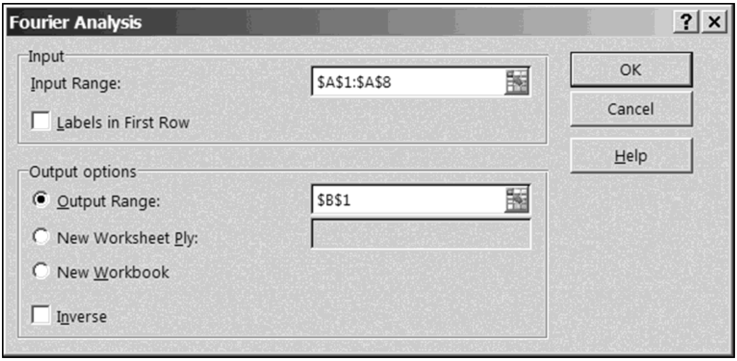
3) after activating text box *Output Range*, enter \$B\$1 into it (Fig. 5.16);

4) click on the *OK* button to get the procedure execution results (Fig. 5.17).



	A	B	C	D	E	F	G
1	2						
2	1.707107						
3	1						
4	1.707107						
5	2						
6	0.292893						
7	-1						
8	0.292893						
9							

Fig. 5.15. The Excel worksheet with the Fourier analysis source data



Input

Input Range: $A$1:$A$8

□ Labels in First Row

Output options

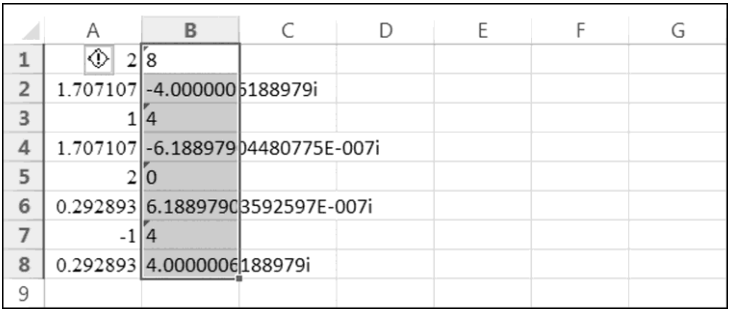
● Output Range: $B$1

○ New Worksheet Ply:

○ New Workbook

□ Inverse

Fig. 5.16. The *Fourier Analysis* window before the procedure execution



	A	B	C	D	E	F	G
1	2	8					
2	1.707107	-4.00000006188979i					
3	1	4					
4	1.707107	-6.18897904480775E-007i					
5	2	0					
6	0.292893	6.18897903592597E-007i					
7	-1	4					
8	0.292893	4.0000006188979i					
9							

Fig. 5.17. The Excel worksheet with the Fourier analysis results

If we locate the mouse pointer on an exclamation mark (near the B1 cell), the following information appears: *The number in this cell is formatted as text or preceded by an apostrophe*. We should not be afraid of this information.

According to Fig. 5.17, the calculation results (in cells B1:B8) are the coefficients of the Fourier series in complex form: letter “i” (in cells B2, B4, B6 and B8) means the imaginary unit, $i = \sqrt{-1}$.

The complex representation of the Fourier series [3] can be written as

$$f(t) = c_0 + \sum_{k=1}^{\infty} \left[c_k e^{ik\omega t} + c_{-k} e^{-ik\omega t} \right],$$

where $c_0 = a_0$, c_k and c_{-k} are complex Fourier coefficients, k = 1, 2, 3, ...

For using the procedure execution results, we have to know the following:

1) the values of coefficients c_0 and c_k in front of $e^{ik\omega t}$ (k = 1, 2, 3, 4), multiplied by $n = 8$, are respectively located in cells B1, B2, B3, B4, B5 (the first half of the set of complex Fourier coefficients);

2) the values of coefficients c_{-k} in front of $e^{-ik\omega t}$ (k = 1, 2, 3, 4), multiplied by 8, are respectively located in cells B8, B7, B6, B5 (the second half of the set of complex Fourier coefficients);

3) $c_k = c_{-k} = 0$ for $k > 4$.

For verifying it, we will use the following expressions, which follow from the Euler relation for complex numbers [3]:

$$\sin(k\omega t) = \frac{e^{ik\omega t} - e^{-ik\omega t}}{2i} = \frac{-ie^{ik\omega t} + ie^{-ik\omega t}}{2}, \tag{5.59}$$

$$\cos(k\omega t) = \frac{e^{ik\omega t} + e^{-ik\omega t}}{2}. \tag{5.60}$$

Using (5.59) at $k = 1$ and (5.60) at $k = 2$, we write expression (5.58) in form

$$f(t) = 1 - \frac{i}{2} e^{i\omega t} + \frac{1}{2} e^{i2\omega t} + \frac{1}{2} e^{-i2\omega t} + \frac{i}{2} e^{-i\omega t}. \tag{5.61}$$

Multiplying values $c_0 = 1$, $c_1 = -i/2$, $c_2 = 1/2$, $c_3 = 0$, $c_4 = c_{-4} = 0$, $c_{-3} = 0$, $c_{-2} = 1/2$, $c_{-1} = i/2$ by 8, we obtain the Fourier analysis results depicted in Fig. 5.17. Let us pay attention to the following: the value of both $8c_4$ and $8c_{-4}$ is in the B5 cell.

We see that the complex coefficients in front of functions $e^{ik\omega t}$ and $e^{-ik\omega t}$ in expression (5.61) are conjugate, i.e., differ from each other only in the sign in front of the imaginary unit. This is the property of the expansion of real-valued functions, as $f(t)$, in terms of $e^{\pm ik\omega t}$. The expansion of complex-valued functions in terms of $e^{\pm ik\omega t}$ does not have such property.

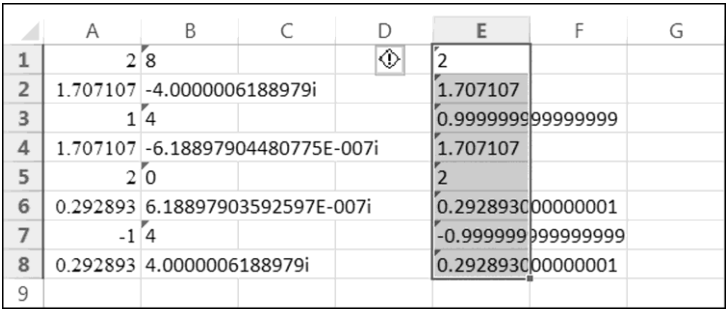
In the bottom left corner of the *Fourier Analysis* window (Fig. 5.16), we see a little square field (element CheckBox) called *Inverse*. When clicking on this field, the check mark appears in it, meaning the switching from the forward transform (analysis) to the backward transform (synthesis).

Let the worksheet, depicted in Fig. 5.17, be active. To verify the correctness of working the Fourier Analysis procedure, we fulfill the following operations:

1) *Data* > *Data Analysis* > *Fourier Analysis* > *OK*;
2) in the *Fourier Analysis* window opened, enter $B$1:$B$8 into text box *Input Range*;
3) after activating text box *Output Range*, enter $E$1 into it;
4) by clicking on element *Inverse*, set the check mark in it;
5) click on the *OK* button for starting the procedure execution.

The execution results are depicted in Fig. 5.18. We see practical coincidence of columns A and E, and that is natural because the forward and backward Fourier transforms are performed successively.

If the mouse pointer is located on an exclamation mark (near the E1 cell), the same information appears as in the case of Fig. 5.17.



	A	B	C	D	E	F	G
1	2	8			2		
2	1.707107	-4.0000006188979i			1.707107		
3	1	4			0.999999999999999		
4	1.707107	-6.18897904480775E-007i			1.707107		
5	2	0			2		
6	0.292893	6.18897903592597E-007i			0.292893000000001		
7	-1	4			-0.999999999999999		
8	0.292893	4.0000006188979i			0.292893000000001		
9							

Fig. 5.18. The Excel worksheet with results of the analysis and synthesis

The main drawback of the Fourier Analysis procedure consists in the following: the number of steps on the period is not arbitrary, it must be a power of 2, i.e., *n* must be equal to 2, 4, 8 (as in the above example), 16, 32, 64 or so on. It is because this procedure realizes the so-called fast Fourier transformation [3].

5.11. Subroutines for the forward and backward discrete Fourier transforms

In this section, we will develop subroutines for the forward and backward discrete Fourier transforms, free of the drawback formulated at the end of the previous section. These subroutines will be used in the next section.

At first, let us obtain a discrete analog of formula (5.53) at *k* = 0, 1, 2, 3, ...

We use designation

$$f(t)\cos(k\omega t) = g(t) \tag{5.62}$$

and consider periodic third-degree spline $S(t)$, which respectively assumes values g_0, g_1, g_2, ..., g_{n-1}, g_n at points 0, t_1, t_2, ..., t_{n-1}, t_n ($g_0 = g_n$). According to expression (4.20), the integral of $S(t)$ equals

$$\int_0^T S(t)dt = \sum_{j=1}^{n} \frac{g_{j-1} + g_j}{2} h - \sum_{j=1}^{n} \frac{M_{j-1} + M_j}{24} h^3 \ , \tag{5.63}$$

where $h = T/n$ is the grid step, M_0, M_1, M_2, ..., M_{n-1}, M_n are the spline moments, $M_0 = M_n$.

Let us consider equation (4.9) in form

$$\alpha_j M_{j-1} + 2M_j + \gamma_j M_{j+1} = \delta_j \, , \tag{5.64}$$

j = 1, 2, …, *n*, taking into account the periodicity of $g(t)$ and $S(t)$, i.e., the following equalities:

$$g_{n+1} = g_1, \quad M_{n+1} = M_1 ,$$

where g_{n+1} and M_{n+1} are the values of the $g(t)$ and $S''(t)$ functions, respectively, at additional node $t_{n+1} = T + h$. Because of the grid step constancy and expressions (4.10), we have

$$\alpha_j = \gamma_j = \frac{1}{2} ,$$

$$\delta_j = \frac{3}{h^2} (g_{j+1} - 2g_j + g_{j-1}) \, .$$

Equation (5.64) takes the following form:

$$\frac{1}{2}M_{j-1}+2M_j+\frac{1}{2}M_{j+1}=\frac{3}{h^2}(g_{j+1}-2g_j+g_{j-1})\,.$$

Summing both sides of the last equation over $j = 1, 2, \ldots, n$, we obtain

$$\frac{1}{2}\sum_{j=1}^{n}M_{j-1}+2\sum_{j=1}^{n}M_j+\frac{1}{2}\sum_{j=1}^{n}M_{j+1}=$$

$$=\frac{3}{h^2}\left[\sum_{j=1}^{n}g_{j+1}-2\sum_{j=1}^{n}g_j+\sum_{j=1}^{n}g_{j-1}\right]$$

or

$$\frac{1}{2}\sum_{j=1}^{n}M_j+2\sum_{j=1}^{n}M_j+\frac{1}{2}\sum_{j=1}^{n}M_j=\frac{3}{h^2}\left[\sum_{j=1}^{n}g_j-2\sum_{j=1}^{n}g_j+\sum_{j=1}^{n}g_j\right]$$

or

$$\sum_{j=1}^{n}M_j=0\,.$$

Expression (5.63) becomes simpler:

$$\int_0^T S(t)dt=\sum_{j=1}^{n}\frac{g_{j-1}+g_j}{2}h=h\sum_{j=1}^{n}g_j\,.$$

Taking into account expression (5.62), we obtain

$$a_k=\frac{2}{T}\int_0^T f(t)\cos(k\omega t)dt=$$

$$=\frac{2}{T}h\sum_{j=1}^{n}f_j\cos(k\omega\, jh)=\frac{2}{n}\sum_{j=1}^{n}f_j\cos\left(k\,j\frac{2\pi}{n}\right).\qquad(5.65)$$

This formula is the required discrete analog of formula (5.53) at $k = 0, 1, 2, 3, \ldots$

Similarly, we can obtain the following discrete analog of formula (5.54):

$$b_k=\frac{2}{n}\sum_{j=1}^{n}f_j\sin\left(k\,j\frac{2\pi}{n}\right),\qquad(5.66)$$

$k = 1, 2, 3, \ldots$

According to (5.52), (5.65) and (5.66), the formulas of the backward and forward discrete Fourier transforms can be written in the following form for an odd value of n:

$$f_j = d_0 + \sum_{k=1}^{(n-1)/2} \left[d_{2k-1} \sin\left(k\, j \frac{2\pi}{n} \right) + d_{2k} \cos\left(k\, j \frac{2\pi}{n} \right) \right], \qquad (5.67)$$

$$d_0 = \frac{1}{n} \sum_{j=1}^{n} f_j\, , \qquad (5.68)$$

$$d_{2k-1} = \frac{2}{n} \sum_{j=1}^{n} f_j \sin\left(k\, j \frac{2\pi}{n} \right), \qquad (5.69)$$

$$d_{2k} = \frac{2}{n} \sum_{j=1}^{n} f_j \cos\left(k\, j \frac{2\pi}{n} \right), \qquad (5.70)$$

where j = 0, 1, 2, ..., n in (5.67), k = 1, 2, 3, ..., $(n-1)/2$ in (5.69) and (5.70).

Similarly, we have the following for an even value of n.

The backward discrete Fourier transform is performed according to formula

$$f_j = d_0 + \sum_{k=1}^{n/2} \left[d_{2k-1} \sin\left(k\, j \frac{2\pi}{n} \right) + d_{2k} \cos\left(k\, j \frac{2\pi}{n} \right) \right], \qquad (5.71)$$

j = 0, 1, 2, ..., n.

The forward discrete Fourier transform is performed according to formulas (5.68) — (5.70) at k = 1, 2, 3, ..., $(n-2)/2$ and

$$d_{n-1} = \frac{1}{n} \sum_{j=1}^{n} f_j \sin(j\pi) = 0, \qquad (5.72)$$

$$d_n = \frac{1}{n} \sum_{j=1}^{n} f_j \cos(j\pi)\, . \qquad (5.73)$$

Let us pay attention to the following:

- the Fourier coefficients do not depend on the period size;
- the number of Fourier coefficients is equal to n, i.e., to the number of equal steps on the period (when n is even, we do not take into account coefficient d_{n-1}, which is equal to zero).

Into Module14 of the BookNM workbook, we enter the following declaration of the subroutine for the forward discrete Fourier transform:

Listing 5.7

```
Sub fouf(ByVal n, ByRef F() As Double, _
    ByRef D() As Double, Optional m)
    Dim m2 As Integer, j As Integer, k2 As Integer
    Dim a As Double, b As Double
    Dim w As Double, z As Double
    Dim c0 As Double, s0 As Double
    Dim c1 As Double, s1 As Double
    Dim c2 As Double, s2 As Double
    Const pi As Double = 3.141592654
    If IsMissing(m) Then
        m2 = n
    ElseIf m < 0 Or m > n Then
        m2 = n
    Else
        m2 = m
    End If
    a = 0
    For j = 1 To n
        a = a + F(j)
    Next j
    D(0) = a / n
    w = 2 * pi / n
    c0 = Cos(w): s0 = Sin(w)
    z = 2 / n
    c1 = 1: s1 = 0
    For k2 = 2 To m2 Step 2
        w = c1 * c0 - s1 * s0
        s1 = s1 * c0 + c1 * s0: s2 = s1
        c1 = w: c2 = c1
        a = 0: b = 0
        For j = 1 To n
            a = a + F(j) * c2
            b = b + F(j) * s2
            w = c2 * c1 - s2 * s1
            s2 = s2 * c1 + c2 * s1: c2 = w
        Next j
        D(k2 - 1) = b * z
        D(k2) = a * z
        If k2 = n Then
            D(k2 - 1) = 0
            D(k2) = D(k2) / 2
        End If
    Next k2
End Sub
```

The subroutine name (`fouf`) occurs from words “Fourier” and “forward”. The subroutine parameters have the following sense:

- `n` is the number of equal steps on the period;
- `F` is an array of the function values;
- `D` is an array intended for the Fourier coefficients;
- `m` is the doubled number of harmonic oscillations, which are of interest (this optional parameter is used for possible reducing the execution time).

The basis of this subroutine are formulas (5.68) — (5.70), (5.72) and (5.73). According to formulas (5.69) and (5.70), the forward discrete Fourier transform requires the calculation of the sine and cosine values, repeated many times. These values can be calculated in one of two ways:

- by the multiple calls of built-in functions `Sin(x)` and `Cos(x)`;
- by the single call of functions `Sin(x)` and `Cos(x)` for calculating the values of $\sin(2\pi/n)$ and $\cos(2\pi/n)$, and by subsequent usage of trigonometric formulas for calculating the values of $\sin(kj2\pi/n)$ and $\cos(kj2\pi/n)$ at $k>1$ and/or $j>1$.

To reduce the execution time, the second way is used, which is based on the following trigonometric formulas [3]:

$$\sin(\alpha+\beta)=\sin(\alpha)\cos(\beta)+\cos(\alpha)\sin(\beta)\,, \qquad (5.74)$$

$$\cos(\alpha+\beta)=\cos(\alpha)\cos(\beta)-\sin(\alpha)\sin(\beta)\,, \qquad (5.75)$$

where α and β are angles.

Into Module15 of the BookNM workbook, we enter the following declaration of the subroutine for the backward discrete Fourier transform:

Listing 5.8

```
Sub foub(ByVal n, ByRef F() As Double, _
    ByRef D() As Double, Optional m)
    Dim m2 As Integer, j As Integer, k2 As Integer
    Dim w As Double, z As Double
    Dim c0 As Double, s0 As Double
    Dim c1 As Double, s1 As Double
    Dim c2 As Double, s2 As Double
    Const pi As Double = 3.141592654
    If IsMissing(m) Then
        m2 = n
    ElseIf m < 0 Or m > n Then
        m2 = n
    Else
        m2 = m
    End If
    w = 2 * pi / n
    c0 = Cos(w): c1 = c0: c2 = c1
    s0 = Sin(w): s1 = s0: s2 = s1
    For j = 1 To n
        z = D(0)
        For k2 = 2 To m2 Step 2
            z = z + D(k2 - 1) * s2 + D(k2) * c2
            w = c2 * c1 - s2 * s1
            s2 = s2 * c1 + c2 * s1: c2 = w
        Next k2
        F(j) = z: w = c1 * c0 - s1 * s0
        s1 = s1 * c0 + c1 * s0: s2 = s1
        c1 = w: c2 = c1
    Next j
    F(0) = F(n)
End Sub
```

The subroutine name (foub) occurs from words “Fourier” and “backward”. The parameters have the following sense:

- n is the number of equal steps on the period;
- F is an array intended for the function values;
- D is an array of the Fourier coefficients;
- m is the doubled number of harmonic oscillations, which are considered.

Formulas (5.67), (5.71), (5.74) and (5.75) are used in the foub subroutine.

5.12. Solving the sound insulation problem

The sound waves, emitted by a sound source (by a vibrating body), propagate in the medium (in a solid body, liquid or gas) in the form of longitudinal oscillations of the density of the medium. The ear of an adult person perceives sound oscillations with frequency $1/T$ from 17 — 20 Hz to about 20 kHz.

To lower the sound level (i.e., the amplitude of the density oscillation), soundproof coverings are used. A technique of constructing such coverings with given acoustic parameters was developed at the Andreyev Acoustics Institute, Moscow. According to Fig. 5.19 (from page http://www.akin.ru/r_comm15.htm of the institute website), the covering is characterized by efficiency with respect to lowering the sound level. As we see, both the calculated and experimental dependences for the covering's efficiency are linear splines whose argument is the logarithm of frequency $k/T = k\omega/2\pi$ of harmonic oscillation (5.55).

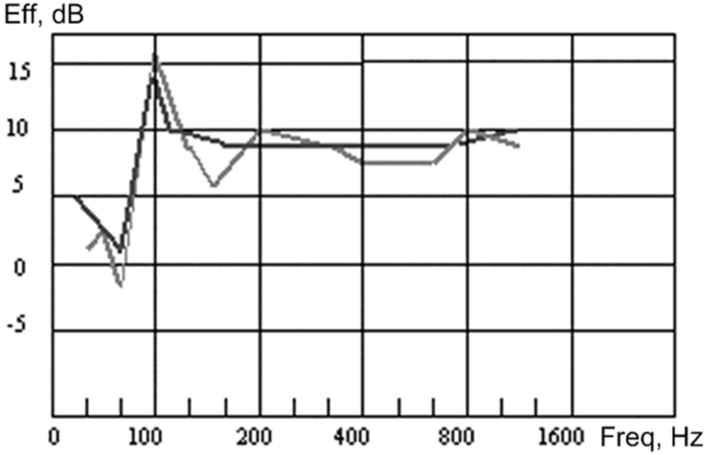


Fig. 5.19. The calculated (dark) and experimental (light) dependences of the efficiency on the frequency: between the fractures, the efficiency is a linear function of the logarithm of frequency (the base of logarithm does not matter due to proportionality of logarithms with different bases [3]: $\log_a x = M \log_b x$)

Let us return to modeling of the bar oscillation in Section 3.20. We will attach weightless membrane M$_s$ (whose area equals *A*) to the bar and consider that this membrane is the sound source, at that, dependence $u(t)$ in Fig. 3.18 is the M$_s$ membrane deviation. Besides, we will place the mechanism in a box with the soundproof covering whose efficiency is defined by the calculated (dark) dependence in Fig. 5.19.

We have to answer the following question: by how many decibels is the sound (from the M$_s$ membrane) attenuated, when it goes through the box's soundproof covering?

Outside the box, we place weightless membrane M$_r$ (whose area equals *A*) of the sound receiver; $u_r(t)$ is the periodic dependence, which characterizes the M$_r$ membrane deviation (Fig. 5.20). We have to determine difference Δ (in decibels) between the intensities of the oscillations of membranes M$_s$ and M$_r$.

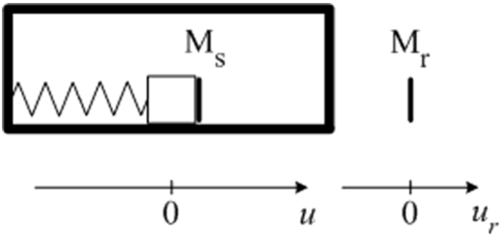


Fig. 5.20. The membranes separated by the soundproof covering

We use the following definition of the intensity of the harmonic oscillation with frequency k/T:

$$P_k = 10 \cdot \lg A_k^2 = 10 \cdot \lg (d_{2k-1}^2 + d_{2k}^2),$$

where lg is the decimal logarithm, A_k is the oscillation amplitude, d_{2k-1}, d_{2k} are the Fourier coefficients of this oscillation, *k* = 1, 2, 3, ...

The intensities of the harmonic oscillations of membranes M$_s$ and M$_r$ are given by formulas

$$P_k^s = 10 \cdot \lg[(d_{2k-1}^s)^2 + (d_{2k}^s)^2], \tag{5.76}$$

$$P_k^r = 10 \cdot \lg[(d_{2k-1}^r)^2 + (d_{2k}^r)^2], \tag{5.77}$$

where d_{2k-1}^s, d_{2k}^s and d_{2k-1}^r, d_{2k}^r are the Fourier coefficients of dependences $u(t)$ and $u_r(t)$, respectively, *k* = 1, 2, 3, ...

The efficiency (in decibels) of the soundproof covering at frequency k/T is equal to

$$\Delta_k = P_k^s - P_k^r . \tag{5.78}$$

The sound oscillation of the environment density can be written in form (5.56). Let us assume that phase $k\omega t + \varphi_k$ of harmonic oscillation (5.55) does not change when the sound goes through the soundproof covering, and amplitude A_k changes only (k = 1, 2, 3, ...). In this case, the sound attenuation is calculated as follows.

1. By means of the `fouf` subroutine, the forward discrete Fourier transform of the $u(t)$ dependence, depicted in Fig. 3.18 ($n = 25$), is performed, that is, Fourier coefficients d_0^s, d_{2k-1}^s, d_{2k}^s of this dependence are calculated, k = 1, 2, ..., 12.
2. The summary intensity of the oscillation of membrane M$_\mathrm{s}$ is calculated according to formula

$$P_s = 10 \cdot \lg\left\{ \sum_{k=1}^{12} [(d_{2k-1}^s)^2 + (d_{2k}^s)^2] \right\} . \tag{5.79}$$

3. For every frequency k/T, the values of

$$d_{2k-1}^r = d_{2k-1}^s 10^{-\Delta_k/20} , \tag{5.80}$$

$$d_{2k}^r = d_{2k}^s 10^{-\Delta_k/20} \tag{5.81}$$

are calculated, where Δ_k is the efficiency of the soundproof covering at the considered frequency, k = 1, 2, ..., 12. Formulas (5.80) and (5.81) follow from the logarithm properties [3] and formulas (5.57), (5.76) — (5.78).

4. The summary intensity of the oscillation of membrane M$_\mathrm{r}$ is calculated according to formula

$$P_r = 10 \cdot \lg\left\{ \sum_{k=1}^{12} [(d_{2k-1}^r)^2 + (d_{2k}^r)^2] \right\} . \tag{5.82}$$

5. The required sound attenuation is calculated according to the following formula:

$$\Delta = P_s - P_r . \tag{5.83}$$

To obtain dependence $u_r(t)$ of the M$_\mathrm{r}$ membrane deviation, we have to perform the backward discrete Fourier transform by means of the `foub` subroutine according to formula (5.67), which includes $d_0^r = 0$ and the values of d_1^r, d_2^r, d_3^r, d_4^r, ..., d_{23}^r, d_{24}^r calculated according to formulas (5.80) and (5.81).

As the source data for a program, intended for solving our problem, we use table Listing 5.9, which includes:

- the results of solving the oscillation equation (Fig. 3.18);
- the dependence of the covering's efficiency on the frequency.

Listing 5.9

M	0.001	
K	800	
L	100	
F	t	u
0.00E+00	0.00E+00	6.62E-04
0.00E+00	2.00E-03	-5.89E-03
0.00E+00	4.00E-03	5.89E-03
0.00E+00	6.00E-03	-1.60E-03
0.00E+00	8.00E-03	-3.07E-03
0.00E+00	1.00E-02	4.66E-03
0.00E+00	1.20E-02	-2.57E-03
0.00E+00	1.40E-02	-1.01E-03
0.00E+00	1.60E-02	3.20E-03
0.00E+00	1.80E-02	-2.67E-03
0.00E+00	2.00E-02	2.91E-04
0.00E+00	2.20E-02	1.87E-03
0.00E+00	2.40E-02	-2.27E-03
0.00E+00	2.60E-02	9.54E-04
5.00E+00	2.80E-02	8.20E-04
1.00E+01	3.00E-02	1.65E-02
5.00E+00	3.20E-02	1.77E-02
0.00E+00	3.40E-02	-1.46E-02
0.00E+00	3.60E-02	1.48E-03
0.00E+00	3.80E-02	1.03E-02
0.00E+00	4.00E-02	-1.25E-02
0.00E+00	4.20E-02	5.17E-03
0.00E+00	4.40E-02	4.59E-03
0.00E+00	4.60E-02	-9.23E-03
0.00E+00	4.80E-02	6.32E-03
0.00E+00	5.00E-02	6.62E-04

N	8
frequency	efficiency
17	5
45	1
100	13.8
126	10
158	8.8
720	8.8
1100	10
1600	10

The subtable with the dependence of the covering’s efficiency on the frequency includes:

- *N*, the number of the frequency values;
- the frequency values (in hertzs) in column *frequency*;
- the efficiency values (in decibels) in column *efficiency*, according to the dark dependence in Fig. 5.19.

The program follows:

Listing 5.10

```
Sub main()
    Dim T() As Double
    Dim U() As Double
    Dim D() As Double
    Dim m As Integer
    Dim N As Integer
    Dim j As Integer, k2 As Integer
    Dim i As Integer
    Dim sb As String, se As String
    Dim fr As Double, eff As Double
    Dim ln_frequ() As Double
    Dim efficiency() As Double
    Dim MOM() As Double
    Dim Ps As Double, Pr As Double
    Dim w As Double
    m = Selection.Rows.Count         'quantity of rows
    N = Selection.Cells(1, 5)
    ReDim T(0 To m - 5)
    ReDim U(0 To m - 5)
    ReDim D(0 To m - 5)
    ReDim ln_frequ(1 To N)
    ReDim efficiency(1 To N)
    ReDim MOM(1 To N)
    For j = 0 To m - 5
        T(j) = Selection.Cells(j + 5, 2)
        U(j) = Selection.Cells(j + 5, 3)
    Next j
    sb = Selection.Cells(5, 2).Address
    se = Selection.Cells(m, 3).Address
0:  Call graph(sb, se, "t, s", "u, m")
    For i = 1 To N
        ln_frequ(i) = Log(Selection.Cells(i + 2, 4))
        efficiency(i) = Selection.Cells(i + 2, 5)
        MOM(i) = 0
    Next i
1:  Call fouf(m - 5, U, D)
    D(0) = 0
    fr = 1 / T(m - 5)     'value of main frequency 1/T
'Calculating value of Ps:
    w = 0
    For k2 = 2 To m - 5 Step 2
        w = w + D(k2 - 1) ^ 2 + D(k2) ^ 2
    Next k2
    Ps = 10 * Log(w) / 2.302585093
'Calculating value of Pr:
    w = 0
    For k2 = 2 To m - 5 Step 2
2:      Call si(1, N, ln_frequ, efficiency, MOM, _
        Log(k2 / 2 * fr), eff)
        D(k2 - 1) = D(k2 - 1) * 10 ^ (-eff / 20)
        D(k2) = D(k2) * 10 ^ (-eff / 20)
        w = w + D(k2 - 1) ^ 2 + D(k2) ^ 2
    Next k2
    Pr = 10 * Log(w) / 2.302585093
'Calculating sound attenuation:
3:  w = Ps - Pr
4:  MsgBox "delta =" & Str(Round(w, 3)) & " dB"
'Synthesis of time dependence:
5:  Call foub(m - 5, U, D)
    sb = Selection.Cells(5, 6).Address
    se = Selection.Cells(m, 7).Address
    Selection.Cells(4, 6) = "t"
    Selection.Cells(4, 7) = "ur"
    For j = 0 To m - 5
        Selection.Cells(j + 5, 6) = T(j)
        Selection.Cells(j + 5, 7) = U(j)
    Next j
6:  Call graph(sb, se, "t, s", "ur, m")
    Range("O33").Select
End Sub
```

Operator 0 of this program creates the $u(t)$ graph of the bar deviation, i.e., of the deviation of membrane M_s, which is the sound source. For that, the graph subroutine (Section 4.8) is used for the first time.

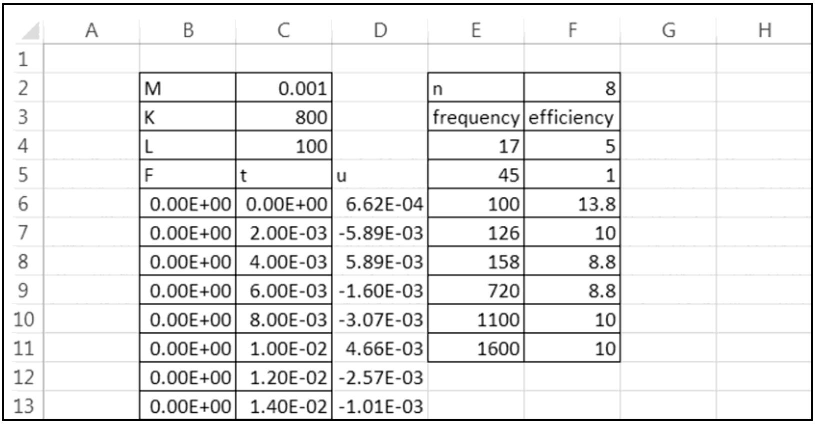
Further, dependence $u(t)$ is expanded into the Fourier series by calling the `fouf` subroutine for the forward discrete Fourier transform (operator `1`). By means of the first `k2` cycle, the value of P_s is calculated according to formula (5.79).

By means of the second `k2` cycle, the value P_r is calculated according to formulas (5.80) — (5.82). In this case, the linear spline, determined by the dependence of *efficiency* on *frequency* in tabular form, is used. The argument of the linear spline is *ln_ frequ* — the logarithm of frequency or the so-called logarithmic frequency (see operator `2` and the caption to Fig. 5.19).

Operator `3` calculates the sound attenuation according to formula (5.83). The resulting value is rounded up to three decimal places and put into the standard window (operator `4)`.

By calling the `foub` subroutine for the backward discrete Fourier transform, periodic time dependence $u_r(t)$ of the deviation of membrane M$_r$, which is the sound receiver, is synthesized (operator `5`). Operator `6` creates the $u_r(t)$ graph. For that, the `graph` subroutine is used for the second time.

The source data for program Listing 5.10 are the values given in complex table Listing 5.9 (Fig. 5.21). Before running the program, this Excel table (range B2:F31) must be selected.



	A	B	C	D	E	F	G	H
1								
2		M	0.001		n	8		
3		K	800		frequency	efficiency		
4		L	100		17	5		
5		F	t	u	45	1		
6		0.00E+00	0.00E+00	6.62E-04	100	13.8		
7		0.00E+00	2.00E-03	-5.89E-03	126	10		
8		0.00E+00	4.00E-03	5.89E-03	158	8.8		
9		0.00E+00	6.00E-03	-1.60E-03	720	8.8		
10		0.00E+00	8.00E-03	-3.07E-03	1100	10		
11		0.00E+00	1.00E-02	4.66E-03	1600	10		
12		0.00E+00	1.20E-02	-2.57E-03				
13		0.00E+00	1.40E-02	-1.01E-03				

Fig. 5.21. The Excel table with the source data

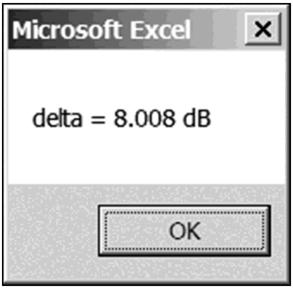
After starting the program execution:

1) the $u(t)$ graph appears;

2) the window with the Δ value of the sound attenuation appears (Fig. 5.22);

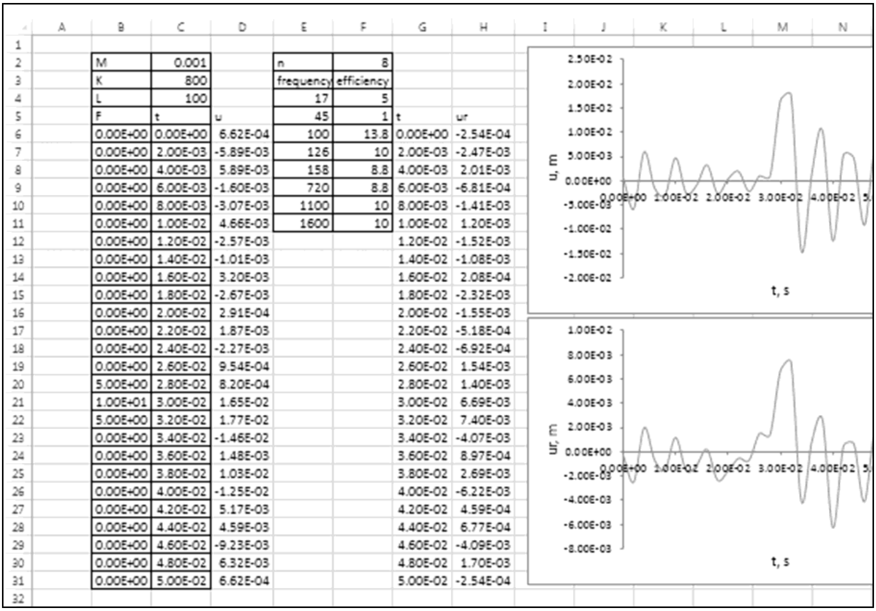
3) after clicking on *OK*, the $u_r(t)$ dependence and its graph appear.

Fig. 5.23 shows the Excel worksheet upon termination of the program execution. The automatically constructed graphs of dependences $u(t)$ and $u_r(t)$ are on this worksheet; Fig. 5.24 shows them completely.



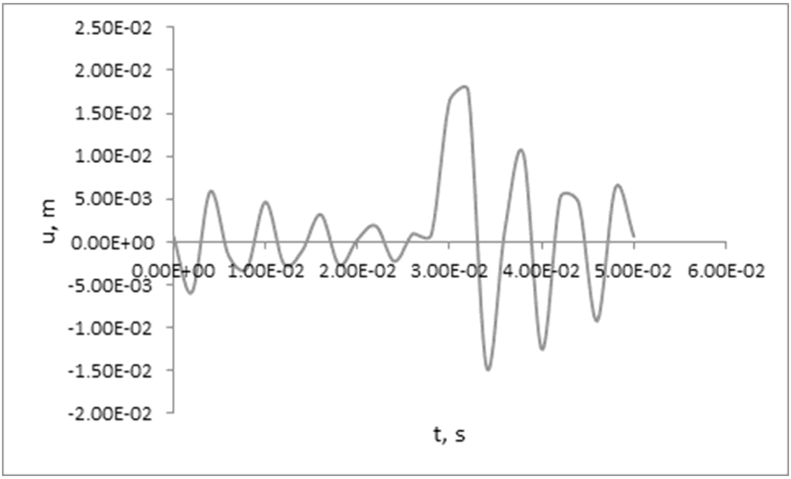
delta = 8.008 dB

Fig. 5.22. Window with the calculated sound attenuation

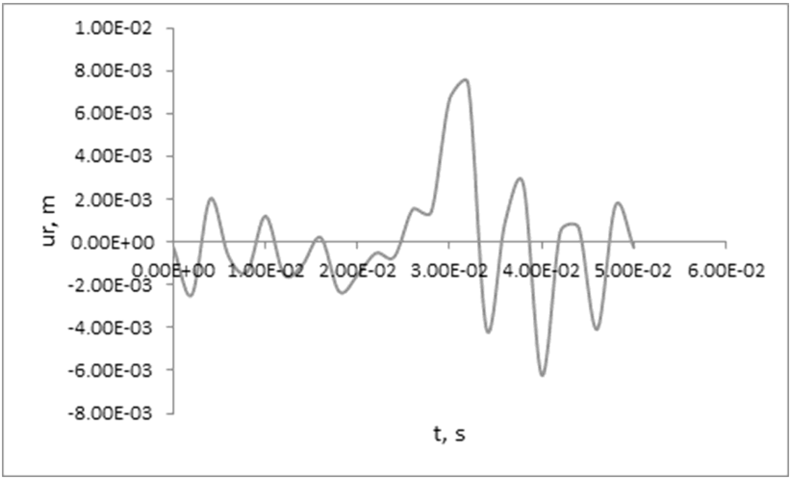


	A	B	C	D	E	F	G	H
1								
2		M	0.001		n	8		
3		K	800		frequency	efficiency		
4		L	100		17	5		
5		F	t	u	45	1	t	ur
6		0.00E+00	0.00E+00	6.62E-04	100	13.8	0.00E+00	-2.54E-04
7		0.00E+00	2.00E-03	-5.89E-03	126	10	2.00E-03	-2.47E-03
8		0.00E+00	4.00E-03	5.89E-03	158	8.8	4.00E-03	2.01E-03
9		0.00E+00	6.00E-03	-1.60E-03	720	8.8	6.00E-03	-6.81E-04
10		0.00E+00	8.00E-03	-3.07E-03	1100	10	8.00E-03	-1.41E-03
11		0.00E+00	1.00E-02	4.66E-03	1600	10	1.00E-02	1.20E-03
12		0.00E+00	1.20E-02	-2.57E-03			1.20E-02	-1.52E-03
13		0.00E+00	1.40E-02	-1.01E-03			1.40E-02	-1.08E-03
14		0.00E+00	1.60E-02	3.20E-03			1.60E-02	2.08E-04
15		0.00E+00	1.80E-02	-2.67E-03			1.80E-02	-2.32E-03
16		0.00E+00	2.00E-02	2.91E-04			2.00E-02	-1.55E-03
17		0.00E+00	2.20E-02	1.87E-03			2.20E-02	-5.18E-04
18		0.00E+00	2.40E-02	-2.27E-03			2.40E-02	-6.92E-04
19		0.00E+00	2.60E-02	9.54E-04			2.60E-02	1.54E-03
20		5.00E+00	2.80E-02	8.20E-04			2.80E-02	1.40E-03
21		1.00E+01	3.00E-02	1.65E-02			3.00E-02	6.69E-03
22		5.00E+00	3.20E-02	1.77E-02			3.20E-02	7.40E-03
23		0.00E+00	3.40E-02	-1.46E-02			3.40E-02	-4.07E-03
24		0.00E+00	3.60E-02	1.48E-03			3.60E-02	8.97E-04
25		0.00E+00	3.80E-02	1.03E-02			3.80E-02	2.69E-03
26		0.00E+00	4.00E-02	-1.25E-02			4.00E-02	-6.22E-03
27		0.00E+00	4.20E-02	5.17E-03			4.20E-02	4.59E-04
28		0.00E+00	4.40E-02	4.59E-03			4.40E-02	6.77E-04
29		0.00E+00	4.60E-02	-9.23E-03			4.60E-02	-4.09E-03
30		0.00E+00	4.80E-02	6.32E-03			4.80E-02	1.70E-03
31		0.00E+00	5.00E-02	6.62E-04			5.00E-02	-2.54E-04
32								

Fig. 5.23. The calculated $u_r(t)$ dependence and graphs



a



b

Fig. 5.24. The automatically created graphs: a — $u(t)$; b — $u_r(t)$

We advise the reader to return to the task on p. 279 for calculating the Fourier coefficients of dependences $v(t)$ and $i(t)$.

Chapter 6. Numerical Methods for Nonlinear Programming

For solving a series of applied problems, in particular the ones involving optimization, we must find minimum point $\mathbf{x}^*$ of linear or nonlinear non-negative function $F(\mathbf{x}) = F(x_1, x_2, ..., x_n)$, where $\mathbf{x} = (x_1, x_2, ..., x_n)$ is a point (vector) of the *n*-dimensional space, $n \geq 1$. The $F(\mathbf{x})$ function being minimized is called an objective function.

Minimizing a linear objective function with linear constraints is called linear programming. Minimizing a nonlinear objective function (with or without constraints) is called nonlinear programming.

The minimization, which does not require knowledge of mathematical expressions for partial derivatives of the objective function with respect to its arguments, is called the search for the minimum point. The importance of search methods follows from the fact that expressions for the partial derivatives are often unavailable.

In the first section of this chapter, tasks of linear and nonlinear programming are solved by means of the Solver add-in. In the fifth chapter of book [2], we demonstrated that Solver for Excel 2007 can give an incorrect result upon minimizing nonlinear function

$$F(x_1,\ x_2) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2 \tag{6.1}$$

known as the Rosenbrock function. We did not observe this drawback in the later versions of Excel (2010 and 2013). However, the minimization method, realized in Solver for the later versions of Excel, is less efficient than the Powell method [14] considered in this chapter (p. 454).

The $\mathbf{x}^*$ point is called a local minimum of function $F(\mathbf{x})$ if a neighborhood of $\mathbf{x}^*$ exists, where inequality $F(\mathbf{x}) \geq F(\mathbf{x}^*)$ is satisfied. Function $F(\mathbf{x})$ can be unimodal, with one local minimum, or multimodal, with several local minima.

Two subroutines are developed that are intended to find a local minimum of the $F(\mathbf{x})$ function of one or several variables. In the case of several variables, these subroutines realize the coordinate-descent and Powell methods.

Possibilities of the developed minimization subroutines are demonstrated on the following mathematical and applied tasks:

- optimizing the dimensions of a one-liter tin can;
- determining the equilibrium state of a four-spring mechanical system;
- minimizing a nonlinear function with nonlinear constraints and a tabular function of two variables;
- determining the local minima of a multimodal function of two variables.

Besides, the minimization subroutines are used:

- in the shooting method for solving the nonlinear differential equation with boundary conditions;
- in the least-squares method for determining the production function.

It is obvious that the problem of maximizing positive function $G(\mathbf{x})$ is equivalent to the problem of minimizing function $F(\mathbf{x}) = 1/G(\mathbf{x})$. Therefore, the minimization subroutines of this chapter can be used to find the maxima of function $G(\mathbf{x})$.

6.1. Minimizing linear and nonlinear functions of several variables by the Solver add-in

We used the Solver add-in for solving the nonlinear algebraic equation (Section 4.5). Let us consider two more examples of using this add-in — for solving problems of linear and nonlinear programming.

Initially, we will solve the following transportation problem of linear programming from article [15].

There are three points (A_1, A_2, A_3) for sending identical loads and four points (B_1, B_2, B_3, B_4) for receiving them. Let c_{ij} be the expenses for relocation of one load from A_i to B_j (i = 1, 2, 3, j = 1, 2, 3, 4). We have to find the minimum of total expenses for the required relocations from points A_1, A_2, A_3 to points B_1, B_2, B_3, B_4.

More precisely, the source data for the problem include:

- the matrix of expenses for transportation of one load looking like

$$\mathbf{C}=\begin{bmatrix} c_{11} & c_{12} & c_{13} & c_{14} \\ c_{21} & c_{22} & c_{23} & c_{24} \\ c_{31} & c_{32} & c_{33} & c_{34} \end{bmatrix}=\begin{bmatrix} 3 & 4 & 5 & 6 \\ 4 & 4 & 2 & 1 \\ 2 & 1 & 6 & 8 \end{bmatrix};$$

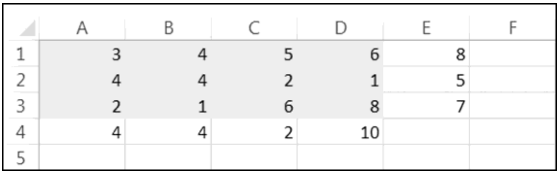
- the number of loads at points A_1, A_2, A_3: $a_1 = 8$, $a_2 = 5$, $a_3 = 7$;
- the requirement for loads at points B_1, B_2, B_3, B_4: $b_1 = 4$, $b_2 = 4$, $b_3 = 2$, $b_4 = 10$ (note that $b_1 + b_2 + b_3 + b_4 = a_1 + a_2 + a_3$).

We have to find:

- the minimum of total expenses for the load relocations from points A_1, A_2, A_3 to points B_1, B_2, B_3, B_4;
- the corresponding number of loads relocated from point A_i to point B_j (i = 1, 2, 3, j = 1, 2, 3, 4).

For solving the formulated problem, let us put:

- the **C** matrix into range A1:D3 on an Excel worksheet;
- values $a_1 = 8$, $a_2 = 5$, $a_3 = 7$ into cells E1:E3 of the same worksheet;
- values $b_1 = 4$, $b_2 = 4$, $b_3 = 2$, $b_4 = 10$ into cells A4:D4 (Fig. 6.1).



	A	B	C	D	E	F
1	3	4	5	6	8	
2	4	4	2	1	5	
3	2	1	6	8	7	
4	4	4	2	10		
5						

Fig. 6.1. The Excel worksheet with the source data:
the shaded range corresponds to the **C** matrix

We will use matrix

$$\mathbf{X} = \begin{bmatrix} x_{11} & x_{12} & x_{13} & x_{14} \\ x_{21} & x_{22} & x_{23} & x_{24} \\ x_{31} & x_{32} & x_{33} & x_{34} \end{bmatrix},$$

where x_{ij} is the number of loads relocated from A$_i$ to B$_j$.

Product $c_{ij}x_{ij}$ denotes the expenses for the load relocation from point A$_i$ to point B$_j$ (i = 1, 2, 3, j = 1, 2, 3, 4). The total expenses for the relocations are equal to

$$F = \sum_{i,\,j} c_{ij}x_{ij}\ , \tag{6.2}$$

where the summation is over i = 1, 2, 3 and j = 1, 2, 3, 4.

Formula (6.2) gives the linear function of 12 variables,

$$F = F(x_{11},\ x_{12}, x_{13},\ x_{14}, x_{21},\ x_{22}, x_{23},\ x_{24}, x_{31},\ x_{32}, x_{33},\ x_{34})\ ,$$

whose arguments are the **X** matrix elements. We have to minimize this function with linear constraints

$$\sum_{j\,=\,1}^{4} x_{ij} = a_i\,, \tag{6.3}$$

i = 1, 2, 3,

$$\sum_{i\,=\,1}^{3} x_{ij} = b_j\,, \tag{6.4}$$

j = 1, 2, 3, 4.

Continuing to fill the Excel worksheet, we allot cells A5:D7 for the **X** matrix elements, which are unknown, and put arbitrary numbers into these cells (for example, units). Into cell F2, we enter formula

```
=SUMPRODUCT(A1:D3;A5:D7)
```

corresponding to mathematical formula (6.2). Into cells E5, E6 and E7, we enter formulas

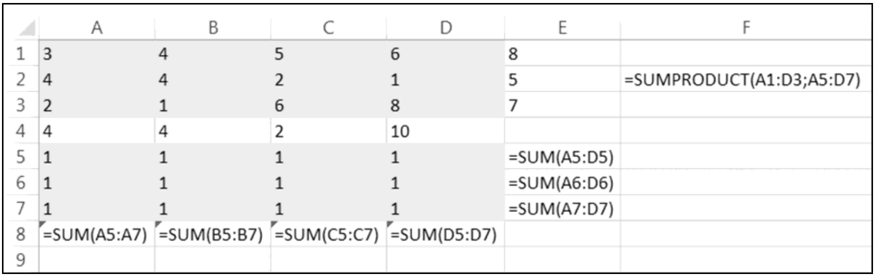
```
=SUM(A5:D5)
=SUM(A6:D6)
=SUM(A7:D7)
```

corresponding to the left-hand side of constraints (6.3). Into cells A8, B8, C8 and D8, we enter formulas

```
=SUM(A5:A7)
=SUM(B5:B7)
=SUM(C5:C7)
=SUM(D5:D7)
```

corresponding to the left-hand side of constraints (6.4).

Upon tuning Excel so that formulas are displayed in cells (p. 193), the worksheet takes the form depicted in Fig. 6.2. We return to the customary tuning of Excel when the calculated results are in cells.



	A	B	C	D	E	F
1	3	4	5	6	8	
2	4	4	2	1	5	=SUMPRODUCT(A1:D3;A5:D7)
3	2	1	6	8	7	
4	4	4	2	10		
5	1	1	1	1	=SUM(A5:D5)	
6	1	1	1	1	=SUM(A6:D6)	
7	1	1	1	1	=SUM(A7:D7)	
8	=SUM(A5:A7)	=SUM(B5:B7)	=SUM(C5:C7)	=SUM(D5:D7)		
9						

Fig. 6.2. The Excel worksheet with the formulas in the cells: the shaded top and bottom ranges correspond to the **C** and **X** matrices, respectively

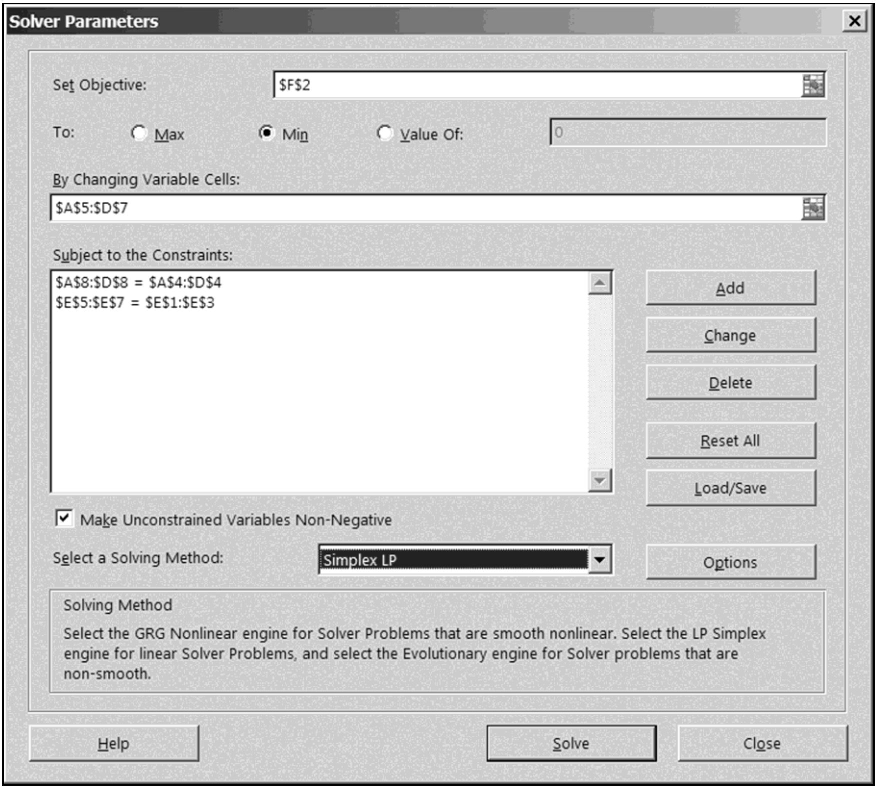
Let us fulfill the following operations:

1) *Data* > *Solver* in area *Analysis*;
2) in the *Solver Parameters* window, enter $F$2 into text box *Set Objective*;
3) turn on option *Min*;
4) enter $A$5:$D$7 into text box *By Changing Variable Cells*;
5) by means of clicks on the *Add* button, successively enter conditions $E$5:$E$7=$E$1:$E$3 and $A$8:$D$8=$A$4:$D$4 into box *Subject to the Constraints* (in addition to the equal sign, we can use comparison signs “less than or equal to” and “greater than or equal to” in the constraints);

6) enter *Simplex LP* into box *Select a Solving Method* by means of the drop-down list (Fig. 6.3);

7) click on the *Solve* button;

8) in the *Solver Results* window opened, click on *OK*.



Solver Parameters

Set Objective: $F$2

To: ○ Max ● Min ○ Value Of: 0

By Changing Variable Cells: $A$5:$D$7

Subject to the Constraints:

$A$8:$D$8 = $A$4:$D$4
$E$5:$E$7 = $E$1:$E$3

☑ Make Unconstrained Variables Non-Negative

Select a Solving Method: Simplex LP

Solving Method

Select the GRG Nonlinear engine for Solver Problems that are smooth nonlinear. Select the LP Simplex engine for linear Solver Problems, and select the Evolutionary engine for Solver problems that are non-smooth.

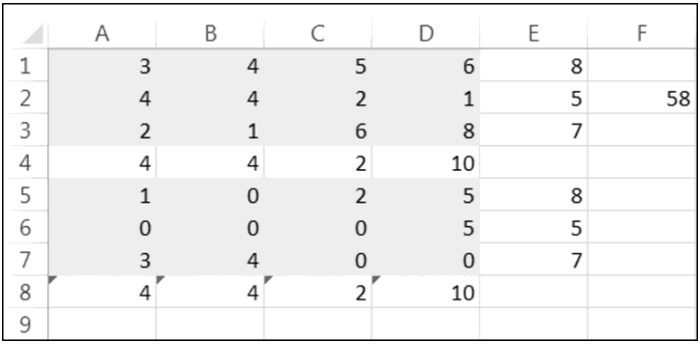
Fig. 6.3. The *Solver Parameters* window before minimizing linear function (6.2) with linear constraints (6.3) and (6.4)

Results of minimizing function (6.2) with constraints (6.3) and (6.4) appear in cell F2 and range A5:D7 (Fig. 6.4).

Value 58 in cell F2 gives the minimum of total expenses for the load relocations. The contents of range A5:D7 suggest the distribution of the load relocations for minimizing the total expenses.

For example:

- value $x_{11} = 1$ (in cell A5) means that one load must be relocated from point A$_1$ to point B$_1$;
- value $x_{24} = 5$ (in cell D6) means that five loads must be relocated from point A$_2$ to point B$_4$.



	A	B	C	D	E	F
1	3	4	5	6	8	
2	4	4	2	1	5	58
3	2	1	6	8	7	
4	4	4	2	10		
5	1	0	2	5	8	
6	0	0	0	5	5	
7	3	4	0	0	7	
8	4	4	2	10		
9						

Fig. 6.4. The worksheet upon termination of minimizing function (6.2) with constraints (6.3) and (6.4)

According to the source data for the problem solved by us, equality

$$\sum_{i=1}^{3} a_i = \sum_{j=1}^{4} b_j$$

is satisfied. If this equality is not satisfied, that is,

$$\sum_{i=1}^{3} a_i > \sum_{j=1}^{4} b_j ,$$

we should introduce virtual point B$_5$ (for receiving the loads) and consider that

$$b_5 = \sum_{i=1}^{3} a_i - \sum_{j=1}^{4} b_j ,$$

$$c_{15} = c_{25} = c_{35} = 0 .$$

Further, we use the above method for solving the problem of minimizing the total expenses for the load relocations from points A$_1$, A$_2$, A$_3$ to points B$_1$, B$_2$, B$_3$, B$_4$, B$_5$. The calculated values of x_{15}, x_{25}, x_{35} are the numbers of loads remaining at points A$_1$, A$_2$, A$_3$.

According to the fourth chapter of book [2], the considered operation mode of Solver for Excel 2007 can be used for solving the system of linear algebraic equations (3.49). Solver for the later versions of Excel does not have this ability.

Below, we will minimize nonlinear function (6.1), the Rosenbrock function, by means of the Solver add-in. The graph of this rather popular test function is depicted in Fig. 6.5, which is taken from the following page of Wikipedia, the free encyclopedia: http://en.wikipedia.org/wiki/Rosenbrock_function.

The following is obvious:

- nonlinear function (6.1) is continuous and non-negative;
- point $\mathbf{x}^*$ of the function minimum has coordinates $x_1^* = x_2^* = 1$, and $F(1,\ 1) = 0$;
- point $\mathbf{x}^*$ is in "ravine" (on the dark stripe in Fig. 6.5).

Let us show that the Rosenbrock function does not have other minimum points.

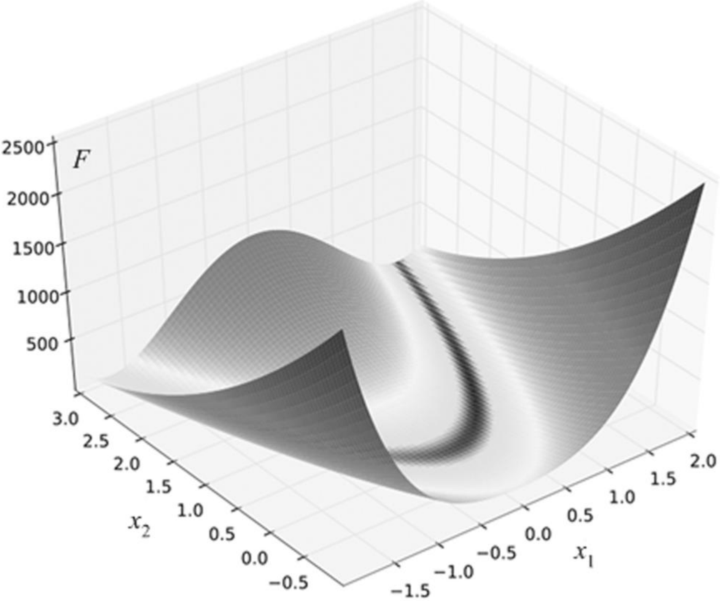


Fig. 6.5. Graphic of Rosenbrock function $F(x_1,\ x_2) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2$

The partial derivatives of function (6.1) are defined by the following algebraic expressions:

$$\partial F / \partial x_1 = 400x_1^3 - 400x_1x_2 + 2x_1 - 2, \qquad (6.5)$$

$$\partial F / \partial x_2 = 200(x_2 - x_1^2). \qquad (6.6)$$

According to the necessary condition for an extreme value [3], the minimum points of function $F(x_1,\ x_2)$ belong to the set of points $(x_1,\ x_2)$ that are solutions of the system of equations

$$\partial F / \partial x_1 = 0,$$

$$\partial F / \partial x_2 = 0$$

or

$$400x_1^3 - 400x_1x_2 + 2x_1 - 2 = 0,$$

$$x_2 - x_1^2 = 0.$$

According to the second equation, $x_2 = x_1^2$. Substituting this expression into the first equation, we have $x_1 = 1$. Consequently, $x_2 = x_1^2 = 1$.

Thus, function (6.1) has the unique minimum point with coordinates $x_1^* = x_2^* = 1$, i.e., the function is unimodal, and high-quality minimization tools must find this point under any initial approximation.

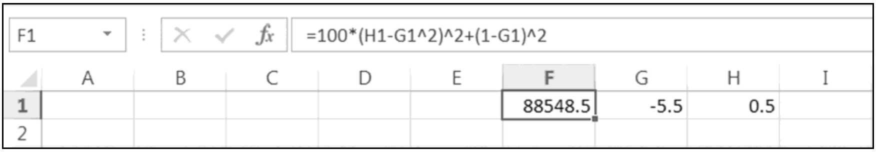
To minimize function (6.1) by means of the Solver add-in, we assume that:

- the G1 and H1 cells contain the values of x_1 and x_2, respectively;
- the F1 cell contains the value of $F(x_1,\ x_2)$.

Into cell F1, we enter formula

```
=100*(H1-G1^2)^2+(1-G1)^2
```

corresponding to mathematical formula (6.1). Into cells G1 and H1, we respectively enter values -5.5 and 0.5 defining the initial approximation of the minimum point: x_1 = -5.5 and x_2 = 0.5 (Fig. 6.6).


F1 =100*(H1-G1^2)^2+(1-G1)^2

	A	B	C	D	E	F	G	H	I
1						88548.5	-5.5	0.5	
2									

Fig. 6.6. The worksheet before minimizing nonlinear function (6.1)

Let us fulfill the following operations:

1) *Data* > *Solver* in area *Analysis*;

2) in the *Solver Parameters* window opened, enter $F$1 into text box *Set Objective*;

3) turn on option *Min*;

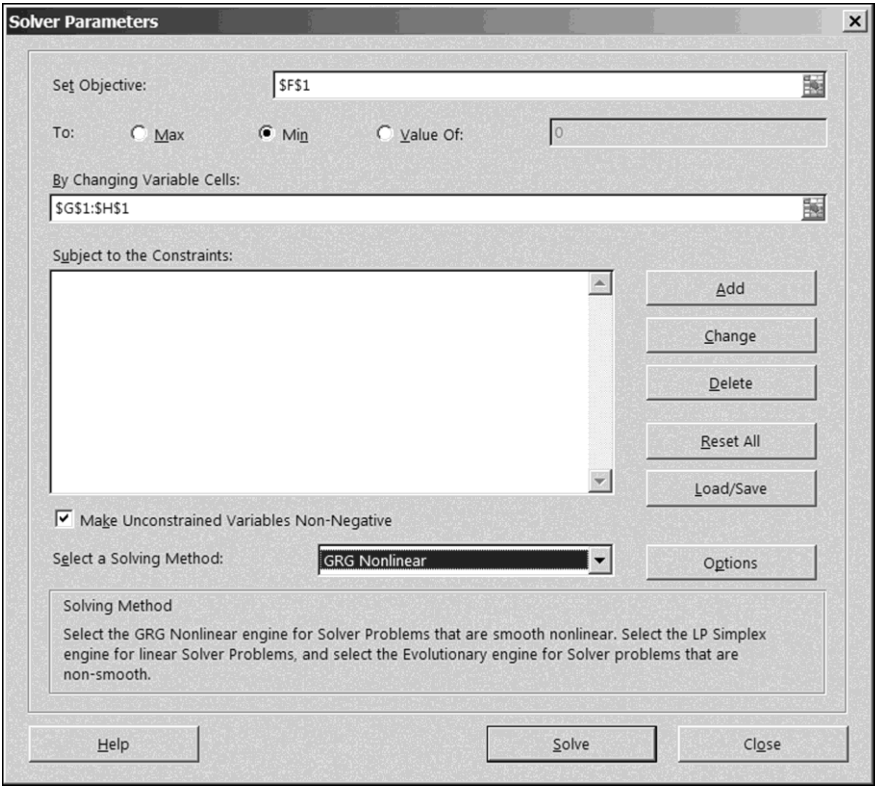
4) enter $G$1:$H$1 into text box *By Changing Variable Cells*;

5) enter *GRG Nonlinear* into box *Select a Solving Method* by means of the drop-down list (Fig. 6.7)

6) click on the *Solve* button;

7) click on the *OK* button in the *Solver Results* window.

The result of minimizing the Rosenbrock function is given in Fig. 6.8.



Solver Parameters

Set Objective: $F$1

To: ○ Max ● Min ○ Value Of: 0

By Changing Variable Cells:
$G$1:$H$1

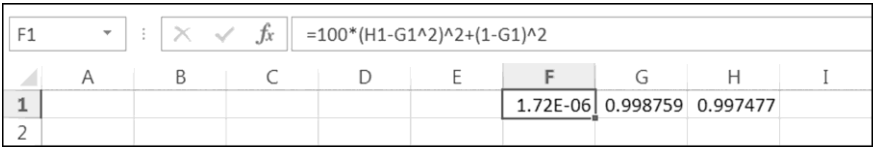
Subject to the Constraints:

☑ Make Unconstrained Variables Non-Negative

Select a Solving Method: GRG Nonlinear

Solving Method
Select the GRG Nonlinear engine for Solver Problems that are smooth nonlinear. Select the LP Simplex engine for linear Solver Problems, and select the Evolutionary engine for Solver problems that are non-smooth.

Fig. 6.7. The *Solver Parameters* window before minimizing function (6.1)



F1 =100*(H1-G1^2)^2+(1-G1)^2

	A	B	C	D	E	F	G	H	I
1						1.72E-06	0.998759	0.997477	
2									

Fig. 6.8. The worksheet upon termination of minimizing function (6.1)

The Excel formula, intended for calculating the values of objective function $F(x_1,\ x_2)$, may include Excel functions, in particular, user-defined functions. To demonstrate this, let us consider the following user-defined function corresponding to mathematical function (6.1).

Listing 6.1

```
Function Rosenbrock(x1, x2)
    Rosenbrock = 100 * (x2 - x1 ^ 2) ^ 2 + _
        (1 - x1) ^ 2
End Function
```

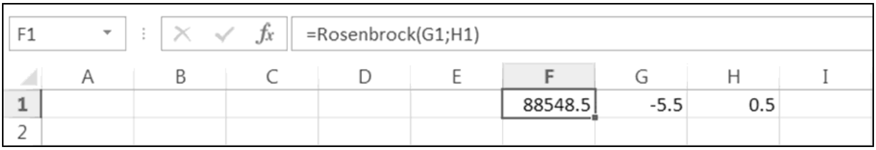
We put this declaration into Module16 of the BookNM workbook and formula

```
=Rosenbrock(G1;H1)
```

into cell F1.

Excel user-defined functions, developed by means of VBA, may be very complicated. Therefore, the Solver add-in allows us to solve quite complex problems.

During the minimization, we can see results of iterations. For demonstrating this possibility of the Solver add-in, we will solve the previous task again (see Fig. 6.9, which is similar to Fig. 6.6).



F1 =Rosenbrock(G1;H1)

	A	B	C	D	E	F	G	H	I
1						88548.5	-5.5	0.5	
2									

Fig. 6.9. The worksheet before minimizing nonlinear function (6.1)

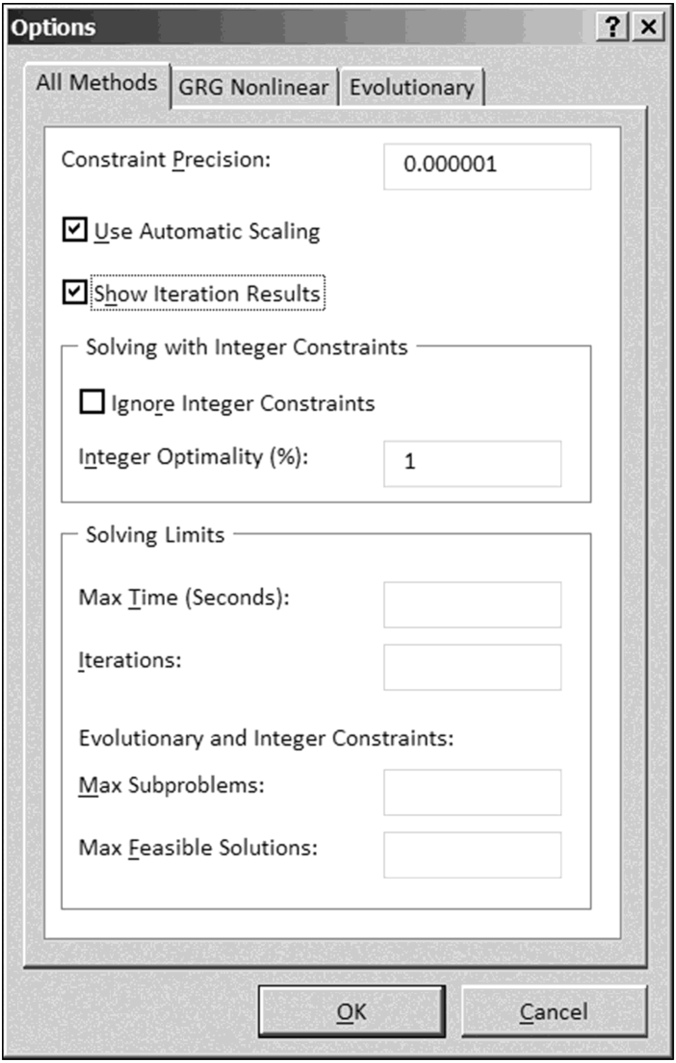
To see results of iterations, we fulfill the following operations:

1) click on the *Options* button in the *Solver Parameters* window depicted in Fig. 6.7;

2) in the *Options* window opened, turn on option *Show Iteration Results* (Fig. 6.10);

3) click on the *OK* button;

4) click on the *Solve* button in the *Solver Parameters* window (Fig. 6.7).



Options

All Methods | GRG Nonlinear | Evolutionary

Constraint Precision: 0.000001

☑ Use Automatic Scaling

☑ Show Iteration Results

Solving with Integer Constraints

☐ Ignore Integer Constraints

Integer Optimality (%): 1

Solving Limits

Max Time (Seconds):

Iterations:

Evolutionary and Integer Constraints:

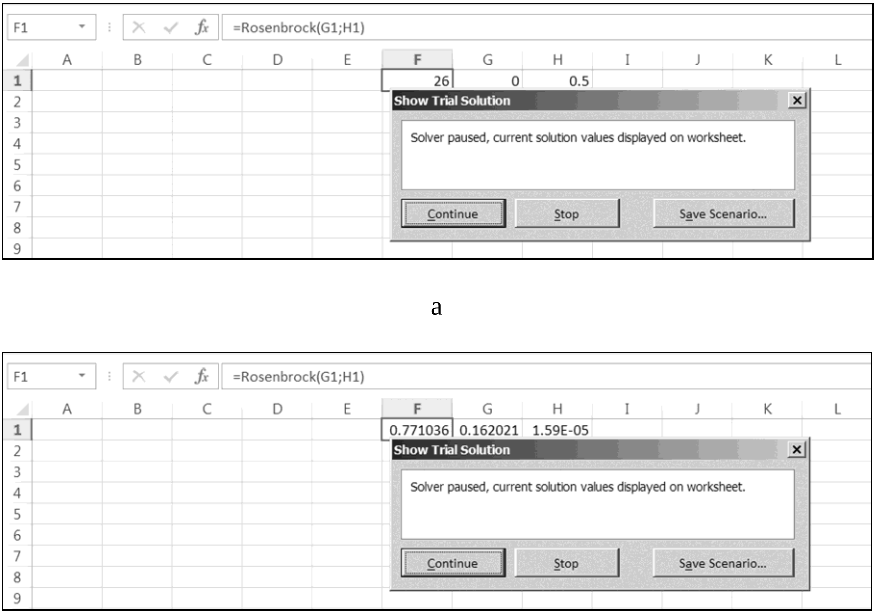
Max Subproblems:

Max Feasible Solutions:

Fig. 6.10. The *Options* window with option *Show Iteration Results* turned on

Fig. 6.11a shows the result of the first iteration. After the next 4 clicks on *Continue* in window *Show Trial Solution*, we see the result of the 5th iteration (Fig. 6.11b). After the next 16 clicks on *Continue*, the *Solver Results* window appears. We click on *OK* in the last window to terminate the execution. Fig. 6.12, which is similar to Fig. 6.8, shows the result. Thus, 21 iterations were performed.

The Powell minimization method (Section 6.5) gives the same result after 5 iterations (p. 454).



F1 =Rosenbrock(G1;H1)

	F	G	H
1	26	0	0.5

Show Trial Solution

Solver paused, current solution values displayed on worksheet.

a

F1 =Rosenbrock(G1;H1)

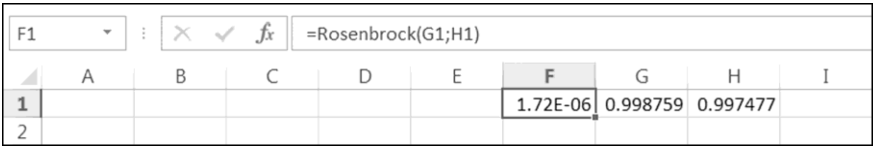
	F	G	H
1	0.771036	0.162021	1.59E-05

Show Trial Solution

Solver paused, current solution values displayed on worksheet.

b

Fig. 6.11. The worksheet after the 1st (a) and 5th (b) iterations



F1 =Rosenbrock(G1;H1)

	A	B	C	D	E	F	G	H	I
1						1.72E-06	0.998759	0.997477	
2									

Fig. 6.12. The worksheet upon termination of minimizing function (6.1)

6.2. Method for minimizing a nonlinear function of one variable

The minimization of a nonlinear function of one variable, $f(x)$, is an iterative process, which needs an initial approximation of the required minimum point. When developing the minimization method, we will assume that $f(x)$ satisfies the following conditions (in the function's domain including the unique minimum point, x^*, and its initial approximation, x^0):

- the $f(x)$ function is continuous and non-negative;
- the derivative, $f'(x)$, is continuous (that is, the function is smooth).

In the algorithm below, the minimization of $f(x)$ includes the following three stages:

1) finding the so-called uncertainty segment, which contains point x^* within itself;
2) reducing the uncertainty segment to one and a half times;
3) reducing the uncertainty segment by means of the parabolic interpolation of $f(x)$.

For finding the uncertainty segment, the movement along the *x* axis must be in the direction of decreasing $f(x)$, at that, each step is twice as large as the previous step (as in the second chapter of book [16]).

When searching the uncertainty segment, only three points are stored, whose designations are *a*, *b*, *c*, at that, point *a* is the last, *b* is the penultimate point, *c* precedes *b*. The function values at these points are also stored: $f_a = f(a)$, $f_b = f(b)$, $f_c = f(c)$. The movement is terminated when function $f(x)$ becomes increasing, i.e., condition $f_a \geq f_b$ is satisfied (Fig. 6.13). Resulting $[a, c]$ is the uncertainty segment.

Let *d* be the midpoint of segment $[a, b]$: $d = (a+b)/2$. After calculating value $f_d = f(d)$ (Fig. 6.14), the condition of the termination of searching the minimum point is checked and, if needed, the uncertainty segment is reduced according to the following algorithm.

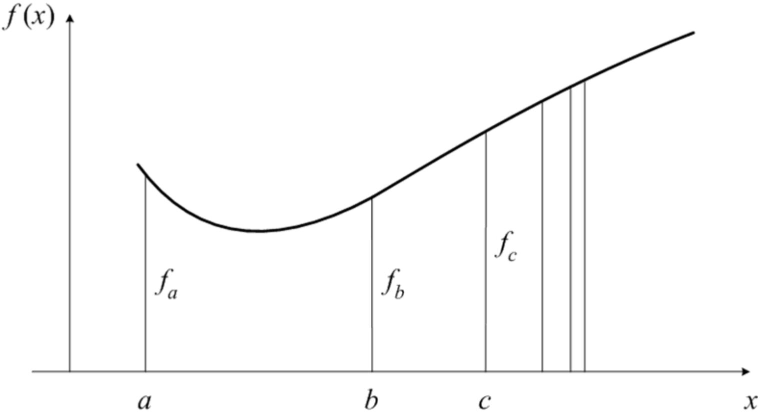


Fig. 6.13. Termination of searching the uncertainty segment: $[a, c]$ is the uncertainty segment; the movement from right to left took place

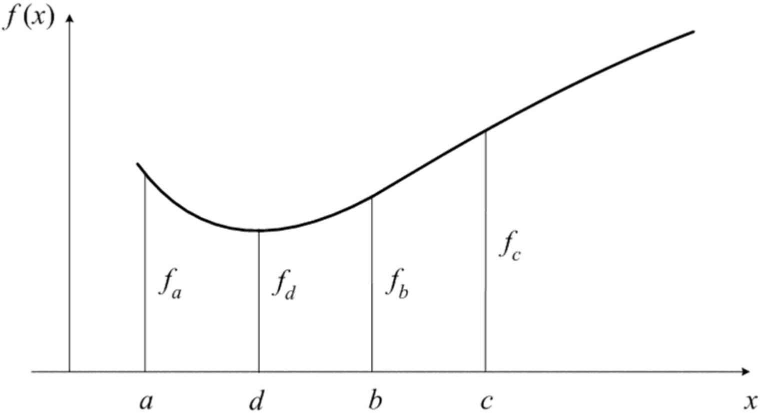


Fig. 6.14. The result of calculating value $f_d = f(d)$:
segments $[a, d]$, $[d, b]$ and $[b, c]$ are equal in length

1. If condition $\left|f_b - f_d\right| < \rho\, f_b$ is satisfied (ρ is a given positive constant), the minimization is terminated, and assignments $x^* = d$, $f(x^*) = f_d$ are performed. Otherwise, the next item is fulfilled.

2. Of the four points (a, b, c, d), three neighboring points are chosen, such that the function value at the midpoint is less than the function values at the endpoints. These three points are denoted by letters a, b, c; the corresponding function values are denoted by f_a, f_b, f_c, respectively. As a result, the uncertainty segment, $[a, c]$, is reduced to one and a half times (Fig. 6.15).

3. The minimum point of the parabola, passing through points A = (a, f_a), B = (b, f_b), C = (c, f_c), is calculated according to formula

$$d = 0.5\frac{f_a(b^2-c^2)+f_b(c^2-a^2)+f_c(a^2-b^2)}{f_a(b-c)+f_b(c-a)+f_c(a-b)}. \qquad (6.7)$$

The derivation of this formula is given below.

4. Value $f_d = f(d)$ is calculated (Fig. 6.15). The jump to the first item is performed.

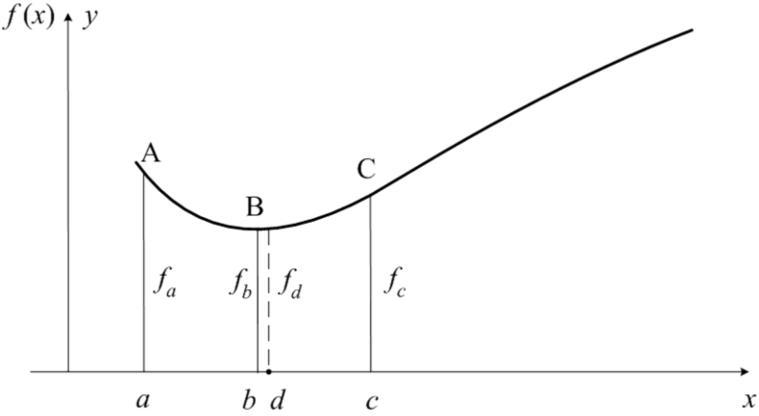


Fig. 6.15. The results of reducing the uncertainty segment to 1.5 times and of the parabolic interpolation: $[a, c]$ is the resulting uncertainty segment; b is the midpoint of $[a, c]$; d is the minimum point of the parabola

Let us derive formula (6.7).

According to the Lagrange interpolation formula [3], the equation of the quadratic parabola, passing through points A, B, C (Fig. 6.15), is as follows:

$$y = \frac{(x-b)(x-c)}{(a-b)(a-c)}f_a + \frac{(x-a)(x-c)}{(b-a)(b-c)}f_b + \frac{(x-a)(x-b)}{(c-a)(c-b)}f_c.$$

Using the basic rules of differentiation [3], we obtain

$$\frac{dy}{dx} = \frac{(x-b)+(x-c)}{(a-b)(a-c)} f_a + \frac{(x-a)+(x-c)}{(b-a)(b-c)} f_b + \frac{(x-a)+(x-b)}{(c-a)(c-b)} f_c .$$

We bring the right-hand side of this expression down to common denominator:

$$\frac{dy}{dx} = \frac{1}{(a-b)(a-c)(b-c)}[2x(b-c) f_a - (b^2-c^2) f_a -$$

$$-2x(a-c) f_b + (a^2-c^2) f_b + 2x(a-b) f_c - (a^2-b^2) f_c] .$$

The coordinate of the minimum point of the considered quadratic parabola is the solution of equation $dy/dx = 0$, that is,

$$2x(b-c) f_a - (b^2-c^2) f_a - 2x(a-c) f_b + (a^2-c^2) f_b +$$

$$+ 2x(a-b) f_c - (a^2-b^2) f_c = 0$$

or

$$2x(b-c) f_a - 2x(a-c) f_b + 2x(a-b) f_c =$$

$$= (b^2-c^2) f_a - (a^2-c^2) f_b + (a^2-b^2) f_c .$$

This solution, denoted by *d*, has form (6.7).

6.3. The coordinate-descent method

Let $x_0 = F(x_1, x_2, ..., x_n)$ be a continuous and non-negative function of *n* variables. Besides, we assume that this function has the unique minimum point and continuous partial derivatives $\partial F / \partial x_1$, $\partial F / \partial x_2$, …, $\partial F / \partial x_n$. Determining the minimum point of $F(x_1, x_2, ..., x_n)$ is an iterative process.

In the coordinate-descent method, each iteration consists of successive minimizations of the *F* function on arguments x_1, x_2, …, x_n: at first, the minimization of the *F* function on x_1 is performed, then on x_2, ..., and on x_n. The iterations are repeated until the condition of the termination of searching the minimum point is satisfied.

The declaration of the `mini` subroutine, realizing the coordinate-descent method, is given below. We enter it into Module17 of the BookNM workbook.

Listing 6.2

```
Sub mini(ByVal n, ByRef x() As Double, _
    ByRef ss() As Double, ByVal rho, Optional alpha)
    Dim fa As Double, fb As Double, fc As Double
    Dim fd As Double
    Dim a As Double, b As Double, c As Double
    Dim d As Double, e As Double
    Dim s() As Double: ReDim s(n)
    Dim y() As Double: ReDim y(n)
    Dim z() As Double: ReDim z(n)
    Dim sss() As Double: ReDim sss(n, n)
    Dim i As Byte, j As Byte
    d = 0
    For j = 1 To n
        For i = 1 To n
            If i <> j Then d = ss(i, j) ^ 2 + d
        Next i
    Next j
    If d <> 0 Then
        MsgBox "mini: Array ss is non-diagonal"
        End            'immediate termination of macro
    End If
    For j = 1 To n
        For i = 1 To n
            sss(i, j) = ss(i, j)
        Next i
    Next j
m1: For i = 0 To n
        y(i) = x(i)
        z(i) = x(i)
    Next i
    For j = 1 To n     'j - number of descent direction
        For i = 1 To n
            s(i) = ss(i, j)
        Next i
        d = 0
        For i = 1 To n
            d = s(i) ^ 2 + d
        Next i
        If d = 0 Then
            For i = 1 To n
                s(i) = sss(i, j)
            Next i
        Else
            For i = 1 To n
                sss(i, j) = s(i)
            Next i
        End If
'Finding uncertainty segment:
        fa = y(0): fb = y(0): fc = y(0)
        a = 0: b = 0: c = 0
        d = 1: e = 1
s1:     For i = 1 To n
            x(i) = y(i) + d * s(i)
        Next i
        Call func(n, x)
        If Not IsMissing(alpha) Then
            If x(0) < alpha Then GoTo m3
        End If
        fd = x(0)
        If fd < fa Then
            fc = fb: fb = fa: fa = fd
            c = b: b = a: a = d
            d = 2 * d + e
            GoTo s1
        Else
            If fa = fb Then
                fb = fd
                b = d
                e = -2 * d
                d = e
                GoTo s1
            End If
        End If
        fc = fb: fb = fa: fa = fd
        c = b: b = a: a = d
        d = (a + b) * 0.5
        For i = 1 To n
            x(i) = y(i) + d * s(i)
        Next i
        Call func(n, x)
        If Not IsMissing(alpha) Then
            If x(0) < alpha Then GoTo m3
        End If
        fd = x(0)
'Reducing uncertainty segment:
s2:     If Abs(fb - fd) < rho * fb Then GoTo s0
        If (c - d) * (d - b) < 0 Then
            If fd < fb Then
                fc = fb: fb = fd
                c = b: b = d
            Else
                fa = fd
                a = d
            End If
        Else
            If fd < fb Then
                fa = fb: fb = fd
                a = b: b = d
            Else
                fc = fd
                c = d
            End If
        End If
        d = fa * (b - c) + fb * (c - a) + fc * (a - b)
        If d = 0 Then GoTo s0
        d = (fa * (b * b - c * c) + _
            fb * (c * c - a * a) + _
            fc * (a * a - b * b)) / (2 * d)
        For i = 1 To n
            x(i) = y(i) + d * s(i)
        Next i
        Call func(n, x)
        If Not IsMissing(alpha) Then
            If x(0) < alpha Then GoTo m3
        End If
        fd = x(0)
        GoTo s2
s0:     For i = 1 To n
            ss(i, j) = x(i) - y(i)
            y(i) = x(i)
        Next i
        y(0) = x(0)
    Next j
'Checking condition of minimization termination:
m3: If Not IsMissing(alpha) Then
        If test(n, x, z, rho, alpha) Then GoTo m1
    Else
        If test(n, x, z, rho) Then GoTo m1
    End If
End Sub
```

The `mini` subroutine under consideration has five parameters, and parameter `alpha` is optional.

The obligatory parameters have the following sense:

- `n` is the number of variables;
- `x` is an array with elements `x(0)`, `x(1)`, `x(2)`, …, `x(n)`, at that, memory cell `x(0)` contains the *F* function's value corresponding to the values of x_1, x_2, …, x_n being in memory cells `x(1)`, `x(2)`, …, `x(n)`, respectively;
- `ss` is a two-dimensional array $n \times n$, corresponding to diagonal matrix **S** of initial steps: memory cell `ss(1, 1)` contains the initial step along the x_1 axis, cell `ss(2, 2)` contains the initial step along the x_2 axis, …, cell

`ss(n, n)` contains the initial step along the x_n axis (the diagonal matrix is a square matrix that has nonzero elements only on its main diagonal);

- `rho` is the relative change in *F*, terminating the minimization along an axis (see ρ in the first item of the algorithm in the previous section).

The above declaration of the `mini` subroutine, Listing 6.2, contains three calls of the `func` subroutine intended for calculation of the *F* function's value (the value of the `x(0)` element) corresponding to the current values of elements `x(1)`, `x(2)`, …, `x(n)`.

Optional parameter `alpha` of the `mini` subroutine is used when we know that the minimum value of non-negative function *F* is equal to zero: when condition $F < \alpha$ is satisfied, operator `GoTo m3` (following the `func` subroutine calls) is performed, and then the `test` function is called.

In the `test` function declaration, conditions must be formulated for terminating the minimization. The call of this function is performed when $F < \alpha$ and also at the end of every iteration, that is, after minimizations of the *F* function on all arguments x_1, x_2, …, x_n.

Function `test` returns `True` to the `mini` subroutine if none of the minimization termination conditions is satisfied. In this case, the next iteration is performed (that is, the minimizations of the *F* function on arguments x_1, x_2, …, x_n are repeated) and the call of function `test` is again performed. The `mini` subroutine execution is terminated when `test` returns `False`.

To control the course of the $x_0 = F(x_1, x_2, ..., x_n)$ function minimization, the `test` function declaration must contain operators for putting the current values of elements `x(0)`, `x(1)`, `x(2)`, …, `x(n)` into cells on the Excel worksheet.

Note that from iteration to iteration in the course of the minimization:

- not only one-dimensional array `x` is being changed, but two-dimensional array `ss`, corresponding to matrix **S** of initial steps for the next iteration, is also being changed;
- when changing array `ss`, the **S** matrix remains diagonal in the considered coordinate-descent method.

6.4. Examples of using the minimization methods

The `mini` subroutine from the previous section will be used for solving the following two tasks: the optimization of a tin can and the Rosenbrock function minimization.

The optimization of a tin can.

According to this task, we have to determine a variant of the cylindrical tin can of a given volume, which is the best in terms of the amount of tin required.

The volume and total area of a right circular cylinder [3] are respectively defined by formulas

$$V = \pi r^2 h,$$
$$A = 2\pi r(r + h),$$

where r is the base radius, h is the height. The first formula leads to

$$h = \frac{V}{\pi r^2}. \tag{6.8}$$

By means of this expression, we eliminate h from the second formula:

$$A(r) = 2\pi r^2 + 2\frac{V}{r}. \tag{6.9}$$

For given volume V of the tin can, the total area is defined by the last formula. We have to find the minimum point of the $A(r)$ function for positive values of the r argument.

Because the objective function, $A(r)$ defined by (6.9), has a simple form, the optimization task can be solved analytically. For that, according to [3], we must fulfill the following:

1) find real roots of equation

$$\frac{dA}{dr} = 0; \tag{6.10}$$

2) choose those of the roots, which are positive and satisfy the following inequality:

$$\frac{d^2 A}{dr^2} > 0 .$$

Using formula (6.9) and the basic rules of differentiation [3], we obtain

$$\frac{dA}{dr} = \frac{2}{r^2}(2\pi r^3 - V) .$$

Thus, equation (6.10) becomes

$$2\pi r^3 - V = 0 .$$

This equation has only one real root:

$$r^* = \sqrt[3]{\frac{V}{2\pi}} . \qquad (6.11)$$

The second derivative of the $A(r)$ function equals

$$\frac{d^2 A}{dr^2} = \frac{4}{r^3}(\pi r^3 + V) .$$

We see that the second derivative is positive for any positive value of *r*. Therefore, formula (6.11) gives the desired optimal value of *r*.

According to formula (6.11), for $V = 1000\,\text{cm}^3$ (that is, in the case of the one-liter can), the optimal base radius is equal to r^* = 5.41926 cm. Formulas (6.8) and (6.9) lead to the following optimal values of the height and total area of the tin can: h^* = 10.83853 cm, A^* = 553.5810 cm^2.

For optimizing the one-liter can by means of subroutine `mini`, we enter the following text of program `main`, subroutine `func` and function `test` into Module1 of the BookNM workbook.

Listing 6.3

```
Dim nf As Long                     'counter of calls of func
Dim nt As Long                     'counter of calls of test

Sub main()
    Dim x() As Double
    Dim ss() As Double
    Dim n As Byte
    n = 1                               'number of variables
    ReDim x(n)
    ReDim ss(1 To n, 1 To n)
    x(1) = Range("Sheet2!G1").Value
    ss(1, 1) = Range("Sheet2!G2").Value
    If ss(1, 1) ^ 2 = 0 Then
        Range("Sheet2!A1").Value = _
            "Initial step must be increased"
        End
    End If
    nf = 0
    nt = 0
0:  Call func(n, x)     'it must be before minimization
1:  Call mini(n, x, ss, 1E-6)
End Sub

Sub func(ByVal n, ByRef x() As Double)
    Const pi As Double = 3.141592654
    nf = nf + 1
    x(0) = 2 * pi * x(1) ^ 4 + 2 * 1000 / x(1) ^ 2
End Sub

Function test(ByVal n, ByRef x() As Double, _
    ByRef z() As Double, ByVal rho, Optional alpha) _
    As Boolean
    nt = nt + 1
    Range("Sheet2!A" & CStr(nt)) = x(0)
    Range("Sheet2!B" & CStr(nt)) = x(1)
    Range("Sheet2!C" & CStr(nt)) = nt
    Range("Sheet2!D" & CStr(nt)) = nf
2:  If Abs(z(0) - x(0)) < n * rho * z(0) Then
        test = False
    Else
        test = True
    End If
    If Not IsMissing(alpha) Then
        If x(0) < alpha Then test = False
    End If
3:  If n = 1 Or nt = 1048576 Then test = False
End Function
```

When executing the `test` function, the values of `x(0)` = $F(\chi_1)$, `x(1)` = χ_1, `nt` and `nf` are put into the `nt`-th row on the Sheet2 worksheet.

The sense of the last two variables follows:

- `nt` is the current number of the iterations, i.e., of the `test` function calls;
- `nf` is the current number of the calculated values of objective function $F(\chi_1)$, i.e., of the `func` function calls.

In other words, the `nt` variable is the counter of the `test` function calls, the `nf` variable is the counter of the `func` function calls. The Excel row number may be considered as the iteration number (we will use this in Section 6.7).

The `mini` subroutine execution is terminated when the relative change of the $F(\chi_1)$ function becomes less than $\rho = 10^{-6}$: conditional operator `2` (where `n` is equal to unity) includes variables `z(0)` and `x(0)`, i.e., the $F(\chi_1)$ function values at the iteration's beginning and end, respectively.

Due to operator `3`, the `mini` subroutine execution also terminates when the number of arguments is equal to one or the Sheet2 worksheet does not contain empty rows. We will encounter operator

```
If n = 1 Or nt = 1048576 Then test = False
```

more than once in this chapter.

According to (6.9), objective function $F(\chi_1)$ has the following form:

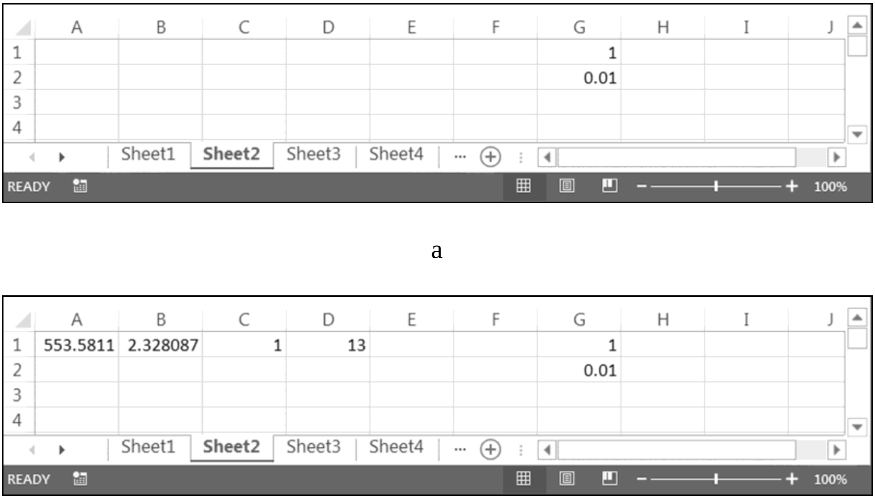
$$F(\chi_1) = A(\chi_1^2) = 2\pi\,\chi_1^4 + 2\frac{V}{\chi_1^2}\,. \qquad (6.12)$$

We made substitution $r = \chi_1^2$ in order to have only non-negative values of the r radius in the course of the A area minimization. The tin can volume is given in the `func` subroutine; it is equal to $V = 1000$.

The initial approximation of the minimum point, `x(1)` = χ_1, and initial step `ss(1, 1)` = s_{11} are respectively taken from cells G1 and G2 on the Sheet2 worksheet. The sign of s_{11} determines the direction of the initial step, and $|\,s_{11}\,|$ determines its size. Before the `mini` subroutine call (operator `1`), the initial value of objective function `x(0)` = $F(\chi_1)$ must be defined. For that, operator `0` is used, which calculates the `x(0)` value corresponding to the `x(1)` value by means of the `func` subroutine.

Fig. 6.16 shows the Sheet2 worksheet (a) before and (b) after the execution of code Listing 6.3. According to Fig. 6.16a, the initial value of the χ_1 variable equals 1, initial step s_{11} equals 0.01.

According to Fig. 6.16b, only one iteration was performed for the minimization of function (6.12) because it is a function of one variable (see operator 3); the iteration result is in cells A1 and B1. The final value of χ_1 (which is in the B1 cell) is equal to 2.328087. In this case, the radius of the tin can base, $r = \chi_1^2$ = 5.41999, is close to the optimal value, r^* = 5.41926, calculated according to formula (6.11). During the search for the minimum point, 13 values of the objective function were calculated. The final value of the objective function (which is in the A1 cell) is equal to A = 553.5811.



	A	B	C	D	E	F	G	H	I	J
1							1			
2							0.01			
3										
4										

a

	A	B	C	D	E	F	G	H	I	J
1	553.5811	2.328087	1	13			1			
2							0.01			
3										
4										

b

Fig. 6.16. The Sheet2 worksheet (a) before and (b) after minimization of $F(\chi_1)$

We see conditional operator 2 in code Listing 6.3. It can be replaced with one of the following two assignment operators with logical expression in the right-hand side:

```
test = Not Abs(z(0) - x(0)) < n * rho * z(0)
```

or

```
test = Abs(z(0) - x(0)) >= n * rho * z(0)
```

The Rosenbrock function minimization.

To find the minimum point of nonlinear function (6.1) by the coordinate-descent method, we change the text of program `main`, subroutine `func` and function `test` as follows:

Listing 6.4

```
Dim nf As Long                      'counter of calls of func
Dim nt As Long                      'counter of calls of test

Sub main()
    Dim x() As Double
    Dim ss() As Double
    Dim n As Byte
    Dim i As Byte, j As Byte
    Dim d As Double
    n = 2                                'number of variables
    ReDim x(n)
    ReDim ss(1 To n, 1 To n)
    For j = 1 To n
        x(j) = Worksheets("Sheet2").Cells(1, 6 + j)
        For i = 1 To n
            ss(i, j) = Worksheets("Sheet2"). _
                Cells(1 + i, 6 + j)
        Next i
    Next j
    For j = 1 To n
        d = 0
        For i = 1 To n
            d = ss(i, j) ^ 2 + d
        Next i
        If d = 0 Then
            Range("Sheet2!A1").Value = _
                "You must increase" & Str(j) & _
                "-th initial step"
            End
        End If
    Next j
    nf = 0
    nt = 0
    Call func(n, x)     'it must be before minimization
1:  Call mini(n, x, ss, 1E-6, 1E-6)
End Sub
```

```
Sub func(ByVal n, ByRef x() As Double)
    nf = nf + 1
    x(0) = 100 * (x(2) - x(1) ^ 2) ^ 2 + _
        (1 - x(1)) ^ 2
End Sub

Function test(ByVal n, ByRef x() As Double, _
    ByRef z() As Double, ByVal rho, Optional alpha) _
    As Boolean
    Dim j As Byte
    nt = nt + 1
    For j = 0 To n
        Worksheets("Sheet2").Cells(nt, j + 1) = x(j)
    Next j
    Worksheets("Sheet2").Cells(nt, n + 2) = nt
    Worksheets("Sheet2").Cells(nt, n + 3) = nf
2:  If Abs(z(0) - x(0)) < n * rho * z(0) Then
        test = False
    Else
        test = True
    End If
    If Not IsMissing(alpha) Then
3:      If x(0) < alpha Then test = False
    End If
    If n = 1 Or nt = 1048576 Then test = False
End Function
```

According to conditional operators `2` and `3`, the `test` function returns `False` into the `mini` subroutine when at least one condition is satisfied of the following two:

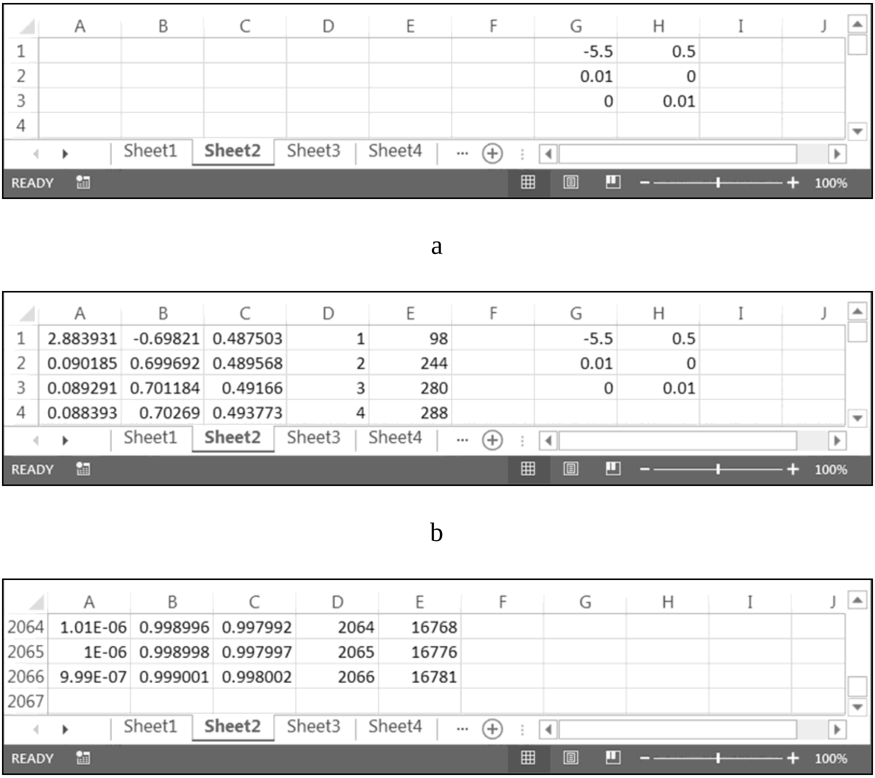
- the relative change of objective function $F(x_1, x_2)$ on one iteration becomes less than $2\rho = 2 \cdot 10^{-6}$ (the number of arguments is equal to 2);
- the objective function value becomes less than $\alpha = 10^{-6}$.

In this case, the search of the minimum point of the Rosenbrock function terminates.

When executing the `test` function, the values of `x(0)` = $F(x_1, x_2)$, `x(1)` = x_1, `x(2)` = x_2 and the current values of `nt` and `nf` (that is, the numbers of calls of `test` and `func`, respectively) are put into cells on the Sheet2 worksheet.

Fig. 6.17 shows worksheet Sheet2 (a) before and (b, c) after executing code Listing 6.4.

Coordinates `x(1)` = x_1, `x(2)` = x_2 of the initial approximation of the minimum point and also initial steps `ss(1, 1)` = s_{11}, `ss(2, 2)` = s_{22} are respectively taken from ranges G1:H1 and G2:H3 on worksheet Sheet2 (Fig. 6.17a). Before calling the `mini` subroutine, the initial value of the objective function is calculated by calling the `func` subroutine. The result of the Rosenbrock function minimization is located in range A2066:C2066 (Fig. 6.17c).



	A	B	C	D	E	F	G	H	I	J
1							-5.5	0.5		
2							0.01	0		
3							0	0.01		
4										

a

	A	B	C	D	E	F	G	H	I	J
1	2.883931	-0.69821	0.487503	1	98		-5.5	0.5		
2	0.090185	0.699692	0.489568	2	244		0.01	0		
3	0.089291	0.701184	0.49166	3	280		0	0.01		
4	0.088393	0.70269	0.493773	4	288					

b

	A	B	C	D	E	F	G	H	I	J
2064	1.01E-06	0.998996	0.997992	2064	16768					
2065	1E-06	0.998998	0.997997	2065	16776					
2066	9.99E-07	0.999001	0.998002	2066	16781					
2067										

c

Fig. 6.17. The Sheet2 worksheet (a) before and (b, c) after the execution of code Listing 6.4

According to Fig. 6.17a, the initial values of the variables equal x_1^0 = -5.5, x_2^0 = 0.5 (we used these initial values in Section 6.1 when minimizing the Rosenbrock function by the Solver add-in), initial steps s_{11} and s_{22} are equal to 0.01, and s_{12} = s_{21} = 0.

Fig. 6.17b and 6.17c show the results of the initial and final iterations. According to Fig. 6.17c, the result of the execution of code Listing 6.4 is x_1^* = x_2^* = 1, and during the execution:

- 2066 iterations were performed;
- 16781 values of objective function (6.1) were calculated.

Further, we will consider the Powell minimization method. According to Fig. 6.18 (similar to Fig. 6.17c), the method usage reduces:

- the number of iterations to 5;
- the number of the calculated values of objective function (6.1) to 376.

6.5. The Powell minimization method

Let $\mathbf{x} = (x_1, x_2, ..., x_n)$ be a point of the *n*-dimensional space, $n \geq 2$. We have to find the unique minimum point of non-negative and continuous function $F(\mathbf{x})$, which has continuous partial derivatives $\partial F / \partial x_1$, $\partial F / \partial x_2$, …, $\partial F / \partial x_n$.

As mentioned earlier, the minimization of the $F(\mathbf{x})$ function is an iterative process. According to the Powell method (based on a good theory [14]), each iteration consists of successive minimizations of the $F(\mathbf{x})$ function along directions $\mathbf{S}^1$, $\mathbf{S}^2$, ..., $\mathbf{S}^n$ (which are defined by the $\mathbf{S}$ matrix), at that, a set of directions for the next iteration is formed. More precisely, one iteration includes the following seven stages.

1. For $i = 1, 2, \ldots, n$, vectors $\mathbf{x}^i$ are defined according to recurrence formula

$$\mathbf{x}^i = \mathbf{x}^{i-1} + \lambda_i^* \mathbf{S}^i ,$$

where $\mathbf{x}^0$ is the initial approximation of the minimum point for the given iteration, λ_i^* is the minimum point of the following function of one variable:

$$f_i(\lambda) = F(\mathbf{x}^{i-1} + \lambda \mathbf{S}^i) .$$

2. The maximum change in the $F(\mathbf{x})$ function,

$$\Delta = \max_{1 \leq i \leq n} \{F(\mathbf{x}^{i-1}) - F(\mathbf{x}^i)\} ,$$

is defined. Integer *k* is defined as the serial number of the direction along which this change has happened.

3. Values $F_a = F(2\mathbf{x}^n - \mathbf{x}^0)$, $F_b = F(\mathbf{x}^n)$, $F_c = F(\mathbf{x}^0)$ are defined.

4. If $F_a - F_c < 0$ and

$$2(F_c - 2F_b + F_a)\left(\frac{F_c - F_b - \Delta}{F_a - F_c} \right)^2 < \Delta ,$$

then the 5th item is fulfilled. Otherwise, $\mathbf{x}^n$ is assigned to $\mathbf{x}^0$, and the jump to the 7th item is performed without changing the set of minimization directions.

5. Two vectors are defined according to formulas

$$\mathbf{S}^{n+1} = \mathbf{x}^n - \mathbf{x}^0,$$

$$\mathbf{x}^* = \mathbf{x}^n + \lambda^* \mathbf{S}^{n+1},$$

where λ^* is the minimum point of the following function of one variable:

$$f_{n+1}(\lambda) = F(\mathbf{x}^n + \lambda \mathbf{S}^{n+1}).$$

6. The new set of the n minimization directions (matrix $\mathbf{S}$) for the next iteration is formed as follows:

$$\mathbf{S}^1, \ldots, \mathbf{S}^{k-1}, \mathbf{S}^{k+1}, \ldots, \mathbf{S}^n, \mathbf{S}^{n+1},$$

where k is the integer defined in the 2nd item. Assignment $\mathbf{x}^0 = \mathbf{x}^*$ is performed.

7. If none of the termination conditions of the $F(\mathbf{x})$ function minimization is satisfied, the following iteration is performed.

The method of Section 6.2 is used for minimizing $f_1(\lambda)$, $f_2(\lambda)$, …, $f_n(\lambda)$, $f_{n+1}(\lambda)$.

The declaration of the `minim` subroutine, realizing the Powell method, is given below. We enter it into Module18 of the BookNM workbook.

Listing 6.5

```
Sub minim(ByVal n, ByRef x() As Double, _
    ByRef ss() As Double, ByVal rho, Optional alpha)
    Dim fa As Double, fb As Double, fc As Double, _
    fd As Double
    Dim a As Double, b As Double, c As Double, _
    d As Double, e As Double, dm As Double
    Dim s() As Double: ReDim s(n)
    Dim y() As Double: ReDim y(n)
    Dim z() As Double: ReDim z(n)
    Dim sss() As Double: ReDim sss(n, n)
    Dim i As Byte, j As Byte, k As Byte, m As Byte
    For j = 1 To n
        For i = 1 To n
            sss(i, j) = ss(i, j)
        Next i
    Next j
m1: For i = 0 To n
        y(i) = x(i)
        z(i) = x(i)
    Next i
    dm = 0
    For j = 1 To n    'j - number of descent direction
        For i = 1 To n
            s(i) = ss(i, j)
        Next i
        d = 0
        For i = 1 To n
            d = s(i) ^ 2 + d
        Next i
        If d = 0 Then
            For i = 1 To n
                s(i) = sss(i, j)
            Next i
        Else
            For i = 1 To n
                sss(i, j) = s(i)
            Next i
        End If
'Finding uncertainty segment:
        fa = y(0): fb = y(0): fc = y(0)
        a = 0: b = 0: c = 0
        d = 1: e = 1
s1:     For i = 1 To n
            x(i) = y(i) + d * s(i)
        Next i
        Call func(n, x)
        If Not IsMissing(alpha) Then
            If x(0) < alpha Then GoTo m3
        End If
        fd = x(0)
        If fd < fa Then
            fc = fb: fb = fa: fa = fd
            c = b: b = a: a = d
            d = 2 * d + e
            GoTo s1
        Else
            If fa = fb Then
```

```
                fb = fd
                b = d
                e = -2 * d
                d = e
                GoTo s1
            End If
        End If
        fc = fb: fb = fa: fa = fd
        c = b: b = a: a = d
        d = (a + b) * 0.5
        For i = 1 To n
            x(i) = y(i) + d * s(i)
        Next i
        Call func(n, x)
        If Not IsMissing(alpha) Then
            If x(0) < alpha Then GoTo m3
        End If
        fd = x(0)
'Reducing uncertainty segment:
s2:     If Abs(fb - fd) < rho * fb Then GoTo s0
        If (c - d) * (d - b) < 0 Then
            If fd < fb Then
                fc = fb: fb = fd
                c = b: b = d
            Else
                fa = fd
                a = d
            End If
        Else
            If fd < fb Then
                fa = fb: fb = fd
                a = b: b = d
            Else
                fc = fd
                c = d
            End If
        End If
        d = fa * (b - c) + fb * (c - a) + fc * (a - b)
        If d = 0 Then GoTo s0
        d = (fa * (b * b - c * c) + _
            fb * (c * c - a * a) + _
            fc * (a * a - b * b)) / (2 * d)
        For i = 1 To n
            x(i) = y(i) + d * s(i)
        Next i
        Call func(n, x)
        If Not IsMissing(alpha) Then
            If x(0) < alpha Then GoTo m3
        End If
        fd = x(0)
        GoTo s2
s0:     d = y(0) - x(0)
        If d > dm Then
            dm = d
            m = j
        End If
        For i = 1 To n
            ss(i, j) = x(i) - y(i)
            y(i) = x(i)
        Next i
        y(0) = x(0)
    Next j
    If n = 1 Then GoTo m3
'Last descent:
    For i = 1 To n
        x(i) = 2 * y(i) - z(i)
    Next i
    Call func(n, x)
    If Not IsMissing(alpha) Then
        If x(0) < alpha Then GoTo m3
    End If
    fa = x(0): fb = y(0): fc = z(0)
    a = fa - fc
    If a >= 0 Then GoTo m2
    a = (fc - fb - dm) / a
    If 2 * (fc - 2 * fb + fa) * a ^ 2 >= dm Then _
        GoTo m2
    For j = m To n - 1
        k = j + 1
        For i = 1 To n
            ss(i, j) = ss(i, k)
        Next i
    Next j
    For i = 1 To n
```

```
        s(i) = y(i) - z(i)
    Next i
'Finding uncertainty segment:
    fa = y(0): fb = y(0): fc = y(0)
    a = 0: b = 0: c = 0
    d = 1: e = 1
h1: For i = 1 To n
        x(i) = y(i) + d * s(i)
    Next i
    Call func(n, x)
    If Not IsMissing(alpha) Then
        If x(0) < alpha Then GoTo m3
    End If
    fd = x(0)
    If fd < fa Then
        fc = fb: fb = fa: fa = fd
        c = b: b = a: a = d
        d = 2 * d + e
        GoTo h1
    Else
        If fa = fb Then
            fb = fd
            b = d
            e = -2 * d
            d = e
            GoTo h1
        End If
    End If
    fc = fb: fb = fa: fa = fd
    c = b: b = a: a = d
    d = (a + b) * 0.5
    For i = 1 To n
        x(i) = y(i) + d * s(i)
    Next i
    Call func(n, x)
    If Not IsMissing(alpha) Then
        If x(0) < alpha Then GoTo m3
    End If
    fd = x(0)
'Reducing uncertainty segment:
h2: If Abs(fb - fd) < rho * fb Then GoTo h0
    If (c - d) * (d - b) < 0 Then
```

```
        If fd < fb Then
            fc = fb: fb = fd
            c = b: b = d
        Else
            fa = fd
            a = d
        End If
    Else
        If fd < fb Then
            fa = fb: fb = fd
            a = b: b = d
        Else
            fc = fd
            c = d
        End If
    End If
    d = fa * (b - c) + fb * (c - a) + fc * (a - b)
    If d = 0 Then GoTo h0
    d = (fa * (b * b - c * c) + _
        fb * (c * c - a * a) + _
        fc * (a * a - b * b)) / (2 * d)
    For i = 1 To n
        x(i) = y(i) + d * s(i)
    Next i
    Call func(n, x)
    If Not IsMissing(alpha) Then
        If x(0) < alpha Then GoTo m3
    End If
    fd = x(0)
    GoTo h2
h0: For i = 1 To n
        ss(i, n) = x(i) - y(i)
        y(i) = x(i)
    Next i
    y(0) = x(0)
'Writing result of descents into array x:
m2: For i = 0 To n
        x(i) = y(i)
    Next i
'Checking condition of minimization termination:
m3: If Not IsMissing(alpha) Then
        If test(n, x, z, rho, alpha) Then GoTo m1
    Else
        If test(n, x, z, rho) Then GoTo m1
    End If
End Sub
```

The `minim` subroutine parameters have the same sense as the corresponding parameters of the `mini` subroutine (p. 435). However, two-dimensional array `ss` may be non-diagonal.

In the `minim` subroutine, elements `ss(1, j)`, `ss(2, j)`, ..., `ss(n, j)` of the `ss` array, corresponding to the *j*-th column ($\mathbf{S}^j$) of the $\mathbf{S}$ matrix, define the *j*-th descent direction and the initial step along this direction, which equals $s_j = \sqrt{s_{1j}^2 + s_{2j}^2 + \ldots + s_{nj}^2}$, $j = 1, 2, \ldots, n$. The descent directions, defined by the `ss` array (matrix $\mathbf{S}$), is generally changing from iteration to iteration during the $F(x_1, x_2, \ldots, x_n)$ function minimization.

Let us test the new subroutine by means of the Rosenbrock function. For this purpose, we replace line

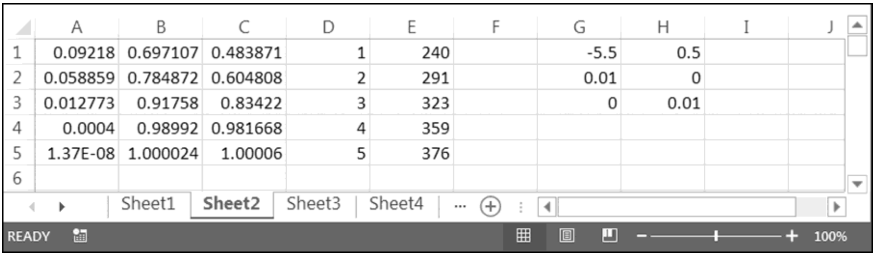
```
1:  Call mini(n, x, ss, 1E-6, 1E-6)
```

with line

```
1:  Call minim(n, x, ss, 1E-6, 1E-6)
```

in Listing 6.4. We leave subroutine `func`, function `test` and the initial data without change.

The result of the Rosenbrock function minimization by the Powell method is located in cells A5:C5 (Fig. 6.18).



	A	B	C	D	E	F	G	H	I	J
1	0.09218	0.697107	0.483871	1	240		-5.5	0.5		
2	0.058859	0.784872	0.604808	2	291		0.01	0		
3	0.012773	0.91758	0.83422	3	323		0	0.01		
4	0.0004	0.98992	0.981668	4	359					
5	1.37E-08	1.000024	1.00006	5	376					
6										

Fig. 6.18. The Sheet2 worksheet upon termination of the code execution

We see that the obtained minimum point is the same as when using the Solver add-in and coordinate-descent method: $x_1^* = x_2^* = 1$. The minimization of function (6.1) requires:

- 5 iterations of the Powell method (Fig. 6.18);
- 21 iterations of Solver (p. 427);
- 2066 iterations of the coordinate-descent method (p. 445).

Thus, the Powell method is more efficient than the Solver add-in and coordinate-descent method for the Rosenbrock function. This is due to the following:

1) minimum point $\mathbf{x}^* = (1, 1)$ is located in the ravine (p. 422);

2) in the Powell method, the initial descent directions, which are parallel to the x_1 and x_2 axes, are being converted to the descent directions, $\mathbf{S}^1$ and $\mathbf{S}^2$, oriented along the ravine bottom (Fig. 6.19).

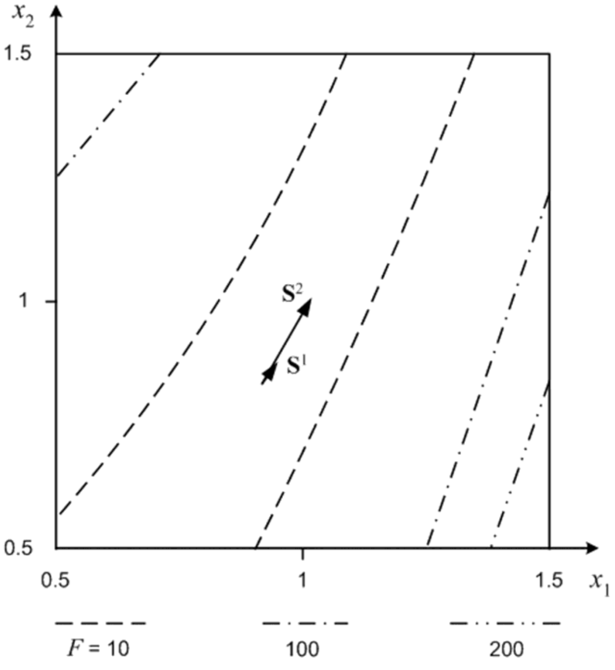


Fig. 6.19. The level curves of the Rosenbrock function in the neighborhood of the minimum point and vectors $\mathbf{S}^1$ and $\mathbf{S}^2$ for the 4th iteration

The iteration results, $(x_1^1,\ x_2^1)$, $(x_1^2,\ x_2^2)$, $(x_1^3,\ x_2^3)$, $(x_1^4,\ x_2^4)$, …, approach minimum point $\mathbf{x}^* = (1,\ 1)$ along the ravine bottom. This assertion follows from Fig. 6.18 and 6.19.

6.6. Determining the equilibrium state of a four-spring system

Below, we will determine the equilibrium state of the mechanical system depicted in Fig. 6.20. This system is formed by four weightless springs of the same length, $\lambda = \sqrt{2}$ meter (without a load), but with different elastic constants. The springs are located in the plane of the paper with axes x_1 and x_2. One ends of the springs are fastened together, the other ends are attached to tops A_1, A_2, A_3, A_4 of an imaginary square with the 2-meter side.

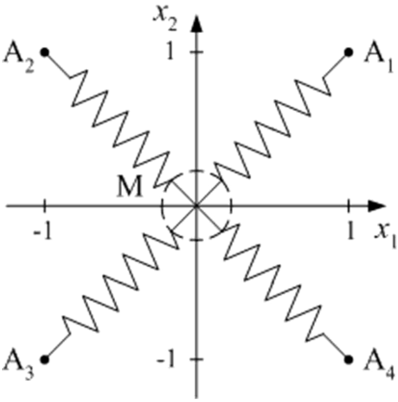


Fig. 6.20. The four-spring system: the x_3 axis is directed “towards us” and passes through the springs junction point with zero coordinates

The outside force, vector **f** directed along the x_3 axis (for example, the force of gravity), acts on body M attached to the springs junction. The value of this force, $f = \pm |\,\mathbf{f}\,|$, is given, and this value is negative if vector **f** and axis x_3 do not coincide in direction (according to the common practice in physics), i.e., the vector is directed “from us” (Fig. 6.20). The elastic constants of the springs equal K_1, K_2, K_3, K_4, respectively.

We have to find coordinates x_1, x_2, x_3 of the springs junction point in the equilibrium state, which is the result of a damped oscillation of the mechanical system. The damping occurs, for example, because of the air resistance: the resistance force is proportional to the velocity of body M and is directed opposite to the velocity vector.

According to the principle of minimum potential energy of the elasticity theory, the work of the elasticity forces and of the outside force for relocation of the M body from the origin of coordinates to the point with coordinates x_1, x_2, x_3 (that is, the potential energy of body M) assumes its minimum value in the equilibrium state. We will use this principle.

Up to a constant, the potential energy of body M is equal to

$$F(x_1,\ x_2,\ x_3) = \sum_{m=1}^{4} \frac{K_m}{2}\left[\sqrt{(x_1 - a_{m1})^2 + (x_2 - a_{m2})^2 + (x_3 - a_{m3})^2} - \lambda\right]^2 - \\ - f\,x_3 + C\,, \qquad (6.13)$$

where a_{m1}, a_{m2} are the first two coordinates of point A$_m$, which are equal to ±1, a_{m3} = 0 is the third coordinate of point A$_m$ (m = 1, 2, 3, 4), C is a positive constant introduced to ensure the positivity of function $F(x_1,\ x_2,\ x_3)$. Difference $\sqrt{(x_1 - a_{m1})^2 + (x_2 - a_{m2})^2 + (x_3 - a_{m3})^2} - \lambda$ is the change in the length of the m-th spring.

For the minimization of function (6.13), initially, we use the coordinate-descent method. The corresponding code is given below.

Listing 6.6

```
Dim nf As Long                  'counter of calls of func
Dim nt As Long                  'counter of calls of test
Dim K1 As Double, K2 As Double
Dim K3 As Double, K4 As Double
Dim f As Double, C As Double

Sub main()
    Dim x() As Double
    Dim ss() As Double
    Dim n As Byte, i As Byte, j As Byte
    Dim d As Double
    K1 = Range("Sheet2!G5").Value
    K2 = Range("Sheet2!H5").Value
    K3 = Range("Sheet2!I5").Value
    K4 = Range("Sheet2!J5").Value
    f = Range("Sheet2!G6").Value
    C = Range("Sheet2!H6").Value
    n = 3                             'number of variables
    ReDim x(n)
    ReDim ss(1 To n, 1 To n)
    For j = 1 To n
1:      x(j) = Worksheets("Sheet2").Cells(1, 6 + j)
        For i = 1 To n
2:          ss(i, j) = Worksheets("Sheet2"). _
                Cells(1 + i, 6 + j)
        Next i
    Next j
    For j = 1 To n
        d = 0
        For i = 1 To n
            d = ss(i, j) ^ 2 + d
        Next i
        If d = 0 Then
            Range("Sheet2!A1").Value = _
                "You must increase" & Str(j) & _
                "-th initial step"
            End
        End If
    Next j
    nf = 0
    nt = 0
    Call func(n, x)    'it must be before minimization
3:  Call mini(n, x, ss, 1E-6)
End Sub

Sub func(ByVal n, ByRef x() As Double)
    Const lambda As Double = 1.414213562
    nf = nf + 1
    x(0) = K1 / 2 * (Sqr((x(1) - 1) ^ 2 + _
        (x(2) - 1) ^ 2 + x(3) ^ 2) - lambda) ^ 2 + _
        K2 / 2 * (Sqr((x(1) + 1) ^ 2 + _
        (x(2) - 1) ^ 2 + x(3) ^ 2) - lambda) ^ 2 + _
        K3 / 2 * (Sqr((x(1) + 1) ^ 2 + _
        (x(2) + 1) ^ 2 + x(3) ^ 2) - lambda) ^ 2 + _
        K4 / 2 * (Sqr((x(1) - 1) ^ 2 + _
        (x(2) + 1) ^ 2 + x(3) ^ 2) - lambda) ^ 2 - _
        f * x(3) + C
End Sub

Function test(ByVal n, ByRef x() As Double, _
    ByRef z() As Double, ByVal rho, Optional alpha) _
    As Boolean
    Dim j As Byte
    nt = nt + 1
    For j = 0 To n
        Worksheets("Sheet2").Cells(nt, j + 1) = x(j)
    Next j
    Worksheets("Sheet2").Cells(nt, n + 2) = nt
    Worksheets("Sheet2").Cells(nt, n + 3) = nf
    If Abs(z(0) - x(0)) < n * rho * z(0) Then
        test = False
    Else
        test = True
    End If
    If Not IsMissing(alpha) Then
        If x(0) < alpha Then test = False
    End If
    If n = 1 Or nt = 1048576 Then test = False
End Function
```

The `test` function returns `False` when the relative change of function (6.13) on one iteration becomes less than $3\rho = 3\cdot10^{-6}$ (the number of arguments is equal to 3). In this case, the search of the minimum point terminates.

When executing the `test` function, the values of `x(0)`, `x(1)`, `x(2)`, `x(3)`, `nt` and `nf` are put into cells on the Sheet2 worksheet.

By means of operators `1` and `2`, coordinates `x(1)`, `x(2)`, `x(3)` of the initial approximation of the minimum point and initial steps `ss(1, 1)`, `ss(2, 2)`, `ss(3, 3)` are respectively taken from ranges G1:I1 and G2:I4 on the Sheet2 worksheet. Assignment operators `1` and `2` interpret the contents of the empty cells as zero.

Elastic constants K_1, K_2, K_3, K_4 of the springs (in units of $\mathrm{N/m}$) are respectively taken from cells G5, H5, I5, J5. The value of force *f* (in newtons) is taken from G6; constant *C* is taken from H6.

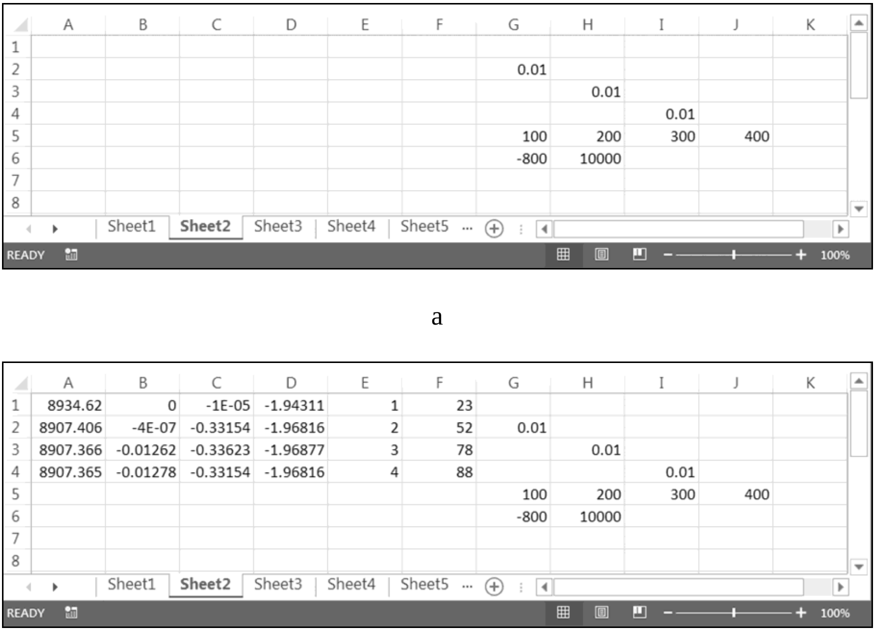
According to Fig. 6.21a:

- the initial values of variables x_1, x_2, x_3 are equal to zero;

- initial steps s_{11}, s_{22}, s_{33} are equal to 0.01;
- the elastic constants have the following values: K_1 = 100, K_2 = 200, K_3 = 300, K_4 = 400;
- the force: f = -800;
- the constant: C = 10000.

According to Fig. 6.21b:

- the springs junction point has coordinates x_1^* = -0.01278, x_2^* = -0.33154, x_3^* = -1.96816 in the equilibrium state (the coordinates are in meters);
- the number of iterations equals 4;
- 88 values of objective function (6.13) were calculated during the minimization.



	A	B	C	D	E	F	G	H	I	J	K
1											
2							0.01				
3								0.01			
4									0.01		
5							100	200	300	400	
6							-800	10000			
7											
8											

a

	A	B	C	D	E	F	G	H	I	J	K
1	8934.62	0	-1E-05	-1.94311	1	23					
2	8907.406	-4E-07	-0.33154	-1.96816	2	52	0.01				
3	8907.366	-0.01262	-0.33623	-1.96877	3	78		0.01			
4	8907.365	-0.01278	-0.33154	-1.96816	4	88			0.01		
5							100	200	300	400	
6							-800	10000			
7											
8											

b

Fig. 6.21. The Sheet2 worksheet (a) before and (b) after the code execution

The use of the Powell method for the minimization of function (6.13) is not difficult. For this purpose, we have to replace `mini` with `minim` in operator `3`.

Upon the change in the minimization method, only the number of the calculated values of objective function (6.13) changes markedly — from 88 to 95.

The above minimization problem is the only one in this chapter whose solution by the Powell method is not more efficient than by the coordinate-descent method.

6.7. Minimization with nonlinear constraints

Minimizing nonlinear function $x_0 = F(x_1, x_2, ..., x_n)$ with constraints is a frequently encountered problem. Often the following *m* inequalities play the role of these constraints:

$$\begin{aligned} &C_1(x_1, x_2, ..., x_n) \ge 0, \\ &C_2(x_1, x_2, ..., x_n) \ge 0, \\ &\cdot \quad \cdot \quad \cdot \quad \cdot \quad \cdot \quad \cdot \\ &C_m(x_1, x_2, ..., x_n) \ge 0, \end{aligned} \tag{6.14}$$

where $C_i(x_1, x_2, ..., x_n)$ is a given dependence, $1 \le i \le m$. If this dependence is nonlinear, the corresponding inequality is called a nonlinear constraint. Later we will consider a constraint of the equality type.

In the case of simple dependences $C_i(x_1, x_2, ..., x_n)$, $1 \le i \le m$, the formulated minimization problem can be solved by replacing the variables.

For $n = 1$, we already replaced the variable for solving the task of optimizing a tin can in Section 6.4. In order to have only non-negative values of radius *r* in the course of minimization of area *A*, we minimized a function with argument χ_1 related with *r* as $r = \chi_1^2$, instead of minimizing the $A(r)$ function.

For $n > 1$, we will consider ***the variable replacement method*** on an example of Rosenbrock function (6.1) with constraints $x_1 \le 1$ and $|x_2| \le 0.7$, which can be written in form (6.14),

$$1 - x_1 \ge 0, \tag{6.15}$$

$$0.7 - |x_2| \ge 0. \tag{6.16}$$

We introduce new variables χ_1 and χ_2 defined by equations

$$x_1 = 1 - \chi_1^2, \tag{6.17}$$

$$x_2 = 0.7 \sin \chi_2. \tag{6.18}$$

It is visible that variables x_1 and x_2 satisfy the required inequalities, (6.15) and (6.16), for all values of χ_1 and χ_2 from $-\infty$ to $+\infty$. Substituting expressions (6.17) and (6.18) into formula (6.1), we get the following objective function:

$$G(\chi_1,\ \chi_2) = 100[0.7\sin\chi_2 - (1-\chi_1^2)^2]^2 + \chi_1^4. \qquad (6.19)$$

After obtaining the minimum point of this function, $\boldsymbol{\chi}^* = (\chi_1^*,\ \chi_2^*)$, the required values, x_1^* and x_2^*, must be calculated by means of formulas (6.17) and (6.18).

The text of program `main`, subroutine `func` and function `test`, intended for minimizing function (6.1) with constraints (6.15) and (6.16), has the following form:

Listing 6.7

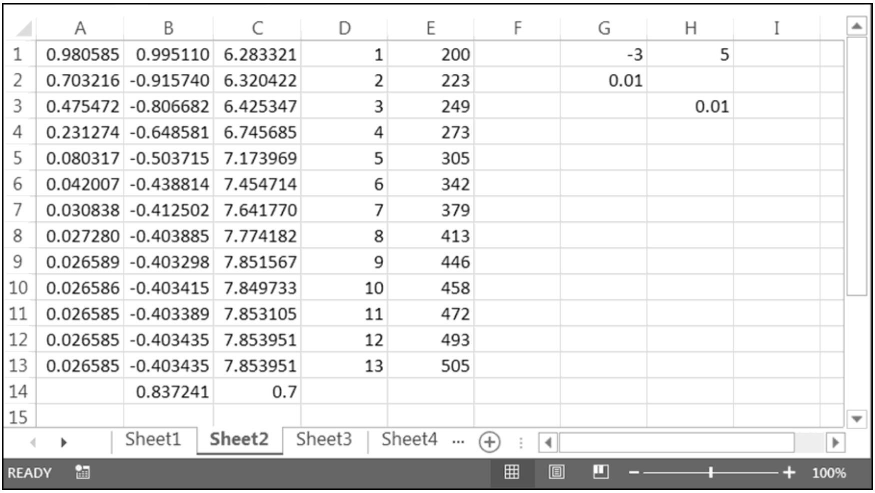
```
Dim nf As Long                    'counter of calls of func
Dim nt As Long                    'counter of calls of test

Sub main()
    Dim x() As Double
    Dim ss() As Double
    Dim n As Byte
    Dim i As Byte, j As Byte
    Dim d As Double
    n = 2                                  'number of variables
    ReDim x(n)
    ReDim ss(1 To n, 1 To n)
    For j = 1 To n
        x(j) = Worksheets("Sheet2").Cells(1, 6 + j)
        For i = 1 To n
            ss(i, j) = Worksheets("Sheet2"). _
            Cells(1 + i, 6 + j)
        Next i
    Next j
    For j = 1 To n
        d = 0
        For i = 1 To n
            d = ss(i, j) ^ 2 + d
        Next i
        If d = 0 Then
            Range("Sheet2!A1").Value = _
                "You must increase" & Str(j) & _
                "-th initial step"
            End
        End If
    Next j
    nf = 0
    nt = 0
    Call func(n, x)     'it must be before minimization
1:  Call minim(n, x, ss, 1E-6, 1E-6)
2:  Worksheets("Sheet2").Cells(nt + 1, 2) = _
        1 - x(1) ^ 2
3:  Worksheets("Sheet2").Cells(nt + 1, 3) = _
        0.7 * Sin(x(2))
End Sub

Sub func(ByVal n, ByRef x() As Double)
    Dim x1 As Double, x2 As Double
    nf = nf + 1
    x1 = 1 - x(1) ^ 2
    x2 = 0.7 * sin(x(2))
    x(0) = 100 * (x2 - x1 ^ 2) ^ 2 + x(1) ^ 4
End Sub

Function test(ByVal n, ByRef x() As Double, _
    ByRef z() As Double, ByVal rho, Optional alpha) _
    As Boolean
    Dim j As Byte
    nt = nt + 1
    For j = 0 To n
        Worksheets("Sheet2").Cells(nt, j + 1) = x(j)
    Next j
    Worksheets("Sheet2").Cells(nt, n + 2) = nt
    Worksheets("Sheet2").Cells(nt, n + 3) = nf
    If Abs(z(0) - x(0)) < n * rho * z(0) Then
        test = False
    Else
        test = True
    End If
    If Not IsMissing(alpha) Then
        If x(0) < alpha Then test = False
    End If
    If n = 1 Or nt = 1048576 Then test = False
End Function
```

From code Listing 6.4 for solving the minimization problem without constraints, the above code differs in the following:

- in operators of the `func` subroutine because it is now intended for calculating values of objective function $G(\chi_1, \chi_2)$ according to formula (6.19) instead of (6.1);
- in operator `1` because we use the Powell method in the new program;
- in the presence of operators `2` and `3`, which correspond to formulas (6.17) and (6.18), respectively.

After obtaining the values of χ_1^* and χ_2^*, operators `2` and `3` calculate the values of x_1^* and x_2^*, i.e., the solution of the minimization problem for the Rosenbrock function with constraints (6.15) and (6.16).

Fig. 6.22 shows the Sheet2 worksheet upon termination of the code execution.



	A	B	C	D	E	F	G	H	I
1	0.980585	0.995110	6.283321	1	200		-3	5	
2	0.703216	-0.915740	6.320422	2	223		0.01		
3	0.475472	-0.806682	6.425347	3	249			0.01	
4	0.231274	-0.648581	6.745685	4	273				
5	0.080317	-0.503715	7.173969	5	305				
6	0.042007	-0.438814	7.454714	6	342				
7	0.030838	-0.412502	7.641770	7	379				
8	0.027280	-0.403885	7.774182	8	413				
9	0.026589	-0.403298	7.851567	9	446				
10	0.026586	-0.403415	7.849733	10	458				
11	0.026585	-0.403389	7.853105	11	472				
12	0.026585	-0.403435	7.853951	12	493				
13	0.026585	-0.403435	7.853951	13	505				
14		0.837241	0.7						
15									

Fig. 6.22. The worksheet upon termination of the code execution

According to Fig. 6.22:

- the initial approximation of the minimum point, defined by the values of cells G1 and H1, has coordinates χ_1^0 = -3 and χ_2^0 = 5;
- the initial steps, defined by the values of cells G2:H3, are directed along the χ_1 and χ_2 axes and are equal to 0.01;

- the coordinates of the minimum point of function $G(\chi_1, \chi_2)$ are χ_1^* = -0.403 and χ_2^* = 7.854;
- the required values are x_1^* = 0.837 and x_2^* = 0.7; they are located in cells B14 and C14.

When replacing `minim` with `mini` in operator `1` (that is, when using the coordinate-descent method) the number of iterations increases from 13 to 254, and the number of the calculated values of objective function $G(\chi_1, \chi_2)$ increases from 505 to 2372.

The application of the variable replacement method, which was considered, is very limited. For minimizing nonlinear function $F(x_1, x_2, ..., x_n)$ with constraints (6.14), ***the penalty function method*** is more universal. According to this method, functions

$$G_k(\mathbf{x}) = F(\mathbf{x}) + 2^k[D_1(\mathbf{x}) + D_2(\mathbf{x}) + ... + D_m(\mathbf{x})] \qquad (6.20)$$

of vector argument $\mathbf{x} = (x_1, x_2, ..., x_n)$, $k \geq 0$, are being sequentially minimized for k = 0, 1, 2, ... The value of k determines the “weight” of sum $D_1(\mathbf{x}) + D_2(\mathbf{x}) + ... + D_m(\mathbf{x})$, i.e., its contribution to the $G_k(\mathbf{x})$ function; m is the number of the constraints.

We use formula

$$D_i(\mathbf{x}) = -\min\{C_i^p(\mathbf{x}), 0\},$$

where the p power is an odd natural number, for example 3, $1 \leq i \leq m$.

According to the last formula:

- $D_i(\mathbf{x}) = 0$ when $C_i(\mathbf{x}) \geq 0$, i.e., the i-th inequality of (6.14) is satisfied;
- $D_i(\mathbf{x}) = -C_i^p(\mathbf{x}) > 0$ when $C_i(\mathbf{x}) < 0$, i.e., the i-th inequality of (6.14) is unsatisfied.

We can combine functions (6.20) into the following single function with additional argument x_{n+1}:

$$G(\mathbf{x}, x_{n+1}) = F(\mathbf{x}) + x_{n+1}[D_1(\mathbf{x}) + D_2(\mathbf{x}) + ... + D_m(\mathbf{x})]. \qquad (6.21)$$

The original problem with constraints is reduced to a sequence of problems without constraints: we must find the minimum point of each of functions $G_0(\mathbf{x}) = G(\mathbf{x}, 1)$, $G_1(\mathbf{x}) = G(\mathbf{x}, 2)$, $G_2(\mathbf{x}) = G(\mathbf{x}, 4)$, ..., $G_k(\mathbf{x}) = G(\mathbf{x}, 2^k)$, ...

It is obvious that the obtained sequence of minimum points $\mathbf{x}_0^*$, $\mathbf{x}_1^*$, $\mathbf{x}_2^*$, ..., $\mathbf{x}_k^*$, ... converges to required solution $\mathbf{x}^*$ of the minimization problem for function $F(\mathbf{x})$ with constraints (6.14).

For a fixed value of *k*, one of the considered methods for unconstrained minimization (the coordinate-descent or Powell method) minimizes function $G_k(\mathbf{x}) = G(\mathbf{x},\, 2^k)$, at that, minimum point $\mathbf{x}_{k-1}^*$ of $G_{k-1}(\mathbf{x}) = G(\mathbf{x},\, 2^{k-1})$ is used as the initial approximation of the $\mathbf{x}_k^*$ minimum point of function $G_k(\mathbf{x}) = G(\mathbf{x},\, 2^k)$.

We must specify initial approximation $\mathbf{x}^0$ of minimum point $\mathbf{x}_0^*$ of function $G_0(\mathbf{x}) = G(\mathbf{x},\, 1)$. The $\mathbf{x}^0$ point may be considered as the initial approximation of required $\mathbf{x}^*$.

Functions (6.20) are called the penalty functions. Let also:

- function (6.21) of form

$$G(x_1,\, x_2,\, ...,\, x_n,\, x_{n+1}) = F(\mathbf{x}) + x_{n+1}[D_1(\mathbf{x}) + D_2(\mathbf{x}) + ... + D_m(\mathbf{x})]$$

be called the penalty function;

- summand $x_{n+1}[D_1(\mathbf{x}) + D_2(\mathbf{x}) + ... + D_m(\mathbf{x})]$ be called the penalty.

The penalty function method allows to solve the minimization problem for Rosenbrock function (6.1) with constraint $x_1^2 + x_2^2 \le 1$, which can be written in form (6.14),

$$C_1(x_1,\; x_2) \ge 0\,, \tag{6.22}$$

where

$$C_1(x_1,\; x_2) = -x_1^2 - x_2^2 + 1. \tag{6.23}$$

To use the Powell method for minimizing function (6.1) with this constraint, we must enter the following code into Module1.

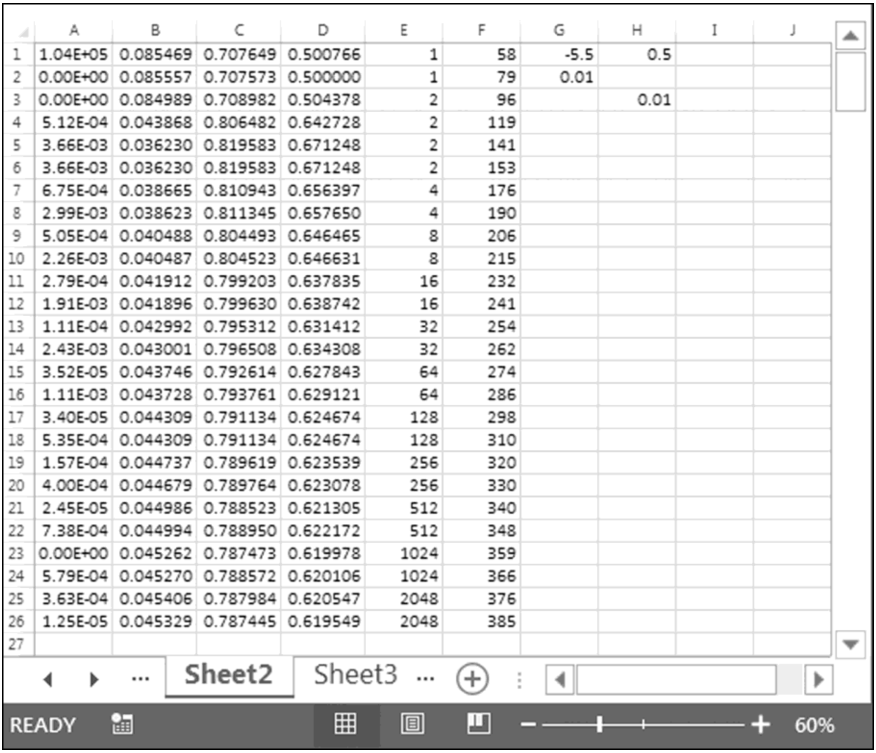
Listing 6.8

```
Const DBL_MAX = 1E+308
Dim nf As Long                  'counter of calls of func
Dim nt As Long                  'counter of calls of test
Dim m As Byte
Dim y0 As Double, z0 As Double
```

```
Sub main()
    Dim x() As Double
    Dim ss() As Double
    Dim n As Byte, i As Byte, j As Byte
    Dim d As Double
    n = 2                           'number of variables
    m = 1                           'number of constraints
    ReDim x(-1 To n + m)
    ReDim ss(1 To n, 1 To n)
    For j = 1 To n
        x(j) = Worksheets("Sheet2").Cells(1, 6 + j)
        For i = 1 To n
            ss(i, j) = Worksheets("Sheet2"). _
                Cells(1 + i, 6 + j)
        Next i
    Next j
    For j = 1 To n
        d = 0
        For i = 1 To n
            d = ss(i, j) ^ 2 + d
        Next i
        If d = 0 Then
            Range("Sheet2!A1").Value = _
                "You must increase" & Str(j) & _
                "-th initial step"
            End
        End If
    Next j
    nf = 0
    nt = 0
    x(n + 1) = 1
    Call func(n, x)     'it must be before minimization
    z0 = x(0): y0 = DBL_MAX
1:  Call minim(n, x, ss, 1E-3, 1E-6)
End Sub

Sub func(ByVal n, ByRef x() As Double)
    Dim c As Double
    nf = nf + 1
    c = -x(1) ^ 2 - x(2) ^ 2 + 1
2:  If c > = 0 Then
        x(-1) = 0
    Else
        x(-1) = x(3) * (-c ^ 3)        'penalty, p = 3
    End If
    x(0) = 100 * (x(2) - x(1) ^ 2) ^ 2 + _
        (1 - x(1)) ^ 2 + x(-1)
End Sub

Function test(ByVal n, ByRef x() As Double, _
    ByRef z() As Double, ByVal rho, Optional alpha) _
    As Boolean
    Dim j As Integer
    nt = nt + 1
    For j = -1 To n + m
        Worksheets("Sheet2").Cells(nt, j + 2) = x(j)
    Next j
    Worksheets("Sheet2").Cells(nt, n + m + 3) = nf
    If Abs(z0 - x(0)) < n * rho * z0 Then
        If Abs(y0 - x(0)) < n * rho * y0 Then
            test = False
        Else
            x(n + 1) = x(n + 1) * 2
            If x(n + 1) > DBL_MAX Then
                test = False
            Else
                test = True
            End If
            y0 = x(0)
            Call func(n, x)
        End If
    Else
        test = True
    End If
    z0 = x(0)
    If Not IsMissing(alpha) Then
        If x(0) < alpha Then test = False
    End If
    If n = 1 Or nt = 1048576 Then test = False
End Function
```

The results of the code execution are depicted in Fig. 6.23. As we see, initial approximation $\mathbf{x}^0$, defined by the values of cells G1 and H1, has coordinates

x_1^0 = -5.5 and x_2^0 = 0.5. According to the contents of cells G2:H3, the initial steps are directed along the x_1 and x_2 axes and equal 0.01.



	A	B	C	D	E	F	G	H	I	J
1	1.04E+05	0.085469	0.707649	0.500766	1	58	-5.5	0.5		
2	0.00E+00	0.085557	0.707573	0.500000	1	79	0.01			
3	0.00E+00	0.084989	0.708982	0.504378	2	96		0.01		
4	5.12E-04	0.043868	0.806482	0.642728	2	119				
5	3.66E-03	0.036230	0.819583	0.671248	2	141				
6	3.66E-03	0.036230	0.819583	0.671248	2	153				
7	6.75E-04	0.038665	0.810943	0.656397	4	176				
8	2.99E-03	0.038623	0.811345	0.657650	4	190				
9	5.05E-04	0.040488	0.804493	0.646465	8	206				
10	2.26E-03	0.040487	0.804523	0.646631	8	215				
11	2.79E-04	0.041912	0.799203	0.637835	16	232				
12	1.91E-03	0.041896	0.799630	0.638742	16	241				
13	1.11E-04	0.042992	0.795312	0.631412	32	254				
14	2.43E-03	0.043001	0.796508	0.634308	32	262				
15	3.52E-05	0.043746	0.792614	0.627843	64	274				
16	1.11E-03	0.043728	0.793761	0.629121	64	286				
17	3.40E-05	0.044309	0.791134	0.624674	128	298				
18	5.35E-04	0.044309	0.791134	0.624674	128	310				
19	1.57E-04	0.044737	0.789619	0.623539	256	320				
20	4.00E-04	0.044679	0.789764	0.623078	256	330				
21	2.45E-05	0.044986	0.788523	0.621305	512	340				
22	7.38E-04	0.044994	0.788950	0.622172	512	348				
23	0.00E+00	0.045262	0.787473	0.619978	1024	359				
24	5.79E-04	0.045270	0.788572	0.620106	1024	366				
25	3.63E-04	0.045406	0.787984	0.620547	2048	376				
26	1.25E-05	0.045329	0.787445	0.619549	2048	385				
27										

Fig. 6.23. The Sheet2 worksheet upon termination of the code execution

Columns A and B contain the values of penalty `x(-1)` = $x_3 D_1(x_1, x_2)$ and penalty function `x(0)` = $G(x_1, x_2, x_3)$, respectively. Columns C, D and E contain the values of coordinates `x(1)` = x_1 and `x(2)` = x_2 and variable `x(3)` = $x_3 = 2^k$. Column F contains the current value of `nf`, i.e., the current number of the `func` subroutine calls. The coordinates of the obtained minimum point, $\mathbf{x}^*$, are equal to x_1^* = 0.787 and x_2^* = 0.62.

When replacing `minim` with `mini` in operator `1` (that is, when using the coordinate-descent method), the number of iterations increases from 26 to 153, and the number of the calculated values of the objective function increases from 385 to 1313.

The penalty function method allows solving the minimization problem for the $F(x_1,\ x_2, ..., x_n)$ function when the equality type constraints are present among constraints (6.14); for example, $C_1(x_1,\ x_2, ..., x_n) = 0$ may be the first constraint. In this case, we use $D_1(\mathbf{x}) = C_1^q(\mathbf{x})$ in expressions (6.20) and (6.21), where power q is an even natural number, for example 2.

As an example of using this form of the penalty function method, we will consider the minimization problem for Rosenbrock function (6.1) with constraint $x_1^2 + x_2^2 = 1$, which can be written as follows:

$$C_1(x_1,\ x_2) = 0\ ,$$

where the left-hand side is defined by (6.23), i.e., $C_1(x_1,\ x_2) = -x_1^2 - x_2^2 + 1$.

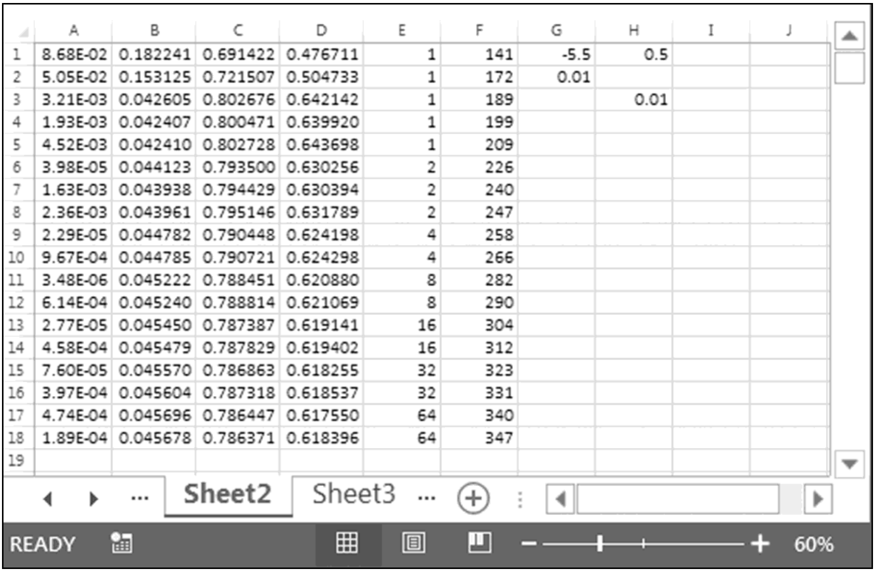
To minimize function (6.1) with this constraint, we must change the `func` subroutine declaration in code Listing 6.8 as follows:

Listing 6.9

```
Sub func(ByVal n, ByRef x() As Double)
    Dim c As Double
    nf = nf + 1
    c = -x(1) ^ 2 - x(2) ^ 2 + 1
2:  If c = 0 Then x(-1) = 0 Else _
        x(-1) = x(3) * c ^ 2           'penalty, q = 2
    x(0) = 100 * (x(2) - x(1) ^ 2) ^ 2 + _
        (1 - x(1)) ^ 2 + x(-1)
End Sub
```

The code execution results are depicted in Fig. 6.24.

As we see, initial approximation $\mathbf{x}^0$ of the minimum point has coordinates x_1^0 = -5.5 and x_2^0 = 0.5, the initial steps are directed along the x_1 and x_2 axes and equal 0.01. The coordinates of the obtained minimum point, $\mathbf{x}^*$, are equal to x_1^* = 0.786 and x_2^* = 0.618. When replacing the `minim` subroutine with `mini`, the number of iterations increases from 18 to 67, the number of the calculated values of the penalty function increases from 347 to 686.



	A	B	C	D	E	F	G	H	I	J
1	8.68E-02	0.182241	0.691422	0.476711	1	141	-5.5	0.5		
2	5.05E-02	0.153125	0.721507	0.504733	1	172	0.01			
3	3.21E-03	0.042605	0.802676	0.642142	1	189		0.01		
4	1.93E-03	0.042407	0.800471	0.639920	1	199				
5	4.52E-03	0.042410	0.802728	0.643698	1	209				
6	3.98E-05	0.044123	0.793500	0.630256	2	226				
7	1.63E-03	0.043938	0.794429	0.630394	2	240				
8	2.36E-03	0.043961	0.795146	0.631789	2	247				
9	2.29E-05	0.044782	0.790448	0.624198	4	258				
10	9.67E-04	0.044785	0.790721	0.624298	4	266				
11	3.48E-06	0.045222	0.788451	0.620880	8	282				
12	6.14E-04	0.045240	0.788814	0.621069	8	290				
13	2.77E-05	0.045450	0.787387	0.619141	16	304				
14	4.58E-04	0.045479	0.787829	0.619402	16	312				
15	7.60E-05	0.045570	0.786863	0.618255	32	323				
16	3.97E-04	0.045604	0.787318	0.618537	32	331				
17	4.74E-04	0.045696	0.786447	0.617550	64	340				
18	1.89E-04	0.045678	0.786371	0.618396	64	347				
19										

Fig. 6.24. The Sheet2 worksheet after the code execution

Besides the above methods, ***the barrier function method*** is used for the $F(\mathbf{x})$ function minimization with nonlinear constraints. It differs from the penalty function method in the form of the second summand in formula (6.20):

$$G_k(\mathbf{x}) = F(\mathbf{x}) + 2^{-k}\,[1/C_1^p(\mathbf{x}) + 1/C_2^p(\mathbf{x}) + \ldots + 1/C_m^p(\mathbf{x})]\,, \qquad (6.24)$$

where p is an even natural number. The $G(\mathbf{x},\ x_{n+1})$ function has the following form:

$$G(\mathbf{x},\ x_{n+1}) = F(\mathbf{x}) + x_{n+1}^{-1}[1/C_1^p(\mathbf{x}) + 1/C_2^p(\mathbf{x}) + \ldots + 1/C_m^p(\mathbf{x})]\,. \quad (6.25)$$

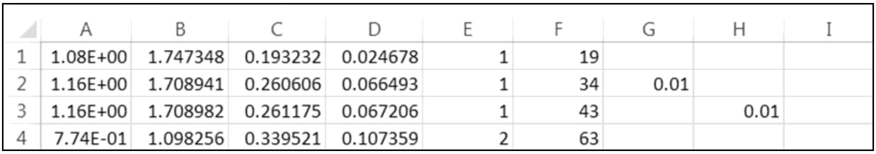
Summand $x_{n+1}^{-1}[1/C_1^p(\mathbf{x}) + 1/C_2^p(\mathbf{x}) + \ldots + 1/C_m^p(\mathbf{x})]$ is called the barrier; functions (6.24) and (6.25) are called the barrier functions.

As we see, function (6.25) is close to (6.21) in form. Therefore, the new code for minimizing function (6.1) with constraint (6.22), (6.23) by the Powell method differs from Listing 6.8 only in the following operator:

```
2:  x(-1) = 1 / x(3) * (1 / c ^ 2)     'barrier, p = 2
```

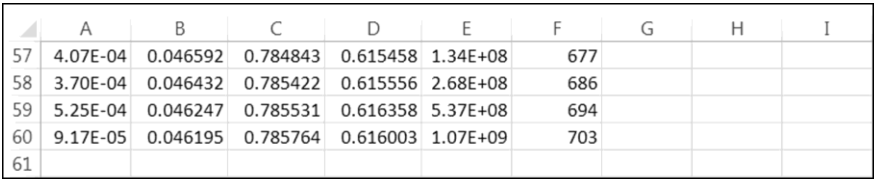
For constraints in the form of inequalities (6.14), initial approximation $\mathbf{x}^0$ of the minimum point must satisfy these inequalities. For example, $\mathbf{x}^0 = (0,\ 0)$ may be when minimizing function (6.1) with constraint (6.22), (6.23) by the new version of code Listing 6.8. Therefore, the method being considered is often called ***the interior point method***.

Fig. 6.25 shows the results of using the new version of code Listing 6.8 (with `minim` in operator `1`).



	A	B	C	D	E	F	G	H	I
1	1.08E+00	1.747348	0.193232	0.024678	1	19			
2	1.16E+00	1.708941	0.260606	0.066493	1	34	0.01		
3	1.16E+00	1.708982	0.261175	0.067206	1	43		0.01	
4	7.74E-01	1.098256	0.339521	0.107359	2	63			

a



	A	B	C	D	E	F	G	H	I
57	4.07E-04	0.046592	0.784843	0.615458	1.34E+08	677			
58	3.70E-04	0.046432	0.785422	0.615556	2.68E+08	686			
59	5.25E-04	0.046247	0.785531	0.616358	5.37E+08	694			
60	9.17E-05	0.046195	0.785764	0.616003	1.07E+09	703			
61									

b

Fig. 6.25. The initial (a) and final (b) iterations

As we see in Fig. 6.25, initial approximation $\mathbf{x}^0$ of the minimum point has zero coordinates, the initial steps are directed along the x_1 and x_2 axes and equal 0.01. The coordinates of the obtained minimum point, $\mathbf{x}^*$, are equal to x_1^* = 0.786 and x_2^* = 0.616. When replacing `minim` with `mini` in operator `1`, the number of iterations increases from 60 to 314, and the number of the calculated values of the barrier function increases from 703 to 2552.

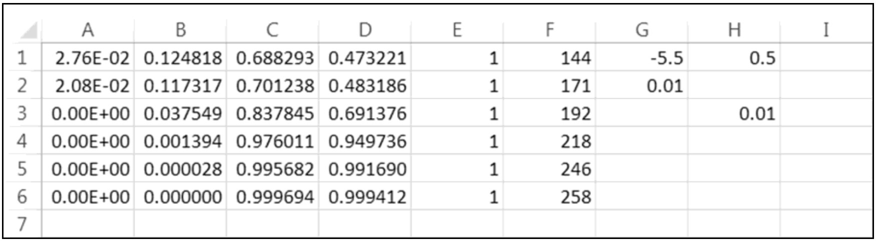
Let us solve one more minimization problem for Rosenbrock function (6.1), when constraint $x_1^2 + x_2^2 \geq 1$ is imposed on the minimum point. This inequality can be written in form (6.22), where $C_1(x_1,\ x_2) = x_1^2 + x_2^2 - 1$.

For minimizing function (6.1) with this constraint by the penalty function and Powell methods, we use Listing 6.8 with `minim` in operator `1` and the following `func` subroutine:

Listing 6.10

```
Sub func(ByVal n, ByRef x() As Double)
    Dim c As Double
    nf = nf + 1
    c = x(1) ^ 2 + x(2) ^ 2 - 1
2:  If c >= 0 Then
        x(-1) = 0
    Else
        x(-1) = x(3) * (-c ^ 3)         'penalty, p = 3
    End If
    x(0) = 100 * (x(2) - x(1) ^ 2) ^ 2 + _
        (1 - x(1)) ^ 2 + x(-1)
End Sub
```

Fig. 6.26 shows the results of using the last version of code Listing 6.8.



	A	B	C	D	E	F	G	H	I
1	2.76E-02	0.124818	0.688293	0.473221	1	144	-5.5	0.5	
2	2.08E-02	0.117317	0.701238	0.483186	1	171	0.01		
3	0.00E+00	0.037549	0.837845	0.691376	1	192		0.01	
4	0.00E+00	0.001394	0.976011	0.949736	1	218			
5	0.00E+00	0.000028	0.995682	0.991690	1	246			
6	0.00E+00	0.000000	0.999694	0.999412	1	258			
7									

Fig. 6.26. The Sheet2 worksheet upon termination of the code execution

As we see in Fig. 6.26, initial approximation $\mathbf{x}^0$ of the minimum point has coordinates x_1^0 = -5.5 and x_2^0 = 0.5, the initial steps are directed along the x_1 and x_2 axes and equal 0.01. The coordinates of the obtained minimum point, $\mathbf{x}^*$, are equal to x_1^* = 1 and x_2^* = 1.

The barrier function method gives an incorrect result.

6.8. Minimization of the multimodal function

The search methods (the coordinate-descent and Powell methods) considered above were developed for finding the minimum point of the so-called unimodal function, $x_0 = F(x_1, x_2, ..., x_n)$, that has a single local minimum. Application of these methods to the multimodal function, with several local minima, gives only one minimum point, which depends on the initial approximation of the minimum point and on the directions and values of initial steps.

Let function $F(x_1, x_2, ..., x_n)$ be multimodal inside its domain, and all the minima are to be found. The solution of this problem may be required for subsequent definition of the global minimum or for solving the maximin problem, i.e., for definition of the function's maximum value among the local minima.

All local minima can be found if the initial approximation of the minimum point is defined by means of the random-number generator and, at that, the code execution is sufficiently long.

For example, let us consider function

$$F(x_1,\ x_2) = (\cos^2 x_1 + \cos^2 x_2)[100(x_2 - x_1^2)^2 + (1 - x_1)^2] \qquad (6.26)$$

in the rectangle given by inequalities

$$0.5 \le x_1 \le 5.5\,,$$

$$0.5 \le x_2 \le 5\,.$$

To find the local minima of function (6.26) inside its domain (the above rectangle), we use the following text of program `main`, subroutine `func` and function `test`:

Listing 6.11

```
Dim nf As Long                'counter of calls of func
Dim nt As Long                'counter of calls of test
Dim np As Long                'counter of minimum points

Sub main()
    Dim x() As Double
    Dim ss() As Double
    Dim n As Byte
    Dim i As Byte, j As Byte
    Dim d As Double
    Dim sec As Long
    Dim min As Byte, hour As Byte
    Dim st As String
    Dim stp As Date
    Dim np_lim As Long
    n = 2                         'number of variables
    ReDim x(n)
    ReDim ss(1 To n, 1 To n)
1:  sec = Range("Sheet2!G4").Value
2:  hour = sec \ 3600
3:  min = (sec - hour * 3600) \ 60
4:  sec = sec - hour * 3600 - min * 60
5:  st = CStr(hour) & ":" & CStr(min) & ":" _
        & CStr(sec)
6:  stp = Now + TimeValue(st)
7:  np_lim = Range("Sheet2!H4").Value
8:  Randomize               'it must be before calling Rnd
9:  np = 0
beg:
10: np = np + 1
11: x(1) = 0.5 + (5.5 - 0.5) * Rnd
12: x(2) = 0.5 + (5 - 0.5) * Rnd
    For j = 1 To n
        For i = 1 To n
            ss(i, j) = Worksheets("Sheet2"). _
                Cells(1 + i, 6 + j)
        Next i
    Next j
    For j = 1 To n
        d = 0
        For i = 1 To n
            d = ss(i, j) ^ 2 + d
        Next i
        If d = 0 Then
            Range("Sheet2!A1").Value = _
                "You must increase" & Str(j) & _
                "-th initial step"
            End
        End If
    Next j
    nf = 0
    nt = 0
    Call func(n, x)     'it must be before minimization
    Call minim(n, x, ss, 1E-6, 1E-6)    'Powell method
13: For j = 0 To n
14:     Worksheets("Sheet2").Cells(np, j + 1) = x(j)
15: Next j
16: Worksheets("Sheet2").Cells(np, n + 2) = nt
17: Worksheets("Sheet2").Cells(np, n + 3) = nf
18: If Now < stp And np < np_lim And np < 1048576 _
        Then GoTo beg
End Sub

Sub func(ByVal n, ByRef x() As Double)
    nf = nf + 1
    x(0) = (Cos(x(1)) ^ 2 + Cos(x(2)) ^ 2) * _
        (100 * (x(2) - x(1) ^ 2) ^ 2 + (1 - x(1)) ^ 2)
End Sub

Function test(ByVal n, ByRef x() As Double, _
    ByRef z() As Double, ByVal rho, Optional alpha) _
    As Boolean
    Dim j As Byte
    nt = nt + 1
    If Abs(z(0) - x(0)) < n * rho * z(0) Then
        test = False
    Else
        test = True
    End If
    If Not IsMissing(alpha) Then
        If x(0) < alpha Then test = False
    End If
End Function
```

From code Listing 6.4, intended for minimizing the Rosenbrock function, the last code differs in operators of the `func` subroutine and presence of operators `1` — `18` in the `main` program. Let us consider the purpose of these additional operators.

Operator `1` assigns the specified limiting execution time (in seconds), being in cell `Sheet2!G4`, to variable `sec` of the `Long` data type. Operators `2` — `5` form the `st` string of format `"hh:mm:ss"`, in which `hh` is one or two digits defining the number of hours, mm is the number of minutes, ss is the number of seconds. Operator 6 determines the moment of the execution termination and assigns it to variable stp (from word "stop") of the Date data type.

Operator 7 assigns the limiting number of found minimum points to variable np_lim of the Long data type. This number is taken from cell Sheet2!H4.

The Randomize operator (labeled by 8) prepares the built-in random-number generator for work; operator 9 nullifies the np counter of the found minimum points.

Operator 10, below label beg (from "beginning"), increases the np counter by 1. Operators 11 and 12 define the coordinates of initial approximation $\mathbf{x}^0$ of the next minimum point by means of the Rnd function returning real numbers uniformly distributed on segment $[0, 1]$.

The $\mathbf{x}^0$ point falls into any place of rectangle $0.5 \le x_1 \le 5.5$ and $0.5 \le x_2 \le 5$ with equal probability.

Below operator 12, we see familiar operators of the function minimization by the Powell method. Upon termination of the minim subroutine execution, operators 13 — 18 are performed.

Operators 13 — 17 put the result of searching the local minimum into the empty row on the Sheet2 worksheet. The result includes:

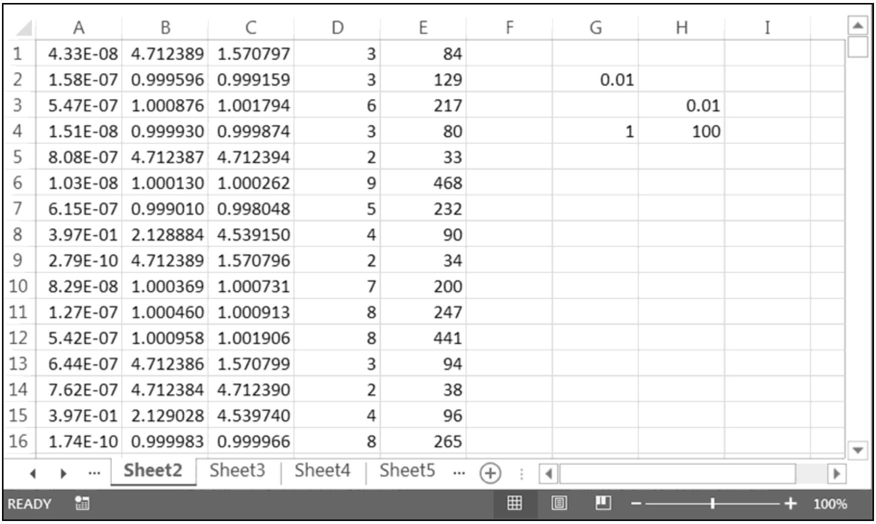
- x(0) — the value of function F;
- x(1), x(2) — the values of arguments x_1 and x_2;
- nt, nf — the numbers of the test and func calls required for finding the local minimum.

Operator 18 performs the jump to label beg if the following three conditions are satisfied simultaneously:

- the execution time is less than the limiting time;
- the number of found minimum points is less than the limiting number;
- empty rows still stay on the Excel worksheet.

After the jump to label beg, the new $\mathbf{x}^0$ point is defined randomly and the minimization is repeated.

Fig. 6.27 shows the beginning of the Sheet2 worksheet after the code execution. The limiting execution time and the limiting number of found minimum points are equal to 1 second and 100, respectively. During the code execution, 100 minimum points were calculated; coordinates of seven of them are given in table "The minimization results" below. The 93 remaining points practically coincide with these seven points or are outside the domain defined by inequalities $0.5 \le x_1 \le 5.5$ and $0.5 \le x_2 \le 5$.



	A	B	C	D	E	F	G	H	I
1	4.33E-08	4.712389	1.570797	3	84				
2	1.58E-07	0.999596	0.999159	3	129		0.01		
3	5.47E-07	1.000876	1.001794	6	217			0.01	
4	1.51E-08	0.999930	0.999874	3	80		1	100	
5	8.08E-07	4.712387	4.712394	2	33				
6	1.03E-08	1.000130	1.000262	9	468				
7	6.15E-07	0.999010	0.998048	5	232				
8	3.97E-01	2.128884	4.539150	4	90				
9	2.79E-10	4.712389	1.570796	2	34				
10	8.29E-08	1.000369	1.000731	7	200				
11	1.27E-07	1.000460	1.000913	8	247				
12	5.42E-07	1.000958	1.001906	8	441				
13	6.44E-07	4.712386	1.570799	3	94				
14	7.62E-07	4.712384	4.712390	2	38				
15	3.97E-01	2.129028	4.539740	4	96				
16	1.74E-10	0.999983	0.999966	8	265				

Fig. 6.27. The first 16 rows on the Sheet2 worksheet
after the code execution

The minimization results

Minimum's number	$x_0 = F(x_1,\ x_2)$	x_1	x_2
1	0	1.57	1.57
2	0	4.71	1.57
3	0	4.71	4.71
4	0	1.57	4.71
5	0	1	1
6	0.397	2.13	4.54
7	0.732	1.88	3.55

When creating table "The minimization results", we applied the Excel filter to the worksheet of Fig. 6.27 as follows:

1) set the number format with two decimal places for columns B and C;
2) select columns A:E;
3) *Data* > *Filter* in area *Sort & Filter* (Fig. 6.28);

4) in the first row of the B column, fulfill *Number Filters* > *Custom Filter* by means of the drop-down list;

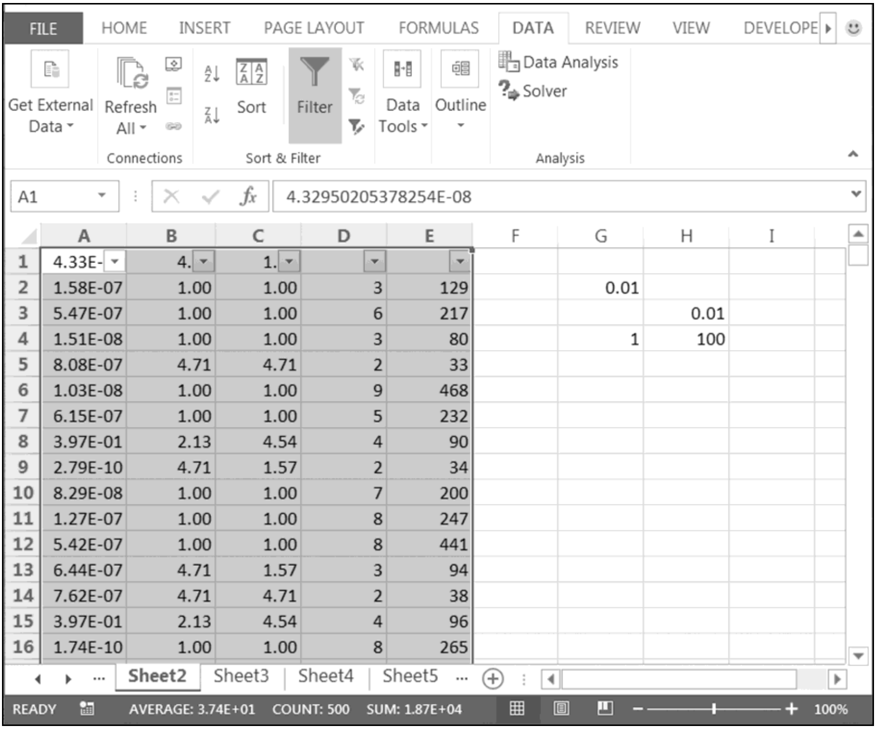
5) in the *Custom AutoFilter* window opened, set the following: ≥ 0.5 and ≤ 5.5 (Fig. 6.29);

6) click on the *OK* button;

7) in the first row of the C column, fulfill *Number Filters* > *Custom Filter*;

8) in the *Custom AutoFilter* window opened, set the following: ≥ 0.5 and ≤ 5 (Fig. 6.30);

9) click on the *OK* button.



A1 | 4.32950205378254E-08

	A	B	C	D	E	F	G	H	I
1	4.33E-	4.	1.						
2	1.58E-07	1.00	1.00	3	129		0.01		
3	5.47E-07	1.00	1.00	6	217			0.01	
4	1.51E-08	1.00	1.00	3	80		1	100	
5	8.08E-07	4.71	4.71	2	33				
6	1.03E-08	1.00	1.00	9	468				
7	6.15E-07	1.00	1.00	5	232				
8	3.97E-01	2.13	4.54	4	90				
9	2.79E-10	4.71	1.57	2	34				
10	8.29E-08	1.00	1.00	7	200				
11	1.27E-07	1.00	1.00	8	247				
12	5.42E-07	1.00	1.00	8	441				
13	6.44E-07	4.71	1.57	3	94				
14	7.62E-07	4.71	4.71	2	38				
15	3.97E-01	2.13	4.54	4	96				
16	1.74E-10	1.00	1.00	8	265				

READY AVERAGE: 3.74E+01 COUNT: 500 SUM: 1.87E+04

Fig. 6.28. The first 16 rows on the Sheet2 worksheet after starting the filter

The first four minima in table “The minimization results” are located at the points, where the first multiplicand of objective function (6.26) is equal to zero:

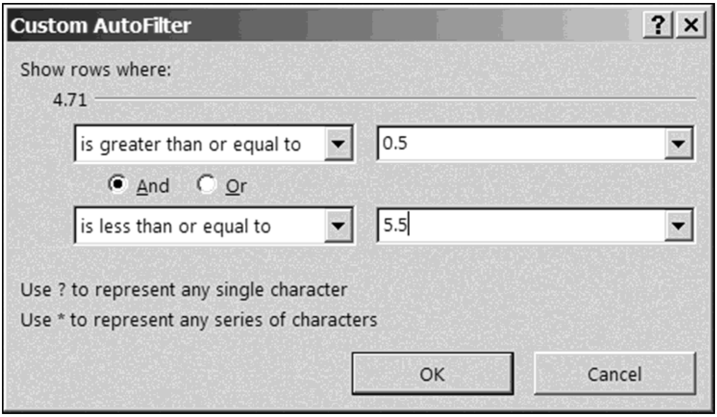
$$\cos^2 x_1 + \cos^2 x_2 = 0$$

because $\pi/2 = 1.57$, $3\pi/2 = 4.71$, i.e., $\cos(1.57) = \cos(4.71) = 0$.

The 5th minimum has coordinates $x_1 = x_2 = 1$, for which the second multiplicand of objective function (6.26) is equal to zero:

$$100(x_2 - x_1^2)^2 + (1 - x_1)^2 = 0 .$$

The 6th and 7th minima are not so obvious: objective function (6.26) has nonzero values at points (2.13; 4.54) and (1.88; 3.55).



Custom AutoFilter

Show rows where:

4.71

is greater than or equal to | 0.5

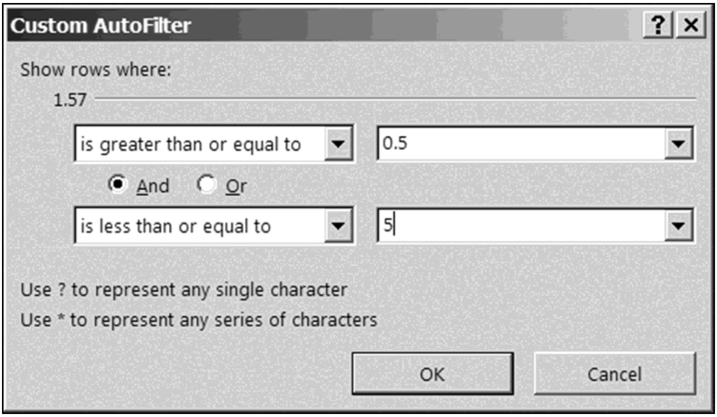
● And ○ Or

is less than or equal to | 5.5

Use ? to represent any single character

Use * to represent any series of characters

Fig. 6.29. Setting the constraints for x_1 during the filter usage



Custom AutoFilter

Show rows where:

1.57

is greater than or equal to | 0.5

● And ○ Or

is less than or equal to | 5

Use ? to represent any single character

Use * to represent any series of characters

Fig. 6.30. Setting the constraints for x_2 during the filter usage

Strictly speaking, the above method for minimizing the multimodal function does not guarantee finding all local minima inside the function's domain. However, the probability of this is close to unity if the code execution is sufficiently long.

Often, we do not believe in the adequacy of the obtained solution of a minimization problem, even when we know that the objective function is unimodal. In this case, the use of this section's method is a good idea.

We advise the reader to write a program for finding the global minimum of function

$$F(x_1,\ x_2) = f_1^2(x_1) + f_2^2(x_2)$$

inside rectangle $a_1 \le x_1 \le b_1$ and $a_2 \le x_2 \le b_2$, where $f_1(x_1)$ and $f_2(x_2)$ are functions from Appendix 4, segments $[a_1,\ b_1]$ and $[a_2,\ b_2]$ are the domains of functions $f_1(x_1)$ and $f_2(x_2)$, respectively.

In addition, ***we advise the reader*** to use the random-number generator for defining both the initial approximation of the minimum point and the initial steps along the x_1 and x_2 axes.

By the way, the last minimum in table "The minimization results" (with coordinates x_1 = 1.88 and x_2 = 3.55) is the solution of the following maximin problem: to determine the maximum value of function (6.26) among the local minima inside rectangle $0.5 \le x_1 \le 5.5$ and $0.5 \le x_2 \le 5$. The global minimum is equal to zero; it is reached at the first five minimum points.

6.9. Minimization of the tabular function

To demonstrate the possibilities of the numerical methods for nonlinear programming, we used simple functions that can be differentiated analytically. For example, we easily derived expressions (6.5) and (6.6) for the partial derivatives of the Rosenbrock function. In this case, methods of the third chapter in book [16], using the first and second partial derivatives of the objective function, are effective for the minimization.

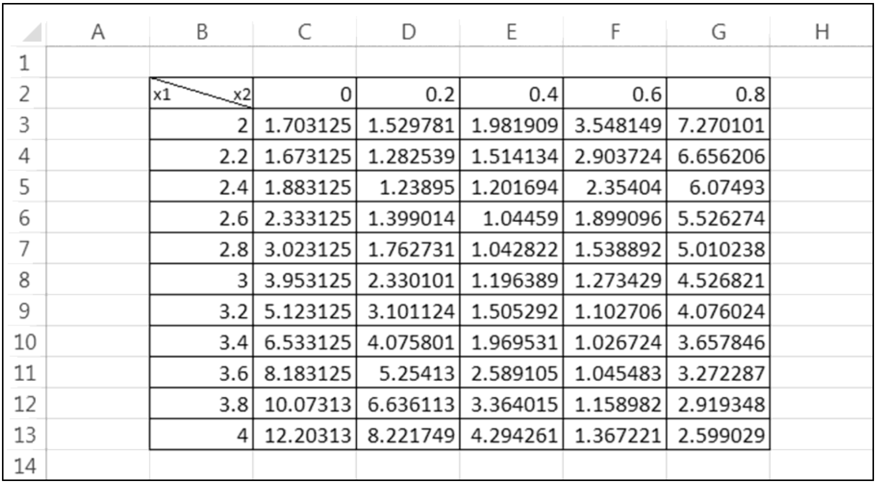
In practice, it is often necessary to minimize a function in tabular form or a function whose values are calculated implicitly, for example, by solving any equation. In these cases, the analytical differentiation of the function being minimized is impossible, and the search methods take on special significance.

We will consider the minimization of a positive tabular function of two variables. The values of arguments x_1 and x_2 and of function $x_0 = F(x_1, x_2)$ are given in table Listing 6.12 and Fig. 6.31.

Listing 6.12

The positive tabular function of two variables

x_1 \ x_2	0	0.2	0.4	0.6	0.8
2	1.703125	1.529781	1.981909	3.548149	7.270101
2.2	1.673125	1.282539	1.514134	2.903724	6.656206
2.4	1.883125	1.23895	1.201694	2.35404	6.07493
2.6	2.333125	1.399014	1.04459	1.899096	5.526274
2.8	3.023125	1.762731	1.042822	1.538892	5.010238
3	3.953125	2.330101	1.196389	1.273429	4.526821
3.2	5.123125	3.101124	1.505292	1.102706	4.076024
3.4	6.533125	4.075801	1.969531	1.026724	3.657846
3.6	8.183125	5.25413	2.589105	1.045483	3.272287
3.8	10.07313	6.636113	3.364015	1.158982	2.919348
4	12.20313	8.221749	4.294261	1.367221	2.599029



	A	B	C	D	E	F	G	H
1								
2		x1 x2	0	0.2	0.4	0.6	0.8	
3		2	1.703125	1.529781	1.981909	3.548149	7.270101	
4		2.2	1.673125	1.282539	1.514134	2.903724	6.656206	
5		2.4	1.883125	1.23895	1.201694	2.35404	6.07493	
6		2.6	2.333125	1.399014	1.04459	1.899096	5.526274	
7		2.8	3.023125	1.762731	1.042822	1.538892	5.010238	
8		3	3.953125	2.330101	1.196389	1.273429	4.526821	
9		3.2	5.123125	3.101124	1.505292	1.102706	4.076024	
10		3.4	6.533125	4.075801	1.969531	1.026724	3.657846	
11		3.6	8.183125	5.25413	2.589105	1.045483	3.272287	
12		3.8	10.07313	6.636113	3.364015	1.158982	2.919348	
13		4	12.20313	8.221749	4.294261	1.367221	2.599029	
14								

Fig. 6.31. The Excel table with source data

According to Fig. 6.31:

- cells B3:B13 contain values $x_{1,i}$ of argument x_1 ($0 \le i \le k$, $k = 10$), i.e., grid nodes $x_{1,0}$, $x_{1,1}$, ..., $x_{1,k}$ on axis x_1;
- cells C2:G2 contain values $x_{2,j}$ of argument x_2 ($0 \le j \le r$, $r = 4$), i.e., grid nodes $x_{2,0}$, $x_{2,1}$, ..., $x_{2,r}$ on axis x_2;
- cells C3:G13 contain the $F(x_1, x_2)$ function values.

The code intended for minimizing the $F(x_1, x_2)$ function follows:

Listing 6.13

```
Const DBL_MAX = 1E+308
Dim x1() As Double            'grid nodes on axis x1
Dim x2() As Double            'grid nodes on axis x2
Dim k As Integer              'number of segments on x1
Dim r As Integer              'number of segments on x2
Dim ff() As Double            'values of function F
Dim f1() As Double            'values of splines
Dim f2() As Double            'values of function of x2
Dim mm() As Double            'values of moments about x2
Dim m1() As Double            'values of moments about x1
Dim m2() As Double          'values of moments about x2
Dim nf As Long              'counter of calls of func
Dim nt As Long              'counter of calls of test
Dim no As Integer           'shift for output

Sub main()
    Dim x(2) As Double
    Dim ss(1 To 2, 1 To 2) As Double
    Dim i As Integer, j As Integer
    Dim min As Double
    k = Selection.Rows.Count - 2
    r = Selection.Columns.Count - 2
    ReDim x1(k)
    ReDim x2(r)
    ReDim ff(k, r)
    ReDim f1(k)
    ReDim f2(r)
    ReDim mm(k, r)
    ReDim m1(k)
    ReDim m2(r)
    For i = 0 To k
        x1(i) = Selection.Cells(2 + i, 1)
    Next i
    For j = 0 To r
        x2(j) = Selection.Cells(1, 2 + j)
    Next j
    For i = 0 To k
        For j = 0 To r
            ff(i, j) = Selection.Cells(2 + i, 2 + j)
        Next j
    Next i
'Calculation of 2D array of moments about x2:
    For i = 0 To k
        For j = 0 To r
            f2(j) = ff(i, j)
        Next j
0:      Call mos(0, r, x2, f2, 0, 0, 0, 0, m2)
        For j = 0 To r
            mm(i, j) = m2(j)
        Next j
    Next i
'Specifying initial approximation of minimum point:
    min = DBL_MAX
    For i = 0 To k
        For j = 0 To r
            If ff(i, j) < min Then
                min = ff(i, j)
                x(1) = x1(i)
                x(2) = x2(j)
            End If
        Next j
    Next i
    no = i + 2
'Specifying initial steps:
    ss(1, 1) = 0.01: ss(1, 2) = 0
    ss(2, 1) = 0: ss(2, 2) = 0.01
'Searching minimum point:
    nf = 0
    nt = 0
    Call func(2, x)    'it must be before minimization
    Selection.Cells(no, 1) = x(0)
    Selection.Cells(no, 2) = x(1)
    Selection.Cells(no, 3) = x(2)
    Call minim(2, x, ss, 1E-6)          'Powell method
'Outputting minimum point:
    Selection.Cells(no + 1, 1) = x(0)
    Selection.Cells(no + 1, 2) = x(1)
    Selection.Cells(no + 1, 3) = x(2)
    Selection.Cells(no + 1, 4) = nt
    Selection.Cells(no + 1, 5) = nf
End Sub

Sub func(ByVal n, ByRef x() As Double)
    Dim i As Integer
    Dim j As Integer
    nf = nf + 1
    For i = 0 To k
        For j = 0 To r
            f2(j) = ff(i, j)
            m2(j) = mm(i, j)
        Next j
1:      Call si(0, r, x2, f2, m2, x(2), f1(i))
    Next i
2:  Call mos(0, k, x1, f1, 0, 0, 0, 0, m1)
```

```
3:  Call si(0, k, x1, f1, m1, x(1), x(0))
End Sub

Function test(ByVal n, ByRef x() As Double, _
    ByRef z() As Double, ByVal rho, Optional alpha) _
    As Boolean
    Dim j As Byte
    nt = nt + 1
    If Abs(z(0) - x(0)) < n * rho * z(0) Then
        test = False
    Else
        test = True
    End If
    If Not IsMissing(alpha) Then
        If x(0) < alpha Then test = False
    End If
End Function
```

We enter this code into Module1 of the BookNM workbook. The source data are in the Excel table (Fig. 6.31); this table (range B2:G13) should be selected before the code execution.

Let us consider the three main stages of the code execution.

First of all, the moments of cubic splines $S_0(x_2)$, $S_1(x_2)$, …, $S_k(x_2)$, defined by grid functions $f_0(x_2) = F(x_{1,0},\ x_2)$, $f_1(x_2) = F(x_{1,1},\ x_2)$, …, $f_k(x_2) = F(x_{1,\,k},\ x_2)$, are calculated by means of operator 0:

- $f_0(x_2)$ and $S_0(x_2)$ correspond to range C3:G3;
- $f_1(x_2)$ and $S_1(x_2)$ correspond to range C4:G4;
- $f_k(x_2)$ and $S_k(x_2)$ correspond to range C13:G13.

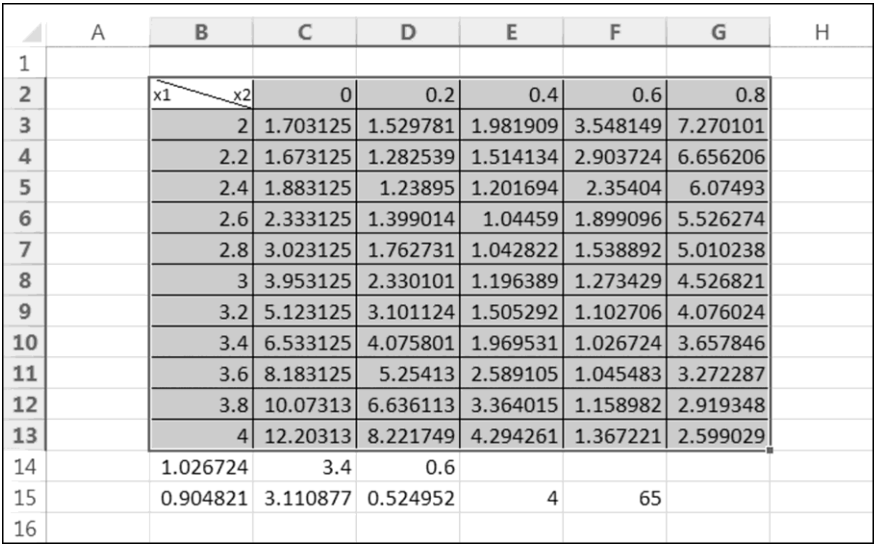
These moments are stored in two-dimensional array mm. Later this array is used (by means of one-dimensional array m2) in the func subroutine for the spline interpolation (operator 1).

Further, coordinates x(1) and x(2) of initial approximation $\mathbf{x}^0$ of the $\mathbf{x}^*$ minimum point are specified as follows:

- the minimum of range C3:G13 (Fig. 6.31) is found: min = 1.026724 (cell F10);
- the point, where function F equals min, is taken as the initial approximation: x(1) = 3.4, x(2) = 0.6.

In last turn, two-dimensional array `ss` of initial steps is specified and the search for the minimum point is performed by the Powell method. The following data are put into the Excel worksheet:

- initial array `x` into cells B14:D14 (Fig. 6.32);
- final array `x` and the final values of `nt` and `nf` into cells B15:F15.



	A	B	C	D	E	F	G	H
1								
2		x1 x2	0	0.2	0.4	0.6	0.8	
3		2	1.703125	1.529781	1.981909	3.548149	7.270101	
4		2.2	1.673125	1.282539	1.514134	2.903724	6.656206	
5		2.4	1.883125	1.23895	1.201694	2.35404	6.07493	
6		2.6	2.333125	1.399014	1.04459	1.899096	5.526274	
7		2.8	3.023125	1.762731	1.042822	1.538892	5.010238	
8		3	3.953125	2.330101	1.196389	1.273429	4.526821	
9		3.2	5.123125	3.101124	1.505292	1.102706	4.076024	
10		3.4	6.533125	4.075801	1.969531	1.026724	3.657846	
11		3.6	8.183125	5.25413	2.589105	1.045483	3.272287	
12		3.8	10.07313	6.636113	3.364015	1.158982	2.919348	
13		4	12.20313	8.221749	4.294261	1.367221	2.599029	
14		1.026724	3.4	0.6				
15		0.904821	3.110877	0.524952	4	65		
16								

Fig. 6.32. The code execution results

Let us use the following notations: χ_1 and χ_2 are the current values of variables `x(1)` and `x(2)`, respectively.

The value of objective function $\chi_0 = F(\chi_1, \chi_2)$ is the result of the `func` subroutine execution. This value is calculated as follows:

1) by means of the `si` subroutine (operator `1`), grid function $f(x_1)$ is determined according to the following formulas:

$$f(x_{1,0}) = S_0(\chi_2), \quad f(x_{1,1}) = S_1(\chi_2), \quad \ldots, \quad f(x_{1,k}) = S_k(\chi_2);$$

2) by means of the `mos` subroutine (operator `2`), the moments of spline $S(x_1)$, corresponding to $f(x_1)$, are calculated;

3) by means of the `si` subroutine (operator `3`), the value of $S(\chi_1)$ is calculated, which is considered as the objective function's value:

$$\mathtt{x(0)} = \chi_0 = F(\chi_1, \chi_2) = S(\chi_1).$$

According to Fig. 6.32, the found minimum point of the $F(x_1, x_2)$ function, given by table Listing 6.12, has coordinates x_1^* = 3.111 and x_2^* = 0.525, and the minimum value of $F(x_1, x_2)$ is equal to $F(x_1^*, x_2^*)$ = 0.905.

We advise the reader to develop a noniterative method and corresponding program for minimizing a positive tabular function of two variables (similar to the method and program of Section 4.6). Listing 6.12 must be used for testing the program.

Besides splines $S_0(x_2)$, $S_1(x_2)$, ..., $S_k(x_2)$ considered above, cubic splines $S^0(x_1)$, $S^1(x_1)$, ..., $S^r(x_1)$ must be used. The last set of splines is defined by grid functions $f^0(x_1) = F(x_1, x_{2,0})$, $f^1(x_1) = F(x_1, x_{2,1})$, ..., $f^r(x_1) = F(x_1, x_{2,r})$:

- $f^0(x_1)$ and $S^0(x_1)$ correspond to range C3:C13 in Fig. 6.31;
- $f^1(x_1)$ and $S^1(x_1)$ correspond to range D3:D13;
- $f^r(x_1)$ and $S^r(x_1)$ correspond to range G3:G13.

In the next two chapters, we will minimize a function whose values are calculated implicitly, more precisely, by solving the initial value problems for the system of differential equations.

6.10. Solving the nonlinear differential equation by the shooting method

Minimization of an implicit function may be required to solve the following boundary value problem on segment $[a, b]$:

$$\frac{d^2u}{dx^2} = F\left(x, u, \frac{du}{dx}\right), \tag{6.27}$$

$$u(a) = A, \tag{6.28}$$

$$u(b) = B, \tag{6.29}$$

where *A* and *B* are given parameters, *F* is a nonlinear function of variables *x*, *y* and *z*.

Problem (6.27) — (6.29) was solved by the quasilinearization method in Section 3.12. Below, this problem will be solved by the shooting method.

We introduce unknown function

$$v(x) = \frac{du}{dx}.$$

This expression can be written in the following equation form:

$$\frac{du}{dx} = E(x, u, v), \tag{6.30}$$

where *E* is a function of simple form: $E = v$.

Equation (6.27) becomes

$$\frac{dv}{dx} = F(x, u, v). \tag{6.31}$$

In Section 5.2, the method for solving the system of equations (6.30) and (6.31) was developed for initial conditions (6.28) and

$$v(a) = Q. \tag{6.32}$$

According to the shooting method, problem (6.27) — (6.29) can be solved by the repeated solution of system (6.30), (6.31) with initial conditions (6.28) and (6.32) for different values of *Q* until satisfaction of condition (6.29) at point *b*.

Let us consider the shooting model, which is a good illustration of the shooting method (this explains the method name).

If we neglect the air resistance, then Newton's second law gives the projectile trajectory, $u(x)$, described by equation (6.27), where

$$F = -g(1+Q^2)/V^2 .$$

In this expression:

- $Q = \dfrac{du}{dx}(a)$ is the slope of the gun barrel located at point $(a,\ A)$ with coordinates $x = a$ and $u = A$;
- V is the projectile velocity at the moment of leaving the barrel;
- g is the free fall acceleration.

Let us assume that the target is at point $(b,\ B)$ with coordinates $x = b$ and $u = B$. Solving the system of equations (6.30) and (6.31) with initial conditions (6.28) and (6.32) at various values of Q, we simulate the shooting when varying the slope of the gun barrel (Fig. 6.33).

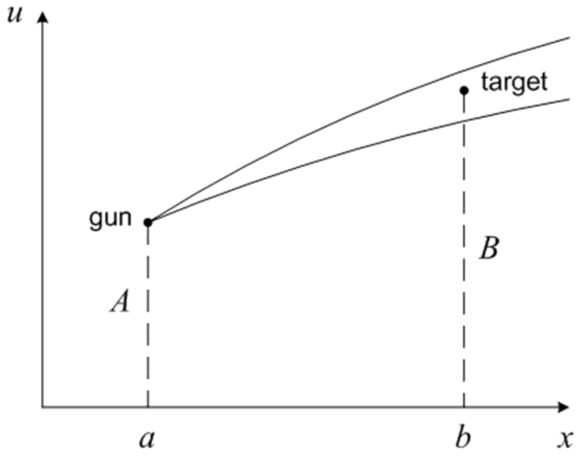


Fig. 6.33. Graphic image of the $u(x)$ solution of system (6.30), (6.31) with initial conditions (6.28) and (6.32) at two values of Q

To obtain slope Q^*, at which the projectile hits the target at point (b, B), we have to solve algebraic equation

$$u(Q,\ b) = B, \tag{6.33}$$

where $u(Q,\ x)$ is the result of solving the system of differential equations (6.30) and (6.31) with initial conditions (6.28) and (6.32) at given Q.

The above algebraic equation can be solved by the bisection, secant or Steffensen method (Sections 4.5 and 5.5). We will solve it by minimizing the following function:

$$G(Q) = [u(Q,\ b) - B]^2\,. \tag{6.34}$$

The method for solving algebraic equation (6.33) by minimizing function (6.34) is based on the following obvious assertion: the problem of solving the system of nonlinear algebraic equations (5.17) is equivalent to the problem of finding point $\mathbf{x}^*$, at which non-negative function

$$F(\mathbf{x}) = [\,f_1(\mathbf{x}) - \alpha_1]^2 + [\,f_2(\mathbf{x}) - \alpha_2]^2 + ... + [\,f_n(\mathbf{x}) - \alpha_n]^2 \tag{6.35}$$

is equal to 0. Here, $\mathbf{x} = (x_1,\ x_2,\ ...,\ x_n)$ is a point of the n-dimensional space, $n \geq 1$.

Minimizing the $F(x_1,\ x_2,\ ...,\ x_n)$ function, defined by (6.35), is a popular method for solving the system of nonlinear algebraic equations (5.17).

6.11. Modeling of the hammer motion in the piano mechanism

As an example of using the shooting method, we will solve the following task of modeling of the hammer motion in the piano mechanism when rupture of the spring occurs:

- in addition to the initial time moment, $a = 0$, of rupture of the spring, another moment, $b = 20$ ms, is given;
- we know that the hammer's displacement forward is equal to $A_3 = 7$ mm and $B_3 = 15$ mm at moments a and b, respectively;
- we have to determine values A_4 and B_4 of the hammer velocity at these moments.

For solving this task, we use the simplified mathematical model of the piano mechanism: the elastic constant of the spring, k, equals zero in model (5.27) — (5.30). In this case, the hammer's displacement is described by equation

$$\frac{d^2u_3}{dt^2} = E_4\left(u_3, \frac{du_3}{dt} \right), \tag{6.36}$$

where

$$E_4\left(u_3, \frac{du_3}{dt} \right) = \frac{q\left(\frac{du_3}{dt} \right)^2}{p - qu_3}.$$

Second-order nonlinear differential equation (6.36) is considered on time segment $0 \le t \le b$. We have to solve this equation with boundary conditions

$$u_3(0) = A_3, \tag{6.37}$$

$$u_3(b) = B_3. \tag{6.38}$$

As in Section 5.6, let us introduce the hammer velocity, $u_4 = du_3/dt$. We obtain the following system of first-order differential equations:

$$\frac{du_3}{dt} = E_3(u_4), \tag{6.39}$$

$$\frac{du_4}{dt} = E_4(\ u_3,\ u_4\ ), \tag{6.40}$$

where

$$E_3 = u_4\ ,$$

$$E_4 = \frac{qu_4^2}{p - qu_3}\ .$$

According to the shooting method, boundary value problem (6.36) — (6.38) can be solved by the repeated solution of the system of differential equations (6.39) and (6.40) with initial conditions (6.37) and

$$u_4(0) = A_4\ . \tag{6.41}$$

For obtaining desired value A_4^* of the hammer velocity at the initial time moment, we have to minimize the following function similar to (6.34):

$$G(A_4) = [u_3(A_4,\ b) - B_3]^2\ , \tag{6.42}$$

where function $u_3(A_4,\ t)$ is the result of solving the system of equations (6.39) and (6.40) with initial conditions (6.37) and (6.41) at given A_4 .

Function (6.42) will be minimized by the `mini` subroutine. We could use the `minim` subroutine because `mini` and `minim` work equally in the case of a one-variable function (n = 1).

In the source data table given below, values l, τ, p, q and ζ have the same sense as in table Listing 5.3 (p. 376).

Listing 6.14

l	20
tau	1.00E-03
A4	0
p	0.406
q	18.3
zeta	1.00E-09
A3	0.007
B3	0.015

The sense of the remaining values in table Listing 6.14 is as follows:

- A_4 is the initial approximation of the minimum point of function (6.42) in units of m / s;

- A_3 is the given value of $u_3(0)$ in meters;
- B_3 is the given value of $u_3(b)$ in meters, $b = l\tau$.

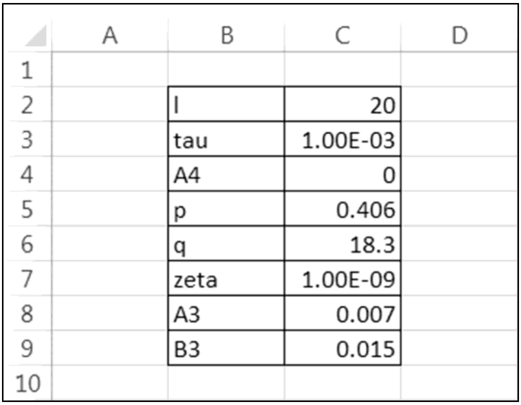
The code for solving the problem of minimizing function (6.42) has the following form:

Listing 6.15

```
Dim x(0 To 1) As Double      'array for minimization
Dim nf As Long               'counter of calls of func
Dim nt As Long               'counter of calls of test
Dim l As Integer, tau As Double
Dim A4 As Double
Dim p As Double, q As Double
Dim zeta As Double
Dim A3 As Double, B3 As Double
Dim tau2 As Double
Dim uu() As Double
Dim jj() As Integer

Sub main()
    Dim ss(1 To 1, 1 To 1) As Double
    Dim i As Integer
    Dim sb As String, se As String
    l = Selection.Cells(1, 2)
    tau = Selection.Cells(2, 2)
    A4 = Selection.Cells(3, 2)
    p = Selection.Cells(4, 2)
    q = Selection.Cells(5, 2)
    zeta = Selection.Cells(6, 2)
    A3 = Selection.Cells(7, 2)
    B3 = Selection.Cells(8, 2)
    tau2 = tau / 2
    ReDim uu(1 To 4, 0 To l)
    ReDim jj(0 To l)
    x(1) = A4                'initial approximation of A4
    ss(1, 1) = 1E-6          'initial step along A4 axis
    nf = 0: nt = 0
    Call func(1, x)    'it must be before minimization
    Call mini(1, x, ss, 1E-12, 1E-12)
    MsgBox "A4 = " & CStr(Round(x(1), 3)) & " m/s"
    MsgBox "B4 = " & CStr(Round(uu(4, l), 3)) & " m/s"
    Selection.Cells(9, 1) = "t"
    Selection.Cells(9, 2) = "u3"
    Selection.Cells(9, 3) = "u4"
    Selection.Cells(9, 4) = "j max"
    For i = 0 To l           'movement along time axis
        Selection.Cells(10 + i, 1) = i * tau
        Selection.Cells(10 + i, 2) = uu(3, i)
        Selection.Cells(10 + i, 3) = uu(4, i)
        Selection.Cells(10 + i, 4) = jj(i)
12:     Selection.Cells(10 + i, 4). _
        HorizontalAlignment = xlCenter      'alignment
    Next i
    sb = Selection.Cells(10, 1).Address
    se = Selection.Cells(10 + l, 2).Address
13: Call graph(sb, se, "t, s", "u3, m")
End Sub

Sub e_functions(ByRef x() As Double, _
    ByRef e() As Double)
    e(3) = x(4)
    e(4) = q * x(4) ^ 2 / (p - q * x(3))
End Sub

Sub fx_jacobian(ByRef x() As Double, _
    ByRef fx() As Double)
    Dim m3 As Double
    fx(3, 3) = 1: fx(3, 4) = -tau2
    m3 = p - q * x(3)
    fx(4, 3) = -tau2 * (q * x(4) / m3) ^ 2
    fx(4, 4) = 1 - tau * q * x(4) / m3
End Sub

Sub func(ByVal n, ByRef x() As Double)
    Dim m As Integer
    Dim i As Integer, j As Integer
    Dim u(3 To 4) As Double
    Dim xx(3 To 4) As Double
    Dim z(3 To 4) As Double
    Dim e(3 To 4) As Double
    Dim a(3 To 4, 3 To 4) As Double
    Dim b(3 To 4) As Double
    Dim alpha(3 To 4) As Double
    Dim max As Double
    nf = nf + 1
1:  u(3) = A3: u(4) = x(1)              'values at t=0
    For m = 3 To 4
        uu(m, 0) = u(m)
    Next m
    jj(0) = 0
    For i = 1 To l           'movement along time axis
2:      Call e_functions(u, e)
        For m = 3 To 4
3:          alpha(m) = u(m) + tau2 * e(m)
4:          xx(m) = u(m) + tau * e(m)
        Next m
        For j = 1 To 1000          'Newton iterations
5:          Call fx_jacobian(xx, a)
6:          Call e_functions(xx, e)
            For m = 3 To 4
7:              b(m) = alpha(m) - _
                (xx(m) - tau2 * e(m))
            Next m
8:          Call gauss(2, a, b, z, 2, 2)
            For m = 3 To 4
9:              xx(m) = xx(m) + z(m)
            Next m
            max = 0
            For m = 3 To 4
                If Abs(z(m)) > max Then _
                max = Abs(z(m))
            Next m
10:         If max < zeta Then Exit For
        Next j
        For m = 3 To 4
11:         u(m) = xx(m)
            uu(m, i) = u(m)
        Next m
        jj(i) = j
    Next i
14: x(0) = (u(3) - B3) ^ 2
End Sub

Function test(ByVal n, ByRef x() As Double, _
    ByRef z() As Double, ByVal rho, Optional alpha) _
    As Boolean
    nt = nt + 1
    If Abs(z(0) - x(0)) < n * rho * z(0) Then
        test = False
    Else
        test = True
    End If
    If Not IsMissing(alpha) Then
        If x(0) < alpha Then test = False
    End If
    If n = 1 Or nt = 1048576 Then test = False
End Function
```

This code for minimizing function (6.42), actually, is a combination of two codes — for optimizing a tin can (Section 6.4) and for simulation of the piano mechanism (Section 5.6). Operator `14` defines the objective function's form.

The source data for the code are specified in table Listing 6.14 (Fig. 6.34). We must select this table before the code execution.



	A	B	C	D
1				
2		l	20	
3		tau	1.00E-03	
4		A4	0	
5		p	0.406	
6		q	18.3	
7		zeta	1.00E-09	
8		A3	0.007	
9		B3	0.015	
10				

Fig. 6.34. The Excel table with the source data

During the execution:

1) the window with calculated velocity $A_4 = A_4^* = 0.294\,\mathrm{m/s}$ appears (Fig. 6.35);

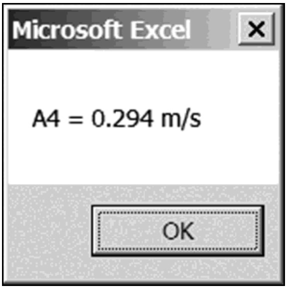
2) after clicking on the *OK* button, the window with calculated velocity $B_4 = B_4^* = 0.623\,\mathrm{m/s}$ appears (Fig. 6.36);

3) after clicking on button *OK*, the following results, depicted in Fig. 6.37, appear:

- the values of *t*, u_3 and u_4;
- the numbers of the Newton iterations, j_{max};
- the graph of dependence $u_3(t)$.

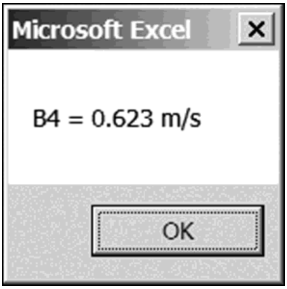
The last graph is the result of the `graph` subroutine execution (operator `13`). On p. 328, we used Fig. 6.37 to demonstrate the features of this subroutine intended for automatic creation of graphs.

In section "Instead of Conclusions", we will need the dependences of the calculated values of A_4 and B_4 versus the number of steps on segment $0 \le t \le b$. These dependences, depicted in Fig. 6.38, were obtained by executing code Listing 6.15 for *l* = 2, 3, 5, 10, 20, 30. The time step is equal to $\tau = b / l$, where *b* = 0.02.



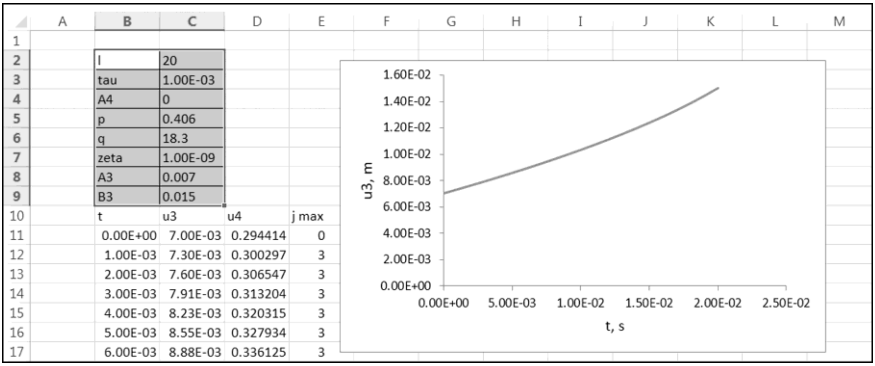
A4 = 0.294 m/s

Fig. 6.35. Window with the calculated value of the hammer velocity at moment $a = 0$



B4 = 0.623 m/s

Fig. 6.36. Window with the calculated value of the hammer velocity at moment $b = 20\,\text{ms}$



	A	B	C	D	E
1					
2		l	20		
3		tau	1.00E-03		
4		A4	0		
5		p	0.406		
6		q	18.3		
7		zeta	1.00E-09		
8		A3	0.007		
9		B3	0.015		
10		t	u3	u4	j max
11		0.00E+00	7.00E-03	0.294414	0
12		1.00E-03	7.30E-03	0.300297	3
13		2.00E-03	7.60E-03	0.306547	3
14		3.00E-03	7.91E-03	0.313204	3
15		4.00E-03	8.23E-03	0.320315	3
16		5.00E-03	8.55E-03	0.327934	3
17		6.00E-03	8.88E-03	0.336125	3

Fig. 6.37. The code execution results, which include the $u_3(t)$ graph

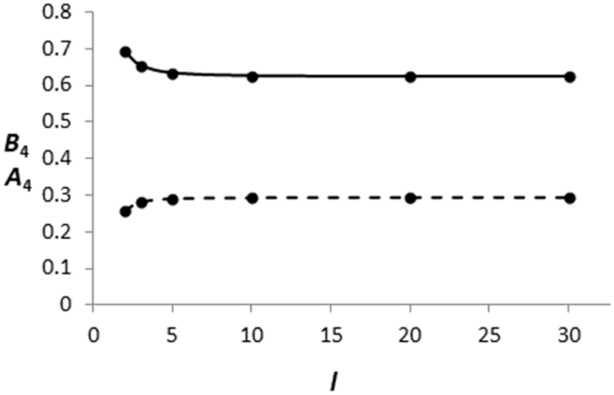


Fig. 6.38. Velocities A_4 (dashed curve) and B_4 (continuous curve) versus the number of time steps, i.e., dependences $A_4(l)$ and $B_4(l)$

6.12. Nonlinear programming and the least-squares method

In Section 5.8, we considered the least-squares method to solve the task of determining the production function, more precisely, the functional dependence of the wheat productivity on the land quality. Let us return to this question.

According to the least-squares method, to get required values L_1, ..., L_κ, ..., L_n of the linear spline at the grid nodes, $z_1 < ... < z_\kappa < ... < z_n$, we have to find the minimum point of non-negative function (5.43) of form

$$G(L_1, ..., L_\kappa, ..., L_n) = \sum_{j=1}^{\nu} [L(x_j) - u_j]^2 .$$

In Section 5.9, we minimized this function by solving the system of linear algebraic equations (5.48). In this section, we will use the Powell method for the minimization.

Below is a code for finding required values L_1, ..., L_κ, ..., L_n.

Listing 6.16

```
Dim nf As Long                  'counter of calls of func
Dim nt As Long                  'counter of calls of test
Dim m As Integer
Dim XX() As Double
Dim UU() As Double
Dim ZZ() As Double

Sub main()
    Dim x() As Double
    Dim ss() As Double
    Dim n As Integer
    Dim i As Integer
    Dim j As Integer
    Dim d As Double
    m = Selection.Rows.Count             'quantity of rows
    n = Selection.Cells(1, 2)            'number of nodes
    ReDim XX(3 To m)
    ReDim UU(3 To m)
    ReDim ZZ(1 To n)
    ReDim x(n)
    ReDim ss(1 To n, 1 To n)
    For j = 3 To m
        XX(j) = Selection.Cells(j, 1)
        UU(j) = Selection.Cells(j, 2)
    Next j
    For i = 1 To n
        ZZ(i) = Selection.Cells(2 + i, 3)
    Next i
    For j = 1 To n
        x(j) = Cells(1, 6 + j)
        For i = 1 To n
            ss(i, j) = Cells(1 + i, 6 + j)
        Next i
    Next j
    For j = 1 To n
        d = 0
        For i = 1 To n
            d = ss(i, j) ^ 2 + d
        Next i
        If d = 0 Then
            Cells(1, 1).Value = _
                "You must increase" & Str(j) & _
                "-th initial step"
            End
        End If
    Next j
    nf = 0
    nt = 0
    Call func(n, x)    'it must be before minimization
0:  Call minim(n, x, ss, 1E-6)          'Powell method
'Output of results:
    For j = 0 To n
        Selection.Cells(m + 2, j + 1) = x(j)
    Next j
    Selection.Cells(m + 2, n + 2) = nt
    Selection.Cells(m + 2, n + 3) = nf
    Selection.Cells(2, 4) = "L"
    For i = 1 To n
```

```
        Selection.Cells(2 + i, 4) = x(i)
    Next i
End Sub

Sub func(ByVal n, ByRef x() As Double)
    Dim j As Integer, L As Double
    nf = nf + 1
    x(0) = 0
    For j = 3 To m
        If XX(j) <= ZZ(2) Then
            L = ((ZZ(2) - XX(j)) * x(1) + _
            (XX(j) - ZZ(1)) * x(2)) / (ZZ(2) - ZZ(1))
        Else
            L = ((ZZ(3) - XX(j)) * x(2) + _
            (XX(j) - ZZ(2)) * x(3)) / (ZZ(3) - ZZ(2))
        End If
        x(0) = x(0) + (L – UU(j)) ^ 2
    Next j
End Sub

Function test(ByVal n, ByRef x() As Double, _
    ByRef z() As Double, ByVal rho, Optional alpha) _
    As Boolean
    Dim j As Byte
    nt = nt + 1
    If Abs(z(0) - x(0)) < n * rho * z(0) Then
        test = False
    Else
        test = True
    End If
    If Not IsMissing(alpha) Then
        If x(0) < alpha Then test = False
    End If
End Function
```

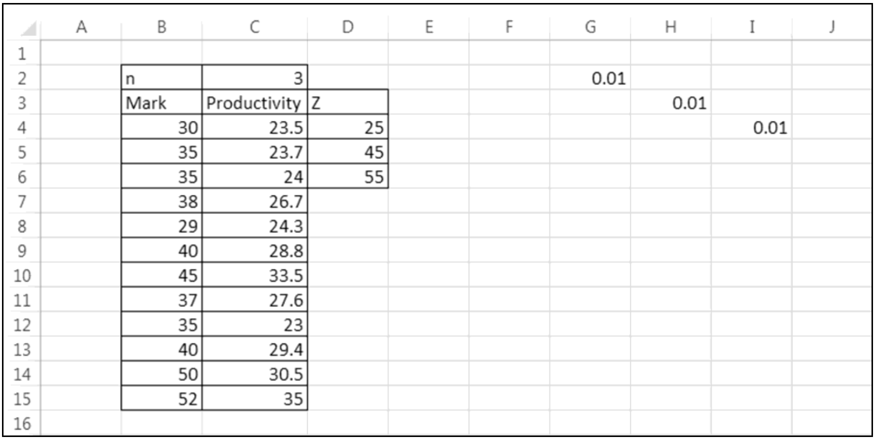
This code has features of program Listing 5.6.

In addition to table Listing 5.5 (with the source data for program Listing 5.6), the G1:I4 range contains the source data for code Listing 6.16 (Fig. 6.39):

- range G1:I1 contains zero initial approximations of required coordinates `x(1) =` L_1, `x(2) =` L_2, `x(3) =` L_3 of the minimum point;

- range G2:I4 contains the **S** matrix of initial steps.

Arrays `XX`, `UU`, `ZZ` are used in Listing 6.16 instead of arrays `X`, `U`, `Z` in Listing 5.6. Before the code execution, we must select the B2:D15 range.



	A	B	C	D	E	F	G	H	I	J
1										
2		n	3				0.01			
3		Mark	Productivity	Z				0.01		
4		30	23.5	25					0.01	
5		35	23.7	45						
6		35	24	55						
7		38	26.7							
8		29	24.3							
9		40	28.8							
10		45	33.5							
11		37	27.6							
12		35	23							
13		40	29.4							
14		50	30.5							
15		52	35							
16										

Fig. 6.39. The source data

According to Fig. 6.40, four iterations were performed and 104 values of objective function (5.43) were calculated during the minimization by the Powell method (by means of the `minim` subroutine, see operator `0`). The calculated values L_1, L_2, L_3 of the linear spline at grid nodes $z_1 < z_2 < z_3$ are respectively placed in cells E4, E5 and E6 (and also in cells C17, D17 and E17). Naturally, these values are the same as when using program Listing 5.6.

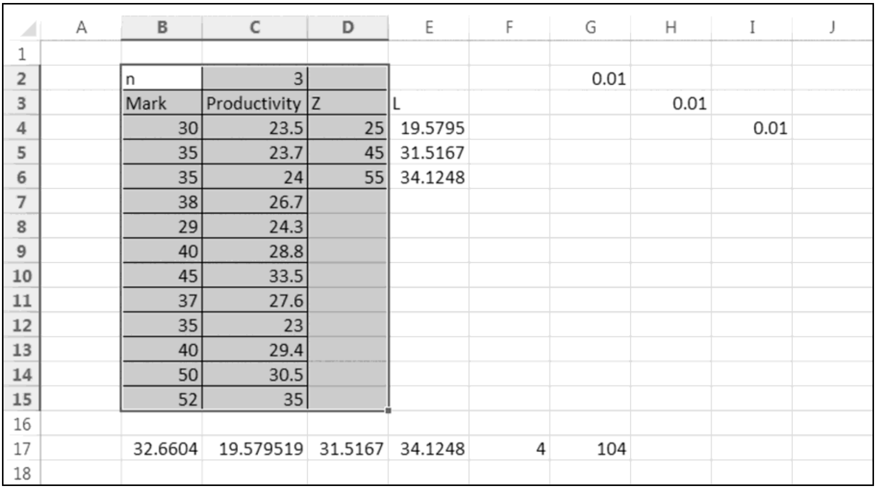
When using the `minim` subroutine in the least-squares method, we can easily switch to another form of the required functional dependence, for example, to quadratic form

$$F(C_1, C_2, C_3, x) = C_1 + C_2 x + C_3 x^2 \ . \tag{6.43}$$

The values of constants C_1, C_2 and C_3 must be determined by minimizing the following non-negative function similar to (5.42):

$$G(C_1, C_2, C_3) = \sum_{j=1}^{\nu} [F(C_1, C_2, C_3, x_j) - u_j]^2 \ , \tag{6.44}$$

where x_j and u_j are the values given in columns *Mark* and *Productivity* of the source data table depicted in Fig. 6.39, $1 \le j \le v$ ($v = 12$ is the number of land plots).



	A	B	C	D	E	F	G	H	I	J
1										
2		n	3				0.01			
3		Mark	Productivity	Z	L			0.01		
4		30	23.5	25	19.5795				0.01	
5		35	23.7	45	31.5167					
6		35	24	55	34.1248					
7		38	26.7							
8		29	24.3							
9		40	28.8							
10		45	33.5							
11		37	27.6							
12		35	23							
13		40	29.4							
14		50	30.5							
15		52	35							
16										
17		32.6604	19.579519	31.5167	34.1248	4	104			
18										

Fig. 6.40. The execution results

The following code is intended for finding the values of constants C_1, C_2 and C_3 by means of the Powell minimization method.

Listing 6.17

```
Dim nf As Long                   'counter of calls of func
Dim nt As Long                   'counter of calls of test
Dim m As Integer
Dim XX() As Double
Dim UU() As Double

Sub main()
    Dim x() As Double
    Dim ss() As Double
    Dim n As Integer
    Dim i As Integer
    Dim j As Integer
    Dim d As Double
    m = Selection.Rows.Count         'quantity of rows
    n = Selection.Cells(1, 2)        'number of nodes
    ReDim XX(3 To m)
    ReDim UU(3 To m)
    ReDim x(n)
    ReDim ss(1 To n, 1 To n)
    For j = 3 To m
        XX(j) = Selection.Cells(j, 1)
        UU(j) = Selection.Cells(j, 2)
    Next j
    For j = 1 To n
        x(j) = Cells(1, 6 + j)
        For i = 1 To n
            ss(i, j) = Cells(1 + i, 6 + j)
        Next i
    Next j
    For j = 1 To n
        d = 0
        For i = 1 To n
            d = ss(i, j) ^ 2 + d
        Next i
        If d = 0 Then
            Cells(1, 1).Value = _
                "You must increase" & Str(j) & _
                "-th initial step"
            End
        End If
    Next j
    nf = 0
    nt = 0
    Call func(n, x)     'it must be before minimization
0:  Call minim(n, x, ss, 1E-6)          'Powell method
'Output of results:
    For j = 0 To n
        Selection.Cells(m + 2, j + 1) = x(j)
    Next j
    Selection.Cells(m + 2, n + 2) = nt
    Selection.Cells(m + 2, n + 3) = nf
    Selection.Cells(2, 4) = "C"
    For i = 1 To n
        Selection.Cells(2 + i, 4) = x(i)
    Next i
End Sub

Sub func(ByVal n, ByRef x() As Double)
    Dim j As Integer
    Dim F As Double
    nf = nf + 1
    x(0) = 0
    For j = 3 To m
        F = x(1) + x(2) * XX(j) + x(3) * XX(j) ^ 2
        x(0) = x(0) + (F - UU(j)) ^ 2
    Next j
End Sub

Function test(ByVal n, ByRef x() As Double, _
    ByRef z() As Double, ByVal rho, Optional alpha) _
    As Boolean
    Dim j As Byte
    nt = nt + 1
    If Abs(z(0) - x(0)) < n * rho * z(0) Then
        test = False
    Else
        test = True
    End If
    If Not IsMissing(alpha) Then
        If x(0) < alpha Then test = False
    End If
End Function
```

This code slightly differs from Listing 6.16.

For code Listing 6.17, the source data are almost the same as for code Listing 6.16:

- column *Z* may be absent (Fig. 6.41);
- range G1:I1 contains zero initial approximations of required coordinates `x(1) =` C_1, `x(2) =` C_2, `x(3) =` C_3 of the minimum point;
- range G2:I4 contains the **S** matrix of initial steps.

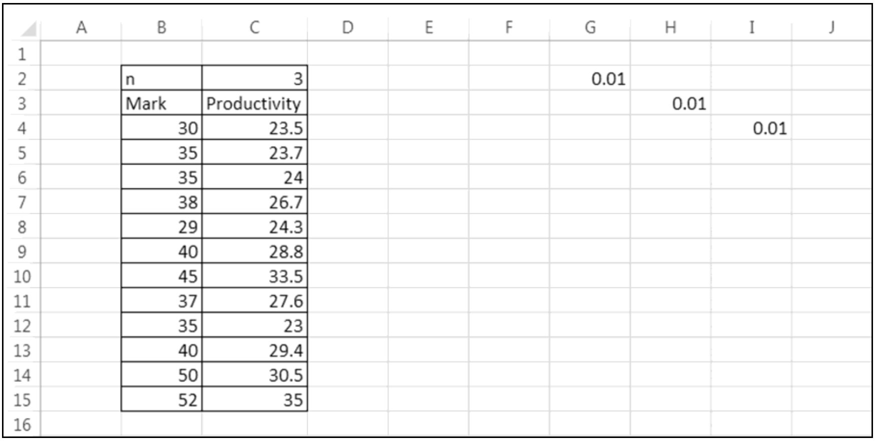
Before the code execution, we must select the B2:C15 range.

According to Fig. 6.42, four iterations were performed and 92 values of objective function (6.44) were calculated during the minimization by the Powell method. The calculated values, C_1^*, C_2^* and C_3^*, are respectively placed in cells

E4, E5 and E6 (and also in cells C17, D17 and E17). Fig. 6.43 shows the graph of quadratic dependence (6.43) with the calculated constants:

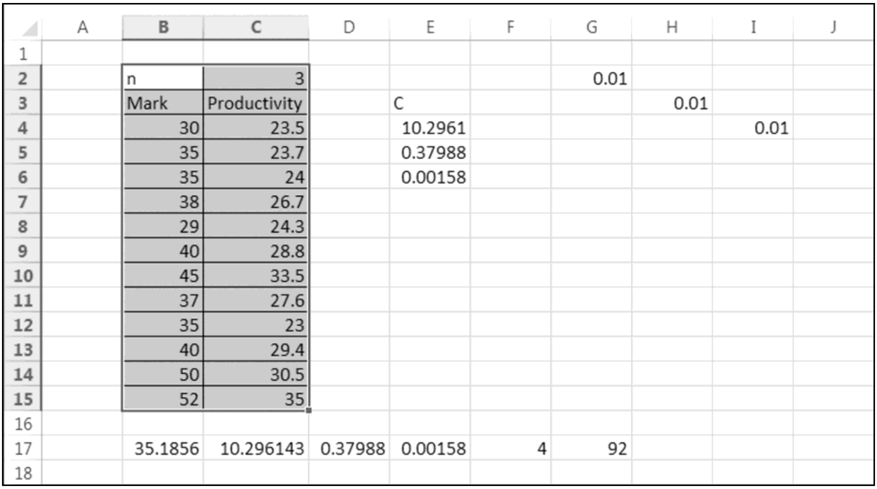
$$F(C_1^*,\, C_2^*,\, C_3^*,\, x) = C_1^* + C_2^* x + C_3^* x^2 \;,$$

where C_1^* = 10.2961, C_2^* = 0.37988 and C_3^* = 0.00158.



	A	B	C	D	E	F	G	H	I	J
1										
2		n	3				0.01			
3		Mark	Productivity					0.01		
4		30	23.5						0.01	
5		35	23.7							
6		35	24							
7		38	26.7							
8		29	24.3							
9		40	28.8							
10		45	33.5							
11		37	27.6							
12		35	23							
13		40	29.4							
14		50	30.5							
15		52	35							
16										

Fig. 6.41. The source data



	A	B	C	D	E	F	G	H	I	J
1										
2		n	3				0.01			
3		Mark	Productivity		C			0.01		
4		30	23.5		10.2961				0.01	
5		35	23.7		0.37988					
6		35	24		0.00158					
7		38	26.7							
8		29	24.3							
9		40	28.8							
10		45	33.5							
11		37	27.6							
12		35	23							
13		40	29.4							
14		50	30.5							
15		52	35							
16										
17		35.1856	10.296143	0.37988	0.00158	4	92			
18										

Fig. 6.42. The execution results

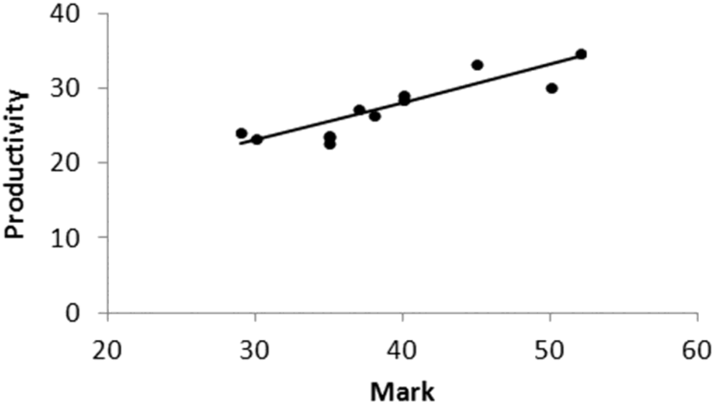


Fig. 6.43. The experimental points and the quadratic line of the functional dependence of the wheat productivity on the land quality mark

Instead of Conclusions

For solving mathematical and applied tasks, an alternative to the numerical method is the analytical method aimed at obtaining a solution in the formula form. It is interesting to compare these two methods.

The obvious advantage of numerical methods over analytical ones is their generality. However, numerical methods have a disadvantage consisting in the approximate nature of the numerical solution of a task. Therefore, for improving the accuracy, we have to run the program (realizing the numerical method) several times while changing the values of the numerical method parameters. The program execution must be repeated until the execution results become independent of the parameter values. It is these results that are reliable.

As an illustration, let us return to the task of modeling of the hammer motion in the piano mechanism when the spring is ruptured (Section 6.11).

In this task:

- the numerical method parameter is l — the number of steps on the time segment considered, $0 \le t \le b = 20\,\text{ms}$;
- the results of solving the task are A_4 and B_4 — the hammer velocities at the initial time moment and in 20 ms, respectively.

It is obvious that the accuracy of the numerical solution of the formulated task improves with increasing l.

We solved the task for different values of l and depicted dependences $A_4(l)$ and $B_4(l)$ in Fig. 6.38. According to these dependences, the values of A_4 and B_4 do not depend on l beyond 10. Therefore, the values obtained at $l = 20$ (A_4 = $0.294\,\text{m/s}$ and B_4 = $0.623\,\text{m/s}$) are authentic within the limits of the used mathematical model of the hammer motion when the spring is ruptured.

The question of losing the numerical solution accuracy was also considered at the end of Section 3.7.

Analytical and numerical methods are used successively for solving many problems. For example, the integral of a complex function may be a result of the analytical solution, and the numerical integration is required for obtaining the values of this integral: according to formula (5.41) on p. 385, the numerical integration of function $f(x)$ must be done to complete the analytical solution of the initial value problem for the Bernoulli differential equation.

One more example of successive usage of analytical and numerical methods is in Section 4.12, where:

1) the first derivative was excluded from differential equation (4.60) by substitution (4.62);

2) the resulting problem was solved numerically by the cubic spline method;

3) the original problem for differential equation (4.60) was solved by using formula (4.62).

Guided by the material of Section 2.14, we can create Personal Macro Workbook with our program modules. However, we can go even further, namely, can modify our program modules for using them as the Excel add-ins. For this purpose, package Microsoft Visual Studio can be used. Let us consider how to tune Microsoft Visual Studio 2010 for this.

According to Section 1.24, the *Start Page* window opens when starting Microsoft Visual Studio 2010. Further, we must:

1) click on the *New Project* hyperlink;

2) in the left area of the *New Project* window opened, click on the plus sign against *Other Languages*;

3) in the open list, click on the plus sign against *Visual Basic*;

4) in the open list, click on the plus sign against *Office*, and further click on line *2010* (or *2007*);

5) in the list of the central area of the *New Project* window, click on line *Excel 2010 Add-In*.

As a result, the necessary operation mode of Visual Studio is set. The following information in the right area of the *New Project* window speaks about it: *A project for creating a managed code add-in for Excel 2010*.

Further, we must specify the project name and the project folder location by using text boxes *Name* and *Location*. At clicking on the *OK* button, the window of Visual Basic Environment appears. This window includes the code window with automatically generated lines.

Information on creating an add-in for Excel is available in the following internet resource: http://msdn.microsoft.com/en-us/library/.

The idea of writing this book was spurred by book [17], the best seller of the end of the 1960s. The last book convinced the author that learning numerical methods is more effective if it follows learning programming and builds on the skills of programming. In this case, the learner has the possibility to grasp difficult material on numerical methods by developing the program modules that realize these methods.

In conclusion, the author would like to share the experience of using Excel for solving scientific and engineering problems.

Instead of Conclusions

In Pulsar R&D Manufacturing Company, Moscow, tabular processor Excel, equipped with macros, is used as a preprocessor and postprocessor for complicated programs written earlier in the Visual C++ programming language (a part of Microsoft Visual Studio) for mathematical modeling of microwave transistors [18]. In other words, the source data for the modeling are prepared in Excel and the results are processed in Excel, in particular, graphs are constructed.

As we know, the Visual Basic programming language includes the `Shell` function (p. 106), which allows running, from Excel, the executable file of an arbitrary program, in particular, of a program written in Visual C++. By using the `Shell` function, the author created the system of mathematical modeling based on Excel. In this system, programs (written in Visual C++) and Excel exchange data by means of text files.

When developing a program in Visual C++ (or in any other programming language), we should pay attention to the following: the text file with the tabulation character, as the substring connector (p. 95), is not only the text file, but also the Excel workbook containing one worksheet. When we open such file with Excel, substrings appear in separate cells, and Excel interprets a substring, which contains digits and other features of a number, as the corresponding number.

The author hopes that he has managed to achieve the goals formulated at the beginning of the book. It remains to wish the reader successful solution of interesting problems.

Appendix 1.
Data Types of Visual Basic and VBA

Data type	Memory cell size	Values of variable / constant or comment
`Boolean` (logical)	2 B	`True` (logical unit), `False` (logical zero)
`Byte` (short integer unsigned)	1 B	Integers from 0 to 255
`Integer` (integer)	2 B	Integers from -32768 to 32767
`Long` (long integer)	4 B	Integers from -2147483648 to 2147483647
`Currency` (scaled integer)	8 B	Numbers with four decimal places from -922337203685477.5808 to 922337203685477.5807
`Single` (single-precision)	4 B	Numbers with a fractional part from $-3.402823 \cdot 10^{38}$ to $-1.401298 \cdot 10^{-45}$ for negative values and from $1.401298 \cdot 10^{-45}$ to $3.402823 \cdot 10^{38}$ for positive values
`Double` (double-precision)	8 B	Numbers with a fractional part from $-1.79769313486231 \cdot 10^{308}$ to $-4.94065645841247 \cdot 10^{-324}$ for negative values and from $4.94065645841247 \cdot 10^{-324}$ to $1.79769313486232 \cdot 10^{308}$ for positive values
`Date` (date-time)	8 B	Date from 1 January 100 to 31 December 9999 and time from 0:00:00 to 23:59:59

Data type	Memory cell size	Values of variable / constant or comment
`String`	10 B + 1 B per character for string of variable length	String length from 0 to 2^{31} characters
	1 B per character for string of fixed length	String length from 1 to 2^{16} characters
`Variant`	16 B if the choice by context gives data type `Boolean`, `Byte`, `Integer`, `Long`, `Currency`, `Single`, `Double` or `Date`	Values correspond to the `Boolean`, `Byte`, `Integer`, `Long`, `Currency`, `Single`, `Double` or `Date` data type
	22 B + 1 B per character if the choice is not made	Values correspond to `String` of variable length
`Object`	4 B	The memory cell stores the object reference
User-defined	Depends on the quantity of fields and their data types	Created by the `Type` operator

Appendix 2.
Greek and Russian Alphabets Denoted by Latin Letters

The Greek alphabet with English names of the letters

Lowercase (small) letter	**Uppercase (capital) letter**	**English name**
α	Α	Alpha
β	Β	Beta
γ	Γ	Gamma
δ	Δ	Delta
ε	Ε	Epsilon
ζ	Ζ	Zeta
η	Η	Eta
θ	Θ	Theta
ι	Ι	Iota
κ	Κ	Kappa
λ	Λ	Lambda
μ	Μ	Mu
ν	Ν	Nu
ξ	Ξ	Xi
ο	Ο	Omicron
π	Π	Pi
ρ	Ρ	Rho
σ	Σ	Sigma
τ	Τ	Tau
υ	Υ	Upsilon
φ	Φ	Phi
χ	Χ	Chi
ψ	Ψ	Psi
ω	Ω	Omega

The Russian alphabet denoted by Latin letters

Russian	Latin
а; А	a; A
б; Б	b; B
в	v
г	g
д	d
е; Е	ye, e; Ye, E
ё; Ё	yo; Yo
ж	zh
з	z
и	i
й	y
к	k
л	l
м	m
н	n
о	o

Russian	Latin
п	p
р	r
с	s
т	t
у	u
ф	f
х	kh, h
ц	ts
ч	tch, ch
ш	sh
щ	sch
ы	y
э	e
ю	yu
я	ya
ъ, ь	apostrophe

Appendix 3.
The Main Mathematical Functions

The main mathematical functions of Visual Basic

Call of the function	Return value, mathematical designation and domain of the function
`Abs(x)`	Absolute value of *x*, $\mid x \mid$
`Atn(x)`	Arctangent of *x*, $\arctan\ x$
`Cos(x)`	Cosine of *x*, $\cos x$
`Exp(x)`	Exponential function, $e^x = \exp x$ (Fig. 3.6)
`Fix(x)`	The result of truncating the fractional part of *x*
`Int(x)`	The greatest integer not exceeding *x*
`Log(x)`	Natural logarithm of *x*, $\ln x$, at $x > 0$
`Sgn(x)`	Sign function: -1, 0 or 1, depending on the sign of *x*
`Sin(x)`	Sine of *x*, $\sin x$
`Sqr(x)`	Square root of *x*, $\sqrt{x}$, at $x \geq 0$
`Tan(x)`	Tangent of *x*, $\tan x$, at $x \neq (0.5 + k)\pi$, *k* is an integer

Examples of using the above functions

The Visual Basic operators for calculating the values of trigonometric function $\cot x$, of inverse trigonometric functions $\arcsin\ x$, $\arccos x$ and $\text{arccot}\ x$ and of decimal logarithm $\lg x$ are given below.

```
cot_x = Cos(x) / Sin(x)                   'if Sin(x) <> 0

arcsin_x = Atn(x / Sqr(1 - x ^ 2))        'if Abs(x) < 1
arccos_x = Atn(Sqr(1 - x ^ 2) / x)
                                     'if x > 0 And x <= 1
arccot_x = Atn(1 / x)                     'if x > 0

lg_x = Log(x) / 2.302585093               'if x > 0
```

Appendix 4.
Material for Tasks

The table below, taken from [19], contains 31 functions $f(x)$ and their domains $[a, b]$ with the following properties:

- function $f(x)$ is continuous and monotonous on segment $[a, b]$;
- the signs of $f(x)$ on the left and right boundaries of $[a, b]$ are different.

This table is used in tasks for the reader (the tasks begin with words "***we advise the reader***"). The $f(a)$ values, given in the table, are intended to help the reader debug his programs.

By using this table, a teacher can create 31 variants of tasks for exams. The variant number may be a student's birthday.

No. of variant	**Function *f*(*x*)**	***a***	***b***	***f*(*a*)**
1	$(3.8-3\sin\sqrt{x})/0.35-x$	2	3	0.3905776
2	$[3+\sin(3.6x)]^{-1}-x$	0	0.85	0.3333333
3	$\cos\sqrt{1-0.3x^3}-x$	0	1	0.5403023
4	$\sin\sqrt{1-0.4x^2}-x$	0	1	0.841471
5	$0.25x^3-x-1.2502$	2	3	-1.2502
6	$0.1x^2-x\ln x$	1	2	0.1
7	$3x-4\ln x-5$	2	4	-1.772589
8	$e^x-e^{-x}-2$	0	1	-2
9	$x+\sqrt{x}+\sqrt[3]{x}-2.5$	0.4	1	-0.7307382
10	$\tan x-(\tan^3 x)/3+(\tan^5 x)/5-1/3$	0	0.8	-0.3333333
11	$\cos(2/x)-2\sin(1/x)+1/x$	1	2	-1.099089
12	$\sin(\ln x)-\cos(\ln x)+2\ln x$	1	3	-1
13	$\ln x-x+1.8$	2	3	0.4931472

No. of variant	Function *f*(*x*)	*a*	*b*	*f*(*a*)
14	$0.4+\arctan\sqrt{x}-x$	1	2	0.1853982
15	$x\tan x-1/3$	0.2	1	-0.2927913
16	$\tan(0.55x+0.1)-x^2$	0.4	1	0.171389
17	$2-\sin(1/x)-x$	1.2	2	0.0598231
18	$1+\sin x-\ln(1+x)-x$	0	1.5	1
19	$-\cos(x^{0.52}+2)-x$	0.5	1	0.4029458
20	$\sqrt{\ln(1+x)+3}-x$	2	3	0.024503
21	$e^x+\ln x-10x$	3	4	-8.815851
22	$3x-14+e^x-e^{-x}$	1	3	-8.649598
23	$2\ln^2 x+6\ln x-5$	1	3	-5
24	$2x\sin x-\cos x$	0.4	1	-0.6095263
25	$\cos x-\exp(-x^2/2)+x-1$	1	2	-0.0662284
26	$\sqrt{1-x}-\tan x$	0	0.9	1
27	$\sin(x^2)+\cos(x^2)-10x$	0	1	1
28	$e^x+\sqrt{1+e^{2x}}-2$	-1	0	-0.5665994
29	$\sqrt{1-x}-\cos\sqrt{1-x}$	0	0.9	0.4596977
30	$\tan(x/2)-\cot(x/2)+x$	1	2	-0.2841852
31	$x-\cos x$	0.5	2.5	-0.3775826

Appendix 5.
Analytical Method for Solving the Cubic Algebraic Equation

Below, we will present Method 3 of handbook [3] for solving cubic equation $z^3 + 3p\,z + 2q = 0$, where p and q are nonzero real numbers.

Let us consider variable $r = \pm\sqrt{|p|}$, the sign coinciding with the sign of q, i.e., $\gamma = q/r^3 > 0$. The auxiliary value, φ, and the roots, z_1, z_2 and z_3, are determined according to the following table.

$p < 0$		$p > 0$
$q^2 + p^3 \le 0$ or $0 < \gamma \le 1$	$q^2 + p^3 > 0$ or $\gamma > 1$	
$\varphi = \arccos\gamma =$ $= \operatorname{arctan}\dfrac{\sqrt{1-\gamma^2}}{\gamma}$	$\varphi = \operatorname{Arcosh}\gamma =$ $= \ln\left(\gamma + \sqrt{\gamma^2 - 1}\right)$	$\varphi = \operatorname{Arsinh}\gamma =$ $= \ln\left(\gamma + \sqrt{\gamma^2 + 1}\right)$
$z_1 = -2r\cos\dfrac{\varphi}{3}$	$z_1 = -2r\,\operatorname{ch}\dfrac{\varphi}{3}$	$z_1 = -2r\,\operatorname{sh}\dfrac{\varphi}{3}$
$z_2 = 2r\cos\dfrac{\pi - \varphi}{3}$	$z_2 = r\,\operatorname{ch}\dfrac{\varphi}{3} + i\sqrt{3}\,r\,\operatorname{sh}\dfrac{\varphi}{3}$	$z_2 = r\,\operatorname{sh}\dfrac{\varphi}{3} + i\sqrt{3}\,r\,\operatorname{ch}\dfrac{\varphi}{3}$
$z_3 = 2r\cos\dfrac{\pi + \varphi}{3}$	$z_3 = r\,\operatorname{ch}\dfrac{\varphi}{3} - i\sqrt{3}\,r\,\operatorname{sh}\dfrac{\varphi}{3}$	$z_3 = r\,\operatorname{sh}\dfrac{\varphi}{3} - i\sqrt{3}\,r\,\operatorname{ch}\dfrac{\varphi}{3}$

As we see, the above formulas for calculating the roots depend on the signs of p and $q^2 + p^3$ (i is the imaginary unit). The following mathematical functions are used:

- ch, sh — the hyperbolic cosine and sine:
  $\operatorname{ch} x = \cosh x = (e^x + e^{-x})/2$, $\operatorname{sh} x = \sinh x = (e^x - e^{-x})/2$;
- Arcosh, Arsinh — the area-hyperbolic cosine and sine.

Appendix 6. Realization of the Tangent Method by Using the Excel Circular Reference

To use the circular reference feature, let us tune Excel as follows:

1) click on the *File* button;

2) *Options* > *Formulas*;

3) turn on option *Automatic* in area *Calculation options* of the *Excel Options* window;

4) turn on option *Enable iterative calculation*;

5) for example, set 100 for the limiting number of iterations in text box *Maximum Iterations*;

6) for example, set 0.001 for the final relative error in text box *Maximum Change*;

7) click on the *OK* button.

If a formula, placed in an Excel cell, contains the reference to the same cell (maybe not directly but indirectly, through a series of other references), we say that the circular reference exists. The cyclic reference is used when we want to realize an iterative or recurrence process.

Let us use the cyclic reference for solving nonlinear algebraic equation $f(x)=0$, where $f(x)=x-\cos\, x-1.5$, by the tangent method (Section 5.5). For that, we fulfill the following operations on an Excel worksheet.

1. Into cell G2, intended for variable *x*, enter an initial approximation of the solution, for example 2.
2. Into cell F2, enter formula

```
=G2-COS(G2)-1.5
```

corresponding to mathematical formula $f(x)=x-\cos\, x-1.5$. When clicking on the tick button of the Excel formula bar, value 0.916147 appears in cell F2.

3. Into cell E2, enter formula

```
=1+SIN(G2)
```

corresponding to mathematical formula $f'(x)=1+\sin\, x$. When clicking on the tick button, value 1.909297 appears in the E2 cell.

4. Into cell G2, intended for variable *x*, enter formula

```
=G2-F2/E2
```

corresponding to mathematical formula

$$x^{j+1} = x^{j} - \frac{f(x^{j})}{f'(x^{j})}$$

of the tangent method. When we click on the tick button, values 1.999373, 0 and 1.535394 appear in cells E2, F2 and G2, respectively.

The multiple circular reference to cell G2 gives the last three values. Thus, G2 contains the result of solving equation $x - \cos x - 1.5 = 0$, which is equal to x^{*} = 1.535394.

In a similar way, the cyclic reference can be used for solving equation $f(x) = 0$ by other iterative methods of Sections 4.5 and 5.5.

References List

1. G. Z. Garber, *Bases of Programming in Visual Basic and VBA for Excel 2007* (in Russian), Moscow: SOLON-PRESS, 2008.
2. G. Z. Garber, *Bases of Programming in VBA for Excel and of Numerical Methods* (in Russian), Moscow: PRINTKOM, 2009.
3. I. N. Bronshtein, K. A. Semendyayev, G. Musiol, H. Muehlig, *Handbook of Mathematics*, 5th edition, Springer, 2007.
4. A. A. Samarskii, *The Theory of Difference Schemes*, Marcel Dekker, 2001.
5. T. Y. Na, *Computational Methods in Engineering: Boundary Value Problems*, Academic Press, 1979.
6. G. Z. Garber, E. V. Kostyukov, Yu. A. Kuznetsov, *Two-dimensional modeling of a cell of photosensitive silicon charge-coupled device for color television cameras* (in Russian), Electronic Engineering. Series 2. Semiconductor Devices, 1989, no. 3, pp. 40 – 44.
7. W. Shockley, *Electrons and Holes in Semiconductors, with Applications to Transistor Electronics*, Van Nostrand Reinhold, 1950.
8. H. Gould, J. Tobochnik, *An Introduction to Computer Simulation Methods: Applications to Physical Systems*, Addison-Wesley, 1988.
9. J. H. Ahlberg, E. N. Nilson, J. L. Walsh, *The Theory of Splines and Their Applications*, Academic Press, 1967.
10. G. Z. Garber, *Model for simulation of AlGaAs-GaAs power heterostructure FETs*, Proceedings of EUROCON 2005 – The International Conference on “Computer as a Tool”, Belgrade: IEEE, 2005, vol. 1, pp. 867 – 870.
11. G. Z. Garber, *Method for calculating a small-signal equivalent circuit of extremely high frequency heterostructural field-effect transistors*, Journal of Communications Technology and Electronics, 2005, vol. 50, no. 7, pp. 822 – 825.
12. A. Oledzki, *Dynamics of piano mechanisms*, Mechanism and Machine Theory, 1972, vol. 7, pp. 373 – 385.

13. S. N. Volkov, *Land Management. Economic and Mathematical Methods and Models* (in Russian), Moscow: KOLOS, 2001.

14. M. J. D. Powell, *An efficient method for finding the minimum of a function of several variables without calculating derivatives*, The Computer Journal, 1964, vol. 7, no. 2, pp. 155 – 162.

15. M. I. Korobochkin, *Solving optimization problems of linear and nonlinear programming in Excel* (in Russian), Land Management, Cadastre and Monitoring of Lands, 2006, no. 12, pp. 73 – 76.

16. D. M. Himmelblau, *Applied Nonlinear Programming*, McGraw-Hill, 1972.

17. D. D. McCracken, W. S. Dorn, *Numerical Methods and Fortran Programming*, John Wiley and Sons, 1965.

18. G. Z. Garber, *Experience of using Microsoft Excel for mathematical modeling of microwave transistors* (in Russian), Electronic Engineering. Series 2. Semiconductor Devices, 2012, no. 2, pp. 22 – 27.

19. V. B. Glagolev, *Visual Basic 6.0. Collection of tasks for laboratory and practical classes* (in Russian), Publishing House of Moscow Power Engineering Institute, 2000.

Subject Index

A

Abs 230, 517
Activate 161, 162, 167
ActiveCell 148, 155, 156, 158, 174
ActiveChart 245
ActiveSheet 155, 168, 169, 245
ActiveWorkbook 155, 162, 163
Add 152, 162, 164, 167, 170, 245
Add Watch 19, 72, 73, 75
Address 172, 174, 245
And 48 – 52
Append 101, 102, 106, 108
Application 148, 152, 154 – 160, 165
Array
  static 68
  dynamic 71
As 23, 29, 77, 78, 81, 90, 101, 103
Atn 45, 315, 517
Auto_Open 159, 160
Average 154, 155

B

Boolean 23, 29, 43, 48, 439, 513
ByRef 80
Byte 23, 24, 43, 513
ByVal 80

C

Calculate 158
Calculation 156, 159
Call 79
Caption 117, 120, 121
Case 56, 57
Case Else 57
CBool 43
CByte 43
CCur 43
CDate 43
CDbl 43
Cells 148, 167, 169, 172, 173
Charts 245
CheckBox 125 – 127, 151
CInt 43, 44, 51, 153
Clear 172
Clear All Breakpoints 19
CLng 43
Close 102 – 104, 161, 164
Color 169
Columns 142, 172, 173, 188, 189
CommandButton 120, 151
Const 29, 30, 210
Constant
  built-in 31
  user-defined 29
Cos 46, 517
Count 142, 162, 168, 188, 189
Course
  backward 225
  forward 225
CSng 43
CStr 43, 44, 96, 153
Currency 23, 24, 27, 29, 43, 513
CVar 43
Cycle
  Do…Loop 61
    Do Until…Loop 62, 63, 103
    Do While…Loop 61, 62
    Do…Loop Until 64
    Do…Loop While 63
  For Each…Next 172
  For…Next 58
  While…Wend 61

Subject Index

D
Date   25, 26, 29, 43, 47, 170, 513
Delete   167, 245
Dialogs   156 – 158
Dim   23 – 25, 80, 210
Do   61 – 64
Double   24, 29, 43, 513

E
Else   54, 55, 188, 403
ElseIf   55, 403
End   180, 181, 433
End Function   77, 78, 139
End If   54, 55, 188, 403
End Select   57
End Sub   16, 79, 80
End Type   90, 92
End With   91 – 93, 156 – 158
EOF   103, 104
Erase   75
Event of object   151
exe   106, 132, 135, 138
Exit Do   64
Exit For   60, 245
Exit Function   83
Exit Sub   83, 309
Exp   216, 220 – 222, 517

F
False   29, 48, 49, 126, 513
fb   210, 211, 213
FileFormat   161
FileName   161
Fix   51, 517
foba   252, 253, 258
fobas   334, 335, 338, 347
Font   118, 156, 169
For   58 – 60, 101, 103
For Each   172
forbac   262 – 264
forbacs   266
Format   100, 170
Formula   148, 171, 174
FormulaR1C1   171
forwback   271, 272, 277
foub   404, 405, 411
fouf   402, 404, 411
FreeFile   101, 103
FullName   161, 162
Func9   188 – 190
Function   77, 78, 139

G
gaus   228, 229, 234
gauss   230, 232, 234, 236, 497
GoTo   52, 54, 308
graph   173, 327, 328, 338

H
Height   117
Hypotenuse   139 – 141

I
If   53 – 55, 188, 403
IIf   56
In   172
Input   101 – 104, 108
InputBox   85, 87, 148
Int   51, 517
Integer   23, 24, 43, 513
ios   298 – 300
IsMissing   82, 83, 290
Italic   156, 169

J
Jacobian matrix   364

K
Kill   106

L
Label   118, 151
LBound   74, 75
LCase   96
Left   97, 98
Len   97
Line Input   102, 103
LOF   103, 104
Log   45, 315, 517
Long   23, 24, 25, 43, 513
Loop   61 – 64
LTrim   95, 96

M
MDETERM 224
Method
  barrier function 472
  bisection 306, 309
  interior point 473
  of object 151, 152
  penalty function 466
  secant 310, 311
  Steffensen 370
  tangent 366
  trapezoidal 384
  variable replacement 462
Mid 97, 98
mini 432, 435, 436, 439, 442
minim 447, 453, 464, 486
MINVERSE 224
MkDir 103 – 105
MMULT 224
Mod 39, 40, 42, 56
mos 288, 289, 294
MsgBox 85 – 88, 98

N
Name 120, 161, 162, 167, 169
Next 58 – 60, 172
Not 48 – 50, 82
Now 25, 26, 47, 100, 153, 159, 476
NumberFormat 148, 245

O
Object 151, 166, 514
Offset 172, 174
OnTime 159, 160
Open 101, 103
Operator
  case 56
  conditional 53
  conditional jump 54
  unconditional jump 52
Option Base 69
Option Explicit 24
Optional 82, 83, 230
Or 48 – 51, 112
Output 101 – 103

P
Password 161
Path 161, 163
PERSONAL.XLSB 184 – 186, 189, 190
Preserve 73, 74
Print 102, 103
Private 81, 121
Procedure
  built-in 85
  Fourier Analysis 396
  user-defined
    function 77
    subroutine 79
Property of object 151, 152
Public 81
Public Const 82

Q
QBColor 169
Quit 158, 165

R
Randomize 76, 476, 478
Range 148, 152 – 154, 171, 172, 178
ReDim 71, 73 – 75, 142, 210
ReferenceStyle 148
Replace 97
Reset 20
Right 97, 98
RmDir 106
Rnd 46, 75, 76, 476, 478
Rosenbrock 425
Round 45, 46, 51, 300
Rows 142, 172, 173, 188, 189
RTrim 95, 96
Run 19
Run To Cursor 19

S
Save 161, 162
SaveAs 161, 164
Select 148, 153, 172, 174, 178, 328
Select Case 56, 57
Selection 148, 173, 174, 181, 328
Set 152, 164, 170, 174, 182, 328

Subject Index

SetFocus 134
Sgn 45, 309, 517
Shell 106, 135, 138, 512
Show 134, 136, 156 – 158
si 290, 291, 294, 324
Sin 81, 517
Single 24, 42, 43, 513
Space 96
Sqr 17, 46, 517
Step 58 – 60
Step Into 19, 82, 114
Step Out 82
Step Over 82
Str 27, 43, 44, 300
String 27, 43, 94, 514
Sub 16, 79, 80
Sum 154, 155, 174
Sweep
  backward 202
  forward 202

T
table 315, 316, 318, 324
Tan 517
Text 121, 123, 125
TextAlign 120
TextBox 120 – 122, 125, 151
Then 53 – 55, 188, 403
Time 25, 26, 47, 100, 159
TimeValue 99, 100, 159, 476
To 58 – 60, 68, 71
Toggle Breakpoint 19
TRANSPOSE 141, 142, 224, 295
TRANSPOSEA 142, 143, 224
Trim 95, 96
True 48, 49, 126, 513
Type 90, 92
TypeName 188

U
UBound 74, 75
UCase 96
Until 62 – 64, 71, 103
UserForm 116, 151

V
Val 27, 33 – 35
Value 122, 126, 156, 172 – 174
Variable
  local 81
  module 81
  public 81
Variant 24, 25, 43, 82, 514
vbCrLf 98, 99, 126, 148
vbTab 95

W
Wend 61
While 61 – 63
Width 117
With 91 – 93, 156 – 158
Workbook 161, 164
Workbooks 152, 161, 162, 164
Worksheet 167 – 170
Worksheets 152 – 155, 167 – 170

X
xlCalculationAutomatic 156
xlCalculationManual 156, 159
XLSTART 186

Подписано в печать 21.06.2013
Формат 60x88/16. Печ. л. 33
Заказ № 125

Отпечатано в полн. соотв. с электронной версией заказчика
в ООО «ИПЦ «Маска»
117246, Москва, Научный проезд, 20