MINISTRY OF EDUCATION AND SCIENCE OF UKRAINE

NATIONAL TECHNICAL UNIVERSITY
"KHARKIV POLYTECHNIC INSTITUTE"

O. Zakovorotniy

O. Lipchanska

FUNDAMENTALS OF COMPUTATIONAL INTELLIGENCE

Part 1

Laboratory workshop
for carrying out laboratory work
for full-time and part-time students
by speciality Computer Engineering and Computer Science

Approved by
editorial and publishing
council of the NTU "KhPI" University,
protocol № 12 dated 29.12.2021.

Kharkiv
NTU "KhPI"
2022

UDC 004.89
ББК 32.973-02
З 19

Reviewers:
V. D. Kovalov, Doctor in Technical Science, Professor, Rector of Donbass State Machine-building Academy, Laureate of the State Prize of Ukraine in the field of science and technology;
G. F. Kryvulia, Doctor in Technical Science, Professor, professor of the Computer Engineering Design Automation department, Kharkiv National University of Radio Electronics.

Наведено теоретичні відомості про пакет MATLAB та способи побудови в ньому нечітких множин, алгоритмів нечіткого виведення й нечіткої кластерізації, нечіткого контролера, генетичних алгоритмів і нейронечітких гібридних мереж. Теоретичний матеріал підкріплений великою кількістю прикладів із використання описаних нечітких систем.

Призначено для студентів денної та заочної форм навчання за напрямом "Комп'ютерна інженерія" й "Комп'ютерні науки".

Іл. 20. Табл. 8. Бібліог. 11 назв.

З 19 **Zakovorotniy O.** Fundamentals of Computational Intelligence / O. Zakovorotniy, O. Lipchanska: Laboratory workshop. – Kharkiv: NTU "KhPI", 2022. – 160 p. – In engl. language.

ISBN 978-617-578-129-6

Theoretical information about the MATLAB package and methods for constructing fuzzy sets, fuzzy inference and fuzzy clustering algorithms, a fuzzy controller, genetic algorithms and neuro-fuzzy hybrid networks in it is given. The theoretical material is supported by a large number of examples on the use of the described fuzzy systems.

Designed for full-time and part-time students in the direction of "Computer Engineering" and "Computer Science".

Pict. 20. Tabl. 8. Referenc. 11 names.

UDC 004.89
ББК 32.973-02

ISBN 978-617-578-129-6

INTRODUCTION

The *Fuzzy Sets Theory* was developed in 1965 when professor Lotfi Zadeh of the University of Berkeley published the fundamental work Fuzzy Sets in the Information and Control journal. The adjective "fuzzy" was included into the title of the new theory in order to distance itself from classical crisp mathematics and Aristotelian logic, which operate with crisp concepts: "belongs – does not belong", "true – false". The concept of a fuzzy set was born by L. Zadeh as dissatisfaction with the mathematical methods of classical systems theory, which forced his to achieve artificial accuracy, inappropriate in many systems of the real world, especially in the so-called humanistic systems that include people"

The beginning of the practical application of fuzzy sets theory can be considered 1975, when Mamdani and Assilian built the first fuzzy controller to control a simple steam engine. In 1982, Holmblad and Ostergad developed the first industrial fuzzy controller, which was implemented in the control of the cement firing process at a plant in Denmark. The success of the first industrial controller based on fuzzy linguistic rules "If – then" led to a surge of interest in fuzzy set theory among mathematicians and engineers. Somewhat later, Bartholomew Kosko proved the *Fuzzy Approximation Theorem*, according to which any mathematical system can be approximated by a system based on fuzzy logic. In other words, with the help of natural language statements-rules "If – then", with their subsequent formalization by means of fuzzy set theory, it is possible to accurately reflect an arbitrary relationship "inputs - output" without using the complex apparatus of differential and integral calculus, traditionally used in management and identification.

Systems based on fuzzy sets have been developed and successfully implemented in such areas as process control, transport management, medical diagnostics, technical diagnostics, financial management, stock forecasting, pattern recognition, etc. The range of applications is very wide - from video cameras and household washing machines to air defense missile guidance and control of combat helicopters. Practical experience in the development of fuzzy logical inference systems shows that the time and cost of their design is much less than when using the traditional mathematical apparatus. This ensures the required level of robustness and transparency of models.

Laboratory work 1

BASIC CALCULATIONS IN MATLAB PACKAGE

The purpose of the laboratory work is to obtain and to consolidate knowledge, to form practical skills in working with the MATLAB package when calculating algebraic expressions using built-in mathematical functions.

1.1. Summary of theory

1.1.1. MATLAB Workspace

MATLAB was created by Math Works over fifteen years ago. The work of hundreds of scientists and programmers is aimed at constantly expanding its capabilities and improving the underlying algorithms. Currently MATLAB is a powerful and versatile tool that allows to solve problems which human experience during their activity.

MATLAB 6.x, MATLAB 7 workspace has a convenient interface for accessing many of the MATLAB auxiliary elements.

When you start MATLAB 6.x, the screen displays the workspace shown in Fig. 1.1.

The Workspace contains the following elements:

– menu;

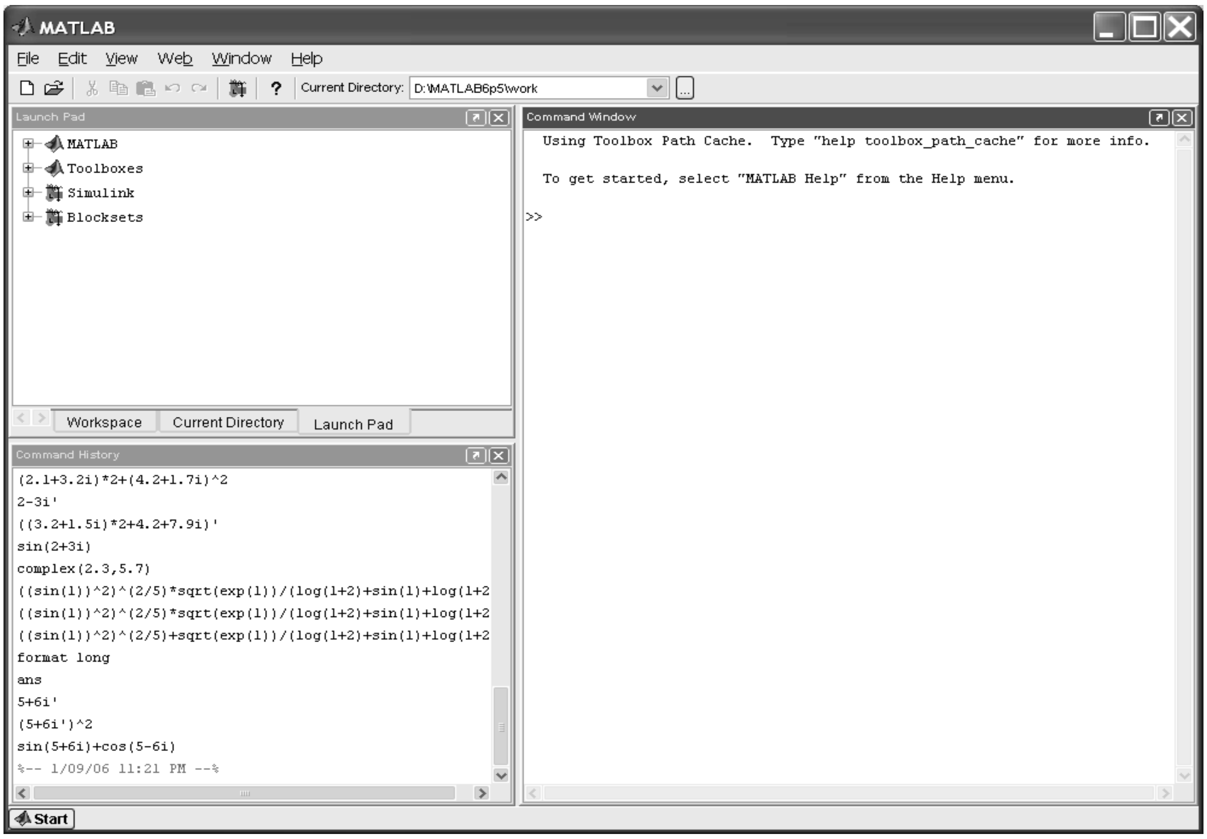
– toolbar with buttons and a drop-down list;

– a panel with Launch Pad and Workspace tabs, from which you can get easy access to various ToolBox modules and the contents of the working environment;

– a panel with the Command History and Current Directory tabs, designed to view and recall previously entered commands, as well as to set the current directory;

– Command Window panel with the command line in which the blinking cursor is located;

– status bar.



```
(2.1+3.2i)*2+(4.2+1.7i)^2
2-3i'
((3.2+1.5i)*2+4.2+7.9i)'
sin(2+3i)
complex(2.3,5.7)
((sin(1))^2)^(2/5)*sqrt(exp(1))/(log(1+2)+sin(1)+log(1+2
((sin(1))^2)^(2/5)*sqrt(exp(1))/(log(1+2)+sin(1)+log(1+2
((sin(1))^2)^(2/5)+sqrt(exp(1))/(log(1+2)+sin(1)+log(1+2
format long
ans
5+6i'
(5+6i')^2
sin(5+6i)+cos(5-6i)
%-- 1/09/06 11:21 PM --%
```

```
  Using Toolbox Path Cache.  Type "help toolbox_path_cache" for more info.

  To get started, select "MATLAB Help" from the Help menu.

>>
```

Fig. 1.1. MATLAB 6.x workspace

All commands described in this lab should be typed at the command line. Symbol » means command line prompt. You do not need to type the character that denotes in the examples. To view the work area, it is convenient to use the scroll bars or the <*Home*>, <*End*> keys to move left or right and <*PageUp*>, <*PageDown*> to move up or down. <*Up*>, <*down*>, <*rigth*>, <*left*> keys using will be discussed additionally. If the command line with the blinking cursor disappeared after moving around the workspace of the Command Window, just press <*Enter*>.

It is important to remember that any command or expression should end with the <*Enter*> key in order for MATLAB to execute that command or evaluate the expression.

Comment 1

If some of the described panels are missing in the MATLAB 6.x workspace, then select the appropriate items from the *View* menu: *Command Window, Command History, Current Directory, Workspace, Launch Pad.*

1.1.2. Arithmetic calculations

MATLAB's built-in math functions let you find the values of various expressions. MATLAB provides the ability to control the result output format. Commands for evaluating expressions have a form common to all high-level programming languages.

Type 1+2 at the command line and press <*Enter*>. As a result, the MATLAB Command Window displays the following:

```
» 1+2
ans =
     3
```

What did MATLAB do? First, it calculated the sum 1 + 2, then wrote the result to the special variable *ans* and printed its value, equal to 3, to the command window. A command line with a blinking cursor is below the answer. It indicates that MATLAB is ready for further computation. You can type new expressions at the command line and find their values.

If you want to continue working with the previous expression, for example, calculate (1+2) / 4.5, then the easiest way is to use the already existing result, which is stored in the *ans* variable. Type ans / 4.5 in the command line (a point is used when entering decimal fractions) and press <*Enter*>, it turns out:

```
» ans/4.5
ans =
    0.6667
```

Comment 2

The form of the calculations result is displayed depends on the output format set in MATLAB. The following explains how to set the basic output formats.

1.1.3. Calculation result output formats

The required output format is defined by the user via the MATLAB menu. Select *Preferences* from the *File* menu. The *Preferences* dialog box appears. To set the output format, make sure the *Command Window* item is selected in the left panel list. The format is set from the *Numeric format* drop-down list of the *Text display* panel.

We will analyze only the most frequently used formats. Select short from the *Numeric format* drop-down list in MATLAB 6.x. Close the dialog by clicking the <*ok*> button. The *short* floating point format short is now used for displaying the results of calculations, in which only four digits after the decimal point are displayed on the screen. Type 100/3 at the command line and press <*Enter*>.

The result is output in the *short* format:

```
» 100/3
ans = 33.3333
```

This output format is retained for all subsequent calculations, unless a different format is specified. Note there is a possible situation in MATLAB when displaying too large or small numbers, the result does not fit into *short* format. Calculate 100000/3, the result is displayed in exponential form:

```
» 100000/3
```

```
ans =
     3.3333e+004
```

The same form will happen when finding 1/3000:

```
» 1/3000
ans =
     3.3333e–004
```

However, the original setting of the format is preserved during further calculations. The result output will again be in *short* format for small numbers.

In the previous example, the MATLAB package displayed the calculation result in exponential form. The entry 3.3333*e*–004 means $3.3333*10^{-4}$ or 0.00033333. Similarly, you can type numbers in expressions. For example, it is easier to type 10*e*9 or l.0*e*10 than 1,000,000,000, and the result will be the same. A gap between digits and *e* symbol is not allowed when entering, because this will result in an error message:

```
» 10 e9
??? 10 e9
Missing operator, comma, or semi-colon.
```

If you want to get the result of calculations more accurately, then you should select long from the drop-down list. The result will be displayed in long floating point format with fourteen digits after the decimal point. The short e and long e formats are designed to output the result in exponential form with four and fifteen digits after the decimal point, respectively. Information about formats can be obtained by typing the command help with the argument format at the command line:

```
» help format
```

A description of each format appears in the command window.

You can set the output format directly from the command line using the *format* command. For example, to set a long floating point format for outputting the results of calculations, enter the *format long e* command at the command line:

```
» format long e
» 1.25/3.11
ans =
    4.019292604501608e-001
```

Note that the *help format* command displays the format names in capital letters. However, the command to be entered is in lowercase letters. This feature of the built-in *help* takes some getting used to. MATLAB distinguishes between uppercase and lowercase letters. Attempting to type the command in capital letters will result in an error:

```
» FORMAT LONG E
??? FORMAT LONG.
Missing operator, comma, or semi-colon.
```

For better readability of the result, MATLAB displays the result of the calculations missing a blank line after the evaluated expression. However, sometimes it is convenient to place more lines on the screen by selecting the *compact* (*File, Numeric display*) button from the drop-down list. Adding blank lines is provided by selecting *loose* from the *Numeric display* drop-down list.

Comment 3

MATLAB performs all intermediate calculations in double precision, regardless of the set output format.

1.1.4. Elementary functions using

Suppose you want to evaluate the following expression:

$$e^{-2{,}5} \cdot (\ln 11{,}3)^{0{,}3} - \sqrt{(\sin 2{,}45\pi + \cos 3{,}78\pi)/(\mathrm{tg}\, 3{,}3)}\,.$$

Enter this expression on the command line according to MATLAB rules and press <*Enter*>:

```
» exp(–2.5)*log(11.3)^0.3–
sqrt((sin(2.45*pi)+cos(3.78*pi)}/tan(3.3))
```

The answer is displayed in the command window:

```
ans =
    -3.2105
```

When entering the expression, the built-in MATLAB functions are used to calculate the exponent, natural logarithm, square root, and trigonometric functions. What integrated elementary functions can be used and how to call it? Type the command help eifun in the command line, and a list of all built-in elementary functions with their brief description is displayed in the command window. Function arguments are enclosed in round brackets; function names are in lowercase. To enter π, just type *pi* at the command line.

Arithmetic operations in MATLAB are performed in the usual order common to most programming languages:

– exponentiation ^;

– multiplication and division *, /;

– addition and subtraction +, –.

Use round brackets to change the order of arithmetic operators execution.

If now you want to calculate the value of an expression similar to the previous one, for example

$$e^{-2,5}\cdot(\ln 11{,}3)^{0,3}-((\sin 2{,}45\pi+\cos 3{,}78\pi)/(\mathrm{tg}\,3{,}3))^2,$$

it is not necessary to type it again on the command line. You can take advantage of the fact that MATLAB remembers all the commands you enter. To re-enter them into the command line, use the <*up*>, <*down*> keys. Evaluate the given expression by following these steps:

1. Press the <↑>, key, and the previously entered expression will appear in the command line.

2. Make the necessary changes to it, replacing the minus sign with plus and the square root for squaring (use the <*right*>, <*left*>, <*Home*>, <*End*> keys to move along the line with the expression).

3. Calculate the modified expression by pressing <*Enter*>.

It turns out

```
» exp(-2.5)*log(11.3)^0.3+
((sin(2.45*pi)+cos(3.78*pi))/tan(3.3))^2
ans =
     121.2446
```

If you need to get a more accurate result, then you should execute the *format long e* command, then press the <↑> key until the required expression appears in the command line, and calculate it by pressing <*Enter*>

```
» format long e
» exp(-2.5)*log(11.3)^0.3+
((sin.(2.45*pi)+cos(3.78*pi))/tan(3.3))^2
ans =
     1.212446016556763e+002
```

You can display the result of the last calculated expression in a different format without recalculation. Change the format with the *short* command, and then see the value of the *ans* variable by typing it in the command line and pressing <*Enter*>:

```
» format short
» ans
ans =
     121.2446
```

There is a convenient tool for invoking previously entered commands such as the *Command History* panel in the MATLAB 6.x working area. It has the history of commands. The command history contains the time and date of each MATLAB 6.x session. To activate the *Command History* window, select the tab with the same name. The current command in the window is shown with a blue background. If you click on any command in the window with the left mouse button, then this command becomes the current one. To execute it in MATLAB, you need to double-click or select a line with a command using the <*up*>, <*down*> keys and press the <*Enter*> key. An extra command can be removed from the window. To do this, make it current and delete it using the <*Delete*> key. You can select several consecutive commands using the key combination <*Shift*> + <*up*>, <*Shift*> + <*down*> and execute them using <*Enter*> or delete with the <*Delete*> key. Sequential commands can be selected with the left mouse button while holding down the <*Shift*> key. If the commands do not follow one after another, then to select them, use the left mouse button while holding down the <*Ctrl*> key.

When you right-click in the *Command History* area, a pop-up menu appears. Selecting the *Copy* item will copy the command to the Windows clipboard. With *Evaluate Selection* you can execute the marked group of commands. Use the *Delete Selection* item to delete the current command. Use the *Delete to Selection* item to delete all commands up to the current one, use the *Delete Entire History* item to delete all commands.

Some exceptional situations are possible in calculations, for example division by zero, which lead to an error in most programming languages. When a positive number is divided by zero in MATLAB, *inf* (infinity) is obtained, and when a negative number is divided by zero, *–inf* (minus infinity) is obtained and a warning is issued:

```
» 1/0
Warning: Divide by zero.
ans = Inf
```

Dividing zero by zero results in *NaN* (not a number) and also generates a warning:

```
» 0/0
Warning: Divide by zero.
ans = NaN
```

When calculating, for example *sqrt*(–1), no error or warning is raised. MATLAB automatically switches to complex numbers:

```
»sqrt(–1.0)
```

```
ans = 0 + 1.0000i
```

1.1.5. Working with complex numbers

When typing complex numbers in the MATLAB command line, you can use either *i* or *j*. The numbers must be enclosed in round brackets when multiplying, dividing and exponentiating:

```
»(2.1+3.2i)*2+(4.2+1.7i)^2
ans =
     18.9500 + 20.6800i
```

If you do not use round brackets, then only the imaginary part will be multiplied or exponentiated, and the result will be incorrect:

```
» 2.1+3.2i*2+4.2+1.7i^2
ans =
     3.4100 + 6.4000i
```

To calculate the complex conjugate number, an apostrophe is used, which should be typed immediately after the number, without a gap:

```
» 2–3i'
ans =
     2.0000 + 3.0000i
```

If you need to calculate a complex conjugate expression, then the source expression must be enclosed in round brackets:

```
»((3.2+1.5i)*2+4.2+7.9i)'
ans =
     10.6000 – 10.9000i
```

MATLAB allows complex numbers to be used as arguments to built-in elementary functions:

```
» sin(2+3i)
ans =
     9.1545 - 4.1689i
```

The construction of a complex number from its real and imaginary parts is performed using the *complex* function:

```
» complex(2.3, 5.8)
ans =
     2.3000 + 5.8000i
```

How do you know which built-in elementary functions can be used and how to call them? Type the command *help eifun* in the command line, and a list of all built-in elementary functions with a short description is displayed in the command window.

1.2. Individual tasks

1. According the number *N* in the group register list, written in the form *N* = *CM*, where *C* is the rank of tens, *M* is the units rank, calculate the expression given using the table 1.1 and table. 1.2.

2. Replace the sign of the arithmetic operation of multiplication by the sign of the addition operation in the expressions of table.1.1 and table 1.2 and calculate new expressions without fully typing them on the command line, taking advantage of the fact that MATLAB remembers the entered commands.

3. Get the results of calculations of point 2 of the individual task in the *short* and *long* formats.

4. Get a complex number ($a + bi$), where a, b are respectively, the number of letters in your first and last name. Also define:

– complex conjugate number to number ($a + bi$);

– calculate the square of the complex conjugate number;

– calculate the product of the source complex number and the complex conjugate number;

– calculate the expression $\sin(a + bi) + \cos(a - bi)$.

5. Design a laboratory report.

Table 1.1 – Individual task for the units rank M of the number in group register

The units rank M of the number in group register	0 or 5	1 or 6	2 or 7	3 or 8	4 or 9
Expression	$\frac{A^{2/5}\cdot\sqrt{C}}{B+D}$	$\frac{\sqrt{A^{-2}}\cdot B^{1{,}7}}{D+C\cdot A}$	$\frac{A\cdot\sqrt{B+C}}{D^{1/5}/C}$	$\frac{C^{3/7}+\sqrt{D}}{B\cdot\sqrt[3]{A^5}}$	$\frac{A\cdot\sqrt[5]{D}}{C+B^4}$

Table 1.2 – Individual task for the rank of tens C of the number in group register

The rank of tens C of the number in group register	A	B	C	D
0	$(\sin(N))^2$	$\ln(N+2)$	$\exp(N/N^2)$	$A+B$
1	$(\cos(N))^4$	$(\ln(N^4))^2$	$\exp(-N)+1$	$A+C$
2	$(\text{tg}(N))^2$	$(\log(N))^4$	$1/\exp(N)$	B/C
3	$(\text{ctg}(N))^4$	$\log(N+N^3+9)$	$N/\exp(-N)$	$C/(A+B)$

Laboratory work 2

BASIC CALCULATIONS IN MATLAB PACKAGE USING VARIABLES AND VECTORS

The purpose of the laboratory work is to obtain and to consolidate knowledge, to form practical skills in working with the MATLAB package when calculating using variables and vectors.

2.1. Summary of theory

2.1.1. Using Variables in MATLAB

MATLAB provides the ability to work with variables as all programming languages do. Moreover, the user does not have to worry about what values the variable will take (complex, real, or only integers). In order to assign, for example, the value 1.45 to the variable *z*, it is enough to write *z* = 1,45 on the command line. MATLAB will immediately display the value of *z*:

```
» z = 1.45
z =
    1.4500
```

The equals sign is used as an assignment operator. It is often not very convenient to get the result after each assignment. Therefore, MATLAB provides the ability to terminate an assignment statement with a semicolon to suppress the output to the command window. Variable name can be any sequence of letters and numbers with no spaces, starting with a letter. Lowercase and uppercase letters are different, for example, *MZ* and *mz* are two different variables. The number of characters MATLAB can accept in a variable name is 31.

As an exercise in using variables, calculate the value of the following expression:

$$\left(\sin 1{,}3\pi/\ln 3{,}4+\sqrt{\text{tg}\,2{,}75/\text{th}\,2{,}75}\right)/\left(\sin 1{,}3\pi/\ln 3{,}4-\sqrt{\text{tg}\,2{,}75/\text{th}\,2{,}75}\right).$$

Type the following sequence of commands (note the semicolon in the first two assignment operators to suppress the output of intermediate values on the screen):

```
» x = sin(1.3*pi)/log(3.4);
» y = sqrt(tan(2.75)/tanh(2.75));
» z = (x+y)/(x-y)
Z =
    0.0243 - 0.9997i
```

The last assignment operator is not terminated with a semicolon in order to get the value of the initial expression immediately.

Of course, you can enter the whole formula at once and get the same result:

```
»(sin(1.3*pi)/log(3.4)+sqrt(tan(2.75)/tanh(2.75)))/…
(sin(1.3*pi)/log(3.4)-sqrt(tan(2.75)/tanh(2.75)))
ans =
    0.0243 - 0.9997i
```

Please note that the first entry is more compact and clearer than the second! In the second version, the formula did not fit in the command window on one line. It had to write it down in two lines. For this, three dots are put at the end of the first line.

Comment 1

To enter long formulas or commands in the command line, put three dots (in a row, without spaces), press the <*Enter*> key and continue typing the formula on the next line. This way you can place an expression on multiple lines. MATLAB will calculate the entire expression or execute a command after pressing <*Enter*> on the last line (which does not have three consecutive dots).

MATLAB remembers the values of all variables defined during a session. If, after entering the example above, some other calculations were done, and it became necessary to display the value of *x*, then you should simply type *x* in the command line and press <*Enter*>:

```
» x
     -0.6611
```

The variables defined above can be used in other formulas as well. For example, if now you need to calculate the expression

$$\left(\sin 1{,}3\pi/\ln 3{,}4+\sqrt{\mathrm{tg}\, 2{,}75/\mathrm{th}\, 2{,}75}\right)^{3/2},$$

then just enter the following command:

```
» (x-y)^(3/2)
ans =
     -0.8139 + 0.3547i
```

Calling MATLAB functions is flexible enough. For example, you can calculate $e^{3{,}5}$ by calling the *exp* function from the command line:

```
» exp(3.5)
ans =
     33.1155
```

Another way is to use the assignment operator:

```
» t = exp(3.5)
t =
    33.1155
```

Suppose that some of the calculations with variables are done. The rest of the calculations have to be completed during the next session of MATLAB. In this case, you will need to save the variables defined in the working space.

2.1.2. Workspace saving

The easiest way to save the values of all variables is to use the *Save Workspace As…* item in the *File* menu. In this case, the *Save…* dialog box appears, in which you should specify the directory and file name. By default, it is suggested to save the file in the *work* subdirectory of the main MATLAB directory. Leave this directory for now.

How to set directory paths for file search in MATLAB will be explained further. It is convenient to give files names containing the date of work, for example *work20-06-12*. MATLAB will save the results to a file *work20-06-12.mat*. You can close MATLAB now in one of the following ways:

– select *Exit MATLAB* from the *File* menu;

– press the keys <*Ctrl*>+<*Q*>;

– type the *Exit* command in the command line and press <*Enter*>;

– click on the button with a cross in the upper right corner of the MATLAB program window.

In the next session, to restore the values of variables, open the *work20-06-12.mat* file using the *Open* item of the *File* menu. Now all the variables defined in the previous session are available. They can be used in newly entered commands.

You can also save and restore working space variables from the command line. Use *save* and *load* commands. At the end of the MATLAB session, it is necessary to execute the command

```
» save work20-06-12
```

The extension can be omitted, MATLAB will save the working space variables in the *work20-06-12.mat* file. To read variables at the beginning of the next session, enter the command

```
» load work20-06-12
```

To get detailed information about *save* and *load* commands type *help save* or *help load* at the command line.

Comment 2

Variables in files with the *mat* extension are stored in binary form. Viewing these files in any text editor do not provide any information about the variables and their values.

MATLAB has the ability to write executable commands and results to a text file (keep a log of work), which can be easily read or printed from a text editor then. The *diary* command is used to start logging. The argument to the *diary* command should be the name of the file that stores the job log. The commands typed further and the results of their execution will be written to this file, for example, a sequence of commands

```
» diary d20-06-12.txt
» a1 = 3;
» a2 = 2.5;
» a3 = a1 + a2
» a3 =
     5.5000
» save work20-06-12
» quit
```

performs the following actions:

- Opens the *d20-06-12.txt* file.
- Performs calculations.
- Saves the variables in the binary file *work.20-06-12.mat*.
- Saves the work log to the *d20-06-12.txt* file in the *work* subdirectory of the MATLAB root directory and closes MATLAB.

Look at the contents of the *d20-06-12.txt* file using any text editor, for example, standard Windows *NotePad* program. The file contains the following text:

```
a1 = 3;
a2 = 2.5;
a3 = a1+a2
a3 =
     5.5000
save work20-02-06
quit
```

Start MATLAB again and enter the command *load work20-06-12* or open the *work20-06-12.mat* file using the menu as described above.

2.1.3. Variables View

Working with a large number of variables, it is necessary to know which variables have already been used and which have not. For this purpose, the *who* command is used. It displays a list of used variables in the MATLAB command window:

```
» who
Your variables are:
a1   a2   a3
```

Whos command allows to get more detailed information about variables in the table form:

```
 »
Name   Size   Bytes  Class
a1     1x1    8      double
a2     1x1    8      double
a3     1x1    8      double
Grand total is 3 elements using 24 bytes
```

The first column *Name* consists of the names of used variables. The content of the *Size* column is determined by the basic principle of MATLAB. The MATLAB program presents all data as arrays. The variables *al*, *a2* and *a3* are one-by-one two-dimensional arrays. Each of the variables is eight bytes, as indicated in the *Bytes* column. Finally, the last *Class* column indicates the *double array* type of the variables, i.e. an array of double precision numbers. The line below the table says that there are three elements, i.e. variables use twenty four bytes. It turns out that representing all data in MATLAB as arrays has certain advantages.

Clear command is used to free all variables from memory. If the list of variables is specified in the arguments (separated by a space), then only they will be freed from memory.

For example:

```
» clear a1 a3
» who
Your variables are:
a2
```

Starting with version 6.0, a convenient tool for viewing the variables of the working space has appeared, the *Workspace* panel. It is necessary to activate the tab of the same name in order to switch to it. This panel contains a table similar to the one displayed by the *whos* command. Double-clicking on the row corresponding to each variable displays its contents in a separate window. It is especially useful when working with arrays. The *Workspace* panel toolbar allows to remove unnecessary variables, save and open the workspace.

2.1.4. Working with arrays

It is very important to understand correctly how to use arrays. This provide the effective work in MATLAB in particular, graphing, solving problems in linear algebra, data processing, statistics and many others. This subsection describes calculations with vectors.

An array is an ordered, numbered collection of homogeneous data. The array has a name. Arrays differ in the number of dimensions: one-dimensional, two-dimensional, multi-dimensional. Items are accessed using an index. In MATLAB, the numbering of array elements starts at one. This means that the indices must be greater than one or equal to one.

It is important to understand that a vector, row vector, or matrix are mathematical objects, and one-dimensional, two-dimensional, or multidimensional arrays are ways of storing these objects in a computer. Further, the words vector and matrix will be used if the object is of more interest than the way it is stored. The vector can be written in column (column vector) and in row (row vector). Column vectors and row vectors will often be referred to simply as vectors. A distinction will be made in cases where the way the vector is stored in MATLAB is important. Vectors and matrices are denoted in italics, and the corresponding arrays are denoted in regular monospaced font, for example: "vector *a* is contained in array a", "write matrix *R* into array R".

Input, addition and subtraction of vectors

Let's start working with arrays with a simple example of calculating the sum of vectors:

$$a = \begin{pmatrix} 1{,}3 \\ 5{,}4 \\ 6{,}9 \end{pmatrix}, \; b = \begin{pmatrix} 7{,}1 \\ 3{,}5 \\ 8{,}2 \end{pmatrix}.$$

Use arrays *a* and *b* to store vectors. Enter array *a* on the command line using square brackets and separating the vector elements with semicolons:

```
» a = [1.3; 5.4; 6.9]
a =
    1.3000
    5.4000
    6.9000
```

Since the entered expression is not terminated with a semicolon, the MATLAB package automatically printed the value of the variable *a*. Enter now the second vector

```
» b = [7.1; 3.5; 8.2];
```

The + sign is used to find the sum of vectors. Calculate the sum, write the result to array *c* and display its elements in the command window:

```
» c = a + b
```

```
c =
    8.4000
    8.9000
    15.1000
```

Find out the dimension and size of the array *a* using the built-in functions *ndims* and *size*:

```
» ndims(a)
ans =
    2
» size(a)
ans = 3  1
```

So, vector *a* is stored in a three-by-one two-dimensional array *a* (a column vector of three rows and one column). Similar operations can be done for arrays *b* and *c*. Since numbers in the MATLAB package are represented as one-by-one two-dimensional array, the vector addition uses the same plus sign as the number addition.

Entering a row vector is realized within square brackets. Items are separated with spaces or commas. Operations of addition, subtraction and calculation of elementary functions from row vectors are performed in the same way as with column vectors. As a result, row vectors of the same size as the initial ones are obtained. For example:

```
» s1 = [3 4 9 2]
s1 =
    3 4 9 2
» s2 = [5 3 3 2]
s1 =
    5 3 3 2
```

```
» s3 = s1 + s2
s3 =
    8 7 12 4
```

Comment 3

If addition or subtraction is applied to vectors whose dimensions do not match, an error message is issued.

Naturally, the minus sign should be used to calculate the difference between vectors, and it is more difficult to find the product.

Enter two row vectors:

```
» v1 = [2 -3 4 1];
» v2 = [7 5 -6 9];
```

The .* operation (do not insert a space between dot and asterisk!) results in elementwise multiplication of vectors of the same length. The result is a vector with elements equal to the product of the corresponding elements of the original vectors:

```
» u = v1.*v2
u =
    14  -15  -24   9
```

.^ is used to perform elementwise exponentiation:

```
» p = v1.^2
p =
     4   9   16   1
```

The exponent can be a vector of the same length as the one exponentiated. In this case, each element of the first vector is raised to a power equal to the corresponding element of the second vector:

```
» p = v1.^v2
P =
     128.0000 -243.0000  0.0002  1.0000
```

Division of the corresponding elements of vectors of the same length is performed using the operation ./

```
» d = v1./v2
d =
     0.2857 -0.6000 -0.6667  0.1111
```

Reverse element-wise division (dividing the elements of the second vector by the corresponding elements of the first one) is carried out using the operation .\

```
» dinv = v1.\v2
dinv =
     3.5000 -1.6667 -1.5000  9.0000
```

So, the dot in MATLAB is used not only to enter decimal fractions, but also to indicate that division or multiplication of arrays of the same size should be performed element by element.

Operations with a vector and a number are also considered element-wise. Adding a vector and a number does not generate an error message. MATLAB adds a number to each element of the vector. The same is true for subtraction:

```
» v = [4 6 8 10];
» s = v + 1.2
s =
    5.2000 6.2000 9.2000 11.2000
» r = 1.2 - v
r =
    -2.8000 -4.8000 -6.8000 -8.8000
» r1 = v - 1.2
r1 =
    2.8000 4.8000 6.8000 8.8000
```

You can multiply a vector by a number on both the right and left:

```
» v = [4 6 8 10];
» p = v*2
p =.
    8  12  16  20
» pi = 2*v
pi =
    8  12  16  20
```

It is possible to divide a vector by a number using the / sign:

```
» p = v/2
p =
         2   3   4   5
```

Attempting to divide a number by a vector results in an error message:

```
» p = 2/v
??? Error using ==> /
Matrix dimensions must agree.
```

If you want to divide a number by each element of the vector and write the result into a new vector, then you should use the operation ./

```
» w = [4 2 6];
» d = 12./w
d = 3 6 2
```

All of the above operations apply to both row vectors and column vectors.

The MATLAB feature of representing all data as arrays is very convenient. Suppose, for example, you want to calculate the value of the *sin* function at once for all elements of the vector *c* (which is stored in the array *c*) and write the result to the vector *d*. To obtain a vector *d*, it is enough to use one assignment operator:

```
» d = sin(c)
d =
    0.8546
    0.5010
    0.5712
```

So, the atomic functions built into MATLAB adapt to the kind of arguments; if the argument is an array, then the result of the function will be an array of the same size, but with elements equal to the function value from the corresponding elements of the initial array. Check this out with another example. If you need to find the square root of the elements of the vector *d* with a minus sign, then it is enough to write:

```
» sqrt(-d)
ans =
     0 + 0.9244i
     0 + 0.7078i
     0 + 0.7558i
```

No assignment operator was used, so the MATLAB package wrote the answer to a standard *ans* variable. To determine the length of column vectors or row vectors, use the built-in *length* function:

```
» length(s1)
ans =
     4
```

Multiple column vectors can be compiled into a single column vector by using square brackets and separating the initial column vectors with a semicolon:

```
» v1 = [1; 2];
» v2 = [3; 4; 5];
» v = [v1; v2]
v =
     1
     2
     3
     4
     5
```

Square brackets are also used to concatenate row vectors, but the concatenated row vectors are separated by spaces or commas:

```
» v1 = [1 2];
» v2 = [3 4 5];
» v = [v1 v2]
v =
    1 2 3 4 5
```

Working with vector elements

The elements of a column vector or row vector are accessed using an index. It is enclosed in parentheses after the name of the array in which the vector is stored. If among the environment variables there is an array *v* defined by a row vector

```
» v = [1.3 3.6 7.4 8.2 0.9];
```

then to display, for example, its fourth element, indexing is used:

```
» v(4)
ans =
    8.2000
```

The appearance of an array element on the left side of the assignment operator results in a change in the array

```
» v(2) = 555
v =
    1.3000 555.0000 7.4000 8.2000 0.9000
```

New arrays can be formed from array elements, for example

```
» u = [v(3); v(2); v(1)]
u =
    7.4000
    555.0000
    1.3000
```

Indexing with a vector is used to place certain elements of a vector into another vector in a given order. The fourth, second, and fifth elements of *v* are written to the array *w* as follows:

```
» ind = [4 2 5];
» w = v(ind)
w =
    8.2000 555.0000 0.9000
```

MATLAB provides a convenient way to refer to blocks of consecutive elements of a column vector or row vector. This is done by indexing with a colon. Suppose it is needed to replace with zeros the elements from the second to the sixth in the array *w* corresponding to a row vector of seven elements. Colon indexing allows you solve the task to easily and clearly:

```
» w = [0.1 2.9 3.3 5.1 2.6 7.1 9.8];
» w(2:6) = 0;
» w
w =
    0.1000 0 0 0 0 0 9.8000
```

Assignment *w*(2:6) = 0 is equivalent to a sequence of commands
w(2) = 0; *w*(3) = 0; *w*(4) = 0; *w*(5) = 0; *w*(6) = 0.

Colon indexing is useful when extracting part of a large amount of data into a new array:

```
» w - [0.1 2.9 3.3 5.1 2.6 7.1 9.8];
» wl = w(3:5)
wl =
     3.3000  5.1000  2.6000
```

Make an array *w2* containing *w* elements other than the fourth. In this case, it is convenient to use colon and string concatenation.:

```
» w2 = [w(l:3) w(5:7)]
w2 =
     0.1000 2.9000 3.3000 2.6000 7.1000 9.8000
```

Array elements can be included in expressions. Finding, for example, the geometric mean of the elements of the array *u* can be done as follows:

```
» gm = (u(l)*u(2)*u(3))^(l/3)
gm =
     17.4779
```

Of course, this method is not very convenient for long arrays. In order to find the geometric mean, it is necessary to type all the elements of the array in the formula. There are many special functions in MATLAB that facilitate such calculations.

Applying data processing functions to vectors

Multiplying elements of a column vector or row vector is performed using the function *prod*:

```
» z = [3; 2; 1; 4; 6; 5];
» p = prod(z)
p =
    720
```

The *sum* function is for summing the elements of a vector. It is easy to calculate the arithmetic mean of the elements of the vector *z*:

```
» sum(z)/length(z)
ans =
    3.5000
```

MATLAB also has a special function *mean* for calculating the arithmetic mean:

```
» mean(z)
ans =
    3.5000
```

To determine the minimum and maximum of the vector elements, use the built-in functions *min* and *max*:

```
» m1 = max(z)
m1 =
    6
» m2 = min(z)
m2 =
    1
```

It is often necessary to know not only the value of the minimum or maximum element in an array, but also its index (ordinal number). In this case, the built-in functions *min* and *max* are used with two output arguments, for example

```
» [m, k] = min(z)
m =
    1
k =
    3
```

As a result, the variable *m* will be assigned the value of the minimum element of the array *z*, and the number of the minimum element will be entered into the variable *k*.

For information on the different ways to use functions, type *help* and the function name at the command line. MATLAB will display all sorts of ways to call the function in the command window with additional explanations.

The main functions for working with vectors include the function of ordering a vector in ascending order of its elements *sort*

```
» r = [9.4 -2.3 -5.2 7.1 0.8 1.3];
» R = sort(r)
R =
    -5.2000 -2.3000 0.8000 1.3000 7.1000 9.4000
```

It is possible to order the vector in descending order using the same function *sort*:

```
» R1 = -sort(-r)
R1 =
    9.4000 7.1000 1.3000 0.8000 -2.3000 -5.2000
```

The ordering of elements in ascending order of their modules is performed using the function *abs*:

```
» R2 = sort(abs(r))
R2 =
    0.8000 1.3000 2.3000 5.2000 7.1000 9.4000
```

Calling *sort* with two output arguments results in an array of indices that match the elements of the ordered and initial arrays:

```
» [rs, ind] = sort(r)
rs =
    -5.2000 -2.3000 0.8000 1.3000 7.1000 9.4000
ind =
    3  2  5  6  4  1
```

2.2. Individual tasks

1. Create a log of the lab work.
2. Calculate function values

$$y(x) = \frac{\sin^2 x}{1+\cos x} + e^{-x} \ln x$$

in points 0.2, 0.3, 0.5, 0.8, 1.3, 1.7, 2.5, *N*, *k*, where *N* is the number in group register list; *k* is the numerical value of the expression given in the table 1.1 and table 1.2 in accordance with the number *N*, written in the form *N* = *CM*, where *C* is the rank of hundreds, *M* is the units rank.

3. Form a row vector *v* containing all the values of the *x* argument and the last five values of the function $y(x)$.

4. Get the row vector *v1* by adding 2.1 to each element of the row vector *v*.

5. Calculate: $w = v + v1$; $w1 = v - v1$; $w2 = v1 - v$; $w3 = v$ ./ $v1$; $w4 = v1$.*v; $w5 = v1$.^v.

6. Order the results of addition of vectors *v* + *v1* in ascending order of absolute values of the elements of the sum vector, increasing elements of the sum vector, decreasing elements of the sum vector.

7. Form the third and fifth elements into row vectors w, $w1$, $w2$, $w3$, $w4$, $w5$ column vector *ww*.

8. Determine in the *ww* vector the minimum and maximum element, the sum and product of the vector components.

9. Set the third through sixth elements of the *ww* array to one.

10. Use the *who* and *whos* commands to get information about all variables used in the lab.

11. Record the first and last few lines from the lab log.

12. Design a laboratory report.

Laboratory work 3

BASIC CALCULATIONS IN MATLAB PACKAGE WITH THE USE OF MATRICES

The purpose of the laboratory work is to obtain and to consolidate knowledge, to form practical skills in working with the MATLAB package when calculating using variables, vectors and matrices.

3.1. Summary of theory

3.1.1. Different ways to enter matrices in MATLAB

It is convenient to enter small-sized matrices directly from the command line. Enter a two-by-three matrix

$$A = \begin{pmatrix} 3 & 1 & -1 \\ 2 & 4 & 3 \end{pmatrix}.$$

To store the matrix, use a two-dimensional array named *A*. When entering, keep in mind that matrix *A* can be viewed as a column vector of two elements. Each of these elements is a row vector of length three, therefore, rows are separated by semicolons when typing:

```
» A =[3 1 -1; 2 4 3]
A =
     3   1   -1
     2     4   3
```

To study the basic operations on matrices, we will give a few more examples. Let's consider other input ways. Enter a square matrix of size *three* as described below:

$$B = \begin{pmatrix} 4 & 3 & -1 \\ 2 & 7 & 0 \\ -5 & 1 & 2 \end{pmatrix}.$$

Start typing at the command line

```
» B = [4 3 -1
```

Press the <*Enter*> key. Note that the package did not compute anything. The cursor blinks on the next line without the » symbol. Continue entering the matrix line by line. At the end of each line, press <*Enter*>. End the last line with a closing square bracket, it turns out:

```
2 7 0
-5 1 2]
B = 4 3 -1
     2 7 0
    -5 1 2
```

Another way to enter matrices is to interpret a matrix as a row vector, each element of which is a column vector. For example, a two-by-three matrix

$$C = \begin{pmatrix} 3 & -1 & 7 \\ 4 & 2 & 0 \end{pmatrix}$$

can be entered using the command:

```
» C = [[3; 4] [-1; 2] [7; 0]]
C =
     3 -1  7
     4  2  0
```

Look at the working environment variables by typing *whos* in the command line:

```
A     2x3          48 double array
B     3x3          72 double array
C     2x3          48 double array
```

So, the working environment contains three matrices, two of them are rectangular and one matrix is square.

3.1.2. Accessing matrices elements in the MATLAB package

Matrix elements are accessed using two indices such as row and column numbers, enclosed in round brackets, for example

```
» C(2, 3)
ans =
     0
```

Matrix elements can be included in expressions:

```
» C(1, 1) + C(2, 2) + C(2, 3)
ans = 5
```

The arrangement of the matrix elements in the computer memory determines another way of accessing them. An *m-by-n* matrix *A* is stored as a vector of length *mn*, in which the elements of the matrix are arranged one after the other in columns

```
[A(1,1) A(2,1)...A(m,1)...A(1,n) A(2,n)...A(m,n)].
```

To access the elements of the matrix, you can use one index, which specifies the ordinal number of the matrix element in the vector.

The matrix C defined in the previous subsection is contained in the vector

```
[C(1,1) C(2,1) C(1,2) C(2,2) C(1,3) C(2,3)],
```

which has six components. Matrix elements are accessed as follows:

```
» C(1)
ans =
     3
» C(5)
ans =
     7
```

3.1.3. Matrices operations in the MATLAB package: addition, subtraction, multiplication, transposition and exponentiation

When using matrices operations, remember that matrices must be of the same size for addition or subtraction. And when using multiplying, the number of columns in the first matrix must be equal to the number of rows in the second matrix.

Addition and subtraction of matrices, as well as numbers and vectors, is carried out using the plus and minus signs. Find the sum and difference of the matrices *C* and *A* defined above:

```
» S = A+C
S =
    6  0  6
    6  6  3
» R = C-A
R =
    0  -2  8
    2  -2  -3
```

Make sure the dimension matches, otherwise an error message will be received:

```
» S = A+B
??? Error using ==> ±
Matrix dimensions must agree.
```

The asterisk is used for matrix multiplication:

```
» P = C*B

P =
    -25  9  11
    20  26  -4
```

Multiplication of a matrix by a number is also carried out using an asterisk. You can multiply by a number both to the right and to the left:

```
» P = A*3
P =
    9  3 -3
    6 12 -3
» P = 3*A
P =
    9  3 -3
    6 12  9
```

Transposition of a matrix, like a vector, is done with .′. And the ′ symbol means complex conjugation. For real matrices, these operations lead to the same results:

```
» B'
ans =
     4  2 -5
     3  7  1
    -1  0  2
» B.'
ans =
     4  2 -5
     3  7  1
    -1  0  2
```

Comment 1

If the matrix $A \equiv (a_{ik}),\ i = \overline{1,n},\ k = \overline{1,m}$ is an arbitrary matrix of size $n \times m$, then the matrix transposed with respect to A is a matrix of size $m \times n$: $A' \equiv (a_{ki})$, $k = \overline{1,m},\ i = \overline{1,n}$. Thus, the rows of the matrix A become columns of the matrix A', and the columns of the matrix A become rows of the matrix A'.

A complex conjugate matrix is obtained from the original one in two stages: the transposition of the original matrix is performed, and then all complex numbers are replaced by complex conjugate ones.

Conjugation and transposition of matrices containing complex numbers will result in different matrices:

```
» K = [l-i, 2+3i; 3-5i, l-9i]
K = 1.0000 - 1.0000i   2.0000 + 3.0000i
    3.0000 - 5.0000i   1.0000 - 9.0000i
» K '
ans =
    1.0000 + 1.0000i   3.0000 + 5.0000i
    2.0000 - 3.0000i   1.0000 + 9.0000i
» K.'
ans =
    1.0000 - 1.0000i   3.0000 - 5.0000i
    2.0000 + 3.0000i   1.0000 - 9.0000i
```

Comment 2

When entering row vectors, their elements can be separated by either spaces or commas. When entering the matrix *K*, commas are used for more visual separation of complex numbers in a row.

The integer exponentiation of a square matrix is performed using the operator ^:

```
» B2 = B^2
B2 =
     27  32  -6
     22  55  -2
    -28  -6   9
```

Check your result by multiplying the matrix by itself.

Make sure you have mastered the basic matrix operations in MATLAB. Find the meaning of the following expression

$$(A + C)\, B^3\, (A - C)^{\mathrm{T}}.$$

Consider the priority of the operations: transpose is performed first, then exponentiation, then multiplication, and addition and subtraction are performed last.

```
» (A+C)*B^3*(A-C)'
ans =
     1848   1914
     10290  3612
```

3.1.4. Multiplication of matrices and vectors

A column vector or a row vector in MATLAB is matrices in which one of the dimensions is equal to one. Therefore, all of the above operations are applicable to matrix multiplication by a column vector or to multiply a row vector by a matrix. For example, the calculation of expression

$$[1 \quad 3 \quad -2]\begin{pmatrix} 2 & 0 & 1 \\ -4 & 8 & -1 \\ 0 & 9 & 2 \end{pmatrix}\begin{bmatrix} -8 \\ 3 \\ 4 \end{bmatrix}$$

can be done as follows:

```
» a = [1 3 -2];
» B = [2 0 1; -4 8 -1; 0 9 2];
» c = [-8; 3; 4];
» a*B*c
ans =
     74
```

3.1.5. Linear equations systems solving

Mathematics does not say anything about dividing matrices and vectors, but MATLAB uses the \ character to solve systems of linear equations. Let's solve a system of three equations with three unknowns:

$$\begin{cases} 1{,}2x_1 + 0{,}3x_2 + 0{,}2x_3 = 1{,}3; \\ 0{,}5x_1 + 2{,}1x_2 + 1{,}3x_3 = 3{,}9; \\ -0{,}9x_1 + 0{,}7x_2 + 5{,}6x_3 = 5{,}4. \end{cases}$$

Let us introduce the matrix of coefficients of the system into array *A*, and let us introduce the vector of the right-hand side of the system into array *b*. Let's solve the system using the symbol \:

```
» x = A\b
x =
    1.0000
    1.0000
    1.0000
```

Check the correctness of the answer by multiplying the matrix of coefficients of system *A* by the column vector *x*.

3.1.6. Block matrices

Very often so-called block matrices appear in applications, i.e. matrices made up of disjoint submatrices (blocks). Consider first the construction of block matrices. Enter matrices

$$A = \begin{pmatrix} -1 & 4 \\ -1 & 4 \end{pmatrix},\ B = \begin{pmatrix} 2 & 0 \\ 0 & 5 \end{pmatrix},\ C = \begin{pmatrix} 3 & -3 \\ -3 & 3 \end{pmatrix},\ D = \begin{pmatrix} 8 & 9 \\ 1 & 10 \end{pmatrix}$$

and create a block matrix out of them $K = \begin{pmatrix} A & B \\ C & D \end{pmatrix}$.

Considering that the matrix *K* consists of two rows, its first row contains matrices *A* and *B*, and its second row contains matrices *C* and *D*. The block matrix can be formed as follows:

```
» K = [A B; C D]
K =
     -1   4   2   0
     -1   4   0   5
      3  -3   8   9
     -3   3   1  10
```

The block matrix can be obtained in another way. If we assume that the matrix *K* consists of two columns, the first column contains matrices *A* and *C*, and the second column contains matrices *B* and *D*:

```
» K = [[A; C] [B; D]]
```

The opposite to the construction of block matrices is the problem of block selection. The selection of matrix blocks is carried out by indexing using a colon. Enter the matrix

$$P = \begin{pmatrix} 1 & 2 & 0 & 2 \\ 4 & 10 & 12 & 5 \\ 0 & 11 & 10 & 5 \\ 9 & 2 & 3 & 5 \end{pmatrix}$$

and then select a submatrix with elements a_{22}, a_{23}, a_{32}, a_{33}, by specifying row and column numbers using colons:

```
»P1 = P(2:3,2:3)
P1 =
     10  12
     11  10
```

To select a column or row from a matrix (that is, an array that has one of the dimensions equal to one), use the number of the column or row of the matrix as one of the indices. The other index should be replaced with a colon without specifying limits. For example, write the second row of the matrix *P* into the vector *p*

```
»p = P(2, :)
p =
    4 10 12 5
```

When selecting a block to the end of the matrix, you can not specify its dimensions, but use the element *end*:

```
»p = P(2, 2:end)
p =
    10 12 5
```

3.1.7. Rows and Columns Deleting

In MATLAB, paired square brackets [ ] denote an empty array, which, in particular, allows you to delete rows and columns of a matrix. To delete a row, it is necessary to assign an empty array to it. Remove for example the first row of a square matrix:

```
» M =[2 0 3; 1 1 4; 6 1 3];
» M(1,:)=[];
» M
```

```
M =
     1  1  4
     6  1  3
```

Note the corresponding resizing of the array, which can be checked withи *size*:

```
» size(M)
ans =
     2  3
```

Columns are removed in the same way. To remove multiple consecutive columns (or rows), they need to be assigned an empty array. Remove the second and third column in the array *M*

```
» M(:, 2:3) = []
M =
     1
     6
```

Indexing significantly saves time when entering matrices with a specific structure.

3.1.8. Filling matrices using indexing

Above, there are several ways to enter matrices in MATLAB. However, it is often easier to generate a matrix than to enter one, especially if it has a simple structure. Consider an example of such a matrix:

$$T = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & -1 & -1 \end{pmatrix}.$$

The matrix *T* is generated in three stages:

1. Create a five-by-five array *T* of zeros.

2. Filling the first row with ones.

3. Filling the last row with minus one to the last element.

The relevant MATLAB commands are given below.

```
» A(1:5, 1:5) = 0
A=
    0   0   0   0   0
    0   0   0   0   0
    0   0   0   0   0
    0   0   0   0   0
    0   0   0   0   0

» A(1, :) = 1
A=
    1   1   1   1   1
    0   0   0   0   0
    0   0   0   0   0
    0   0   0   0   0
    0   0   0   0   0
» A(end, 3:end) = -1
A=
    1   1   1   1   1
    0   0   0   0   0
```

```
     0   0   0   0   0
     0   0   0   0   0
     0   0  -1  -1  -1
```

Some special matrices are created in MATLAB using built-in functions.

3.1.9. Special matrices creating

Filling a rectangular matrix with zeros is performed by the built-in function *zeros*. It’s arguments are the number of rows and columns of the matrix:

```
» A = zeros(2, 6)
A =
     0  0  0  0  0  0
     0  0  0  0  0  0
```

One argument of *zeros* function results in the formation of a predetermined size of square matrix:

```
» A = zeros(3)
A =
     0  0  0
     0  0  0
     0  0  0
```

The identity matrix is initialized using the *eye* function:

```
» I = eye(3)
I=
     1  0  0
```

```
    0 1 0
    0 0 1
```

The *eye* function with two arguments creates a rectangular matrix. It’s main diagonal has ones. And the other elements are equal to zero:

```
» I = eye(4, 8)
I =
    1 0 0 0 0 0 0 0
    0 1 0 0 0 0 0 0
    0 0 1 0 0 0 0 0
    0 0 0 1 0 0 0 0
```

A matrix consisting of ones is formed by calling the *ones* function:

```
» E = ones(3, 7)
E =
    1 1 1 1 1 1 1
    1 1 1 1 1 1 1
    1 1 1 1 1 1 1
```

Using one argument in *ones* function results in a square matrix of ones.

MATLAB provides the ability to fill matrices with random elements.

The result of the *rand* function is a matrix of numbers randomly distributed between zero and one.

The result of the *randn* function is a matrix of normally distributed numbers:

```
» R = rand(3,  5)
R =
    0.9501 0.4860 0.4565 0.4447 0.9218
    0.2311 0.8913 0.0185 0.6154 0.7382
    0.6068 0.7621 0.8214 0.7919 0.1763
```

One argument of *rand* and *randn* functions results in square matrices.

It is often necessary to create diagonal matrices, i.e. matrices for which all off-diagonal elements are equal to zero. The *diag* function forms a diagonal matrix from a column vector or row vector by arranging their elements along the diagonal of the matrix:

```
» d = [1; 2; 3; 4];
» D = diag(d)

D =
    1 0 0 0
    0 2 0 0
    0 0 3 0
    0 0 0 4
```

The *diag* function also serves to extract the diagonal of a matrix into a vector, for example

```
» A = [10 1 2; 1 20 3; 2 3 30];
» d = diag(A)

d =
    10
    20
    30
```

3.1.10. Element-wise operations on matrices

Since vectors and matrices are stored in two-dimensional arrays, the application of mathematical functions to matrices and element-wise operations are performed in the same way as for vectors.

Enter two matrices

$$A=\begin{pmatrix} 2 & 5 & -1 \\ 3 & 4 & 9 \end{pmatrix},\ B=\begin{pmatrix} -1 & 2 & 8 \\ 7 & -3 & -5 \end{pmatrix}.$$

The multiplication of each element of one matrix by the corresponding element of another is performed using the operator .* :

```
» C = A.*B
C =
    -2  10 -8
    21 -12 -45
```

To divide the elements of the first matrix by the corresponding elements of the second, use the operator ./. And to divide the elements of the second matrix by the corresponding elements of the first one use the operator .\:

```
» R1 = A./B1


R1 =
    -2.0000  2.5000 -0.1250
     0.4286 -1.3333 -1.8000
» R2 = A.\B1
R2 =
    -0.5000  0.4000 -8.0000
     2.3333 -0.7500 -0.5556
```

Element-wise exponentiation is performed using the .^ operator. The exponent can be a number or a matrix of the same size as the exponential matrix. In the second case, the elements of the first matrix are raised to powers equal to the elements of the second matrix.

3.1.11. Matrix visualization

Sparse matrices are the matrices which have a sufficiently large number of zeros. It is often necessary to know where nonzero elements are located, i.e. get the so called matrix template. For this, MATLAB uses the *spy* function. Let's see the *G* matrix template

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

```
» spy(G)
```

After executing the *spy* command *Figure No.* 1 graphic window appears. Row and column numbers are plotted on the vertical and horizontal axes. Nonzero elements are indicated by markers.

The number of non-zero items is listed at the bottom of the graphics window (*nz* = 19).

The *imagesc* function gives visual information about the ratio of the values of the elements of the matrix. It interprets the matrix as a rectangular image. Each element of the matrix is represented as a square, the color of which corresponds to the size of the element.

In order to find out the correspondence between the color and the size of an element, use the *colorbar* command, which displays a color bar next to the matrix image (*Insert* (in graphics window *Figure No.* 1), *colorbar*).

Finally, for printing on a monochrome printer, it is convenient to obtain a grayscale image using the command *colormap*(*gray*) (*Edit* (in graphics window *Figure No.* 1), *Colormap*, *Colormap Editor*, *Tools*, *gray*). We will work with matrix *G*. Type the commands below and monitor the state of the graphics window:

```
» imagesc(G)
» colorbar
» colormap(gray)
```

The result is a visual representation of the matrix.

3.2. Individual tasks

1. Create a log of the lab work.
2. Enter two matrices *A* and *B* of three by three dimension.
3. Perform addition, subtraction and multiplication operations on matrices.
4. Transpose matrices *A* and *B*.
5. Create a complex matrix *C* from the matrices *A* and *B*. Elements of matrix *A* must become real parts of complex numbers, and elements of matrix *B* must become complex parts, i.e. the relation:

$$c_{kj} = a_{kj} + b_{kj}i,\ k, j = 1, 2, 3.$$

must be satisfied between the elements of the matrices.

6. Define a matrix that is complex conjugate to a matrix *C*.
7. Raise square matrix *A* to the second power using operator ^. Compare the result obtained by multiplying the matrix *A* by itself.

8. Calculate the product of the first row of matrix *A* and matrix *B*, and the product of matrix *B* and the third column of matrix *A*.

9. Solve a system of linear equations

$$a_{11}x + a_{12}y + a_{13}z = d_1,$$
$$a_{21}x + a_{22}y + a_{23}z = d_2,$$
$$a_{21}x + a_{32}y + a_{33}z = d_3,$$

where the coefficients of the equations system are determined by the number according to the list in the group register from the table 3.1.

10. Create transposed matrices A^T and B^T from matrices *A* and *B* and block matrix $K = \begin{pmatrix} A & B \\ B^T & A^T \end{pmatrix}$.

11. Remove second column and third row from matrix *K*.

12. Fill in a rectangular matrix with dimensions of at least 5×8 with zeros using the function *zeros*.

13. Initialize rectangular and square identity matrices that contain at least five ones using the *eye* function.

14. Create rectangular and square matrices of ones with at least four rows using the *ones* function.

15. Create a 4×5 matrix of numbers randomly distributed between zero and *N* using the *rand* function.

16. Form a diagonal matrix from the elements of the first row of a block matrix using the *diag* function

17. Perform the element-wise operations you know about the matrices *A* and *B*.

18. Perform the visualization of a block matrix using commands *spy*, *imagesc*, *colorbar*, *colormap*(*gray*).

19. Record the first and last few lines from the lab log.

20. Design a laboratory report.

Table 3.1 – Coefficients of the equations system

№	a_{11}	a_{12}	a_{13}	a_{21}	a_{22}	a_{23}	a_{31}	a_{32}	a_{33}	d_1	d_2	d_3
1	1,2	0,3	0,2	0,5	2,1	1,3	–0,9	0,7	5,6	1,32	3,91	5,4
2	1,2	0,3	–0,2	0,5	2,1	1,3	–0,9	0,7	5,6	1,32	3,91	5,4
3	–1,2	0,3	–0,2	0,5	2,1	1,3	–0,9	0,7	5,6	1,32	3,91	5,4
4	1,2	–0,3	0,2	0,5	2,1	1,3	–0,9	0,7	5,6	1,32	3,91	5,4
5	–1,2	–0,3	0,2	0,5	2,1	1,3	–0,9	0,7	5,6	1,32	3,91	5,4
6	–1,2	–0,3	–0,2	0,5	2,1	1,3	–0,9	0,7	5,6	1,32	3,91	5,4
7	–1,2	0,3	0,2	0,5	2,1	1,3	–0,9	0,7	5,6	1,32	3,91	5,4
8	1,2	0,3	0,2	0,5	2,1	1,3	–0,9	0,7	5,6	1,60	3,91	5,4
9	1,2	0,3	0,2	0,5	2,1	1,3	–0,9	0,7	5,6	1,60	4,90	5,4
10	1,2	0,3	0,2	0,5	2,1	1,3	–0,9	0,7	5,6	1,81	4,53	5,4
11	1,2	0,3	0,2	0,5	2,1	1,3	–0,9	0,7	5,6	1,81	4,53	5,8
12	1,2	0,3	0,2	0,5	2,1	1,3	–0,9	0,7	5,6	2,83	4,53	5,8
13	1,2	0,3	0,2	0,5	2,1	1,3	–0,9	0,7	5,6	3,83	4,53	5,8
14	1,2	0,3	0,2	0,5	2,1	1,3	–0,9	0,7	5,6	3,32	4,53	5,8
15	1,2	0,3	0,2	0,5	2,1	1,3	–0,9	0,7	5,6	0,32	0,53	5,8
16	1,2	0,3	0,2	0,5	2,1	1,3	–0,9	0,7	–5,6	0,32	0,45	5,8
17	1,2	0,3	0,2	0,5	2,1	1,3	–0,9	0,7	–5,6	–0,32	0,45	5,8
18	1,2	0,3	0,2	0,5	2,1	1,3	–0,9	0,7	–5,6	–0,32	–0,45	5,8
19	1,2	0,3	0,2	0,5	2,1	1,3	–0,9	0,7	–5,6	–0,32	–0,35	5,8
20	1,2	0,3	0,2	0,5	2,1	1,3	–0,9	0,7	–5,6	–0,32	–0,25	–5,8
21	1,2	0,3	0,2	0,5	2,1	1,3	–0,9	0,7	–5,6	–0,32	–0,18	5,0
22	1,2	0,3	0,2	0,5	2,1	1,3	–0,9	0,7	–5,6	–0,32	–0,15	4,0
23	1,2	0,3	0,2	0,5	2,1	1,3	–0,9	0,7	–5,6	–0,32	–0,1	2,0
24	1,2	0,3	0,2	0,5	2,1	1,3	–0,9	0,7	–5,6	–0,10	–0,1	1,0
25	1,2	0,3	0,2	0,5	2,1	1,3	–0,9	0,7	–5,6	0	–0,1	0,5
26	1,2	0,3	0,2	0,5	2,1	1,3	–0,9	0,7	–5,6	0	–0,01	0
27	1,2	0,3	0,2	0,5	2,1	1,3	–0,9	0,7	–5,6	–0,32	–0,18	5,0
28	1,2	0,3	0,2	0,5	2,1	1,3	–0,9	0,7	5,6	1,81	4,53	5,8
29	1,2	0,3	0,2	0,5	2,1	1,3	–0,9	0,7	5,6	2,83	4,53	5,8
30	1,2	0,3	0,2	0,5	2,1	1,3	–0,9	0,7	5,6	3,83	4,53	5,8

Laboratory work 4

BUILDING TABLES OF VALUES AND GRAPHS OF FUNCTIONS IN THE MATLAB PACKAGE

The *purpose of the laboratory work* is to obtain and consolidate knowledge, to develop practical skills in working with the MATLAB package when building tables of values and functions graphs.

4.1. Summary of theory

4.1.1. Building tables of one variable function values in the MATLAB package

Displaying a function in a table is useful if you have a relatively small number of function values. Suppose you want to display a table of function values in the command window

$$y(x) = \frac{\sin^2 x}{1+\cos x} + e^{-x}\ln x$$

in points 0,2; 0,3; 0,5; 0,8; 1,3; 1,7; 2,5.

The task is performed in two stages.

1. Row vector *x* is created. It contains the coordinates of the given points.

2. The values of the function *y(x)* from each element of the vector *x* are calculated. The obtained values are written into the row vector *y*.

The function values should be found for each of the elements of the row vector *x*. Therefore, the operations in the expression for the function should be performed elementwise.

```
» x = [0.2 0.3 0.5 0.8 1.3 1.7 2.5]
x =
    0.2000 0.3000 0.5000 0.8000 1.3000 1.7000 2.5000
» y = sin(x).^2./(1+cos(x))+exp(-x).*log(x)
y =
    -1.2978  -0.8473  -0.2980  0.2030  0.8040  1.2258
1.8764
```

Please note that when trying to use exponentiation ^, division / and multiplication * (which are not elementwise), an error message is displayed already when squaring *sin (x)*:

```
» y = sin(x)^2/(1+coз(x))+exp(-x)*log(x)
??? Error using ==> ^
Matrix must be square.
```

In MATLAB, the * and ^ operations are used to multiply matrices of appropriate sizes and to raise a square matrix to epy second power.

You can make the table more readable by placing the function values directly below the argument values:

```
» x
x =
    0.2000 0.3000 0.5000 0.8000 1.3000 1.7000 2.5000
» y
y =
    -1.2978  -0.8473  -0.2980  0.2030  0.8040  1.2258
1.8764
```

Often it is required to display the value of a function at points of a segment that are equidistant from each other (step). Suppose that it is necessary to display a table of values of the function *y(x)* on the segment [1, 2] with a step of 0.2. You can, of course, enter a row vector of argument values *x* = [1, 1.2, 1.4, 1.6, 1.8, 2.0] from the command line and calculate all the values of the function as described above. However, if the step is not 0.2, but, for example, 0.01, then there is a lot of work to be done on entering the vector *x*.

MATLAB provides a simple creation of vectors, each element of which differs from the previous one by a constant value, i.e. one step. To enter such vectors, use a colon (do not confuse with indexing with a colon). The following two operators produce the same row vectors. You can conditionally type

```
» x = [1, 1.2, 1.4, 1.6, 1.8, 2.0]
x =
    1.0000 1.2000 1.4000 1.6000 1.8000 2.0000
» x = [1:0.2:2]
x =
    1.0000 1.2000 1.4000 1.6000 1.8000 2.0000
```

where *x* = [start value : step: end value].

It is not necessary to care that the sum of the penultimate step value is equal to the final value, for example, when executing the following assignment operator

```
» x = [1:0.2:1.9]
x =
    1.0000 1.2000 1.4000 1.6000 1.8000
```

The row vector will fill up to an element that does not exceed the final value we defined. The step can be negative:

```
» x = [1.9:-0.2:1]
x =
     1.9000 1.7000 1.5000 1.3000 1.1000
```

In the case of a negative step, the start value must be greater than the end value to obtain a non-empty row vector.

To fill a column vector with elements starting at zero and ending at 0.5 in step of 0.1, fill the row vector and then use the transpose operation:

```
» x = [0:0.1:0.5]'
x =
     0
     0.1000
     0.2000
     0.3000
     0.4000
     0.5000
```

Please note that the elements of a vector filled with a colon can only be real, so an apostrophe can be used for transposition instead of a dot with an apostrophe.

A step that is equal to one is allowed not to be specified during automatic filling:

```
» x = [1:5]
x =
     1  2  3  4  5
```

Suppose you want to display a table of function values

$$y(x) = e^{-x}\sin(10x)$$

on the segment [0, 1] with a step 0,05.

To complete this task, you should perform the following actions:

1. Form the row vector *x* using a colon.
2. Calculate the values of *y(x)* from the elements *x*.
3. Write the result to the row vector *y*.
4. Output *x* and *y*.

```
» x = [0:0.05:1];
» y = exp(-x).*sin(10*x);
» x
x =
    Columns 1 through 7
    0 0.0500 0.1000 0.1500 0.2000 0.2500 0.3000
    Columns 8 through 14
    0.3500 0.4000 0.4500 0.5000 0.5500 0.6000 0.6500
    Columns 15 through 21
    0.7000 0.7500 0.8000 0.8500 0.9000 0.9500 1.0000
» y
y = Columns 1 through 7
    0 0.4560 0.7614 0.8586 0.7445 0.4661 0.1045
    Columns 8 through 14
    -0.2472  -0.5073  -0.6233  -0.5816  -0.4071  -0.1533
0.1123
    Columns 15 through 21
    0.3262  0.4431  0.4445  0.3413  0.1676  -0.0291  -
0.2001
```

Row vectors *x* and *y* consist of twenty-one elements and do not fit in one line on the screen, so they are displayed in parts. Since *x* and *y* are stored in one-by-twenty-one two-dimensional arrays, they are displayed in columns, each of which consists of one element. First, *columns 1 through 7* are displayed, then *columns 8 through 14* are displayed, and finally *columns 15 through 21* are displayed. The graphical representation of the function is more visual and convenient.

4.1.2. One variable functions plotting

Graphs functions in the linear scale

MATLAB has well-developed graphical capabilities for data visualization. Let us first consider the construction of the simplest graph of one variable function using the example of the function

$$y(x) = e^{-x}\sin(10x),$$

that is defined on the segment [0, 1]. Displaying a function in a graph form consists of the following steps:

1. Specifies a vector of values for the argument *x*.
2. Calculation of the *y* vector of the function *y(x)* values.
3. Calling the *plot* command to plot the graph.

Commands for specifying the vector *x* and calculating the function are best terminated with a semicolon to suppress their values output to the command window. It is not necessary to put a semicolon after the *plot* command, since it does not output anything to the command window

```
» x = [0:0.05:1];
» y = exp(-x).*sin(10*x);
» plot(x, y)
```

The *Figure No.* 1 window appears on the screen after executing the commands. It includes a graph of the function. The window contains a menu, a toolbar and a graph area. In the following, we will describe the commands specially designed for the graphs design. Now we are interested in the very principle of graphs plotting and some of the basic possibilities' functions visualizing.

Two vectors of the same dimension must be defined, for example *x* and *y*, to plot the function in the MATLAB workspace. The corresponding array *x* contains the values of the arguments, and *y* contains the function values of those arguments.

The *plot* command connects the points with the coordinates ($x(i)$, $y(i)$) using straight lines, automatically scaling the axes for the optimal graph position in the window. When plotting graphs, it is convenient to place the main MATLAB window and the graph window side by side on the screen so that they do not overlap.

The plotted function graph has kinks. For a more accurate plotting, the function *y(x)* must be calculated at a larger number of points on the segment [0, 1], i.e. it is necessary to set a smaller step when entering the vector *x*:

```
» x = [0:0.01:1];
» y = exp(-x).*sin(10*x);
» plot(x, y)
```

The result is a graph of the function as a smoother curve.

It is convenient to compare several functions by displaying their graphs on the same axes. For example, let us build graphs of functions $f(x) = \sin\left(\frac{1}{x^2}\right)$, $f(x) = \sin\left(\frac{1{,}2}{x^2}\right)$ on the segment [–1, –0.3] using the following sequence of commands:

```
» x = [–1:0.005:–0.3];
» f = sin(x.^-2);
» g = sin(1.2*x.^-2);
» plot(x, f, x, g)
```

Functions do not have to be defined on the same segment. In this case, when plotting the graphs, MATLAB selects the maximum segment containing the rest. It is only important to indicate the vectors corresponding to each other in each pair of abscissas and ordinates vectors, for example:

```
» x1 = [–1:0.005:–0.3];
» f = sin(x1.^-2);
» x2 = [–1:0.005:0.3];
» g = sin(1.2*x2.^-2);
» plot(x1, f, x2, g)
```

Similarly, by specifying comma-separated pairs of arguments (abscissa vector, ordinate vector) in *plot*, graphs of an arbitrary number of functions are plotted.

Comment 1

Using *plot* with one argument, that is a vector, results in a "vector graph", i.e. dependence of vector elements values on their numbers. The *plot* argument can also be a matrix; in this case, the graphs of the columns are displayed on the same coordinate axes.

Sometimes you want to compare the behavior of two functions whose values are very different from each other. The function graph with small values practically merges with the abscissa axis, and it is not possible to establish its appearance. In this situation, the *plotyy* function helps, which displays graphs in a window with two vertical axes with a suitable scale.

Compare, for example, two functions: $f(x) = x^{-3}$ and $F(x) = 1000 \cdot (x + 0{,}5)^{-4}$.

```
» x = [0.5:0.01:3];
» f = x.^-3;
» F = 1000*(x+0.5).^-4;
» plotyy(x, f, x, F)
```

As you run this example, notice that the color of the graph matches the color of its corresponding y-axis.

The *plot* function uses a linear scale on both coordinate axes. However, MATLAB provides the user with the ability to plot functions of one variable on a logarithmic or semi-logarithmic scale.

Functions graphs in logarithmic scales

The following functions are used to plot graphs in logarithmic and semi-logarithmic scales:

– *loglog* (logarithmic scale on both axes);

– *semilogx* (logarithmic scale on the abscissa axis only);

– *semilogy* (logarithmic scale on the ordinate only).

The *loglog*, *semilogx*, and *semilogy* arguments are specified as a pair of vectors of abscissa and ordinate values in the same way as for the *plot* function described in the previous paragraph. Let's plot, for example, graphs of functions $f(x = \ln(0{,}5x)$ and $g(x) = \sin(\ln(x))$ on the interval [0,1, 5] in a logarithmic scale along the *x* axis:

```
» x = [0.1:0.01:10];
» f = log(0.5*x);
» g = sin(log(x));
» semilogx(x, f, x ,g)
```

Line properties setting on functions graphs

The constructed graphs of functions should be as easy to read as possible. Often you need to apply markers, change the color of the lines, and in preparation for monochrome printing, you often need to set the line type (solid, dotted, dash-dotted, etc.).

MATLAB provides the ability to control the appearance of graphs plotted with *plot*, *loglog*, *semilogx*, and *semilogy*. This is done by an additional argument placed behind each pair of vectors. This argument is enclosed in apostrophes and consists of three characters that define: color, marker type, and line type. One, two or three positions are used, depending on the required changes.

Table 4.1 shows the possible values of this argument with an indication of the result.

If, for example, you need to plot the first graph with red dotted markers without a line, and the second graph needs to be drawn with a black dotted line, then you should use the command *plot*(*x*, *f*, '*r*.', *x*, *g*, '*k*:').

Table 4.1 – Possible argument values

Color		Marker type		Line type	
y	yellow	.	dot	−	solid
m	pink	*o*	circle	:	dotted
c	light blue	*x*	cross	−.	dash-dotted
r	red	+	plus sign	−−	dashed
g	green	*	asterisk		
b	bkue	*s*	square		
w	white	*d*	rhombus		
k	black	*v*	downward triangle		
		^	upward triangle		
		<	left-pointing triangle		
		>	right-pointing triangle		
		p	five pointed star		
		h	six pointed star		

Functions graphs designing

Whether it is convenient to use graphics or not depends on additional design elements: a coordinate grid, axis labels, title and legend. The grid is applied using the *grid on* command, the axis labels are placed using *xlabel, ylabel*, the title is given by the *title* command. The presence of several graphs on the same axes requires placing a legend with the *legend* command with information about the lines. All of the above commands are applicable to graphs in both linear and logarithmic and semi-logarithmic scales. The following commands display graphs of daily temperature changes, which are provided with all the necessary information

```
» time = [0 4 7 9 10 11 12 13 13.5 14 14.5 15 16 17
18 20 22];
» temp1 = [14 15 14 16 18 17 20 22 24 28 25 20 16 13
13 14 13];
» temp2 = [12 13 13 14 16 18 20 20 23 25 25 20 16 12
12 11 10];
» plot(time, temp1, 'ro-', time, temp2, 'go-')
» grid on
» title('Daily temperatures')
» xlabel('Time (hours))
» ylabel('Temperature (C)')
» legend('March1 10, March 11')
```

When adding a legend, keep in mind that the order and number of arguments to the *legend* command should match the lines on the graph. The last additional argument can be the position of the legend in the graphics window:

- –1 – outside the graph in the upper right corner of the graphics window;

- 0 – the best position within the graph is chosen so as to overlap the graphs themselves as little as possible;
- 1 – in the upper right corner of the graph (this position is used by default);
- 2 – in the upper left corner of the graph;
- 3 – in the lower left corner of the graph;
- 4 – in the lower right corner of the graph.

You can add formulas and change font styles for the graph title, legend, and axis labels using the *TeX* format.

MATLAB displays graphs in different colors. A monochrome printer will print graphs in various shades of gray, that is not always convenient. The *plot* command makes it easy to set the style and color of lines, for example,

```
» plot(x,f,'k-',x,g,'k:')
```

builds the first graph with a solid black line, and the second graph with a black dotted line. The *'k-'* and *'k:'* arguments specify the style and color of the first and second lines. Here *k* means black, and a hyphen or colon mean a solid or dashed line. The window with the graph can be closed by clicking on the button with a cross in the upper right corner

4.1.3. Two variables functions plotting

Plotting a function of two variables on a rectangular variable definition area includes two preliminary stages in MATLAB:

1. Subdividing the definition area with a rectangular grid.
2. Calculation function values at points of grid lines intersection and write them into a matrix.

Let's plot the function $z(x, y) = x^2 + y^2$ on the definition area in the form of a square $x \in [0, l]$, $y \in [0, l]$. It is necessary to divide the square with a uniform grid (for example, with a step of 0.2) and calculate the values of the functions at the nodes indicated by dots.

It is convenient to use two two-dimensional arrays *x* and *y*, six by six, to store information about the coordinates of the nodes. Array *x* consists of identical rows where coordinates *x*l, *x*2, ..., *x*6 are written, and the array *y* contains the identical columns with *y*l, *y*2, ..., *y*6. Let's write the values of the function at the grid nodes into an array *z* of the same dimension (6×6), moreover, to calculate the matrix *Z*, we use the expression for the function, but with element-wise matrix operations. Then, for example, *z*(3, 4) will be exactly equal to the value of the function *z*(*x*, *y*) at the point (*x*3, *y*4). The *meshgrid* function is used in MATLAB to generate *x* and *y* mesh arrays from the coordinates of the nodes. The *mesh* function is used to plot the graph as a wireframe surface. The following operators lead to the appearance of a window with the function graph on the screen (the semicolon is not put at the end of the operators in order to control the generation of arrays):

```
» [X, У] = meshgrid(0:0.2:1,0:0.2:1)
X =
    0 0.2000 0.4000 0.6000 0.8000 1.0000
    0 0.2000 0.4000 0.6000 0.8000 1.0000
    0 0.2000 0.4000 0.6000 0.8000 1.0000
    0 0.2000 0.4000 0.6000 0.8000 1.0000
    0 0.2000 0.4000 0.6000 0.8000 1.0000
    0 0.2000 0.4000 0.6000 0.8000 1.0000

y =
    0      0      0      0      0      0
    0.2000 0.2000 0.2000 0.2000 0.2000 0.2000
    0.4000 0.4000 0.4000 0.4000 0.4000 0.4000
    0.6000 0.6000 0.6000 0.6000 0.6000 0.6000
    0.8000 0.8000 0.8000 0.8000 0.8000 0.8000
    1.0000 1.0000 1.0000 1.0000 1.0000 1.0000


» Z = X.^2+Y.^2
Z =
    0 0.0400 0.1600 0.3600 0.6400 1.0000
    0.0400 0.0800 0.2000 0.4000 0.6800 1.0400
    0.1600 0.2000 0.3200 0.5200 0.8000 1.1600
    0.3600 0.4000 0.5200 0.7200 1.0000 1.3600
    0.6400 0.6800 0.8000 1.0000 1.2800 1.6400
    1.0000 1.0400 1.1600 1.3600 1.6400 2.0000
» mesh(X,Y,Z)
```

MATLAB allows you to apply for more information to the graph, such as color matching to function values. The mesh is generated using the *meshgrid* command, invoked with two arguments. The arguments are vectors. The elements of the vectors correspond to the grid on the rectangular area of the function construction. One argument can be used if the area of function construction is square. To calculate the function, use element-wise operations.

Let's consider the main possibilities provided by MATLAB for visualizing functions of two variables, using the example of plotting a function

$$z(x,y) = 4\sin(2\pi x)\cdot\cos(1{,}5\pi y)\cdot(1-x^2)\cdot y\cdot(1-y)$$

on a rectangular definition area $x \in [-l, l]$, $y \in [0, l]$.

Prepare matrices with coordinates of grid nodes and function values:

```
» [X, Y] = meshgrid(-l:0.05:l, 0:0.05:1);
» Z=4*sin(2*pi*X).*cos(1.5*pi*Y).*(1-X.^2).*Y.*(1-Y);
```

To build a wireframe surface, use the *mesh* function, called with three arguments

```
» mesh(X,Y,Z)
```

The color of the surface lines corresponds to the values of the function. MATLAB only draws the visible portion of the surface.

With the *hidden off* command, you can make the wireframe surface "transparent" by adding a hidden portion. The *hidden on* command removes the invisible part of the surface, returning the graphics to their previous appearance. The *surf* function builds the wireframe surface of the function graph and fills each cell of the surface with a specific color depending on the function values at the points corresponding to the corners of the cell. The color is constant within each cell. Look at the results of the command

```
» surf(X,Y,Z)
```

The *shading flat* command allows you to remove wireframe lines. To obtain a surface smoothly filled with a color depending on the function values, use the command *shading interp*.

With *shading faceted* you can go back to the surface with wireframe lines.

Three-dimensional plots obtained using the commands described above are convenient for getting an idea of the surface shape, but it is difficult to estimate the values of the function from them. MATLAB defines the *colorbar* command, which displays a column next to the graph that sets the correspondence between color and function value. Use *surf* to plot a surface and add color information

```
» surf(X,Y,Z)
» colorbar
```

The *colorbar* command can be used in conjunction with all 3D building functions.

Using a colored surface, it is difficult to draw a conclusion about the function value at one point or another in the *xy* plane. The *meshc* or *surfc* commands allow you to get a more accurate idea of the function behavior. These commands draw a wireframe surface or a color-flooded wireframe surface. Then they place function level lines (lines of function values constancy) on the *xy* plane:

```
» surfc(X,Y,Z)
» colorbar
```

MATLAB allows you to construct a surface composed of level lines using the *contour3* function. This function can be used in the same way as *mesh*, *surf*, *meshc*, and *surfc* described above with three arguments.

The number of level lines is selected automatically. The number of level lines can be specified by the fourth argument to *contour3*. Also, a vector whose elements are equal to the function values displayed as level lines can be specified as the fourth argument in *contour3*.

Setting a vector (the fourth argument *levels*) is convenient when you need to study the behavior of a function in a certain range of its values (a slice of a function).

Plot, for example, a surface consisting of level lines corresponding to function values from 0 to 0.5 in increments of 0.01:

```
» levels = [0:0.01:0.5];
» contour3(X, Y, Z, levels)
» colorbar
```

4.1.4. Contour plots of two variables functions building

MATLAB provides the ability to generate various types of contour plots using the *contour* and *contourf* functions. Let's consider their capabilities using the example of the function

$$z(x,y) = 4\sin(2\pi x)\cdot\cos(1{,}5\pi y)\cdot(1-x^2)\cdot y\cdot(1-y).$$

Using *contour* with three arguments *contour(X, Y, Z)* results in a graph that shows the level lines on the *xy* plane, but without numerical values on them. Such a graph is not very informative, it does not allow you to find out the values of the function on each of the level lines.

Using the *colorbar* command will also not allow you to accurately determine the function's values. Each level line can be assigned a value that the function under study takes on it using the *clabel* function defined in MATLAB. The *clabel* function is called with two arguments: a matrix containing information about the level lines and a pointer to the graph on which the markup should be applied.

The user does not need to create *clabel* arguments himself. The *contour* function, called with two output parameters, not only builds the level lines, but also finds the parameters required for the *clabel*. Use *contour*with output arguments *CMatr* and *h* (the *CMatr* array contains information about the level lines, and the *h* array contains pointers). End the call to *contour* with a semicolon to suppress the displaying of the output parameters and plot the grid:

```
» [CMatr, h] = contour(X, Y, Z);
» clabel(CMatr, h)
» grid on
```

An additional argument to the *contour* function (just like *contour3* described above) can be either the number of level lines or a vector containing the function values for which you want to draw level lines.

Visual information about the change in the function is given by filling a rectangle on the *xy* plane with a color depending on the value of the function at the points of the plane. The *contourf* function is intended for building such graphs. Its use is the same as *contour*. The following example displays a graph that consists of twenty level lines, and the gaps between them are filled with colors corresponding to the values of the function under study:

```
» contourf(X, Y, Z, 20)
» colorbar
```

4.1.5. Functions graphs designing

A simple and effective way to change the color scheme of a graph is to set the color palette using the *colormap* function. The following example demonstrates preparing a function graph for printing on a monochrome printer using the *gray* palette

```
» surfc(X, Y, Z)
» colorbar
» colormap(gray)
» title('Function Graph z(x,y)')
» xlabel('x')
» ylabel('y')
» zlabel('z')
```

Note that the *colormap(gray)* command changes the palette of the graphics window, i.e. the following graphs will also be displayed in this window in gray tones. To restore the original palette value, use the *colormap('default')* command. The color palettes available in MATLAB are shown in table 4.2.

4.1.6. Displaying multiple graphs on one axis

To display several graphs of one variable functions on the same axes, the capabilities of the *plot, plotyy, semilogx, semilogy, loglog* functions were used. They allow you to display graphs of several functions by specifying the corresponding vector arguments in pairs, for example *plot (x, f, x, g)*. However, they cannot be used to unite 3D plots.

Table 4.2 – Color palettes

Palette	Color change
autumn	Smooth change red – orange – yellow
bone	Similar to the *gray* palette, but with a slight shade of blue
colorcube	Each color changes from dark to bright
cool	Shades of cyan and magenta
copper	Shades of copper
flag	Cyclic change red – white – blue – black
gray	Shades of gray
hot	Smooth change black – red – orange – yellow – white
hsv	Smooth change like the colors of the rainbow
jet	Smooth change blue – cyan – red – green – yellow – red
pink	Similar to the *gray* palette, but with a slight shade of brown
prism	Cyclic change red – orange – yellow – green – blue – purple
spring	Shades of magenta and yellow
summer	Shades of green and yellow
vga	Windows palette of sixteen colors
white	One white color
winter	Shade of blue and green

To unite such graphs, use the *hold on* command. It should be set before building a graph. In the following example, two graphs unit (plane and cone) results in their intersection. The cone is defined parametrically by the following dependencies:

$$x(u,v) = 0{,}3 \cdot u \cdot \cos v;\ y(u,v) = 0{,}3 \cdot u \cdot \sin v;\ z(u,v) = 0{,}6 \cdot u;\ u,v \in [-2\pi, 2\pi].$$

To graphically display a cone, you first need to generate a column vector and a row vector using a colon. These vectors contain the values of the parameters on a given interval (it is important that *u* is a column vector, and *v* is a row vector):

```
» u = [-2*pi:0.1*pi:2*pi]';
» v = [-2*pi:0.1*pi:2*pi];
```

Next, the matrices *X*, *Y*, are formed, containing the values of the functions $x(u,v)$ and $y(u,v)$ at the points corresponding to the values of the parameters. Formation of matrices is performed using the outer product of vectors.

Comment 2

The outer product of vectors $a=(a_1,\dots,a_j,\dots,a_N)$, $b=(b_1,\dots,b_k,\dots,b_M)$ is a matrix $C=(c_{jk})$, $j=\overline{1,N}$, $k=\overline{1,M}$ of size $N\times M$, whose elements are calculated by the formula $c_{jk}=a_j b_k$.

Vector *a* is a column vector and is represented in MATLAB as a two-dimensional array of size *N* by one. The column vector *b*, when transposed, goes into a row vector of size one by *M*. The column vector and the row vector are matrices in which one of the dimensions is equal to one. In fact, $C = ab^T$, where the multiplication occurs according to the matrix product rule. The asterisk operator is used to calculate the matrix product in MATLAB. We calculate the outer product for two vectors:

```
» a = [1;2;3];
» b = [5;6;7];
» C = a*b'

C =
 5  6  7
10 12 14
15 18 21
```

Let's form the matrices *X, Y*, necessary for the graphical display of the cone:

```
» X = 0.3*u*cos(v);
» Y = 0.3*u*sin(v);
```

The *Z* matrix must be the same size as the *X* and *Y* matrices. In addition, it must contain values that correspond to the parameter values. If the function $z(u,v)$ included the product of *u* and *v*, hen the matrix *Z* could be filled similarly to the matrices *X* and *Y* using the outer product. On the other hand, the function *z*(*u*,*v*can be represented as $z(u,v) = 0{,}6 \cdot u \cdot g(v)$, where $g(v) \equiv 1$.Therefore, to calculate *Z* you can apply the outer product of vectors u and $g(v)$, where the row vector $g(v)$ has the same dimension as *v*, but consists of ones:

```
» Z = 0.6*u*ones(size(v));
```

All the required matrices for the display of the cone have been created. Plane is defined as follows:

```
» [X,Y] = meshgrid(-2:0.1:2);
» Z = 0.5*X+0.4*Y;
```

Now it is not difficult to write down the complete sequence of commands for constructing intersecting cone and plane:

```
» u = [-2*pi:0.1*pi:2*pi]';
» v = [-2*pi:0.1*pi:2*pi];
» X = 0.3*u*cos(v);
» Y = 0.3*u*sin(v);
» Z = 0.6*u*ones(size(v));
» surf(X, Y, Z)
» [X,Y] = meshgrid(-2:0.1:2);
» Z = 0.5*X+0.4*Y;
» hold on
» mesh(X, Y, Z)
» hidden off
```

As a result of the program, we get the geometric figure shown in Fig. 4.1.

The *hidden off* command is used to show the part of the cone that is under the plane.

Note that the *hold on* command applies to all subsequent plots to the current window. To place graphs in new windows, execute the *hold off* command. The *hold on* command can also be used to position several graphs of one variable functions, for example,

```
» plot(x,f,x,g)
```

is equivalent to the sequence

```
» plot(x,f)
» hold on
» plot(x,g)
```

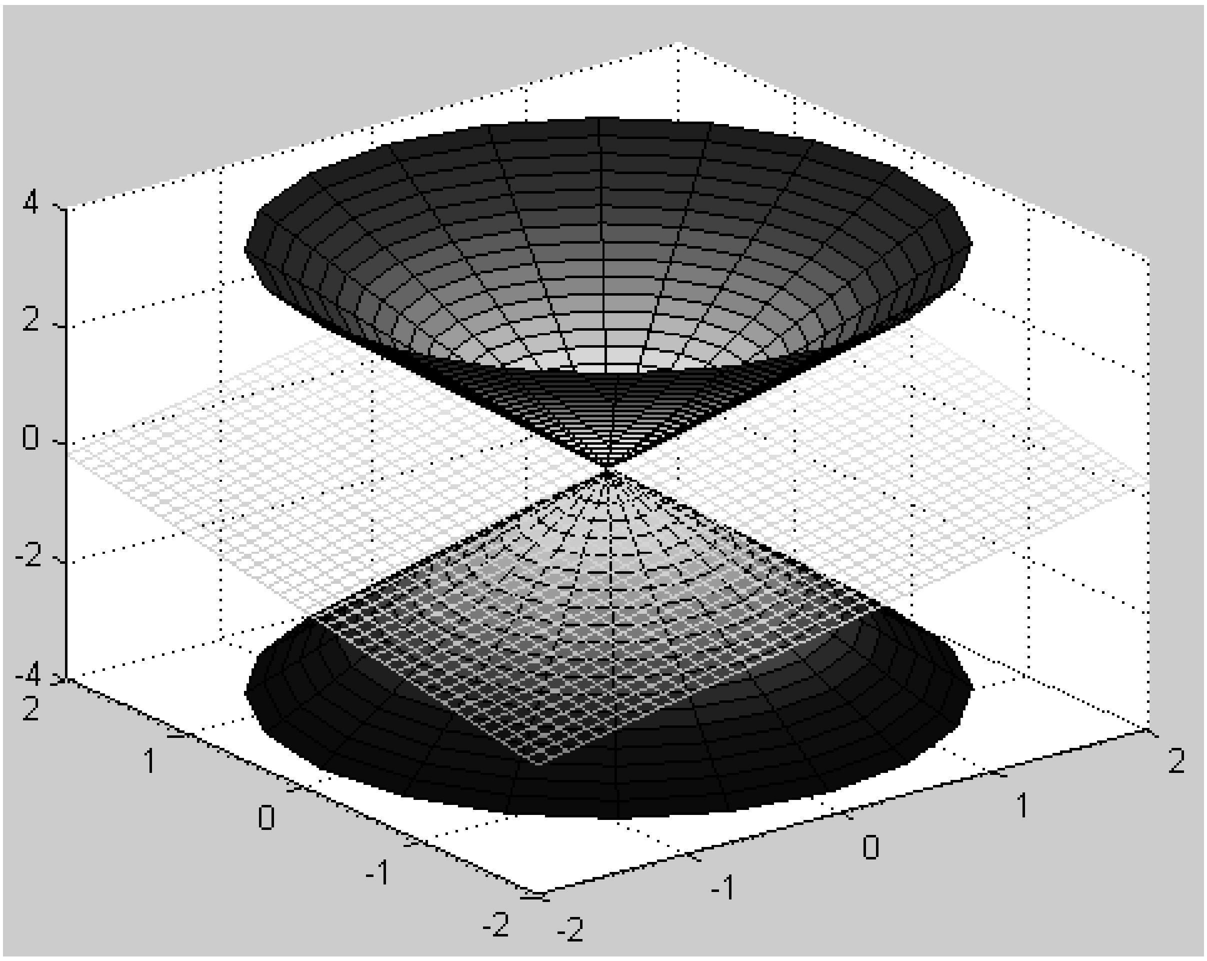


Figure 4.1. The result of the program

4.2. Individual tasks

1. According the number N in the group register list, written in the form $N = CM$, where C is the rank of tens, M is the units rank, in the integer range [N, N+5] calculate the table of values for the expression given using the table 4.3 and table 4.4.

2. Complete task 1 using negative step –1.

3. Display two graphs of the function $f(x)$ from task 1 with steps of 1.0 and 0.05 in the interval [N, N+5] in a linear scale on the same axes.

Using the *plotyy* function, plot the graphs of the functions $f(x)$ and $10^{-2} \cdot f(x) \cdot \sin(x)$ in the interval [N, N+5] in increments of 0.1.

4. Complete task 3 using:

– logarithmic scale on both axes;

– logarithmic scale along the abscissa;

– logarithmic scale along the ordinate axis.

In this step, use six types of markers, six different line colors and different types of lines.

5. Form a matrix and vector sizes, respectively, not less than 5×6 and 1×7. Their first elements are your group journal number. Plot vector and matrix graphs.

6. Give the graph of the function $z(x, y) = x^2 + y^2$ on the defenition area in the form of a square $x \in [0, 1]$, $y \in [0, 1]$ with a step of 0.2 and a graph of a function with a smaller grid step.

7. Construct a transparent and opaque wireframe surface for a function $z(x, y) = 4\sin(2\pi x)\cdot\cos(1{,}5\pi y)\cdot(1-x^2)\cdot y\cdot(1-y)$ on a rectangular defenition area $x \in [-1, 1]$, $y \in [0, 1]$.

Modify the function $z(x,y)$ in some way so that your group register list number appears in the function expression. Report transparent and opaque wireframe surface for your function.

8. Build the wireframe surface of the function $z(x,y)$ using the *surf*(*X*,*Y*,*Z*), *shading flat*, *shading interp* commands and report them.

9. Build the wireframe surface of the function $z(x,y)$ using the *surf*(*X*,*Y*,*Z*) and *colorbar* commands. Report the results.

10. Build the wireframe surface of the function $z(x,y)$ using the *surfc*, *meshc* and *colorbar* commands. Report the results.

11. Build the surfaces of the function $z(x,y)$, consisting of level lines using the *contour3* function with three and four arguments. Report the results.

12. Build contour plots of the function $z(x,y)$ using the *contour*, *contourf*, *clabel* functions. Report the results.

13. Perform three different color schemes for the $z(x,y)$ function graph. Report the results.

14. Draw an intersecting cone and plane.

15. Cross the cone with two different planes.

16. Design a laboratory report.

Table 4.3 – Individual task for the units rank *M* of the number in group register

The units rank *M* of the number in group register	0 or 5	1 or 6	2 or 7	3 or 8	4 or 9
Expression	$\frac{A^{2/3}\cdot\sqrt{C^3}}{(B+D)^2}$	$\frac{ABC}{A^2+D}$	$\frac{A\cdot\sqrt{B}}{D^{1/3}/C^2}$	$\frac{C^{2/5}+\sqrt{D}}{B^{\frac{1}{3}}\cdot\sqrt[5]{A^2}}$	$\frac{AC\cdot\sqrt[3]{D}}{C^{2/3}+B^2}$

Table 4.4 – Individual task for the rank of tens *C* of the number in group register

The rank of tens *C* of the number in group register	*A*	*B*	*C*	*D*
0	$(\sin(N)\cos(N))^2$	$(\ln(N+2))/N$	$(\exp(N/N^2))^{-2}$	$A+B$
1	$1+(\cos(N+1))^2$	$(\ln(N^2))^{-2}$	$\exp(-N)+1$	$A+C$
2	$(\mathrm{tg}(N))^2+N^{-1}$	$(\log(N))^{-2}$	$1+N/\exp(N)$	B/C
3	$\sin(N)/(\mathrm{ctg}(N))^4$	$\log(N^{-3}+N^3)$	$N/\exp(-N)$	$C/(A+B)$

Laboratory work 5

RESEARCH OF FUZZY SETS FORMATION WAYS AND FUZZY SETS OPERATIONS

The *purpose of the laboratory work* is to obtain and consolidate knowledge, to develop practical skills for constructing fuzzy sets using various types of membership functions and to introduce to the most common logical operations on fuzzy sets

5.1. Summary of theory

5.1.1. Membership functions

Triangle membership function

The Fuzzy Logic Tools (FLT) as part of the MATLAB package contains 11 built-in types of membership functions (MF), formed on the basis of piecewise linear functions, Gaussian distribution, sigmoid curve, quadratic and cubic polynomial curves. The simplest MFs are triangular and trapezoidal. The name of the triangular MF is *trimf* (*triangle membership function*). Parametrically, it is nothing more than a set of three points that form a triangle.

Function description:

$$y = trimf\,(x, [a, b, c]),$$

where the vector x is the base set on which the FP is determined. The values a and c define the base of the triangle, b defines its vertex.

Analytically, the triangular MF can be specified as follows (Fig 5.1, *a*):

$$f(x,a,b,c)=\begin{cases}0, x<a,\\ \dfrac{x-a}{b-a}, a\le x\le b,\\ \dfrac{c-x}{c-b}, b\le x\le c,\\ 0, x>c.\end{cases}$$

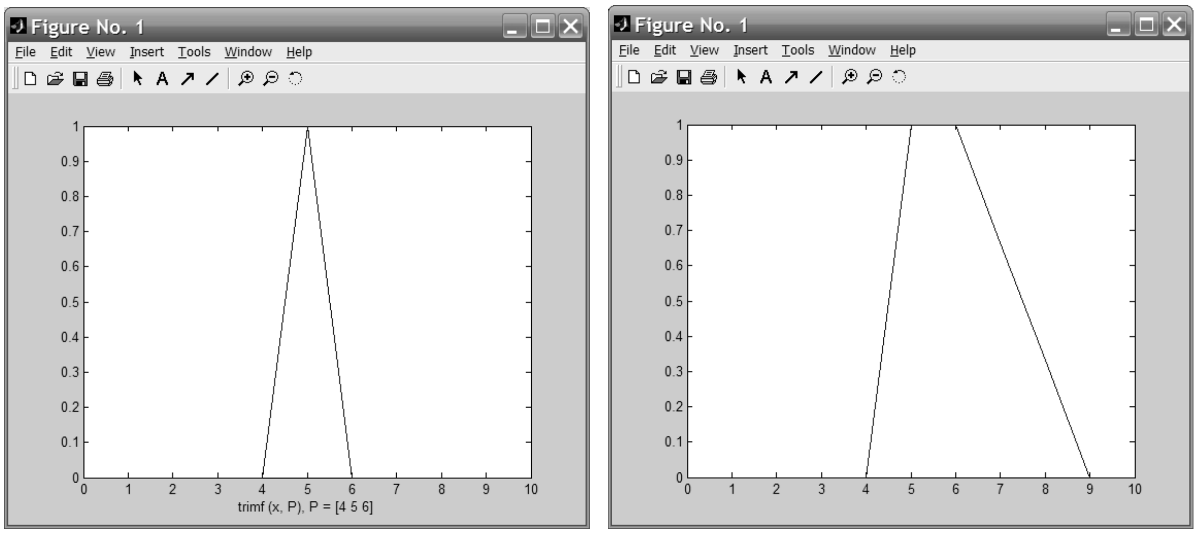


Figure 5.1. Triangular (a) and trapezoidal (b) membership functions

Next, we will consider examples of using various MFs in the system.

Examples are fragments of programs and comments in the MATLAB package language.

Example 5.1. MF *trimf* program.

```
» x = 0 : 0,1 : 10;
» y = trimf (x, [3 6 8]);
» plot (x, y);
» xlabel ('trimf (x, P), P = [3 6 8]');
```

Trapezoidal membership function

Trapezoidal MF *trapmf* (*trapezoid membership function*) differs from the previous function only in that it has an upper base.

Function description:

$$y = trapmf\,(x, [a, b, c, d]),$$

where parameters *a* and *d* are the lower base of the trapezoid; *b* and *c* are the upper base of the trapezoid (Figure 5.1, *б*).

The analytical record of the trapezoidal function is as follows:

$$f(x,a,b,c,d) = \begin{cases} 0, \ x < a, \\ \dfrac{x-a}{b-a}, \ a \le x < b, \\ 1, \ b < x \le c, \\ \dfrac{d-x}{d-c}, \ c < x \le d, \\ 0, \ x > d. \end{cases}$$

Simple and two-sided Gaussian membership function

On the basis of the Gaussian distribution function, you can construct an MF of two types. First type is a simple Gaussian membership function and a the second type is two-sided one, formed using various Gaussian distribution functions. The first is *gaussmf* and the second is *gauss2mf*.

Function description:

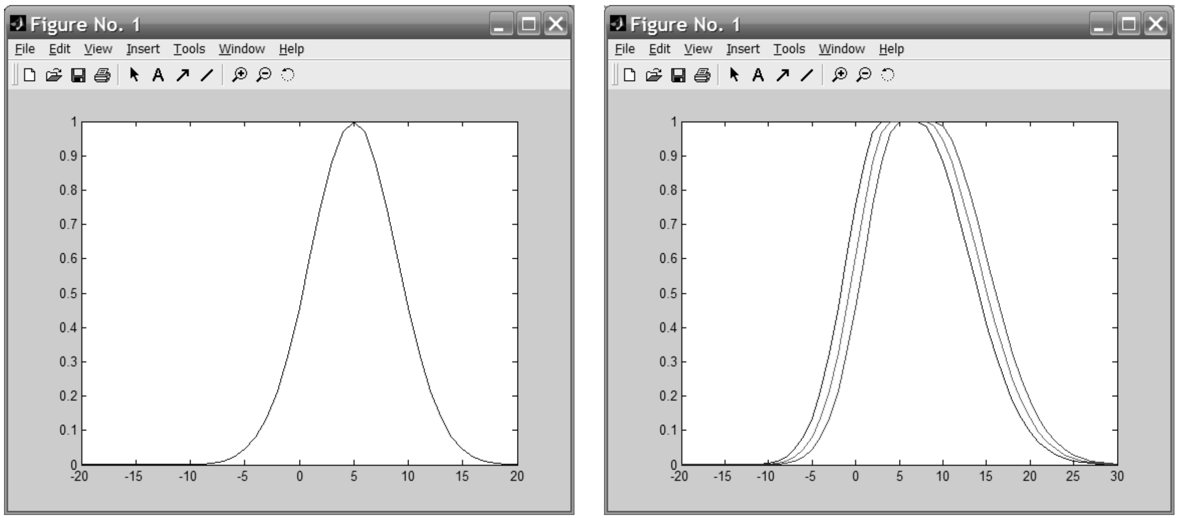
$$y = gaussmf\,(x, [\sigma, c]).$$

Example 5.2. MF *gaussmf* program.

```
» X = -20 : 1 : 20;
» Y = gaussmf (x, [4 5]);
» plot (x, y);
```

Symmetric Gaussian function depends on two parameters σ and *c* (Fig. 5.2, *a*):

$$f(x,\sigma,c)=e^{\frac{-(x-c)^2}{2\sigma^2}}.$$



a *b*

Figure 5.2. Simple (a) and two-sided (b) Gaussian membership functions

Function description:

$$y = gauss2mf\,(x, [\sigma_1, c_1, \sigma_2, c_2]).$$

The last expression is a combination of two different Gaussian distribution functions. The first is determined by parameters σ_1 and c_1 and sets the shape of the left side, and the second one (parameters σ_2, c_2) sets the shape of the MF’s rright side.

If $c_1 < c_2$, then function *gauss2mf* reaches its maximum value at level 1. Otherwise, the maximum value of the function is less than 1 (Fig. 5.2, b).

Example 5.3. MF *gauss2mf* program.

```
» x= [-20 : 30]';
» y1 = gauss2mf (x, [4 3 6 7]);
» y2 = gauss2mf (x, [4 4 6 8]);
» y3 = gauss2mf (x, [4 5 6 9]);
» plot (x,[y1 y2 y3]);
```

The < ' > symbol in the definition line of the base set *x* indicates the transposition of the base set.

Generalized bell-shaped membership function

The next function that allows you to represent fuzzy subjective preferences is the generalized bell-shaped MF (Fig. 5.3) which is denoted *gbellmf* (*generalized bell shape membership function*).

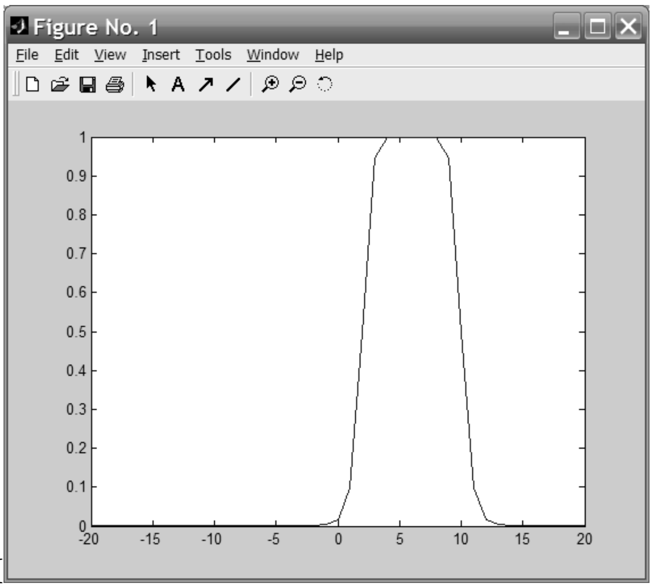
I


Figure 5.3. Generalized bell-shaped membership function

Its difference from the previously considered MF is that a third parameter is added here, which allows a smooth transition between fuzzy sets.

Function description:

$$y = gbellmf\,(x, [a, b, c]).$$

Generalized bell-shaped function depends on three parameters and has the following analytical record:

$$f(x,a,b,c) = \frac{1}{1+\left|\frac{x-c}{a}\right|^{2b}},$$

where *c* determines the location of the MF center; *a* and *b* affect the shape of the curve (Fig. 5.3).

Example 5.4. MF *gbellmf* program.

```
» x= [-20 : 20];
» y = gbellmf (x, [4 5 6]);
» plot (x,y);
```

Membership functions based on the Gaussian distribution function and the generalized bell-shape MF differ by their smoothness and simplicity of writing and are most used in fuzzy sets describing. Despite the fact that Gaussian and bell-shaped MFs possess the property of smoothness, they do not allow the formation of asymmetric MFs. For these purposes, a set of sigmoid functions is provided that can be opened either on the left or on the right, depending on the type of function. Symmetrical and closed functions are synthesized using two additional sigmoids.

Sigmoid membership functions

The main sigmoid MF is designated as *sigmf*, and the additional ones are *dsigmf* and *psigmf*.

Description of the main sigmoid function:

$$y = sigmf\,(x, [a, c]).$$

Analytically, the sigmoid function *sigmf* is written as follows:

$$f\,(x,a,c) = \frac{1}{1+e^{-a(x-c)}}\,.$$

Depending on the sign of the parameter *a*, the considered MF will be open either to the right or to the left (Fig. 5.4, a). This will make it possible to apply it when describing such fuzzy concepts as "very large", "extremely negative", etc.

Description of the additional sigmoid function:

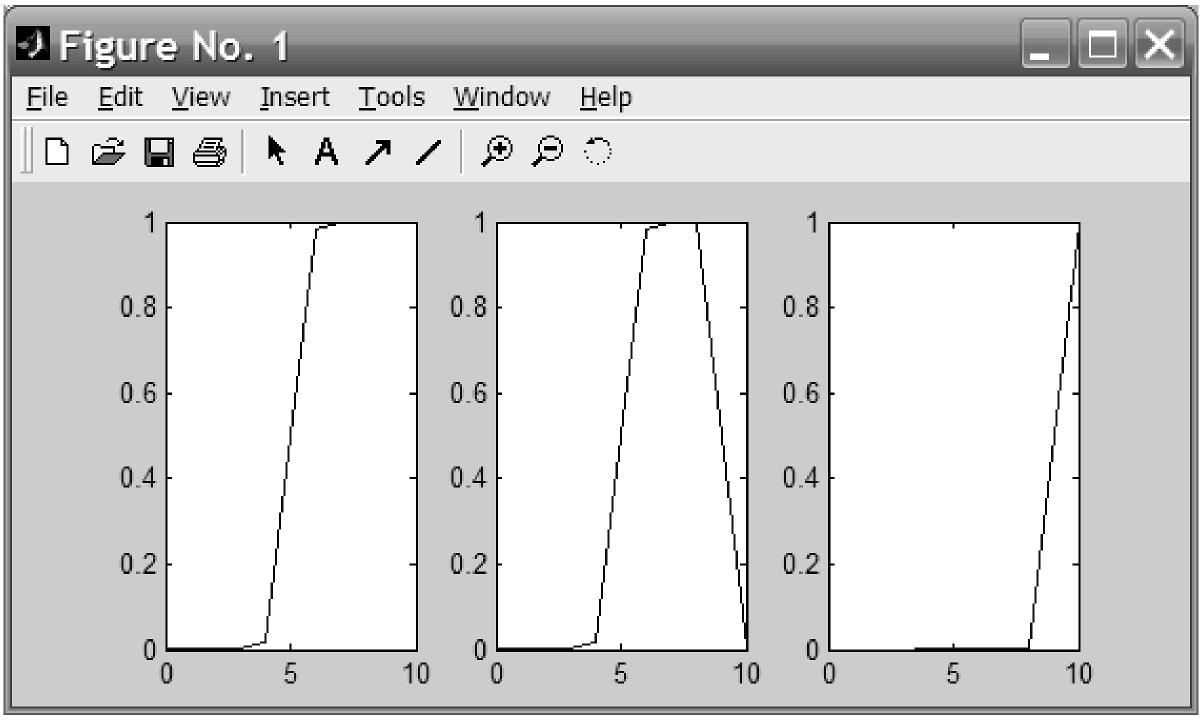
$$y = dsigmf\,(x, ([a_1, c_1, a_2, c_2]) \,.$$

MF *dsigmf* depends on four parameters a_1, c_1, a_2, c_2 and is defined as the difference of two sigmoid functions: $f(x, a_1, c_1) – f(x, a_2, c_2)$ (Fig. 5.4, *b*).

Additional sigmoid function description:

$$y = psigmf\,(x, [a_1, c_1, a_2, c_2]) \,.$$

MF *psigmf*, like the previous function, depends on four parameters a_1, c_1, a_2, c_2 and is defined as the product of two sigmoid functions $f(x, a_1, c_1)$ $f(x, a_2, c_2)$ (Fig. 5.4, *c*).



a *b* *c*

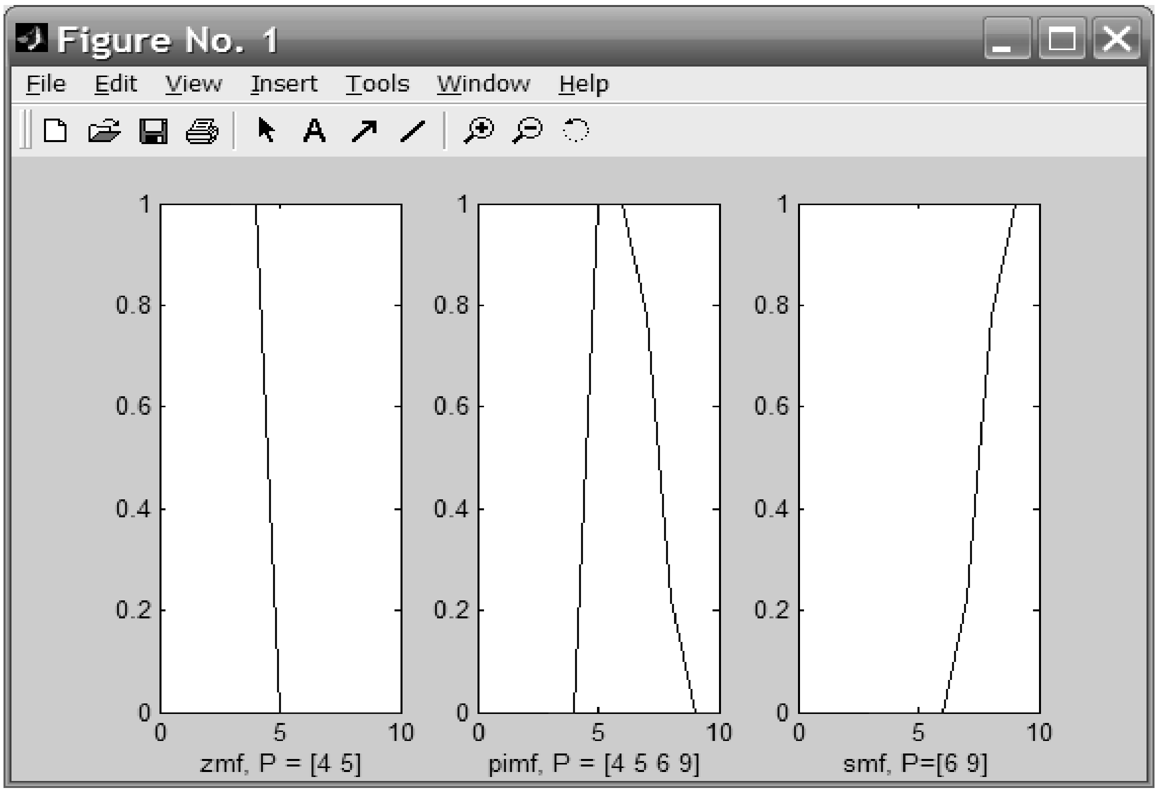
Figure 5.4. Sigmoid membership functions

Example 5.5. Sigmoid functions using program.

```
» x= [0 : 10];
» subplot (1, 3, 1);
» y=sigmf (x,[4 5]);
» plot (x, y);
» subplot (1, 3, 2);
» y = dsigmf (x, [4 5 6 9]);
» plot (x, y);
» subplot (1, 3, 3);
» y = psigmf (x, [4 5 6 9]);
» plot (x, y);
```

Membership functions based on polynomial curves

The *fuzzy logic toolbox* in MATLAB provides the ability to generate MFs based on polynomial curves. The related functions are called *Z*- functions (*zmf*), *PI*- functions (*pimf*) and *S*- functions (*smf*). The function *zmf* is an asymmetric polynomial curve, that is opened on the left (Fig. 5.5, *a*), the function *smf* is a mirror image of the function *zmf* (Fig. 5.5, *c*). Accordingly, the function *pimf* is equal to zero in the right and left limits and takes a value equal to one in the middle of a certain segment (Fig. 5.5, *b*).



a *b* *c*

Figure 5.5. Polynomial membership functions

Function description:

$$y = zmf(x, [a, b]).$$

Parameters *a* and *b* determine the extreme values of the curve (Figure 5.5, *a*).

Function description:

$$y = pimf\,(x, [a, b, c, d]).$$

Parameters *a* and *d* set the function transition to a zero value, and parameters *b* and *c* set the function transition to a unit value (Fig. 5.5, *b*).

Function Description:

$$y = smf\,(x,\,[a,\,b]).$$

Parameters *a* and *b* determine the extreme values of the curve (Fig. 5.5, *c*).

Example 5.6. Polynomial Curve Program.

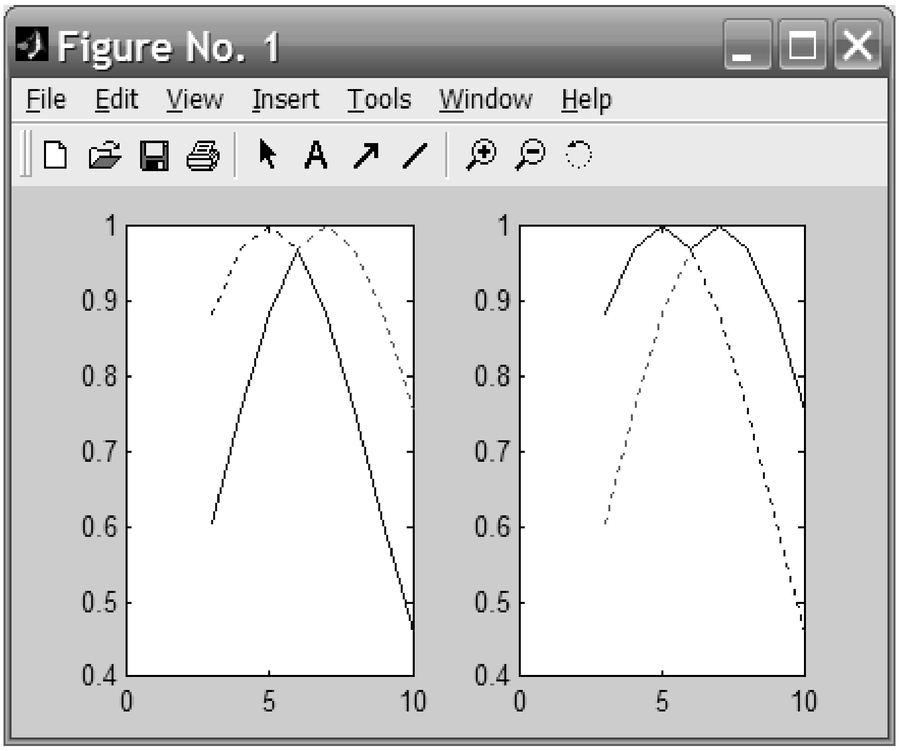
```
» x= [3 : 10];
» subplot(1, 3, 1);
» y = zmf(x, [4 5]);
» plot (x, y);
» xlabel (' zmf, P = [4 5]');
» subplot (1, 3, 2);
» y = pimf(x, [4 5 6 9]);
» plot (x, y);
» xlabel ('pimf, P = [4 5 6 9]');
» subplot (1, 3, 3);
» y = smf (x, [6 9]);
» plot(x, y);
» xlabel ('smf, P=[6 9]')
```

In addition to the functions discussed above that allow you to represent fuzzy sets, MATLAB has the ability to create your own MFs or modify the built-in ones.

5.1.2. Fuzzy set operations

There are three main logical operations with fuzzy sets: conjunction, disjunction and logical negation. In the MATLAB environment, it is possible to define conjunctive and disjunctive operators in terms of minimax and probabilistic interpretations.

Consider the minimax interpretation of logical operators, in which the conjunctive operator represents finding the minimum – *min* (Fig. 5.6, *a*), and the disjunctive operator represents finding the maximum – *max* (Fig.5.6, *b*).



a *b*

Figure 5.6. Intersection (*a*) and union (*b*) of fuzzy sets (minimax interpretation)

Description of conjunctive function: $y = min$ ($[y_1; y_2]$).

Disjunctive function description: $y = max$ ($[y_1; y_2]$).

The parameters *y1* and *y2* represent the initial MFs. The *min* function works with a list of MFs. In MATLAB, the list is formatted with square brackets, and the list elements are separated by semicolons.

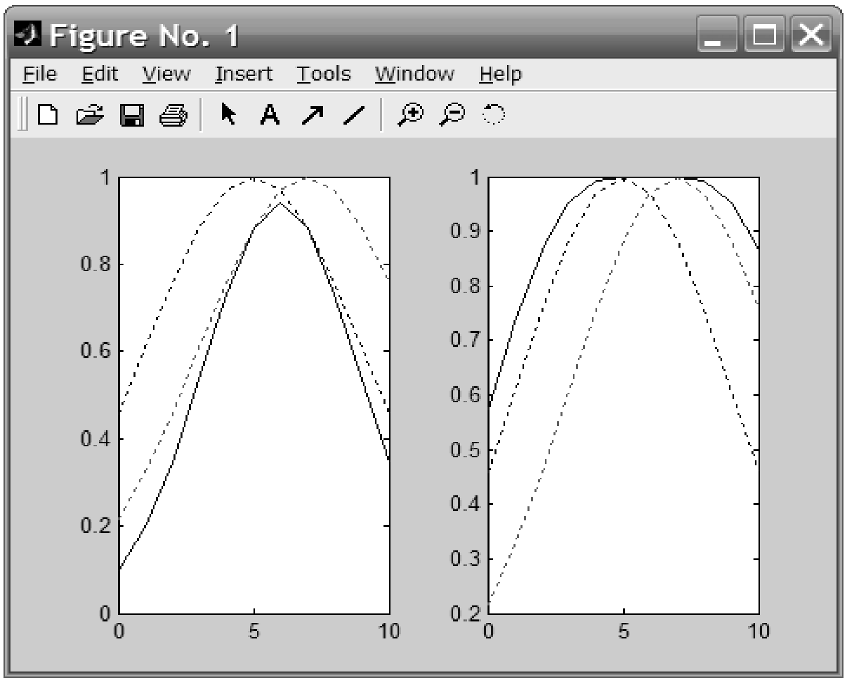
Example 5.7. Program for using *min* and *max* operations.

```
» x = 3 : 0,1 : 10;
» subplot (1, 2, 1);
» y1 = gaussmf (x, [4 5]);
» y2 = gaussmf (x, [4 7]);
» y3 = min ([y1; y2]);
» plot (x, [y1; y2],':');
» hold on;
» plot (x, y3);
» hold off;
» subplot (1, 2, 2);
» y4 = max([y1; y2]);
» plot(x, [y1; y2], ':');
» hold on;
» plot (x, y4);
» hold off.
```

The original MFs are shown with a dashed line on the graphs, and the result of the action of logical operators is shown with a solid line.

The minimax interpretation is the most common in the construction of fuzzy systems. Nevertheless, in practice, an alternative probabilistic interpretation of conjunctive and disjunctive operators is often used. MATLAB provides the corresponding functions.

In the context of this interpretation, the conjunctive operator is the operator for calculating the algebraic product – *prod* (Fig. 5.7, *a*), and the disjunctive operator is the operator for calculating the algebraic sum – *probor* (Fig. 5.7, *b*).



a *b*

Figure 5.7. Intersection (*a*) and union (*b*) of fuzzy sets (probabilistic interpretation)

Function description: y = *prod* ([y_1; y_2])

Function description: y = *probor*([y_1; y_2]).

The parameters *y1* and *y2* are the initial MFs.

Example 5.8. A program for using probabilistic conjunction and disjunction operators.

```
» x = 0 : 0,1 : 10;
» subplot (1, 2, 1);
» y1 = gaussmf (x, [4 5]);
» y2 = gaussmf (x, [4 7]);
» y3 = prod ([y1; y2]);
» plot (x, [y1; y2],':');
» hold on;
» plot(x, y3);
» hold off;
» subplot (1, 2, 2);
» y4 = probor ([y1; y2]);
» plot (x, [y1; y2], ':');
» hold on;
» plot(x, y4);
» hold off.
```

The addition of a fuzzy set is nothing more than a mathematical representation of the verbal expression "NOT *A*" (Fig. 5.8), where *A* is a fuzzy set describing some vague judgment.

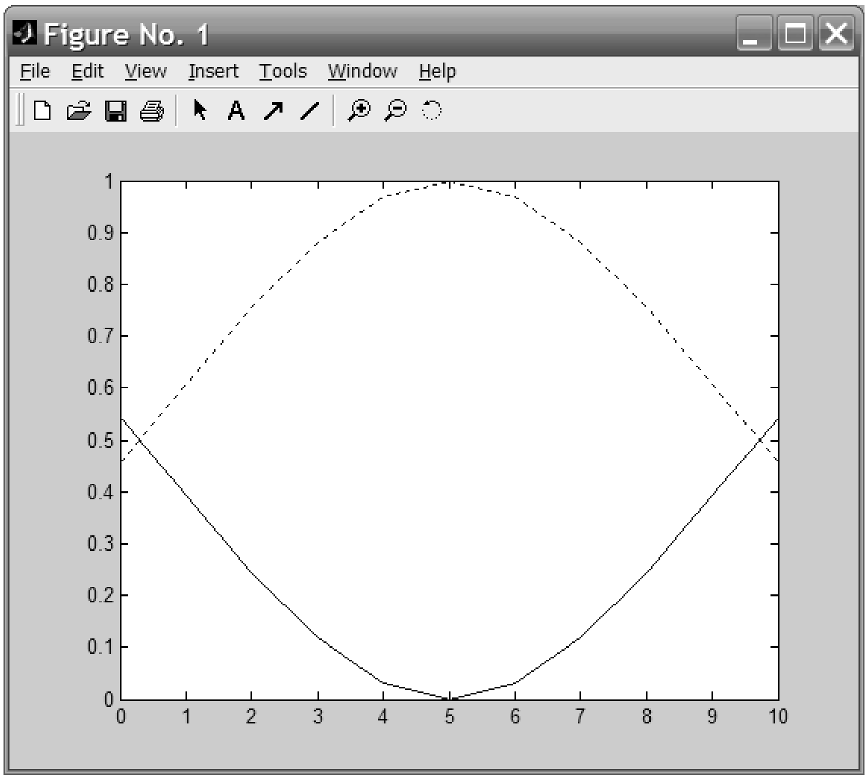


Figure 5.8. A fuzzy set addition

Description of addition function: $y = 1 - y^*$, where y^* is an original MF.

Example 5.9. A program for using addition operation.

```
» x= [0 : 10];
» y1 = gaussmf (x, [3 5]);
» y= 1 - y1;
» plot (x, y1, ':');
» hold on;
» plot(x, y);
» hold off
```

5.2. individual tasks

1. Build a triangular and trapezoidal membership function.

2. Build a simple and two-sided Gaussian membership function formed using various distribution functions.

3. Build a generalized bell-shaped membership function that allows you to represent fuzzy subjective preferences.

4. Build a set of sigmoid functions:

• Main one-sided function, which is open to the left or to the right;

• Additional two-sided function;

• Additional asymmetrical function.

5. Build a set of polynomial membership functions (*Z*-, *PI*- and *S*- functions).

6. Build minimax interpretation of logical operators using minimum and maximum search operations.

7. Build a probabilistic interpretation of conjunctive and disjunctive operators.

8. Build addition of a fuzzy set, which describes a fuzzy proposition and is a mathematical description of verbal expressions, that negates the fuzzy set.

When performing items 1 – 8 of an individual task, the values of variables *a, b, c, d,* etc. must be chosen arbitrarily.

9. Design the lab’s report.

Laboratory work 6

M-FILES AND BASICS OF PROGRAMMING IN MATLAB

The purpose of laboratory work is to obtain and to consolidate knowledge, to form practical skills of working with the MATLAB package when using M-files and executing programs with a nonlinear structure.

6.1. Summary of theory

6.1.1. Work in the M-file editor

Working from the MATLAB command line is difficult if you have to enter many commands and change them frequently. Keeping a diary with the *diary* command and saving the work environment only makes things a little easier.

The most convenient way to execute MATLAB commands is to use M-files, in which you can type commands, execute them all at once or in parts, save them to a file and use them later. The M-file editor is intended for working with M-files. With this editor, you can create your own functions and call them even from the command line.

Open the *File* menu of the main MATLAB window. In the *New* item, select the *M-file* sub-item. The new file opens in the M-file editor window.

Type commands in the editor that lead to the construction of two graphs in one graphic window:

```
» x = [0:0.1:7];
» f = exp(-x);
» subplot(1, 2, 1)
» plot(x, f)
» g = sin(x);
» subplot(1, 2, 2)
» plot(x, g)
```

Now save a file named *mydemo.m* in the *work* subdirectory of the main MATLAB directory by choosing the *Save as* item of the editor's *File* menu. To run all the commands contained in the file for execution, select the *Run* item in the *Debug* menu. The graphic window *Figure No.1* will appear on the screen, containing graphs of functions. If you decide to plot the cosine instead of the sine, then change the line $g = sin(x)$ in the M-file to $g = \cos(x)$ and run all the commands again.

Comment 1

If an error is made while typing and MATLAB cannot recognize the command, then the commands are executed before the incorrectly entered one. After that, an error message is displayed in the command window.

A very convenient feature provided by the M-file editor is the execution of part of the commands. Close the graphical window *Figure No.1*. Select with the mouse, holding the left button, or with the arrow keys while holding down the <*Shift*> key, the first four commands of the program and execute them from the Evaluate *Selection* item of the *Text* menu. Please note that only one graph is displayed in the graphics window, corresponding to the executed commands. Remember that to execute some of the commands, select them and press <*F9*>. Execute the remaining three commands of the program and monitor the state of the graphics window. Practice it yourself, type in any examples from previous labs in the M-file editor and run them.

Individual blocks of the M-file can be provided with comments that are skipped during execution, but are convenient when working with the M-file. Comments in MATLAB start with a percent sign and are automatically highlighted in green, for example:

```
%plotting sin(x) in a separate window
```

Multiple files can be opened in the M-file editor at the same time. The transition between files is carried out using the bookmarks with file names located at the bottom of the editor window.

An existing M-file is opened using the *Open* item of the *File* menu of the working environment, or the M-file editor. You can also open a file in the editor using the MATLAB *edit* command from the command line, specifying the file name as an argument, for example:

```
» edit mydemo
```

The *edit* command without an argument creates a new file.

All examples that are found in this and the following labs are best typed and saved in M-files, supplemented with comments, and executed from the M-file editor. Numerical Methods and Programming in MATLAB requires the creation of M-Files.

6.1.2. Types of M-files

There are two types of M-files in MATLAB: *Script M-Files*, which contain a sequence of commands, and *Function M-Files*, which describe user-defined functions.

You have created a *Script M-File* (*Procedure file*) when you have been reading the previous subsection.

All variables declared in the *Script M-File* become available in the working environment after its execution. Run the *Script M-File* from Section 6.1.1 in the M-file editor.

And type *whos* at the command line to view the contents of the working environment. The description of the variables will appear in the command window:

```
» whos
        Name      Size  Bytes  Class
        f         1x71  568    double array
        g         1x71  568    double array
        x         1x71  568    double array
Grand total  is 213 elements using 1704 bytes
```

Variables defined in one *Script M-File* can be used in other *Script M-Files* and in commands executed from the command line. The execution of the commands contained in the *Script M-File* is carried out in two ways:

1) from the editor of M-files as described above;

2) from the command line or another *Script M-File*, while the name of the M-file is used as a command.

The use of the second way is much more convenient, especially if the created *Script M-File* will be used repeatedly later. In fact, the generated M-file becomes a command that MATLAB understands. Close all graphic windows and type *mydemo* in the command line. A graphic window appears corresponding to the commands in the *mydemo.m Script M-File*. After entering the command *mydemo* MATLAB does the following:

– checks if the entered command is the name of any of the variables defined in the working environment. If a variable is entered, then its value is displayed;

– if no variable is entered, then MATLAB searches for the entered command among the built-in functions. If the command turns out to be a built-in function, then it is executed.

If a non-variable and non-built-in function is entered, MATLAB starts searching for an M-file with the command name and *.m* extension. The search starts from the Current Directory. If the M-file is not found in it, then MATLAB searches the directories set in the search Path. The found M-file is executed in MATLAB.

If none of the above actions led to success, then a message is displayed in the command window, for example:

```
» mydem
??? Undefined function or variable 'mydem'.
```

Typically, M-files are stored in the user's directory. For MATLAB to find them, you must set the paths that indicate the location of the M-files.

Comment 2

There are two reasons why you should keep your own M-files outside the main MATLAB directory. First, when you reinstall MATLAB, files that are contained in subdirectories of the main MATLAB directory may be destroyed. Second, when MATLAB starts up, all the files in the toolbox subdirectory are allocated in the computer memory in some optimal way to increase performance. If you wrote the M-file to this directory, then you can use it only after MATLAB restarting.

6.1.3. Paths setting

Starting with version 6 of MATLAB, it is possible to define the current directory and search path. These properties are set either using the appropriate dialog boxes, or using commands from the command line.

The current directory is determined in the *Current Directory* dialog box of the working environment. The window is present in the working environment if the *Current Directory* item of the *View* menu is selected.

The current directory is selected from the list. If it is not in the list, then it can be added from the *Browse for Folder* dialog box, which is called by clicking the button to the right of the list. The contents of the current directory are displayed in the file table.

Search paths are defined in the *Set Path* dialog box of the path navigator, which is accessed from the *Set Path* item of the *File* menu of the working environment.

To add a directory, click the *Add Folder* button. In the resulting *Browse for Path* dialog box, select the desired directory. Adding a directory with all its subfolders is carried out by clicking the *Add with Subfolders* button. The path to the added directory appears in the MATLAB *search path* field. The search order corresponds to the location of the paths in this field. The first one to look at is the directory, the path to which is located at the top of the list. The search order can be changed or the path to a directory can be deleted. To do this, select the directory in the MATLAB *search path* field and define its position using the following buttons:

Move to Top – поместить вверх списка;

Move Up – move up one position;

Remove – remove from the list;

Move Down – move down one position;

Move to Bottom – move to the bottom of the list.

After making changes, you should save the information about the search paths by clicking the *Save* button. Using the *Default* button, you can restore the default settings, and *Revert* is used to return to the saved ones.

Commands for paths setting

The steps for paths setting in MATLAB 6.x are duplicated by commands. The current directory is set with the *cd* command, for example *cd c:\users\igor*. The *cd* command, called without an argument, display the path to the current directory. To set paths, use the *path* command, called with two arguments:

path (*path*, '*c:\users\igor*') – adds directory *c:\users\igor* with lowest search priority;

path ('*c: \users\igor*',*path*) – adds directory *c:\users\igor* with highest search priority.

Using the *path* command with no arguments results in a list of search paths displayed on the screen. You can remove a path from the list using the *rmpath* command:

rmpath (*'c:\users\igor'*) removes the path to the directory *c:\users\igor* from the list of paths.

Comment 3

Do not delete paths to directories unnecessarily, especially those whose purpose you are not sure about. Removal may cause some of the functions defined in MATLAB to become unavailable.

Example 6.1. Create in the root directory of drive *D* (or any other drive or directory where students are allowed to create their own directories) a directory with your last name, for example, *WORK_IVANOV* and write the M-file *mydemo.m* under the name *mydemo3.m* there. Set the paths to the file and demonstrate that the file is accessible from the command line. Report the results in the lab report.

Solution option:

1. A directory *WORK_IVANOV* is created in the root directory of drive D.
2. The M-file *mydemo.m* is written to the *WORK_IVANOV* directory under the name *mydemo3.m.*
3. Opens the *Set Path* dialog of the *File* menu of the working enviroment MATLAB.
4. The *Add Folder* button is clicked, and a directory *WORK_IVANOV* is selected in the *Browse for Path* dialog box that appears.
5. Adding a directory with all its subfolders is carried out by clicking the *Add with Subfolders* button. The path to the added directory appears in the MATLAB *search path* field.

6. To store the path, press the *Save* key of the *Set Path* dialog box.

7. All actions are checked for correctness by typing the *mydemo3* command from the command line. The graphic window *Figure No.1* containing graphs of functions will appear on the screen.

The above Script M-Files are a sequence of MATLAB commands; they have no input and output arguments. To use numerical methods and when programming your own applications in MATLAB, you must be able to compose Function M-Files that perform the necessary actions with input arguments and return the result in output arguments. In this subsection, a few simple examples are discussed to help you understand how to work with Function M-Files. Function M-Files, like Script M-Files, are created in the M-file editor.

6.1.4. Function M-Files

Function M-Files with one input argument

Suppose that in calculations it is often necessary to use the function

$$e^{-x}\sqrt{\frac{x^2+1}{x^4+0{,}1}}.$$

It is advisable to write a Function M-File once, and then call it wherever it is necessary to calculate this function. Open a new file in the M-file editor and type in the listing text

```
function f = myfun(x)
f= exp(-x)*sqrt((x^2+1)/(x^4+0.1));
```

The word *function* on the first line specifies that the file contains a Function M-File. The first line is the function header, which contains the function name and lists of input and output arguments. In the example shown in the listing, the function name is *myfun*, one input argument is *x* and one output argument is *f*. The header is followed by the body of the function (in this example it consists of one line), where its value is calculated. It is important that the calculated value is written to *f*. A semicolon is supplied to prevent unnecessary information from being displayed on the screen

Now save the file in your working directory. Please note that choosing the *Save* or *Save as* item of the *File* menu brings up a file save dialog box, the *File name* field of which already contains the name *myfun*. Do not change it, save the Function M-File in a file with the suggested name.

Now the created function can be used in the same way as the built-in *sin*, *cos* and others, for example from the command line:

```
» y = myfun(1.3)
y =
    0.2600
```

Own functions can be called from a Script M-File and from another Function M-File.

Warning 1

The directory containing the Function M-Files must be current, or the path to it must be added to the search paths, otherwise MATLAB will not find the function or call another function with the same name instead (if it is in searchable directories).

The Function M-File shown in the listing has one significant drawback. Attempting to compute function values from an array results in an error, not an array of values, as happens when evaluating built-in functions

```
» x = [1.3 7.2];
» y = myfun(x)
??? Error using ==> ^
```

```
Matrix must be square.
Error in ==> C:\MATLABR11\work\myfun.m
On line 2 ==> f = exp(-x)*sqrt((x^2+1)/(x^4+1));
```

If you have studied working with arrays, then eliminating this disadvantage will not cause difficulties. It is necessary to use element-wise operations when calculating the value of a function.

Modify the body of the function as shown in the following listing (don't forget to save the changes in the *myfun.m* file)

```
function f = myfun(x)
f = exp(-x).*sqrt((x.^2+1)./(x.^4+0.1));
```

Now the argument of the *myfun* function can be either a number, or a vector or a matrix of values, for example:

```
» x = [1.3 7.2];
» y = myfun(x)
y =
    0.2600  0.0001
```

The variable *y*, into which the result of calling the *myfun* function is written, automatically becomes a vector of the required size.

Plot the function *myfun* on the segment [0, 4] from the command line or using a Script M-File:

```
x = [0:0.5:4];
y = myfun(x);
plot(x, y)
```

MATLAB provides another option for working with Function M-Files. This is using them as arguments to some commands. For example, a special function *fplot* is used to plot a graph, which replaces the sequence of commands given above. When calling *fplot*, the name of the function whose graph you want to plot is enclosed in apostrophes, the plotting limits are specified in a two-element row vector *fplot*('*myfun*', [0, 4]).

Plot *myfun* graphs with *plot* and *fplot* on the same axes, with *hold on*. Please note that the graph plotted with *fplot* more accurately reflects the behavior of the function, since *fplot* selects the step of the argument, decreasing it in the areas of fast change of the displayed function. Report the results in the laboratory report

Function M-Files with multiple input arguments

Writing Function M-Files with multiple input arguments is almost the same as writing Function M-Files with one argument. All input arguments are placed in a comma-separated list. For example, the following listing contains a Function M-File that calculates the length of the radius vector of a point in three-dimensional space $\sqrt{x^2+y^2+z^2}$.

Function M-File with multiple input arguments listing

```
function r = radius3(x, y, z)
r = sqrt(x.^2 + y.^2 + z.^2);
```

You can now use the *radius3* function to calculate the length of the radius vector, for example:

```
» R = radius3(1, 1, 1)
R =
    1.732
```

In addition to functions with multiple input arguments, MATLAB allows you to create functions that return multiple values, i.e. having multiple output arguments.

Function M-File with multiple output arguments

Function M-Files with multiple output arguments are useful for evaluating functions that return multiple values (called vector functions in mathematics). Output arguments are added comma-separated to the list of output arguments, and the list is enclosed in square brackets. A good example is a function that converts time in seconds into hours, minutes, and seconds. This Function M-File is shown in the following listing.

Listing of the function for converting time in seconds into hours, minutes and seconds

```
function [hour, minute, second] = hms(sec)
hour = floor(sec/3600);
minute = floor((sec-hour*3600)/60);
second = sec-hour*3600-minute*60;
```

When calling Function M-Files with multiple output arguments, the result should be written to a vector of the appropriate length

```
» [H, M, S] = hms(10000)
H =
    2
M =
    46
S =
    40
```

6.1.5. Fundamentals of Programming in MATLAB

The Function M-Files and Script M-Files used in the previous subsections are the simplest examples of programs. All MATLAB commands contained in them are executed sequentially. To solve many more serious problems, it is necessary to write programs in which actions are performed cyclically or, depending on certain conditions, various parts of the programs are executed. Consider the basic operators that specify the sequence of execution of MATLAB commands. Operators can be used in both Script M-Files and Function M-Files. This allows you to create programs with a complex branched structure.

Loop operator for

The operator is designed to perform a specified number of repetitive actions. The simplest use of the *for* operator is as follows:

```
for count = start:step:final
% Next is the text of the program consisting of MATLAB commands

end
```

Here *count* is a loop variable, *start* is its initial value, *final* is its final value, and *step* is the step by which *count* is incremented each time the loop is entered. The loop ends as soon as *count* becomes greater than *final*. The loop variable can take not only integers, but also real values of any sign. Let's analyze the use of the *for* loop operator using some typical examples.

Let it be required to derive a family of curves for $x \in [0, 2\pi]$, which is given by a function depending on the parameter $y(x,a) = e^{-ax}\sin x$, for parameter values from –0,1 to 0,1.

Type Script M-File in the M-file editor and save to file *FORdem1.m*, and run it (from the M-file editor or from the command line by typing the *FORdem1* command in it and pressing <*Enter*>):

```
% Script M-File for plotting a family of curves
x = [0:pi/30:2*pi];
for a = -0.1:0.02:0.1
    y = exp(-a*x).*sin(x);
    hold on
    plot(x, y)
end
```

As a result of *FORdem1* execution, a graphic window will appear (Fig. 6.1), which contains the required family of curves.

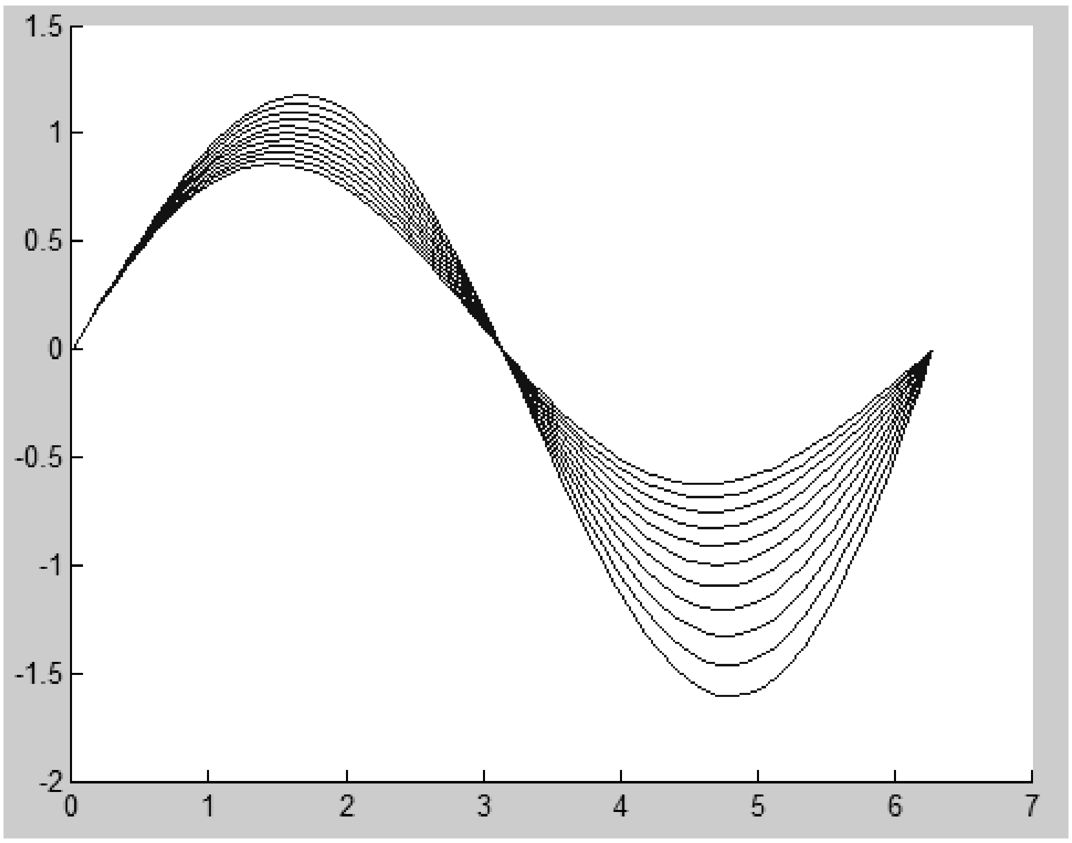


Figure 6.1. Execution result of *FORdem1*

Comment 4

The M-file editor automatically offers to place operators inside the loop, indented from the left edge. Use this opportunity for the convenience of working with the text of the program.

Develop a Script M-File to calculate the sum

$$S = \sum_{k=1}^{10} \frac{1}{k!}.$$

The algorithm for calculating the sum uses the accumulation of the result, i.e. at first the sum is zero ($S = 0$), then 1 is entered into the variable k, the $1/k!$ is calculated, it is added to S and the result is again entered into S. Then k increases by one, and the process continues until the last term becomes 1/10!. Script M-File *Fordem2*, shown in the following listing, calculates the required amount.

Listing of the Script M-File *Fordem2* for calculating the sum

```
% SCRIPT M-FILE FOR CALCULATING THE SUM
% 1/1!+1/2!+ … +1/10!

% Zeroing S to accumulate the amount
S = 0;
% accumulation of the amount in the cycle
for k = 1:10
    S = S + 1/factorial(k);
End
% outputting the result to the command window
S
```

Type Script M-File in the M-File editor, save it in the current directory in the *Fordem2.m* file and execute it. The result will be displayed in the command window, because the last line of the Script M-File *S* contains without a semicolon to display the value of the variable *S*

```
S =
    1.7183
```

Please note that the rest of the lines of the Script M-File, which could result in the display of intermediate values, are terminated with a semicolon to suppress the output to the command window.

The first lines with comments are not accidentally separated by an empty line from the rest of the program text. They are the ones that are displayed when the user, using the *help* command from the command line, receives information about what *Fordem2* is doing.

```
» help Fordem2
 SCRIPT M-FILE TO CALCULATE SUM
 1/1!+1/2!+ … +1/10!
```

When developing Script M-file and Function M-file, do not neglect comments!

All variables used in the Script M-file are made available in the working environment. They are so-called global variables. On the other hand, Script M-file can use all variables entered in the working environment.

Consider the problem of calculating the sum, similar to the previous one, but depending on the variable *x*

$$S = \sum_{k=1}^{10} \frac{x^k}{k!}.$$

To calculate this amount in Script M-file *Fordem2*, you need to change the line inside the *for* loop to

```
S = S + x.^k/factorial(k);
```

Before running the program, you must define the *x* variable on the command line using the following commands:

```
» x = 1.5;
» Fordem2
S =
    3.4817
```

A vector or a matrix can be used as *x*, since in Script M-file *Fordem2,* elementwise operations were used when accumulating the sum.

Before starting *Fordem2*, you must assign a value to the *x* variable. And to calculate the sum, for example, of fifteen terms, you will have to make changes to the text of the Script M-file. It is much better to develop a generic Function M-file that takes the value of *x* and the upper limit of the sum as input arguments, and the value of the sum *S(x)* as its output arguments. The *sumN* Function M-file is shown in the following listing.

Listing of Function M-file for calculating the sum

```
function S = sumN(x, N)
% Function M-file for calculating the sum
% x/1!+x^2/2!+ … +x^N/N!
% using: S = sumN(x, N)

% zeroing S to accumulate the sum
S = 0;
```

```
% accumulation of the amount in the cycle
for m = 1:1:N
     S = S + x.^m/factorial(m);
end
```

The user can learn about using the *sumN* function by typing *help sumN* at the command line. The command window will display the first three lines with comments, separated from the text of the Function M-file by an empty line.

Note that the variables of Function M-file are not global *(m* in *sumN* Function M-file). Attempting to view the value of *m* from the command line results in a message stating that *m* is undefined. If there is a global variable in the working environment with the same name, defined from the command line or in the Script M-file, then it has nothing to do with the local variable in the Function M-file. As a rule, it is better to design your own algorithms in the form of a Function M-file so that the variables used in the algorithm do not change the values of the same global variables of the working environment.

For loops can be nested, while the variables of the nested loops must be different.

The *for* loop is useful for performing repetitive, similar actions when the number is predetermined. A more flexible *while* loop allows you to work around this limitation.

Loop operator while

Let's consider an example for calculating the sum, similar to the example from the previous paragraph. It is required to calculate the sum of a series for a given *x* (expansion in a series $\sin x$):

$$S(x)=\sum_{k=0}^{\infty}(-1)^k\,\frac{x^{2k+1}}{(2k+1)!}.$$

The sum can be accumulated as long as the terms are not too small, for example, more in modulus 10^{-10}. The *for* loop is not enough here, since the number of terms is not known in advance. The solution is to use a *while* loop that runs as long as the loop condition is met:

while loop condition

    MATLAB commands

end

In this example, the loop condition provides that the current term $x^k / k!$ is more than 10^{-10}. A greater sign is used to write this condition. (>). Text of *mysin* Function M-file, calculating the sum of a series, is shown in the following listing.

Listing of *mysin* Function M-file calculating sine by series expansion

```
function S = mysin(x)
% Calculating sine by series expansion
% Using: y = mysin(x), -pi<x<pi
S = 0;
k = 0;
while abs(x.^(2*k+1)/factorial(2*k+1))>1.0e-10
    S = S + (-1)^k*x.^(2*k+1)/factorial(2*k+1);
    k = k + 1;
end
```

Note that the *while* loop, unlike *for*, does not have a loop variable, so we had to assign zero to *k* before the loop began, and inside the loop we increased *k* by one.

The *while* loop condition can contain more than only the > sign. To set the conditions for the execution of the cycle, other relationship operations are also allowed, given in table 6.1.

Table 6.1 – Relationship operations

Designation	Relationship operation
==	Equality
<	Less
>	More
<=	Less or equal
>=	More or equal
~=	Not equal

More complex conditions are specified using logical operators. For example, the condition $-1 \le x < 2$ consists in the simultaneous fulfillment of two inequalities $x \ge -1$ фтв $x < 2$ and is written using the logical operator *and*

```
and(x >= -1, x < 2)
```

or equivalently with the symbol &

```
(x >= -1) & (x < 2)
```

Logical operators and examples of their use are given in table 6.2.

Table 6.2 – Logical operators

Operator	Condition	MATLAB record	Equivalent form
Logical "AND"	$x < 3$ и $k = 4$	*and*(*x* < 3, *k* == 4)	(*x* < 3) & (*k* == 4)
Logical "OR"	$x = 1,\ 2$	*or*(*x* == 1,*x* == 2)	(*x* == 1) | (*x* == 2)
Denying "NOT"	$a \ne 1.9$	*not*(*a* == 1.9)	~(*a* == 1.9)

When calculating the sum of an infinite series, it makes sense to limit the number of terms. If the series diverges due to the fact that its members do not tend to zero, then the condition for a small value of the current term may never be fulfilled and the program will loop. Perform summation by adding a limit on the number of terms to the *while* loop condition of the *mysin* Function M-file:

```
while(abs(x.^(2*k+1)/factorial(2*k+1))>1.0e-10)&
(k<=10000))
```

or in equivalent form

```
while   and(abs(x.^(2*k+1)/factorial(2*k+1))>1.0e-10),
k<=10000)
```

The organization of repetitive actions in the form of loops makes the program simple and understandable; however, it is often required to execute one or another block of commands depending on certain conditions, i.e. use algorithm branching.

Conditional operator if

The conditional *if* operator allows you to create a branching algorithm of command execution in which, when certain conditions are met, the corresponding block of MATLAB operators or commands runs.

The *if* operator can be used in its simple form to execute a block of commands when a certain condition is satisfied, or in an *if-elseif-else* construction to write branching algorithms.

Let it be required to calculate the expression $\sqrt{x^2-1}$. Suppose you are performing calculations in the realm area and you want to display a warning that the result is a complex number. Before evaluating the function, check the value of the argument *x* and display a warning in the command window if the module *x* does not exceed one. Here it is necessary to use the conditional operator *if*, the application of which in the simplest case looks like this:

```
if condition
        commands MATLAB
end
```

If the condition is met, then the MATLAB commands placed between the *if* and *end* are implemented, and if the condition is not met, then the transition to the commands located after the *end* occurs. When recording a condition, the operations shown in table 6.1 are used.

Function M-file, which checks the argument value, is shown in the following listing. The *warning* command is used to display a warning in the command window.

Listing of the *Rfun* Function M-file that checks the value of the argument

```
function f = Rfun(x)
% calculates sqrt(x^2-1)
% displays a warning if the result is complex
% using y = Rfun(x)
% argument validation
if abs(x)<1
    warning(' the result is complex
')
end
% function computation
f = sqrt(x^2-1);
```

Now calling *Rfun* from an argument which is less than one will lead to displaying a warning to the command window:

```
» y = Rfun(0.2)
the result is complex
```

```
y =
     0 + 0.97979589711327i
```

The *Rfun* Function M-file only warns that its value is complex, and all calculations with it continue. If the complex result means a calculation error, then you should stop the execution of the function using the *error* command instead of *warning*.

Branching operator if-elseif-else

In general, the application of the branch operator *if-elseif-else* is as follows:

```
if condition 1
        commands MATLAB
elseif condition 2
        commands MATLAB
………………………
elseif condition N
        commands MATLAB
else
        commands MATLAB
end
```

Depending on the fulfillment of one or another of the *N* conditions, the corresponding branch of the program operates. If none of the *N* conditions are met, then MATLAB commands placed after *else* are implemented. After execution of any of the branches, the operator exits. There can be as many branches as you want, or only two. In the case of two branches, the trailing *else* is used and the *elseif* is skipped. A branch operator must always end with an *end* operator.

Example of using the operator *if-elseif-else*, in which the value of the variable *a* is analyzed and a message about the *a* value is displayed, is shown in the following listing

```
function ifdem(a)
% example of using the operator if-elseif-else
if (a == 0)
    warning('a is equal to zero)
elseif a == 1
    warning('a is equal to one')
elseif a == 2
    warning('a is equal to two')
elseif a >= 3
    warning('a is greater than or equal to three ')
else
    warning('a is less than three, and not equal to
zero, one, two')
end
```

Branching operator switch

A *switch* operator can be used to perform multiple selection or branching. It is an alternative to the operator *if-elseif-else*. In general, the application of the branching operator *switch* is as follows:

```
switch expression
    case value 1
        MATLAB commands
    case value 2
        MATLAB commands
………………………………
```

```
    case value N
        MATLAB commands
    case { value N+1, value N+2, …}
        MATLAB commands
……………………………………
    case { value NM+1, value NM+2,…}
    otherwise
        MATLAB commands
end
```

This operator first evaluates the value of the expression (it can be a scalar numeric value or a string of characters). This value is then compared with the values: value 1, value 2, …, value *N*, value *N*+1, value *N*+2, …, value *NM*+1, value *NM*+2,… (which can also be numeric or string). If a match is found, then the MATLAB commands after the corresponding keyword *case* are executed. Otherwise, MATLAB commands between the *otherwise* and *end* keywords are executed.

There can be any number of lines with the *case* keyword, but there must be one line with the *otherwise* keyword.

After executing any of the branches, the *switch* is exited, while the values specified in other *case* are not checked.

The use of *switch* clarifies the following example:

```
function demswitch(x)
a = 10/5 + x
switch a
    case -1
        warning('a = -1')
    case 0
```

```
          warning('a = 0')
     case 1
          warning('a = 1')
     case {2, 3, 4}
          warning('a equals 2 or 3 or 4')
otherwise
          warning('a is not equal to -1, 0, 1, 2, 3,
4')
end

» x = -4
demswitch(x)
a = -2
» x = 1
demswitch(x)
a = 3
Warning:a equals 2 or 3 or 4
```

Cycle interruption operator break

When organizing cyclic computations, care should be taken to avoid errors inside the loop. For example, suppose you are given an array *x* consisting of integers, and you need to form a new array *y* according to the rule $y(i) = x(i + 1) / x(i)$.

Obviously, the problem can be solved using the *for* loop. But if one of the elements of the original array is equal to zero, then the division will result in *inf*, and subsequent calculations may be useless.

This situation can be prevented by exiting the loop if the current value *x(i)* is equal to zero. The following code fragment demonstrates the use of the *break* operator to interrupt a loop:

```
for x = 1:20
z = x-8;
    if z==0
        break
    end
    y = x/z
end
```

As soon as *z* becomes 0, the loop is aborted.

The *break* operator allows you to prematurely terminate the execution of *for* and *while* loops. Outside of these loops, the *break* operator does not work.

If the *break* operator is used in a nested loop, then it exits only from the inner loop.

6.2. Individual tasks

1. Create M-file *mydemo.m*.

2. Create in the root directory of drive *D* (or any other drive or directory where students are allowed to create their own directories) a directory with your last name, for example, *WORK_IVANOV*, and create the M-file *mydemo.m* under the name *mydemo3.m* there. Set the paths to the file and demonstrate that the file is accessible from the command line.

3. Plot the Function M-file *myfun* using the *plot* and *fplot* commands on the same axes (using *hold on*).

4. Develop the Function M-file *root2*, which only calculates the real roots of a quadratic equation (and gives an error if there are complex roots). In the demos, the second coefficient of the quadratic equation should be equal to your group jornal number.

5. Develop the Function M-file, that calculates the greatest common divisor (GCD) *z* of two natural numbers *x* and *y* using Euclid's algorithm. In one

of the demo examples the greatest common divisor should be 3*N*+1, where *N* is your group jornal number.

Reference Information. The idea behind Euclid's algorithm is based on the fact that if $z = \text{GCD}(x,y)$, then if the numbers are *x* and *y* equal GCD *z* coincides with *x* and *y*, and in the case of inequality of numbers *x* and *y* their difference between larger and smaller, together with the smaller number, has the same greatest common divisor. The algorithm for determining Euclid's GCD can be written as follows:

Step 1. If $x > y$, then go to *step 4.*

Step 2. If $x < y$, then go to *step 5,* otherwise go to *step 3.*

Step 3. $z = x$. The end.

Step 4. Subtract *y* from *x* and assume that this difference is now equal to the value of *x*. Go to *step 1.*

Step 5. Subtract *x* from *y* and assume that this difference is now equal to the value of *y*. Go to *step 1.*

6. Develop Function M-file, that calculates prime numbers not exceeding 150+10*N*, where *N* is your group jornal number.

Reference Information. Positive integers are called prime numbers that are greater than one, which are divisible only by themselves and one without a remainder. One of the simplest algorithms for obtaining primes not exceeding *n* is the algorithm of Eratosthenes, called the sieve of Eratosthenes. It consists of the following steps:

Step 1. Write out sequentially all integers, starting with two and ending with *n*.

Step 2. Specifies the number of $p = 2$.

Step 3. If $p^2 \le n$, then go to *step 4*, otherwise go to *step 6*.

Step 4. In a sequence of numbers, starting with the number*p* + 1, cross out all numbers that are multiples of p, not paying attention to the fact that some of the numbers may have already been crossed out.

Step 5. Consider the first uncrossed number of the sequence after the number *p* as the new value of the number *p*. Return *to step 3* of the algorithm.

Step 6. The process is over. All uncrossed numbers in the sequence are prime.

7. Calculate the sum of the prime numbers found in the task 6.

8. Design a laboratory report.

Laboratory work 7

FUZZY CLUSTERING ALGORITHM RESEARCH

The purpose of laboratory work is to obtain and consolidate knowledge about the fuzzy clustering algorithm, to form practical skills for solving clustering problems using fuzzy logic methods.

7.1. Summary of theory

7.1.1. FCM- clustering algorithm

The fuzzy clustering algorithm is called *FCM*- algorithm (*Fuzzy Classifier Means*, *Fuzzy C-Means*). The purpose of the FCM clustering algorithm is to automatically classify a set of objects that are specified by feature vectors in the feature space. In other words, such an algorithm identifies clusters and classifies objects accordingly. Clusters are represented by fuzzy sets, and furthermore, the boundaries between clusters are also fuzzy.

The *FCM*-clustering algorithm assumes that objects belong to all clusters with a specific MF. The degree of belonging is determined by the distance from the object to the corresponding cluster centers. This algorithm iteratively calculates the centers of clusters and new degrees of belonging of objects.

For a given number K of input vectors x_k $(k = \overline{1,K})$ and N allocated clusters c_j $(j = \overline{1,N})$ any x_k belongs to any c_j $(j = \overline{1,N})$ with membership $\mu_{jk} \in [0,1]$, where j is the cluster number, and k is the input vector number.

The following standardization conditions are taken into account for μ_{jk}:

$$\sum_{j=1}^{N} \mu_{jk} = 1, \ \forall k = 1,\ldots,K;$$

$$0 < \sum_{k=1}^{N} \mu_{jk} \le K, \ \forall j = 1,\ldots,N.$$

The goal of the algorithm is to minimize the sum of all weighted distances $\|x_k - c_j\|$

$$\sum_{j=1}^{N}\sum_{k=1}^{K}(\mu_{jk})^q \|x_k - c_j\| \to \min,$$

where q is fixed parameter set before iterations.

To achieve the above goal, it is necessary to solve the following system of equations:

$$\partial\big/\partial\mu_{jk}\left(\sum_{j=1}^{N}\sum_{k=1}^{K}(\mu_{jk})^q \|x_k - c_j\|\right) = 0, \quad j=\overline{1,N},\ k=\overline{1,K},$$
$$\partial\big/\partial c_j\left(\sum_{j=1}^{N}\sum_{k=1}^{K}(\mu_{jk})^q \|x_k - c_j\|\right) = 0, \quad j=\overline{1,N}.$$

Together with the normalization conditions μ_{jk} this system of equations has the following solution:

$$c_j = \frac{\sum_{j=1}^{N}(\mu_{jk})^q \cdot x_k}{\sum_{k=1}^{K}(\mu_{jk})^q},\ j=\overline{1,N}.$$

(weighted center of gravity) and

$$\mu_{jk} = \frac{1\Big/\|x_k - c_j\|^{1/(q-1)}}{\sum_{j=1}^{N}\left(1\Big/\|x_k - c_j\|^{1/(q-1)}\right)},\ j=\overline{1,N},\ k=\overline{1,K}.$$

The fuzzy clustering algorithm is performed step by step.

Step 1. Initialization.

The following options are selected:

– the required number of clusters$N$, $2 < N < K$;

– the type of distance (for example, Euclidean distance);

– fixed parameter q (usually1,5);

– the initial (at the zero iteration) matrix of the membership functions $U^{(0)} = (\mu_{jk})^{(0)}$ of objects x_k $(k = \overline{1,K})$ taking into account the given initial centers of the clusters c_j $(j = \overline{1,N})$.

Step 2. Clusters centers positions $c_j^{(t)}$ regulation.

At the t-th iteration step, with a known matrix $\mu_{jk}^{(t)}$ the $c_j^{(t)}$ is calculated in accordance with the above solution to the system of equations.

Step 3. Correction of membership values μ_{jk}.

Given the known $c_j^{(t)}$, $\mu_{jk}^{(t)}$ are calculated if $x_k \neq c_j$, otherwise:

$$\mu_{jk}^{(t+1)} = \begin{cases} 1, \text{ if } k = j, \\ 0, \text{ if } k \neq j. \end{cases}$$

Step 4. Stopping the algorithm.

The fuzzy clustering algorithm stops when the following condition is met:

$$\left\| U^{(t+1)} - U^{(t)} \right\| \leq \varepsilon,$$

where$\|\ \ \|$ is a matrix norm (for example, Euclidean norm); ε is preset level of accuracy.

7.1.2. Solving clustering problems

There are two ways to solve clustering problems in MATLAB: using the command line or using the graphical user interface. Let's consider the first of these methods.

To find the centers of clusters, MATLAB has a built-in function *fcm*, which is described below.

Function Description: $[center, U, obj\_fcn] = fcm(data, cluster\_n)$.

The arguments to this function are:

1) *data* is a set of data to be clustered, each line describes a point in a multidimensional space of characteristics;

2) *cluster_n* is the number of clusters (more than one).

The function returns the following parameters:

1) *center* is a matrix of cluster centers, each row of which contains the coordinates of the center of an individual cluster;

2) *U* is the resulting matrix of the membership function;

3) *obj_fcn* is the value of the objective function at each iteration.

Example 7.1. Fuzzy clustering program.

```
//loading data to be clustered from a file
» load fcmdata.dat;
// determination of the clustering center (two
clusters)
» [center, U, obj_fcm] = fcm( fcmdata, 2);
//determination of the maximum degree of membership
//of an individual data item in a cluster
»maxU = max(U);
//distribution of data matrix rows between
//corresponding clusters
»index1 = find (U(1, :) == maxU);
»index2 = find(U(2, :) = maxU);
```

```
//plotting data corresponding to the first cluster
»plot  (fcmdata  (index1,  1),  fcmdata  (index1,  2),'
ko', 'markersize', 5, 'LineWidth' ,1);
»hold on
//plotting data corresponding to the second cluster
»plot(fcmdata  (index2, 1), fcmdata(index2, 2), 'kx',
'markersize', 5, 'LineWidth', 1);
// plotting cluster centers
»plot(center(1, 1), center(1, 2), 'ko', 'markersize',
15, 'LineWidth', 2)
»plot   (center   (2,   1),   center   (2,   2),   'kx',
'markersize', 15, 'LineWidth', 2)
```

The set of data that are subject to clustering, and the found cluster centers for example 7.1 are shown in Fig. 7.1.

The *fcm* function is executed iteratively as long as the changes in the objective function exceed some predetermined threshold.

At each step in the MATLAB command window, the serial number of the iteration and the corresponding current value of the objective function are displayed (Table 7.1).

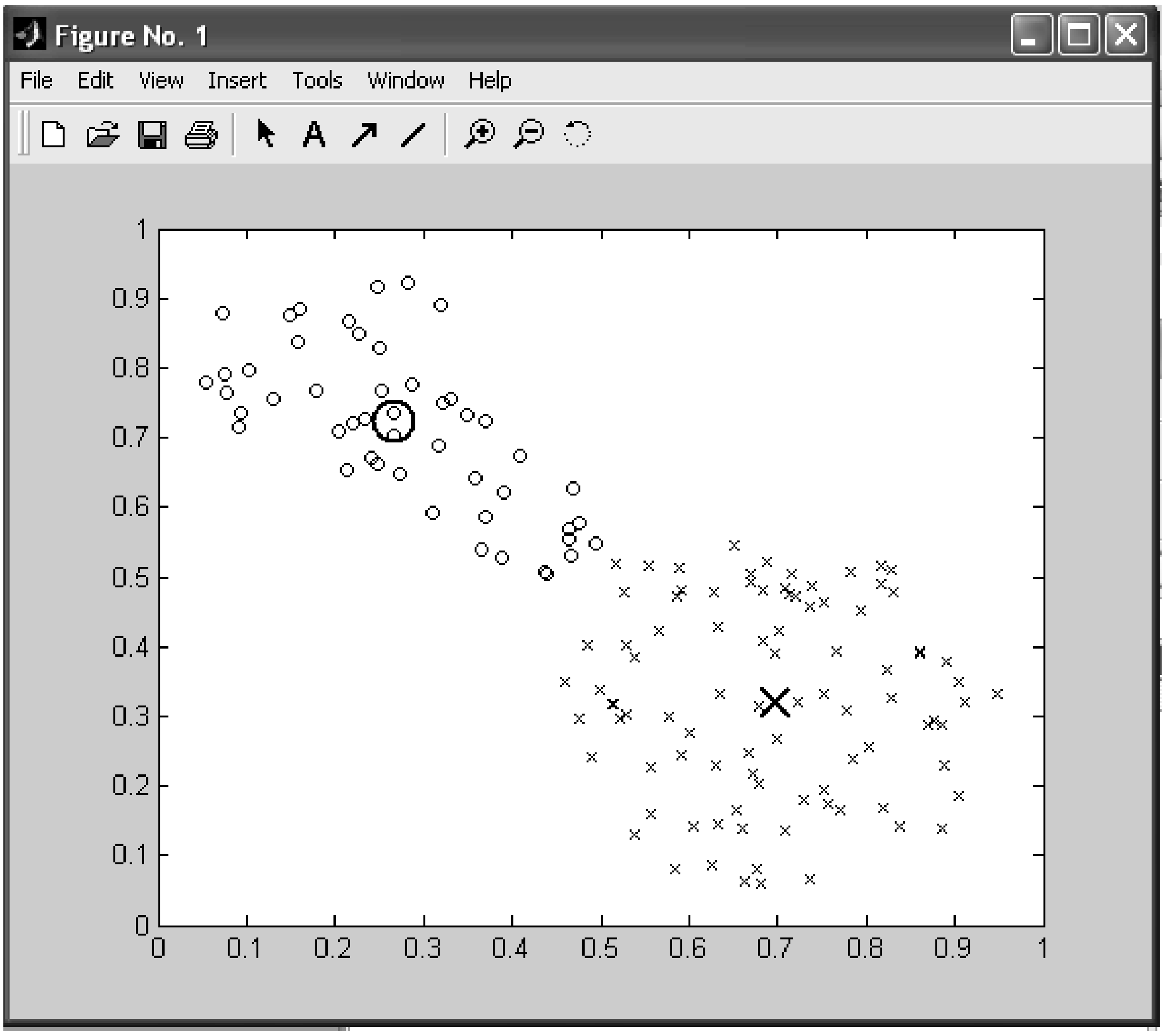


Figure 7.1. The set of analyzed data centers and clusters centers

Table 7.1 – Changing the objective function

Iteration number	Objective function values	Iteration number	Objective function values
1	8,94	7	3,81
2	7,31	8	3,80
3	6,90	9	3,79
4	5,41	10	3,79
5	4,08	11	3,79
6	3,83	12	3,78

To assess the dynamics of changes in the values of the objective function, the *plot(obj_fcm)* command is used. The results of Example 7.1 are shown in Fig. 7.2.

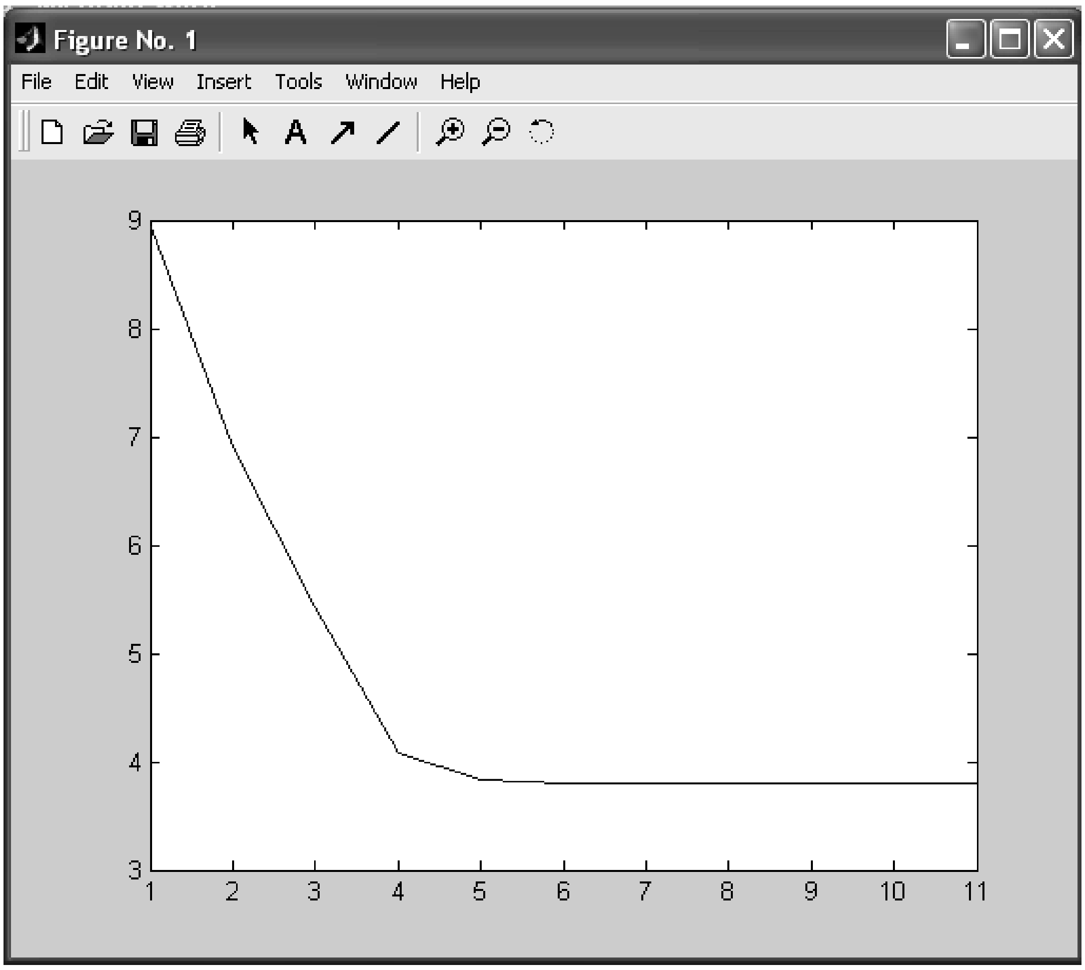


Figure 7.2. Graph of changes in the values of the objective function

The clustering function can be called with an additional set of parameters: *fcm* (*data*, *cluster_n*, *options*). Additional arguments are used to control the clustering process:

– *options*(1) is the exponent for the matrix *U* (default: 2.0);

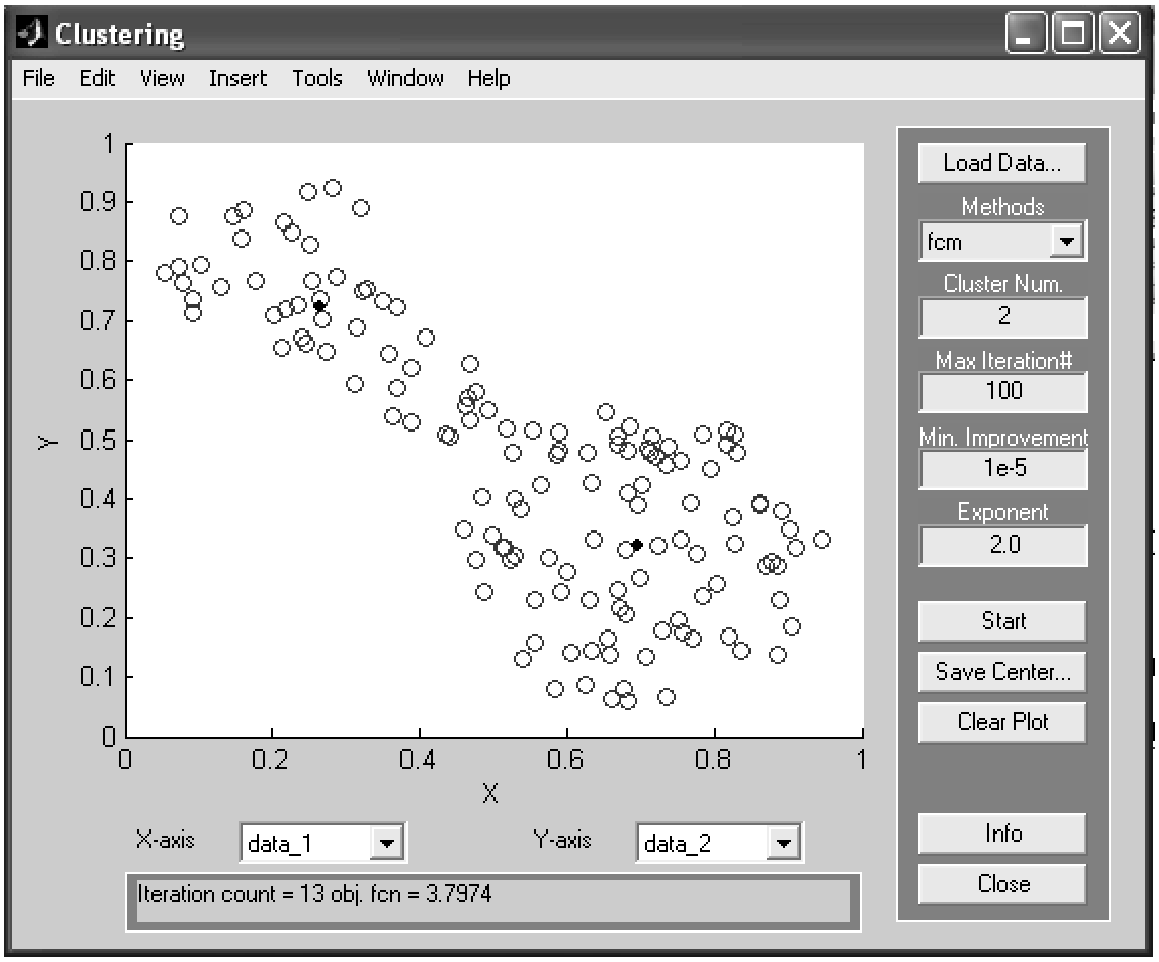
– *options*(2) is the maximum number of iterations (default: 100);

– *options*(3) is the minimum allowable change in the values of the objective function (default: 1*e*–5);

– *options*(4) is the display of information at each step (default: 1).

Here is the example of *fcm* function definition with additional parameters: [*center*, *U*, *obj_fcn*] = *fcm*(*fcmdata*, 2, [2, 100, l*e*–5, 1]).

The second way to solve clustering problems in MATLAB is invoked by the *findcluster* command. The main window of the clustering tool is shown in Fig. 7.3.



Iteration count = 13 obj. fcn = 3.7974

Figure 7.3. Clustering Main Window in MATLAB

The <*Load Data*> button is used to load the source data to be clustered in the following format: each line is a point in the multidimensional space of characteristics, the number of lines corresponds to the number of points (data items). The graphic interpretation of the initial data can be observed in the window of the same name in the main window of the tool.

The choice of the type of clustering algorithm is carried out using the *Methods* drop-down menu (menu item – *fcm*). Next, the parameters of the clustering algorithm are determined:

– number of clusters (input line – *Cluster Num*);

– maximum number of iterations (input line – *Max Iteration*);

– the minimum value of the objective function improvement (input line – *Min. Improvement*);

– exponent at matrix MF (input line – *Exponent*).

After determining the required values of the specified parameters, the clustering algorithm is launched using the <*Start*> button. The number of iterations performed and the value of the objective function can be viewed in the lower part of the main window of the clustering tool.

The coordinates of the found cluster centers can be saved by clicking the <*Save Center ...*> button. Each row of the matrix in the file is a set of coordinates for a separate cluster. The number of rows corresponds to the number of clusters.

7.2. individual tasks

1. It is necessary to formulate a problem from the field of computer technology or programming, for which an automatic classification of a set of objects would be necessary. These objects are defined by feature vectors in the feature space.
2. Solve the formulated problem in MATLAB using the clustering mechanism using fuzzy logic methods, while using the command line and graphical user interface. When using the command line, the clustering function must be called with an additional set of parameters.
3. Find the centers of clusters and plot the graph of change in the values of the objective function.
4. Design a laboratory report.

Laboratory work 8

MODELING A FUZZY SYSTEM WITH FUZZY LOGIC TOOLS

The purpose of laboratory work is to obtain and consolidate knowledge, to form practical skills for constructing a fuzzy system using fuzzy logic tools.

8.1. Summary of theory

MATLAB includes five basic graphical user interface (GUI) tools that provide access to the fuzzy logic toolkit (FLT): fuzzy inference system (FIS) editors, membership functions, inference rules, and rule viewers and inference surfaces viewers tools. These tools are dynamically linked, and changes made in one of them entail changes in others.

The FIS editor provides the ability to form the designed system at a high level of abstraction: the number of input and output variables, the name of the variables.

The membership function (MF) editor is used to define the shape of the MF associated with each variable.

The inference rule editor is used to edit the list of rules that determine the behavior of the designed system.

The Inference Rule Viewer is used for diagnostic purposes and can show, for example, the activity of rules or the form of influence of individual MFs on the result of fuzzy inference.

The Inference Surface Viewer is used to display the dependence of the system output on one or two inputs. In other words, it generates and outputs a map of the inference surface developed by the FIS.

FIS editor. Construction of fuzzy systems according to Mamdani. To build the system being created type *fuzzy* command in the command line of the main MATLAB window. The new FIS editor window contains an input variable, designated *input1*, and an output variable, designated *output1*. By default, FLT offers to create a FIS of the Mamdani type.

In order to add a new variable, select the corresponding item in the *Edit* menu (*Add input* is for an input variable, *Add output* is for an output variable). Changing the name of a variable occurs in steps.

Step 1. The variable that needs to be renamed is marked.

Step 2. The name of the variable is changed by default to the name suggested by the user In the edit field.

Saving the projected system to the MATLAB workspace (to a variable) is performed using the *File – Save to workspace as* ... menu item. In this case, the data is retained until the end of the MATLAB session. To save data on the disk after the end of the session, the corresponding item of the same menu is used – *Save to disk as...* .

MF editor. The next step in building a fuzzy model using FLT is to associate a set of MFs with each input and output variable. This operation is performed in the MF editor in three ways, which can be activated:

– by selecting the *Edit Membership Functions* ... item in the *View* menu;

– by double-clicking on the image of the corresponding variable (input or output);

– by typing the *mfedit* operator on the command line.

With the help of the MF editor, you can display and edit any MF associated with all the input and output variables of the developed FIS.

Binding the MF with a variable name is as follows:

– a variable is selected by name from a set of graphic objects in the MF editor window;

– the range of change of values for the base variable and the visible range for the current variable are indicated;

– in the *Edit* menu, select the *Add MFs...* item. In the window that appears, select the type of MF and their number.

The MF of the current variable is edited in two ways: using the MF graphic window or by changing the MF characteristics (name, type and numerical parameters). When you select the required MF in the graphics window, you can smoothly change the curve using the mouse.

Thus, when constructing a FIS, it is necessary using the MF editor to determine the corresponding functions for each of the input and output variables.

Inference rules editor. After the number of input and output variables is indicated, their names are determined and the corresponding MFs are built, it is necessary to include inference rules in FIS. To do this, select the *Edit Rules...* item in the *View* menu or type the *ruleedit* command in the MATLAB command line ruleedit.

Based on the descriptions of the input and output variables defined in the MF editor, the inference rules editor generates the structure of the rule automatically. The user is only required to bind the values of the input and output variables, choosing from the list of previously set MFs and to determine the logical links between them. It is also allowed to use logical negation (NOT) and change the weights of the rules in the range from 0 to 1.

Inference rules can be displayed in the window in various formats, which are defined by selecting the appropriate item on the *Format* submenu of the *Options* menu. The default is the extended format for displaying inference rules. (*verbose form*):

If (*input_1 is*[*not*] *mf_1j1*) *<and, or>...*(*input_i is*[*not*] *mf_iji*)*...<and,or>*
(*input_n is*[*not*] *mf__njn*) *then*
(*output_1 is*[*not*] *mf_n + 1jn+1*) *<and, or>...*
(*output_k is*[*not*] *mf_k + njk+n*) *<and, or>...*
(*output_m is*[*not*] *mf_m + njm+n*) (*w*),

where *i* is the number of the input variable; *ji* is the number of the MF of the i-th variable; *k* is the number of the output variable; *n* is the number of input variables; *t* is the number of output variables; *w* is the weight of the rule.

Round brackets contain required parameters, square ones contain optional parameters, and angular ones contain alternative parameters (one to choose from).

In addition to the default format, there are two more types of display formats for rules: *symbolic form* and *indexed form*. The symbolic form is as follows:

(*input_1*<~=,==>*mf_1j1*)<&, | >...
(*input_i*<~=,==>*mf_iji*)...<&, | >
(*input_n*<~=,==>*mf_njn*)=>
(*output_1*<~=,==>*mf_n + 1jn+1*)...<&, | >
(*output_k*<~=,==>*mf_k + njk+n*) <&, | >...
(*output_m*<~=,==>*mf_m + njm≠n*) (*W*)

The difference between the symbolic form and the extended form is that instead of verbal interpretation of bundles, symbolic is used (symbols <&> and <|> respectively determine logical AND and logical OR, symbol <~> determines logical negation, and symbol < => > is a separator of the conditional and final parts of the rule (antecedent and consequent).

The general description of the rule of output in index format can be presented as follows:

$$[-]1j1\ldots[-]iji\ldots[-]njn[-]n + 1jn + 1\ldots[-]k + njk + 1\ldots[-]m + njm + n(w){:}\langle 1,2\rangle.$$

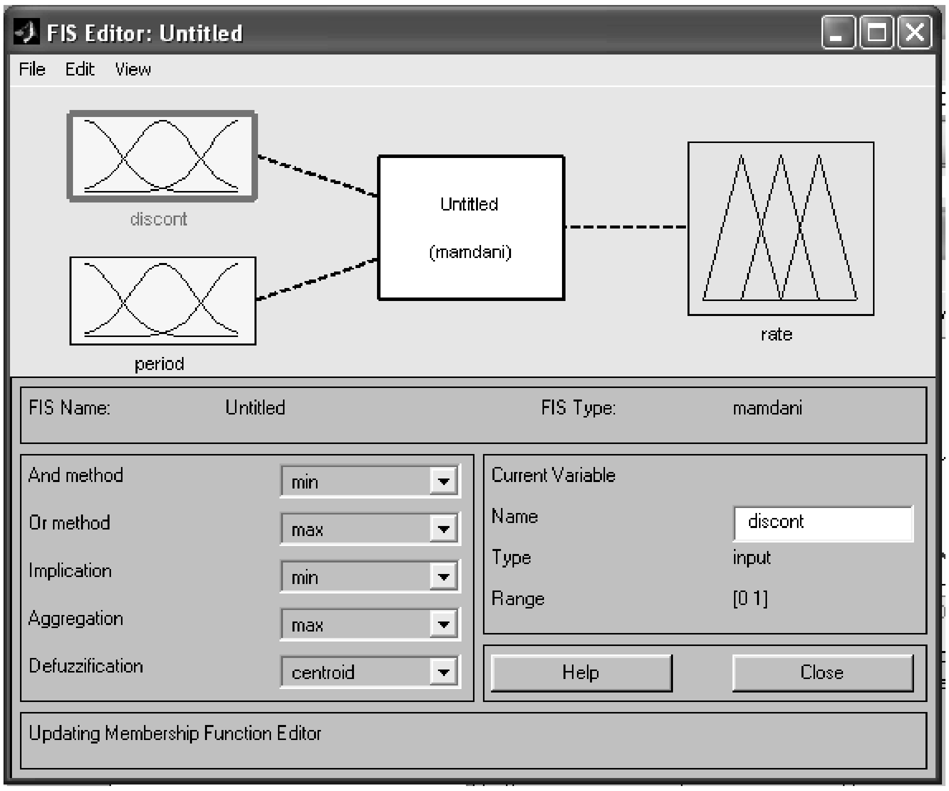
Here the order of the numbers corresponds to the order of the input variables, and the symbol < , > divides the rule into conditional and final parts. The serial number of the corresponding MF is written before the colon, the type of logical connective is written after the colon (< 1 > is logical AND, < 2 > is logical OR). Logical negation is specified by the symbol < – >.

It can be argued that FIS was completely created after defining the rules of output in the editor of the same name.

Example 8.1. Creation of FIS.

Consider the following situation. It is necessary to assess the degree of investment attractiveness of a particular business project based on data on the discount rate and payback period.

Step 1. Call the editor to create the FIS by typing *fuzzy* on the command line. Add an input variable by choosing the *Add input* item from the *Edit* menu. As a result, we get the following FIS structure: two inputs, a Mamdani fuzzy inference mechanism, one output. We declare the first variable as *discont*, and the second one is *period*, which, respectively, will represent the discount rate and the payback period of the business project. The name of the output variable, on the basis of which a decision is made about the degree of investment attractiveness of a business project, is set as *rate*. Let's save the created model under the name *Invest*. The current state of the FIS editor window is shown in Fig. 8.1.



FIS Name: Untitled   FIS Type: mamdani

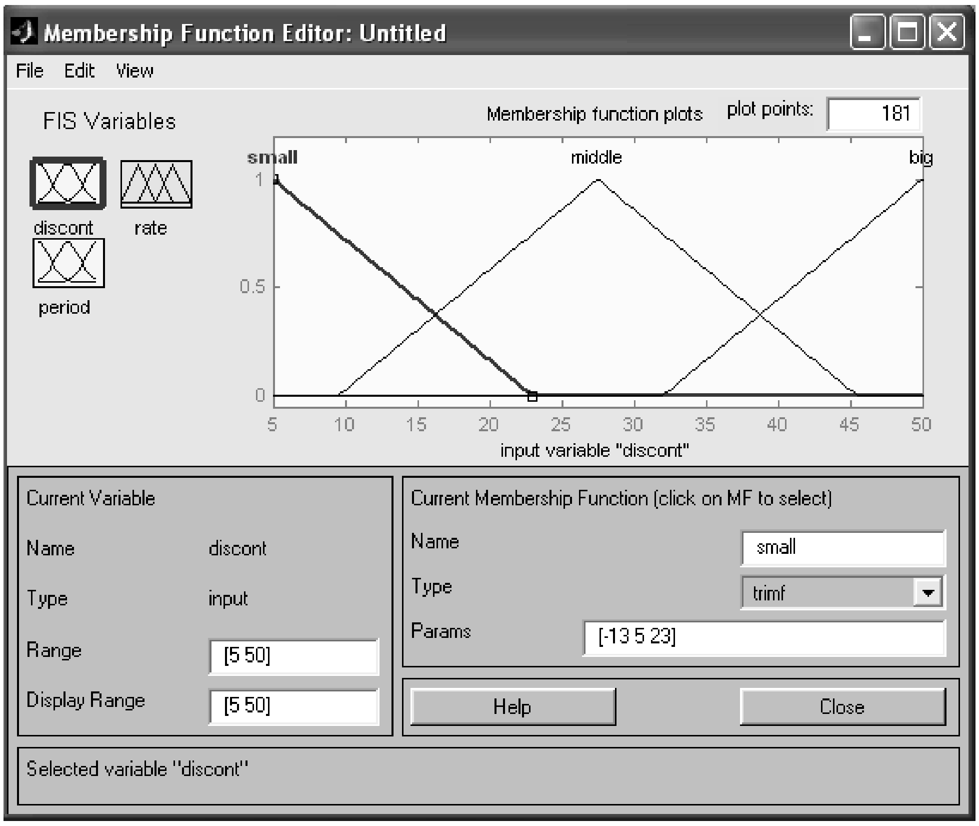
And method: min
Or method: max
Implication: min
Aggregation: max
Defuzzification: centroid

Current Variable
Name: discont
Type: input
Range: [0 1]

Updating Membership Function Editor

Figure 8.1. Fuzzy inference editor window

Step 2. We assign a set of MFs to each input and output variable. This procedure is implemented in the MF editor. For *discont*, we define the range of the base variable (*Range*) from 5 to 50 (unit of measure is percent). We select the same range for its display (*Display Range*). Let's add three MFs, the type of which is *trimf*. Sequentially highlighting individual MFs with the mouse, we will assign names *small, middle, big,* respectively, to a small, medium and large discount rate. The MF editor window in its current state is shown in Fig. 8.2. For the variable *period*, the range of the base variable is set equal to [3, 36] (unit of measurement is months), three MFs of the *gaussmfc* type are assigned with names *short*, *normal*, *long*. Thus, the variable of the payback period of the business project will take on the following values: short, normal and long payback period.



Current Variable
Name discont
Type input
Range [5 50]
Display Range [5 50]

Current Membership Function (click on MF to select)
Name small
Type trimf
Params [-13 5 23]

Selected variable "discont"

Figure 8.2. MF editor window

Finally, we define for the *rate* variable: the base variable changes the value in the range [0, 1], the semantics is described by three MFs of *trimf'* type with names: *bad*, *normal*, *good*.

Step 3. Step 3. The final stage in the FIS construction is to define a set of rules that define the relationship between input and output variables. For this purpose we define the following in the inference rules editor:

IF *discont* = *small* AND *period* = *short* THEN *rate* = *good*

IF *discont* = NOT *small* AND *period* = *long* THEN *rate* = *bad*

IF *discont* = *middle* AND *period* = *normal* THEN *rate* = *normal*

IF *discont* = *big* AND *period* = *short* THEN *rate* = *normal*

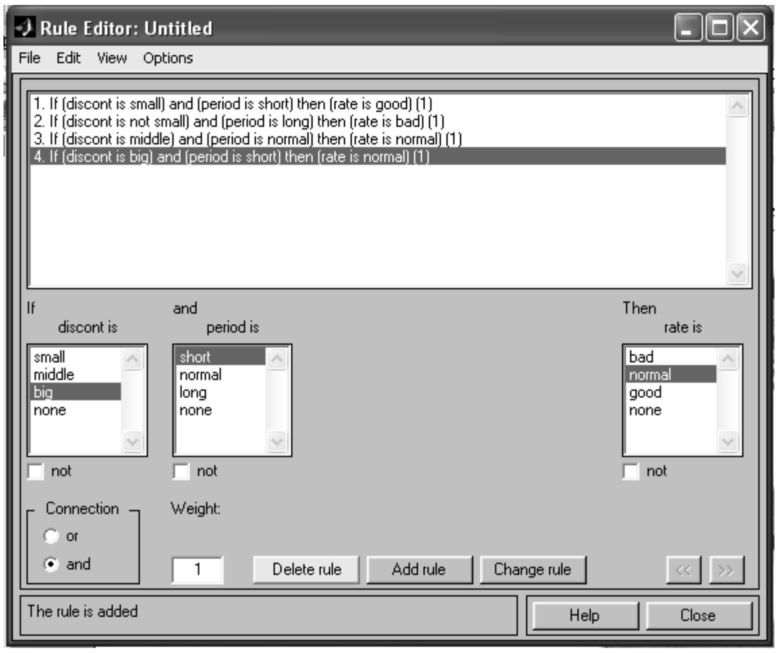
The current state of the inference rules editor window is shown in Fig. 8.3. The specified inference rules are represented in an extended display format as follows:

if(*discont is small*) *and* (*period is short*) *then* (*rate is goad*) (1)

if(*discont is not small*) *and* (*period is long*) *then* (*rate is bad*) (1)

if(*discont is middle*) *and* (*period is normal*) *then* (*rate is normal*) (1)

if(*discont is big*) *and* (*period is short*) *then* (*rate is normal*) (1)



1. If (discont is small) and (period is short) then (rate is good) (1)
2. If (discont is not small) and (period is long) then (rate is bad) (1)
3. If (discont is middle) and (period is normal) then (rate is normal) (1)
4. If (discont is big) and (period is short) then (rate is normal) (1)

If discont is	and period is	Then rate is
small	short	bad
middle	normal	normal
big	long	good
none	none	none

not: ☐   not: ☐   not: ☐

Connection: ○ or   ● and

Weight: 1

The rule is added

Figure 8.3. Inference rules editor window

When changing the form to symbolic, the inference rules will look like this:

(*discont* == *small*) & (*period* == *short*) => (*rate* == *good*) (1)

(*discont* ~= *small*) & (*period* == *long*) => (*rate* == *bad*) (1)

(*discont* == *middle*) & (*period* == *normal*) => (*rate*== *normal*) (1)

(*discont* == *big*) & (*period* == *short*) => (*rate* == *normal*) (1)

Inference rules viewer tool. This inference rule viewer allows you to display the fuzzy inference process and get the result. The main viewer window consists of several graphical windows arranged in rows and columns. The number of lines corresponds to the number of rules for fuzzy inference, and the number of columns corresponds to the number of input and output variables specified in the developed FIS. An additional graphic window is used to display the result of fuzzy inference and defuzzification operation. Each window displays the corresponding MF, the level of its cut (for input variables) and the contribution of an individual MF to the overall result (for output variables).

In the lower part of the main window, you can display the numbers of the inference rules in different inference formats by marking them with the mouse. To change the format in the *Options* menu, select the item *Rule display format*.

Changing the values of input variables is allowed in two ways:

1) by entering a record of an input vector in the *Input* field, the dimension of which is equal to the number of input variables;

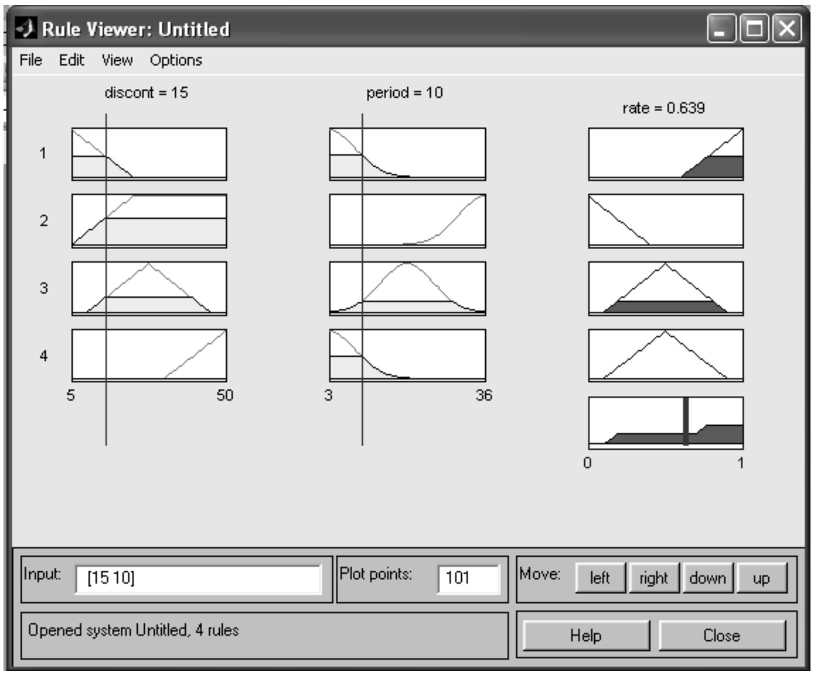
2) by clicking in any graphics window that refers to the input variable.

The input vector in each of these variants of the initial data definition will define a set of red vertical lines.

For the FIS considered in Example 8.1, with the input vector [15 10] (discount rate is 15%, payback period of the business project is 10 months), the result (degree of investment attractiveness) will be 0.639 (Fig. 8.4).

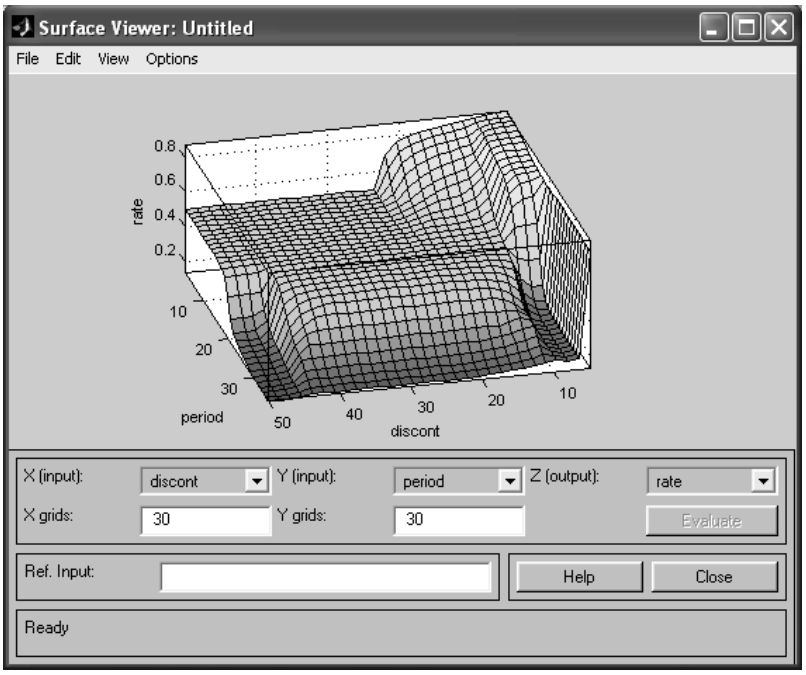
Inference surfaces viewer tool. Inference surfaces viewer tool allows you to build a 3D surface as a dependency of one of the output variables on two input variables. The choice of input and output variables is carried out through the drop-down menus of the main window of the considered software tool. The number of lines to be drawn along the X and Y axes is defined in the X grids, Y grids input fields. The inference surface corresponding to the inference rules of Example 8.1 is shown in Fig. 8.5.

Construction of Sugeno-type fuzzy systems. Let's consider the construction of FIS by two editors – FIS and MF. To build a FIS of the Sugeno type, select the *New FIS* $\rightarrow$ *Sugeno* item in the *File* menu. The number of input and output variables is determined in the same way as when constructing a FIS of the Mamdani type.



Opened system Untitled, 4 rules

Figure 8.4. Inference rules viewer tool window



X (input): discont
Y (input): period
Z (output): rate
X grids: 30
Y grids: 30
Ref. Input:
Ready

Figure 8.5. Inference surfaces viewer tool window

MF editor. For FIS of the Sugeno type, the changes concern only the scheme for determining the MF for the output variables. FLT in MATLAB allows you to develop two types of fuzzy models. The first model is the zero-order Sugeno fuzzy model. The fuzzy inference rule is as follows:

$$if\ x\ is\ A\ and\ y\ is\ B\ then\ z = k,$$

where *A* and *B* are fuzzy sets of the antecedent; *k* is a well-defined consequent constant.

To build such a model, when adding an MF, it is necessary to select the *constant* type and set the numerical value of the corresponding constant as the MF parameter. The second model is the first order Sugeno fuzzy model. For her, the fuzzy inference rule is written as follows:

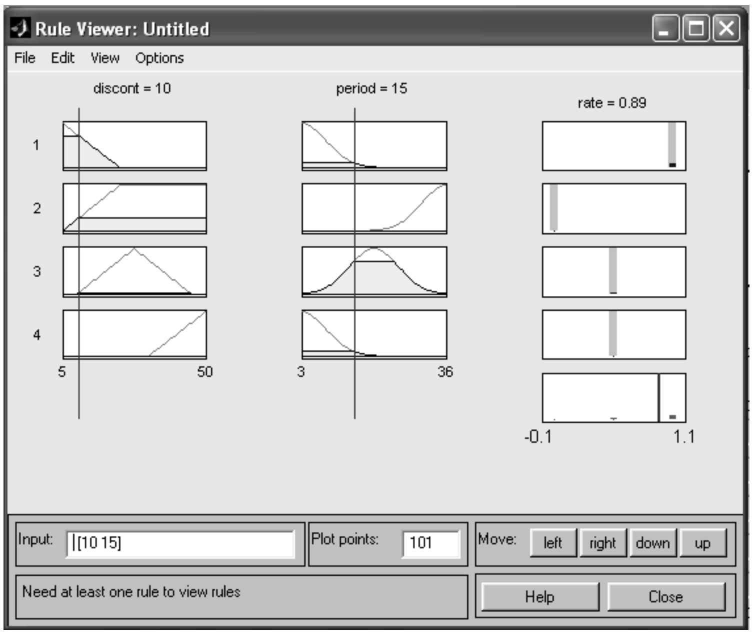
$$\textit{if } x \textit{ as } A \textit{ and } y \textit{ is } B \textit{ then } z = p \cdot x + q \cdot y + r,$$

where p, q and r are constants.

In this case, the MF type is linear. To determine the parameters of the MF, it is necessary to enter a vector, the elements of which correspond to the numerical values of the consequent constants.

The work with the editor of inference rules, as well as with the viewers of rules and the surface of the inference, is carried out in the same way as in the case of constructing a FIS according to Mamdani.

An example of Sugeno fuzzy inference using a zero-order fuzzy model and the inference rules defined above is shown in Fig. 8.6 (the output variable has three values: *bad, normal, good,* which are set, respectively, by three constants – 0, 0.5, 1).



Input: [10 15]

Plot points: 101

Need at least one rule to view rules

Figure 8.6. Inference rules viewer tool window (Sugeno inference)

8.2. Individual tasks

1. It is necessary to formulate an abstract situation from the field of computing or programming and build a fuzzy system for it using a graphical user interface that provides access to the fuzzy logic toolkit and the editor of the fuzzy inference system.

At the same time, the construction of a fuzzy system, in the first case, should be based on the Mamdani principle, and in the second, should be based on the Sugeno principle.

2. When performing point 1 of an individual task, set different ranges of variation of the input and output variables of the fuzzy system, as well as different types of MFs.

3. Construct a graphical display of the inference rules and the surface of solutions of the formulated abstract situation.

4. Comparative analysis of two fuzzy systems built on the principle of Mamdani and Sugeno.

5. Design the lab’s report.

REPORTS DESIGN REQUIREMENTS

Reports are drawn up in English using *Microsoft Word* and *Times New Roman* font without hyphenation. Line spacing is one and a half, paragraph indentation is 1.25 cm. Report volume is no more than 10 printed pages. The report is prepared on A4 sheets using portrait orientation. Top, bottom and right margins are 2.0 cm, left one is 3.0 cm. Headers size is 0 cm, and footers size is 2.0 cm. Numbering is performed at the bottom of the page, font size is 14 *pt*, alignment is center.

The first line contains the last name and initials of the student, as well as the number of the academic group. Font size is 12 *pt*, bold, italic, right alignment.

The second line contains the work number printed in uppercase (font is straight, bold, 14 *pt*, center alignment).

Next, one blank line after the work number the topic of the work is printed in uppercase, (font is straight, bold, 14 *pt*, center alignment).

Then one blank line after the topic of the work the goal of the work is printed. Font size is 14 *pt*, indent is 1.25 *pt*, alignment is justified.

Further, the results of the execution of all individual task points are printed. The font of the text is 14 *pt*, paragraph indentation is 1.25 cm, alignment is justified.

The conclusions of the work are given at the end. They are printed one blank line after the results of the work. Font size is14 *pt*, indent is 1.25 *pt*, justified alignment.

REFERENCES

1. MATLAB Programming Fundamentals [Internet resource]. The MathWorks – Natick, MA: MathWorks, 2021. – 1450 p. – Access mode: https://www.mathworks.com/help/pdf_doc/matlab/matlab_prog.pdf.

2. Otto S.R. An Introduction to Programming and Numerical Methods in MATLAB / S.R. Otto, J.P. Denier. – London: Springer, 2005. – 468 p.

3. Young T. Introduction to Numerical Methods and Matlab Programming for Engineers / T. Young, M. J. Mohlenkamp – Athens: Ohio University, 2021. – 182 p.

4. Zimmermann H.J. Fuzzy Set Theory-and Its Applications / H. J. Zimmermann. – New York: Springer Science+Business Media, 2001. – 525 p.

5. Hooda D.S. Fuzzy Logic Models and Fuzzy Control. An Introduction/ D.S. Hooda, V. Raich – UK: Alpfa Science, 2017. – 408 p.

6. MATLAB Mathematics [Internet resource]. The MathWorks – Natick, MA: MathWorks, 2015. – 634 p. – Access mode: http://www.apmath.spbu.ru/ru/staff/smirnovmn/files/math.pdf.

7. Hunt B.R. A Guide to MATLAB for Beginners and Experienced Users / B.R. Hunt, R.L. Lipsman, J.M. Rosenberg. – USA: Cambridge University Press, 2001. – 347 p.

8. Bassanezi R.C. A First Course in Fuzzy Logic, Fuzzy Dynamical Systems, and Biomathematics: Theory and Applications / R.C. Bassanezi. – New York: Springer Science, 2017. – 304 p.

9. Fuzzy Logic Toolbox: User's Guide / [Internet resource]. The MathWorks – Natick, MA: MathWorks, 2018. – 528 p. – Access mode: https://person.dibris.unige.it/masulli-francesco/lectures/ML-CI/lectures/MAT LAB% 20fuzzy%20toolbox.pdf.

10. Alavala Ch. R. Fuzzy Logic and Neural Networks Basic Concepts and Applications/ Ch. R. Alavala. – Delhi: New Age international publishers, 2007. – 276 p.

11. Yang W.Y. Applied numerical methods using MATLAB / W. Y. Yang, W. Cao, T. Chung, J. Morris. – A John Wiley & sons, inc., publication, 2005. – 511 p.

CONTENTS

NOTES

NOTES

NOTES

Educational edition

ZAKOVOROTNIY Oleksandr
LIPCHANSKA Oksana

FUNDAMENTALS OF COMPUTATIONAL INTELLIGENCE

Part 1

Laboratory workshop
for full-time and part-time students
by Computer Engineering and Computer Science direction
(Engl. language)

The work for the publication was recommended by *V. D. Dmitrienko*
Responsible person for the release is *S. Yu. Leonov*
In the author's edition

Signed for publication 29.12.2021. Format $60 \times 84\ ^1\!/_{16}$. Paper Copy Paper. Printing – risography. Times New Roman Font. Accounting-publishing sheet 6,4. Contractual price.

Publishing house NTU "KhPI".
61002, Kharkiv, Kirpichova str., 2