

Ministry of Education and Science of Ukraine
ODESSA POLYTECHNIC STATE UNIVERSITY

G.M. Goloborodko, V. P. Gugin, U.G. Palenyi, L.M. Perperi

Study manual

**MODERN ENGINEERING AND
MATHEMATICAL PACKAGES OF
COMPUTER SIMULATION**

**For students of the specialty
152 – Metrology and information-measuring technology**

Odessa - 2021

Ministry of Education and Science of Ukraine
ODESSA POLYTECHNIC STATE UNIVERSITY

G.M. Goloborodko, V. P. Gugin, U.G. Palenyi, L.M. Perperi

Study manual

**MODERN ENGINEERING AND
MATHEMATICAL PACKAGES OF
COMPUTER SIMULATION**

**For students of the specialty
152 – Metrology and information-measuring technology**

Approved by the decision of the
Academic Council of the Odessa
Polytechnic National University.
Protocol № 4 of 01.12.2021.

Odessa - 2021

Modern engineering and mathematical packages of computer simulation. Study manual for students of the specialty 152 – Metrology and information-measuring technology / Compiled by G.M. Goloborodko, V. P. Gugin, U.G. Palenyi, L.M. Perperi; Odessa Polytechnic National University. – Odessa: Odessa Polytechnic, 2021. 188 p.

In the study manual discusses the issues of using the freely distributed mathematical package Scilab. The graphical capabilities of the package (plotting graphs and diagrams), programming capabilities in the Scilab environment are described.

The study manual contains a summary of the discipline "Modern engineering and mathematical packages of computer simulation". The study manual is intended for bachelor students studying in the specialty 152 - "Metrology and information-measuring technology" of full-time and part-time forms of study.

Compiled by: G.M. Goloborodko, PhD;
V. P. Gugin, PhD, associate professor;
U.G. Palenyi, PhD;
L.M. Perperi, PhD, associate professor.

Reviewers: V.L. Kostenko, Doctor of Science, professor;
V.N. Tikhenko Doctor of Science, professor.

Сучасні інженерно-математичні пакети комп'ютерного моделювання. Навчальний посібник для студентів спеціальності 152 – Метрологія та інформаційно-вимірвальна техніка / Укладачі Г.М. Голобородько, В.П. Гугнін, Ю.Г. Паленний, Л.М. Перпері; Національний університет «Одеська політехніка». – Одеса: Одеська політехніка, 2021. 188 с.

У навчальному посібнику розглядаються питання використання вільно розповсюдженого математичного пакета Scilab. Описані графічні можливості пакета та можливості програмування в середовищі Scilab. Розглянуто розв'язання математичних задач.

Навчальний посібник містить зміст дисципліни «Сучасні інженерно-математичні пакети комп'ютерного моделювання». Посібник призначений для студентів бакалаврів, які навчаються за спеціальністю 152 – «Метрологія та інформаційно-вимірвальна техніка» денної та заочної форм навчання.

Укладачі: Г.М. Голобородько , канд. техн.наук;
В. П. Гугнін, канд. техн.наук, доцент;
Ю.Г. Паленний, канд. техн.наук;
Л.М. Перпері, канд. техн.наук, доцент.

Рецензенти: В.Л. Костенко, доктор техн. наук, професор;
В.Н. Тіхенко, доктор техн. наук, професор;

PREFACE

Nowadays, mechanical engineering and economics are unthinkable without information technology. Moreover, it is impossible for a specialist in the field of automation and instrumentation to work successfully without the use of modern engineering packages for computer modeling. Great opportunities for computer modeling are provided by the freely distributed software environment SciLab

As a result of studying the discipline are the acquisition of general competence mastery of research methodology and basic methods of scientific knowledge, software development of metrological support of various systems, methods of creating and analyzing process models and gaining knowledge of the theoretical foundations of mathematical modeling and basic techniques for creation mathematical programs and models using the SciLab software environment.

Every year, the capabilities of the Scilab package grow and newer versions of the package appear. Scilab has a similar programming language to Matlab. Package a includes a utility that allows you to convert Matlab documents to Scilab. Scilab allows to work with elementary and a large number of special functions, has powerful tools for working with matrices, polynomials, perform numerical calculations (eg, numerical integration) and solve problems of linear algebra, perform optimizations and simulations using powerful statistical functions and the tools for work with graphs.

This study manual is intended to facilitate study of the course “Modern engineering and mathematical packages of computer simulation” by the students that follow the Bachelor Degree Program of the specialty 152 – Metrology and information-measuring technology.

Study manual will be useful for students that study basic concepts of standardization and metrology within the branches of knowledge: the automation and instrumentation. The material of the study manual can be useful in preparing graduation thesis.

At the time of writing of the study manual, the latest version of Scilab was 6.1.1. It was on this basis this study manual was compiled. The latest version of the package can always be downloaded from the official Scilab website: <http://www.scilab.org>.

Lecture 1. The purpose of the lecture is to get acquainted with the sequence of obtaining and installing the Scilab package on a personal computer, with the layout of the interface windows, rearranging these windows and for what are these windows are needed?

1 FAMILIARIZATION WITH SCILAB PACKAGE.

1.1. Installing the Scilab package on a computer.

A free redistributable version of the package, along with complete documentation in English, can be obtained from SciLab website at www.scilab.org.

Before installing the Scilab 6.1.1 package for Windows operating system, go to the site <http://www.scilab.org> and download the file "scilab-6.1.1_x64" (fig. 1.1).

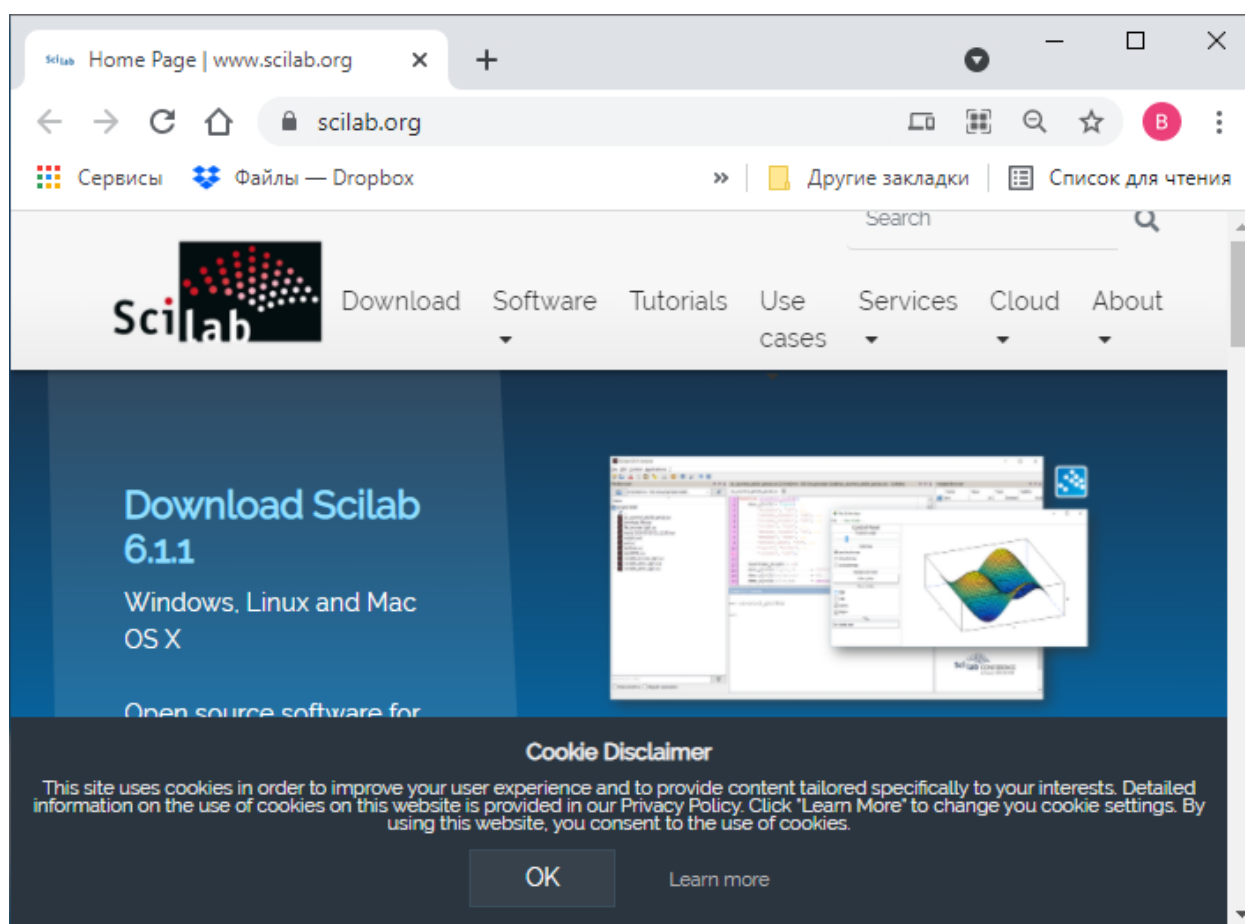


Figure 1.1 – Downloading the file "scilab-6.1.1_x64".

To install the package, run the "scilab-6.0.2.exe" executable file. When you start it, a request will appear about the language in which it is supposed to work (fig. 1.2.). To do this, click by the right button of mouse (RBM) on the drop-down list of suggested languages and press the "OK" button.

The package installation wizard will appear on the computer screen (fig. 1.3.) With the license agreement window, in which, to continue the installation, you need

to click on the button next to the inscription: "I accept the agreement" and then click the "Next" button. The next window prompts you to select the "path" of the package location. Unless you have any special considerations about where to place the package, you can accept the suggested location for "C:\Program Files\scilab-6.1.1" in the "Program Files" folder on the "C: \" drive. If you want to put the package in a different location, select the previously created folder using the "Browse ..." button and click the "Next" button.

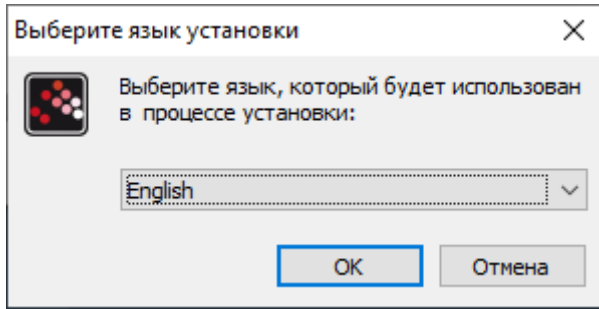


Figure 1.2 – The choice of language for work.

the "Browse ..." button and click the "Next" button.

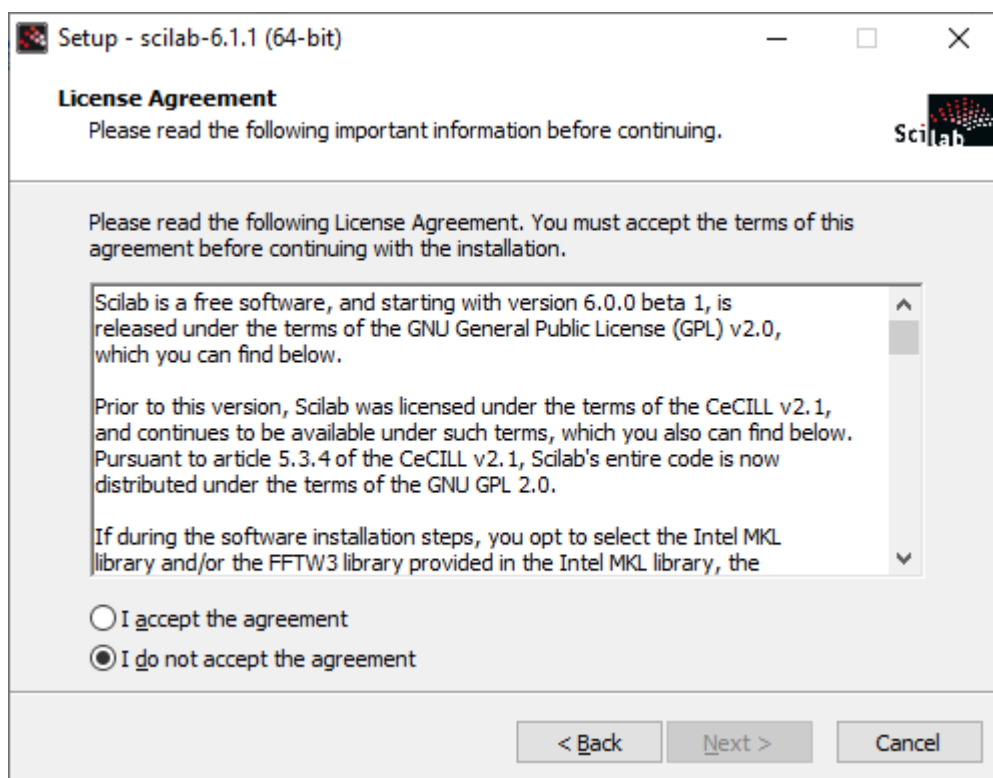


Figure 1.3 – Scilab package installation wizard.

The next window of the Installation Wizard (fig. 1.4) will offer one of three types of installation:

- Full installation;
- Custom installation (installation of the selected components);
- Command Line Minimal Installation (minimal installation).

If you are not an experienced user, you should choice the full installation and click the "Next" button.

The next window of the Installation Wizard will inform you that after installation a shortcut will be created in the "Start" menu for starting Scilab. By default, it will be named "scilab-6.1.1 (64-bit) Desktop". Click the "Next" button and the Installation Wizard will offer a list of additional components available after

installation. It is recommended to use the components suggested by the Installation Wizard. To go to the next step, you need to click the "Next" button.

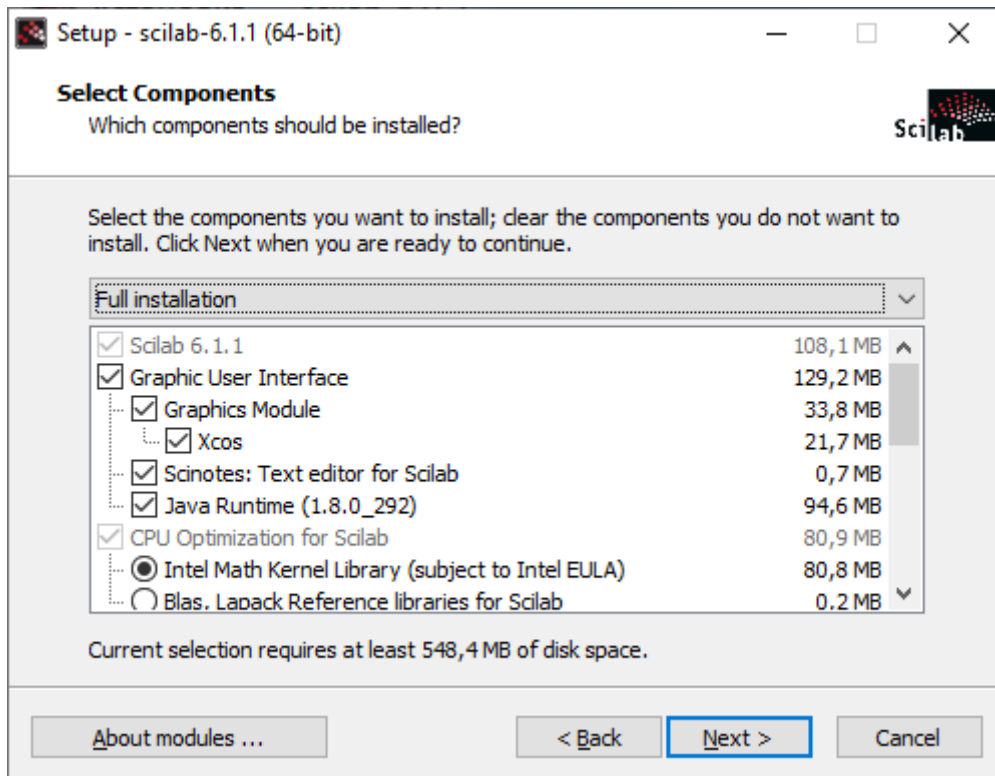


Figure 1.4 – Choice of the package installation type.

The installation wizard will inform you that the package and components are ready to be installed. After reading the report, press the "Install" button.

The process of installing Scilab on a hard disk will be accompanied by a demonstration of the process flow using a bar indicator and may take some time, depending on the performance of your computer (sometimes more than 10 minutes). The installation process ends with the appearance of an information window. If you do not uncheck the box next to "Launch Scilab" in the window that appears, clicking the "Finish" button will start Scilab immediately after clicking this button. Otherwise, the launch can be made from the main menu by



Figure 1.5 – Package launch shortcut.

clicking on the shortcut on the desktop (fig. 1.5).

1.2 Start Scilab.

To start the Scilab package, click by LMB on the shortcut located on the desktop (Fig. 1.5) or launch from the drop-down menu of the [Start] button by clicking on the icon next to the inscription "scilab-6.1.1 (64-bit) Desktop" which is located in the folder "scilab-6.1.1 (64-bit)". The interface of the Scilab package

(further for simplicity called the program) will appear on the monitor. For version 6.1.1 it will look like the one shown in figure 1.6.

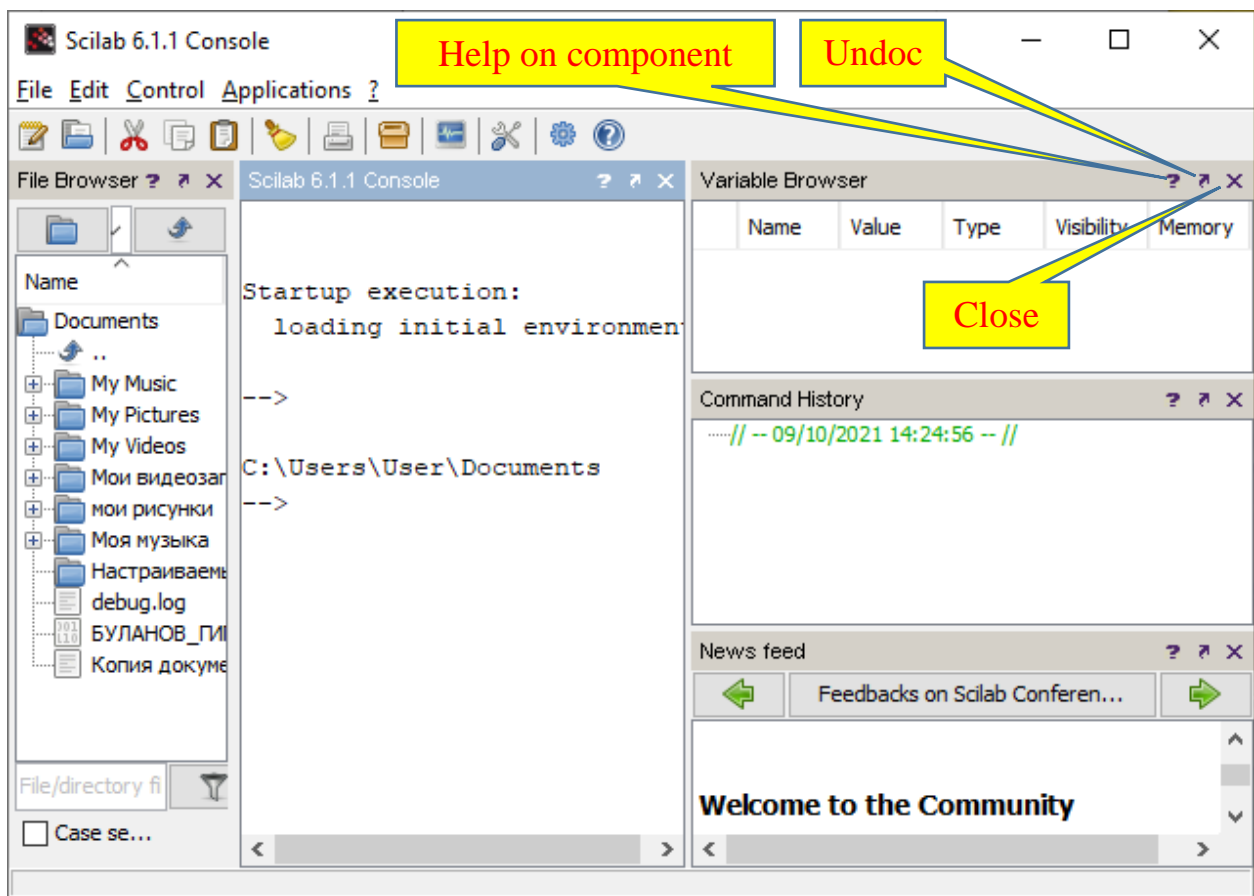


Figure 1.6 – User interface of Scilab-6.1.1.

In figure 1.6 you can see that there are 5 internal windows (or frames) inside the window:

- File Browser;
- Scilab 6.1.1 Console;
- Variable Browser;
- Command History;
- News feed.

At any each time, only one of the visible inner windows can be active, and this is indicated by the highlighted title. In figure 1.6, the Scilab 6.1.1 Console window is active. All five of these inner windows are not rigidly tied to each other and can be excluded from the main window or rearranged again in it.

Do not rush to close the inner windows until you know how to restore them.

Let's conventionally call the view shown in Figure 1.6 a group of windows. At any time, each window can be excluded from the group by pressing the [Undock] button (fig. 1.6), which is represented as a button with an upward arrow. The [Close] button can be used to close the corresponding window. The [Help on component] button opens the Help Browser. Once detached, the window can be moved to any place on the desktop.

To rearrange the windows, you need to click LMB on the strip on which the title of the window which will be moved is located, and keeping the LMB pressed, drag the cursor to the desired position. The gray frame will tell you how the window will be positioned if you release the LMB, and here the following the variants are possible:

- if you drag a any window in the group by the cursor, then the moved window will divide the area horizontally in half and take its position;
- the same as in the previous point, but vertically;
- the window will not break anything, but simply will be attached. In this case, tabs will appear at the bookmarks.

A detached window can be included in any group. To do this, click LMB on its title and perform the same actions. It is very important that you need to grab not the external title of the window, which is generated by the operating system, but the internal title, which has the buttons: [Help on component], [Undock] and [Close].

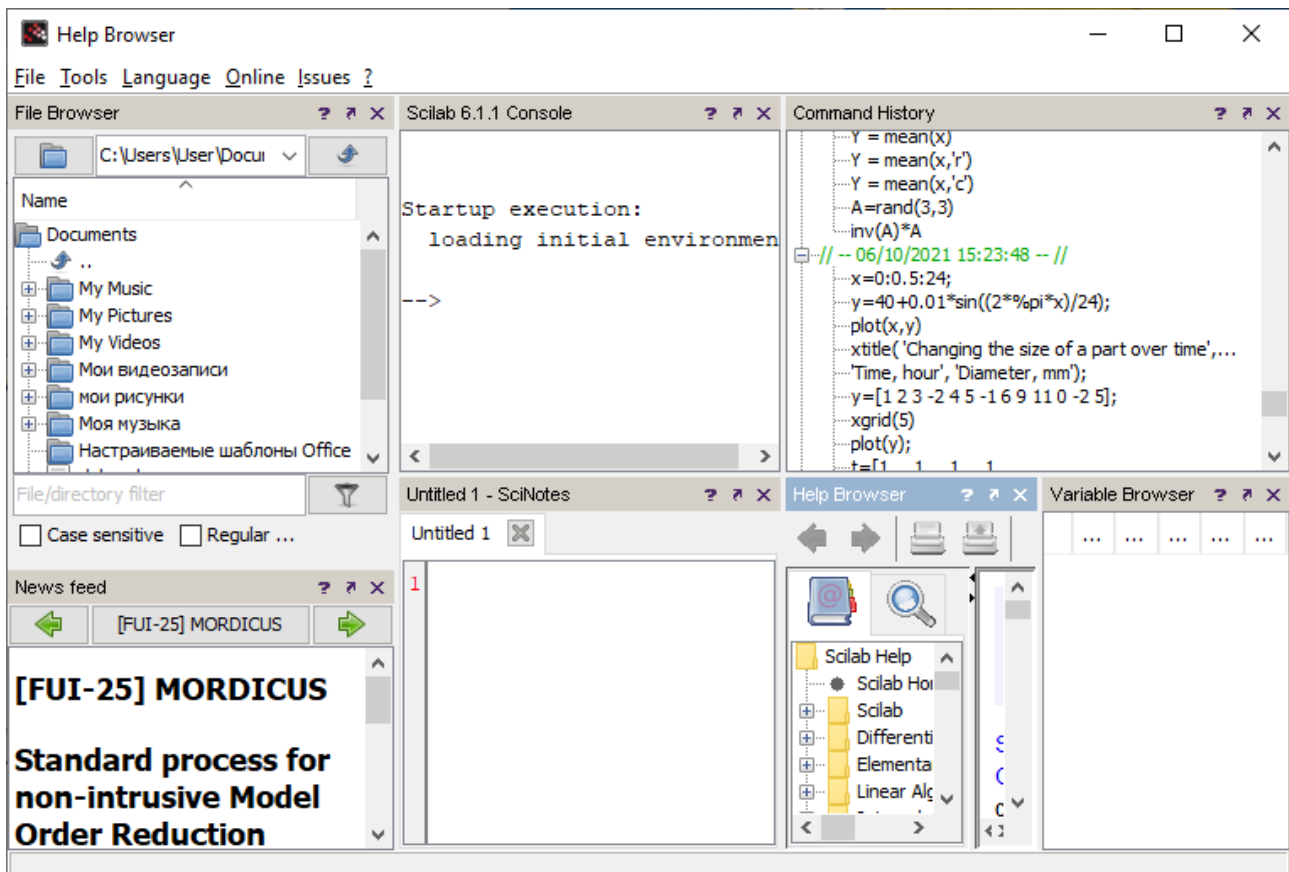


Figure 1.7 – Rearranging Scilab interface windows 6.1.1.

Rearranging takes practice, as one or the other will be suggested at a specific cursor position. Practice a little and do the following exercise.

- Make the active "Scilab 6.1.1 Console window" (further we will just call it Console window or command window);*
- To the Console window, enter the command:*

```
--> scinotes();
```

and push the [Enter] key (don't pay too much attention to its meaning yet). The SciNotes text editor window" (further we will just call it Text editor or SciNotes) should open in front of you;

Press the [F1] key and Help Browser window will open in front of you;

Using the current set of windows, rearrange them as shown in figure 1.7, and then return everything to its original view (as shown in figure 1.6).

During the exercise, you can verify that the software environment has an intuitive interface. Switching to a specific inner window, you will notice that the menu bar and toolbar change according to the purpose of each window. We will get acquainted with the purpose of each button on the toolbar during further work with the Scilab. It is advisable to view all the settings that are provided to the user in graphical mode. For this:

- make the Console window active;
- then select the "Edit" option at the top;
- in the drop-down menu click on "Preferences".

It is strongly discouraged to change the settings unnecessarily.

1.3 The destination of the program windows.

1.3.1 Command window or Console window (Scilab 6.1.1 Console).

The command window (Scilab 6.1.1 Console) is the most important window through which communication with the software environment Scilab takes place. In this window, the user put in commands and receives its results.

Look at the figure 1.6. In the command window, you can observe technical information about loading the environment, after which the system prompts the user to put in a command. The beginning of a line is accompanied by an arrow pointing to the right (-->), which is called a prompt or command prompt. Try to put in the following code:

```
-->2*2+69/25
```

and press the [Enter] key. In response from the Scilab language interpreter, you will receive the following:

```
ans =  
6.76
```

In other words, you passed the algebraic expression that it calculated to the package environment, and the result was placed in the automatically generated variable **ans**, which we will get acquainted with later. After entering this command, there have been changes in the Command History and Variable Browser windows.

Before we move on to the next window, we will enter the second command.

```
--> myVariable=26;
```

With this command, we give order to the Scilab interpreter to allocate memory for a new variable named **myVariable** and assign it the value 26. Notice the semicolon at the end of the command. By entering this symbol, we give order the interpreter not to display the results of program work in the console window.

*Note: Please pay attention, that the Scilab interpreter distinguishes between uppercase and lowercase letters in variable names, therefore **myVariable** and **myvariable** are different variables.*

Next, try entering the command into the Console window:

```
--> anotherVariable=31
```

and you will get the answer that the assignment was happened

```
anotherVariable=
    31.
```

Blocking the output of the result is done in order not to overload the listing of the program code with unnecessary intermediate information. This is especially often used when you are writing large programs.

1.3.2 Variable Browser.

	Name	Value	Type	Visibility	Memory
	anotherVariable	31	Double	local	216 B
	ans	6.76	Double	local	216 B
	myVariable	26	Double	local	216 B

This window is intended for working with variables which are created by the user. If you have entered the previous commands, you can see the result which is shown in figure 1.8.

This window displays all the necessary information about the created variables, in

Figure 1.8 – Window of the Variable Browser.

particular:

- variable name (Name);
- the size of the variable (Value);
- the type of the variable, or in other words, the type of data that this variable currently stores (Type);
- the visibility of the variable (Visibility);
- the size of the reserved memory (Memory) .

1.3.3 Variable Editor.

Double-clicking by LMB on a variable in the Variables Browser opens the Variables Editor window, which is shown in figure 1.9.

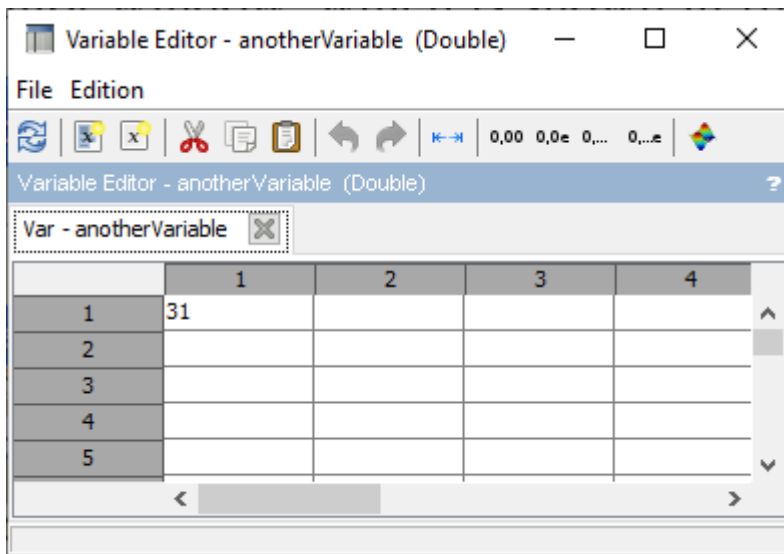


Figure 1.9 – Variables Editor window.

The Variable Editor is another handy tool for working with variables. In the Variable Editor, the variable is represented by a table, because all objects in the Scilab environment, as well as in Matlab, are two-dimensional arrays.

The variables which we have created earlier are simply a degenerate case - an array with one element. We will come back to this question more than once, but for now take a close look at the

Variable Editor.

Figure 1.9 shows the variable **anotherVariable**, which has been assigned the value 31. Change the value of the variable by double-clicking by LMB the value 31 and entering, for example, 62, and then press the [Enter] key or click by the LMB in any place of the Variables Editor window.

You can learn again about the fact that the value of the variable has changed from the command line. Make active the Console window and enter the name of our variable, that is:

```
-->anotherVariable
anotherVariable =
    62.
```

Of course, the shown way of editing a variable is not the most rational, since it is easiest to edit an array with one element in the command line of the Console window. The Variables Editor is used in cases when it is necessary to edit two-dimensional data arrays.

In addition to user-defined variables, Scilab has a number of default system variables defined. In order to see all the variables, including the system ones, that regulate the operation of the program, and that part of the variables that the user can control, do the following:

- make the «Variable Browser» window active;
- on the menu bar at the top, select the «Filter» item;
- in the menu that opens, uncheck the «Hide Scilab Variables» item.



*Study these variables, **but by no means try to edit them.***

1.3.4 Command History.

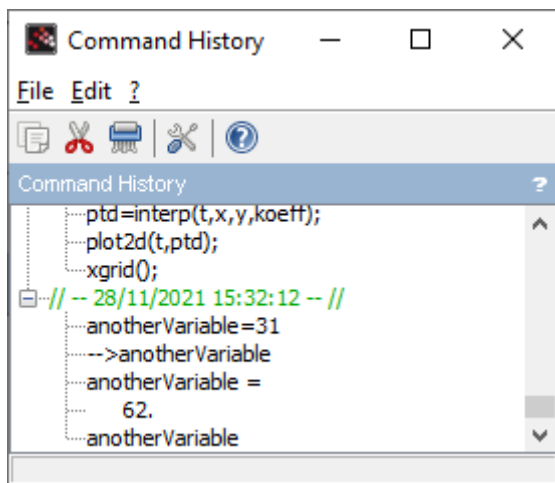



Figure 1.10 – Command History window.

The Command History window displays all the commands that the user entered into the command line during the current session. Figure 1.10 shows the Command History window. You can see the records of the last session that reflect all the commands we entered. All logs are saved by the environment so that you can remember and restore the commands that you entered earlier. This can be useful if you forgot to save the codes and ended the session, or, for another example, if you entered a very long command earlier, and now you need to enter a similar one, but with only minor differences.

However, if you do not need the logs, you can always delete or clear them using the commands from the menu.

1.3.5 Scilab help system.

The easiest way to get help, for most of the questions encountered during work, is to refer to the built-in Help Browser shown in fig.1.11. To call it on the toolbar, click LMB on the icon .

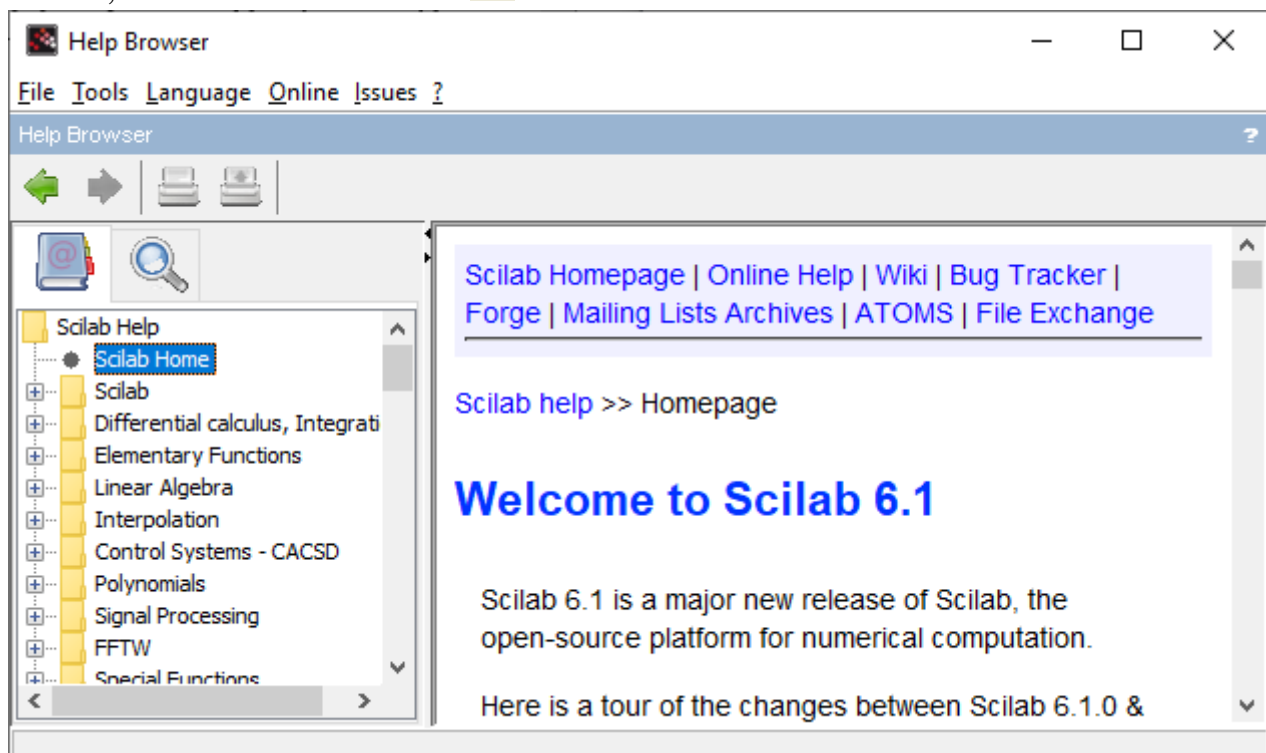


Figure 1.11 – Scilab built-in help window.

When the «Scilab 6.1.1 Console» window is active, the help window can be called in two ways. By pressing the [F1] key, or by typing the «help» command in the command line and pressing the [Enter] key:

```
-->help
```

If you need to view help material on a specific topic, for example, you are interested in the syntax of the function for calculating the **sin**, then you can scroll through the contents in the left frame of the built-in help window, find the section on trigonometric functions, find and select sin, after which a description of this will be displayed in the browser frame functions.

A more convenient way to get information about a specific function, if its name is known, is to use the "help" command in the Command window indicating the name of the function which is interested for you:

```
--> help sin
```

If the specified function does not exist in the Scilab environment or its name is entered with an error, then a beep will sound and nothing will be displayed in the help browser window that opens in the left frame of the help window.

On a computer connected to the Internet, help can also be obtained by contacting the Online Help service at: <https://help.scilab.org>

When you call Scilab help in this way, you can use one of five languages: English - Français - Português - 日本語 - Русский.

1.3.6. Built-in SciNotes text editor.

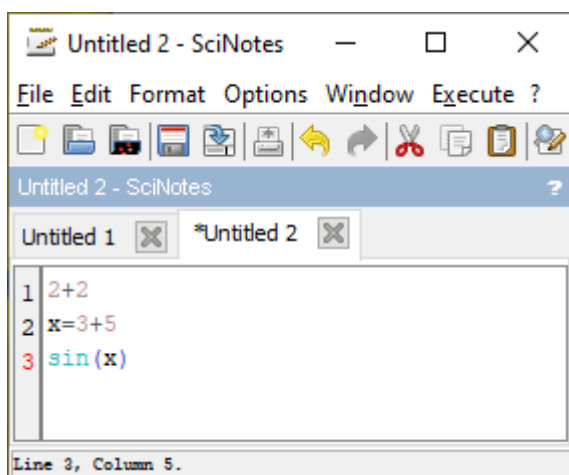


Figure 1.12 – SciNotes text editor window.

type one of the commands:

```
-->editor()
```

SciNotes is a built-in text editor for writing and editing program code executed by the Scilab interpreter (fig. 1.12).

There are several ways to open the SciNotes editor window.

If you select the «Applications» option in the main menu of the Scilab console window and click on it with LMB, a drop-down menu will appear, on which you should select the «SciNotes» option (fig. 1.13).

You can also click LMB on the icon in the toolbar of the Scilab console (fig. 1.13).

The SciNotes editor window can also be opened from the Scilab console window. To do this, on the command line, you should

`-->scinotes()`

Both of the above commands are the same in terms of the result of their execution.

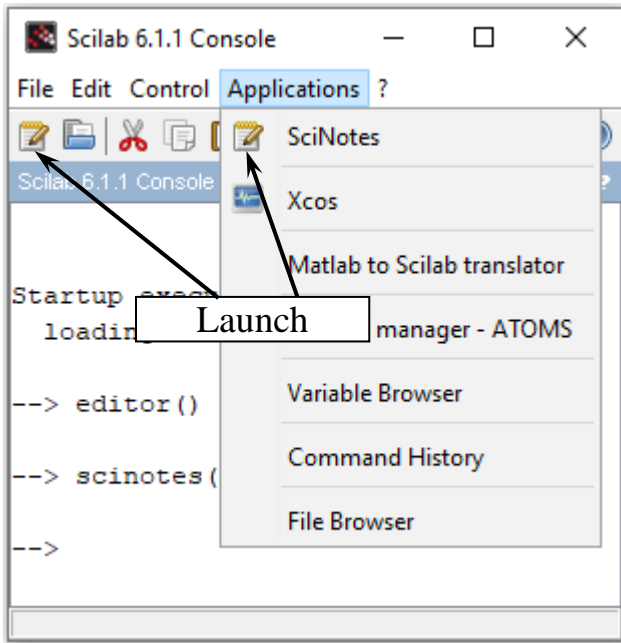


Figure 1.13 – SciNotes launch.

The SciNotes editor allows you to work with several simultaneously open files. In fig. 1.12 shows 2 files which is opened for editing in two tabs.

Run for execution the code of the program which is typed in the SciNotes editor you can use the commands of the «Execute» drop-down menu:

- «... **file with no echo**» - the file will be executed without displaying the program in the console. Be sure to save the file before executing this command. This command can be executed by simultaneously pressing the key combination [CTRL] + [SHIFT] + [E].
- «... **file with echo**» - the file will be executed and displayed in the console. This command can be executed by simultaneously pressing the keys combination [CTRL] + [L].
- «... **until the caret, with echo**»- the selected part of the commands will be executed and displayed in the console, if there is no selected program section in the text editor window, then the command is not executed. The command can be launched by the key combination [CTRL] + [E].

The Scilab software package has other windows with which we will get acquainted as it will be needed.

It is recommended to save programs which was created in the SciNotes text editor, that will make easy to use them in further . With the aim of save the file of the program code, use the «Save» and «Save as» in drop-down menu options. They are located in the main menu bar item «File» of SciNotes. The saved files with the code can be run for execution both in the command line of the Scilab console and when starting the program in SciNotes. For which it is convenient to use the command:

`-->exec('path and file name')`

Note: files with the «.sce» extension, after installing the Scilab package on the computer, are associated by the operating system as executable files with the program code for the Scilab package.

Например в папке «Примеры», на диске «Е» записан файл с именем «открытие файла2.sce» с кодом программы. Команда на выполнение этой программы будет иметь вид:

```
--> ехес ( 'Е : \Примеры\открытие файла2.sce' )
```

Questions for self-examination for the first lecture:

1. *What are the advantages of the Scilab software product?*
2. *Where can you find the "Scilab 6.1.1" package for installation on a computer?*
3. *What is the sequence of installing the "Scilab 6.1.1" package on a computer?*
4. *How many and what windows does the user interface have after starting the program?*
5. *How can the windows of the program interface be rearranged?*
6. *What is the Command Window (Scilab 6.1.1 Console) for?*
7. *Where does the command line put in?*
8. *How to make it so that the result of the calculation is not displayed on the screen after each command line?*
9. *How to set the value of a variable?*
10. *What is the Variable Browser for?*
11. *How can be opened the Variable Editor?*
12. *How can be changed the value of a variable?*
13. *What are objects in the Scilab environment?*
14. *What is the Command History for?*
15. *What is the SciNotes built-in text editor for?*
16. *How can be opened the built-in SciNotes text editor?*
17. *How to run a program using SciNotes?*
18. *How to run a program for execution which was saved in a file ?*

Lecture 2.

The purpose of the lecture is to familiarize yourself with textual comments, the rules for writing mathematical expressions, ways of defining variables and their types and with the dynamic typing of variables, learn how to enter real numbers and change the presentation of calculation results.

2 SCILAB PROGRAMMING LANGUAGE.

2.1 Text comments.

Text comments in Scilab are a line which is starting with the characters `"/"` (double slash). You can use text comments both in the workspace of the console and in the text editor of the program file. In the Scilab console window, the line after the `"/"` characters is not interpreted as a command, and after pressing the [Enter] key

activates the next command line. For example, let's type the following code in the console window:

```
--> // Explanation of the program which have been typing below.  
--> // Example of multiplication:  
--> 2*2
```

and after pressing the [Enter] key on the console screen we get the following text:

```
--> // Explanation of the program which have been typing below.  
--> // Example of multiplication:  
--> 2*2  
ans =  
4.
```

If we implement the same example and type the code in the text editor window, then it will look like this:

```
// Explanation of the program which have been typing below.  
// Example of multiplication:  
2*2
```

After pressing keys [Ctrl] + [L] at the same time, the above calculation example with explanations will appear in the console window.

2.2 Elementary mathematical expressions

The following operators are used to perform the simplest arithmetic operations in Scilab. The description of these operators can be easily found in the "Help" window (see figure 2.1).

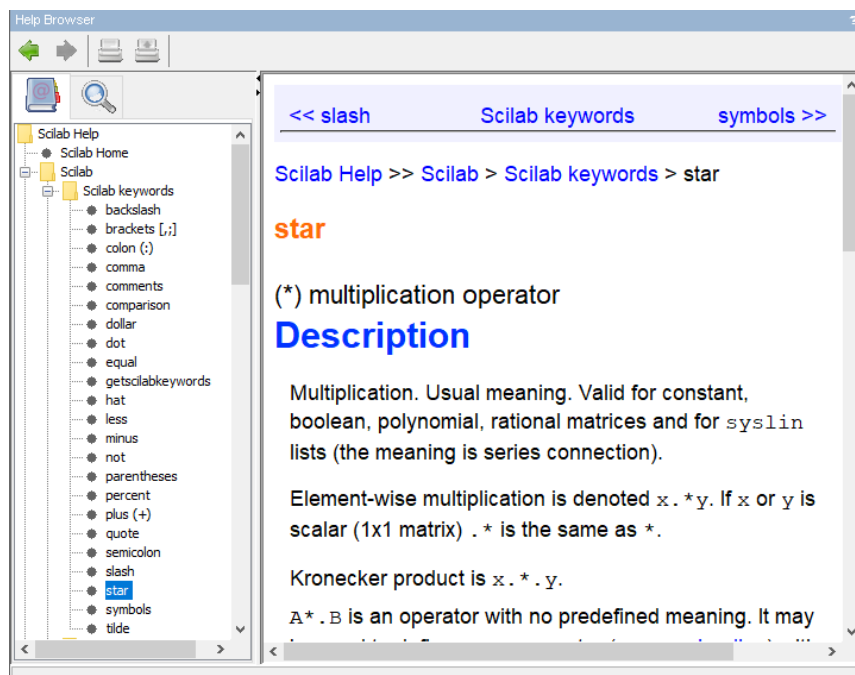


Figure 2.1 – The description of elementary operators in the "Help" window.

- star (*) — multiplication operator.

Description: Multiplication. Valid for constant, Boolean, polynomial, rational matrices.

Example:

```
--> // Basic numerical operations
```

```
--> 2 * 2
```

```
ans =
```

```
4.
```

- slash (/) — operator of right divisions.

Description: When the left operand is divisible by the right operand.

Example:

```
--> a = 4 / 2 // Should be 2
```

```
a =
```

```
2.
```

- backslash (\) — operator of left divisions.

Description: When the right operand is divisible by the left operand.

Example:

```
--> a = 4 \ 2 // Should be 0.5
```

```
a =
```

```
0.5
```

- plus (+) — numerical addition.

Description: For numeric operands, the addition has its usual meaning.

Example:

```
--> 2 + 5 // Should be 7
```

```
ans =
```

```
7.
```

- hat (power) (^) or (**) — exponentiation.

Description: Exponentiation of matrices or vectors by a constant vector.

Example:

```
--> 2^3 // Should be 8
```

```
ans =
```

```
8.
```

```
--> 2**3 // Should be 8
```

```
ans =
```

```
8.
```

- equal (=) assignment, comparison, equal sign

Description: The equal sign (=) is used to denote the assignment of value(s) to variable(s).

Example:

```
--> x=5 // Should be x=5
```

```
x =
```

```
5.
```

Each operator has its own priority and is executed according to its priority. Below is the hierarchy of operators that were presented above.

Priority level	Operators
1	() , [] , « ^ »
2	« * » , « / » or « \ »
3	« + » , « - »
4	« = »

You can calculate the value of an arithmetic expression by entering it into the command line:

```
-->2.35*(1.8-0.25)+1.34^2/3.12
```

and press the [Enter] key.

The result will appear on the console screen:

```
ans =
  4.2180128
```

Let's consider in detail the sequence of calculations in accordance with the priority of the operators:

- 1) $(1.8 - 0.25) = 1.55$
- 2) $2.35 * 1.55 = 3.6425$
- 3) $1.34^2 = 1.7956$
- 4) $1.7956 / 3.12 = 0.5755128$
- 5) $3.6425 + 0.5755128 = 4.2180128$

If the calculated expression is too long, then before pressing the [Enter] key, you must type three or more dots. This will mean that the command will continue on the next line:

```
-->1+2+3+4+5+6....
-->+7+8+9+10....
-->+11+12+13+14+15
ans =
  120.
```

If the semicolon character ";" is specified at the end of the expression, the result of the calculations is not displayed on the console screen:

```
-->2+3;
```

or:
-->3+4
ans =

7.

2.3 Variables in Scilab

In the Scilab console window, you can define variables and then use them in expressions. Any variable must be defined before being used in formulas and expressions. To define a variable, you need to type the name of the variable, then type the "=" symbol and at last you need to type the value of the variable. Here the equal sign is an assignment operator, the effect of which does not differ from similar operators in other programming languages. That is, in general form the assignment operator will be written as:

variable_name = expression_value

then the value of the expression which is specified on the right will be written to the variable whose name is specified on the left. The variable name must be written in one word, if you want to use several words in one variable, then the spaces between the words must be replaced with the underscore symbol "_".

The variable name must not coincide with the names of procedures, functions and system variables predefined in the system and can contain up to 24 characters. Allowed characters in variable names are Latin letters, numbers, as well as symbols "%", "!", "\$", "?". The system distinguishes between uppercase and lowercase letters in variable names. Those. **ABC**, **abc**, **Abc**, **aBc** are the names of different variables.

The expression on the right side of an assignment operator (=) can be a number, arithmetic expression, character string, or symbolic expression. If we are talking about a character or string variable, then the expression on the right side of the assignment operator (=) should be enclosed in single (') or double (") quotes. Concatenation (merging) of strings is carried out using the "+" operator. for example:

Name= 'path'+ "/" + "File1"

If the character ";" semicolon at the end of the expression, the name of the variable and its value are displayed as the result. The presence of the semicolon symbol ";" transfers control to the next command line. This allows variable names to be used to write intermediate results to the computer's memory.

Let's type the following program text in the window of the SciNotes text editor:

```
//-----
```

```

//assigning values to variables a and b
a=2.3
b=-34.7
//assigning values to variables x and y,
// calculating the value of a variable z
x=1;
y=2;
z=(x+y)-a/b
//Error message - variable is not defined
e=c+3^2
//Defining a symbolic variable
d='a'
//Defining a string variable
h='example of writing a string

```

and press the [Ctrl] + [L] keys simultaneously.

The text of the executed program will appear in the console window. After the line "--> e = c + 3 ^ 2", the program will interrupt the execution of calculations and display an error message:

```

-->e=c+3^2
at line 11 of executed file
Undefined variable: c

```

Let's make a correction in the SciNotes text editor window by defining the "c" variable, for example, assigning it a value which will be equal to 1 (c = 1;), and again send the program for execution by pressing the [Ctrl] + [L] keys simultaneously. In the Scilab console window, we get the following result:

```

--> // assigning values to variables x and y,
--> // calculating the value of a variable z
--> x=1;
--> y=2;
--> z=(x+y)-a/b
z =
3.0662824
--> // Error message - variable is not defined
--> c=1
c =

```

```

1.
--> e=c+3^2
e =
10.
--> // Defining a symbolic variable
--> d='a'
d =
"a"
--> // Defining a string variable
--> h=' example of writing a string'
h =
" example of writing a string"

```

To clear the value of a variable, you can use the command

```
clear variable_name
```

If you want to cancel the definitions of all variables which was given for a given program, you can use the **clear** command. The following are examples of how to use this command. Let's type in the Scilab console window:

```

--> //Defining variables x and y
-->x=3; y=-1;

```

and then press the [Enter] key. In the Variable Browser window, the values of the **x** and **y** variables changed their values and became equal to 3 and -1, respectively. If we type **clear x** command and try to view its value in the Variable Browser window, we will see that the **x** variable has been removed from the list of variables. By typing the name of the variable **x** in the Scilab console window, we get the message:

```

-->x
Undefined variable: x

```

If you run the command in the Scilab console window

```
-->clear
```

then we will see that the list of variables has cleared in the Variable Browser window. Such a cleanup is useful if several programs are executed sequentially and the values of variables which was defined in previous programs can affect to the result of the execution, if you forget to redefine them.

2.4 Scilab system variables

If the command does not contain an assignment sign, then by default the calculated value is assigned to the special system variable **ans** (short for the English word answer). Moreover, the resulting value can be used in subsequent calculations, but it is important to remember that the **ans** value changes after each command call without an assignment operator.

For example, type an arbitrary number in the Scilab console window, for example **45.67**, and press the [Enter] key. The **ans** variable will take a value equal to **45.67**. Enter the name of the **ans** variable in the console window, press the [Enter] key and make sure that its value is equal to **45.67**. Let's type the command **2*ans**, press the [Enter] key and see that the value of **ans** has doubled. Below is a listing of the steps which was taken:

```
-->45.67
ans =
    45.67
-->ans
ans =
    45.67
-->2*ans
ans =
    91.34
```

The result of the last calculation that was performed without an assignment operation is always stored in an **ans** variable.

Other system variables in Scilab start with a % character. Below is an incomplete list of them:

- **%eps** — epsilon (floating-point relative accuracy).

Description:

%eps is a predefined variable, $\%eps = 2^{(-16)}$. Calculations are not exact, but performed for a given precision.

Example:

```
--> %eps
%eps =
    2.220D-16
```

- **%i** — imaginary unit.

Description:

%i is imaginary unit, used to enter complex number.

Example:

```
--> %i
%i =
i
```

- **%inf** — infinity.

Description:

returns the representation for positive infinity.

Example:

```
--> %inf
%inf =
Inf
```

- **%nan** — not-a-number

Description:

%nan returns the representation for Not-a-Number (NaN).

Example:

```
--> %nan
%nan =
Nan
```

- **%pi** — ratio of circle's circumference to its diameter.

Description:

%pi returns the floating-point number nearest to the value to π .

Example:

```
--> %pi
%pi =
3.1415927
```

- **%e** — Euler number.

- .

Example:

```
--> %e
%e =
2.7182818
```

All of the above system variables can be used in mathematical expressions, for example:

```
-->a=5.4;b=0.1;
-->F=cos(%pi/3)+(a-b)*%e^2
F =
39.661997
```

The following is an example of an invalid reference to the %pi system variable:

```
-->sin(pi/2)
Undefined variable : pi
```

2.5 Dynamic typing of Scilab variables.

The type of a variable in Scilab can change dynamically depending on the value assigned to that variable. This means that it is possible, for example, to create a variable containing a real value, and then assign the value to this variable as a sequence of characters – strings, as is shown below:

```
-->x=1
  x  =
    1.
-->x+1
 ans =
    2.
-->x="foot"
  x  =
foot
-->x+'boll'
 ans =
football
```

It should be emphasized once again that Scilab is an untyped language, so there is no need to specify the type of a variable before assigning a value to it, and moreover, the type of a variable can change during the lifetime of a variable.

2.6 Entering a real number and displaying the results of calculations.

Numeric results can be represented with floating point (for example, $-3.2E - 6$, $-6.42E + 2$) or with a fixed point (for example, 4.12, 6.05, -17.5489) *.

Note: * Since in some, mainly English-speaking countries, when writing numbers, the integer part is separated from the fractional by a point, the term "floating point" appears in the terminology of these countries. In Ukraine, the integer part of the number from the fractional part is traditionally separated by a comma, therefore In Ukraine, the term "floating comma" is usually used to denote the same concept. Both options of the term "floating point" and "floating comma" can be found in the technical documentation.

Floating point numbers are represented in exponential form $me\pm p$, where m is the mantissa (integer or fractional number with decimal point), p is the order (integer number). In order to convert a number which is represented in exponential form (floating point) to a fixed point form, you need to multiply the mantissa m by ten to the power of the order p .

For example:

$$\begin{aligned} -6.42e+2 &= -6.42 \cdot 10^2 = -642; \\ 3.2e^{-6} &= 3.2 \cdot 10^{-6} = 0.0000032. \end{aligned}$$

When you are entering real numbers, a point is used to separate the fractional part. Examples of input and output of real numbers:

```
-->0.123
ans =
    0.123
-->-6.43e+2
ans =
   -643.
-->3.2e-6
ans =
 0.0000032
```

Consider an example of outputting the value of the system variable π and some variable q , which is defined by the user:

```
-->%pi
%pi =
    3.1415927
-->q=0123.4567890123456789
q =
   123.45679
```

It is easy to see that Scilab outputs only eight significant digits as a result. This is the default floating point format. In order to control the number of bits displayed on the screen or printed, use the `printf` command with a specified format that corresponds to the rules adopted for this command in the *C* language.

For example, let's define some variable g with 20 decimal places. And let's try to display this value in the command line in three ways. To do this, type the following lines in the text editor window and press the [Ctrl] + [L] keys:

```
g=1.12345678901234567890
printf("%1.20f",g)
printf("%1.16f",g)
```

In the Scilab console window, we get the following result:

```
-->g=1.12345678901234567890
g =
    1.1234568
-->printf("%1.20f",g)
1.12345678901234570000
-->printf("%1.16f",g)
1.1234567890123457
```

The example shows that for normal output Scilab outputs only 8 digits after the decimal point, and for a given format no more than 16, and the last of them is rounded, the rest are zeroed.

2.7 Boolean type of variables.

The Boolean type can store the values "true" or "false", In Scilab, "true" is stored in the system variable %t or %T (from English true), and "false" -% f or % F (from English false).

In the table 2.1 boolean and comparison operators that are used in Scilab are listed. Comparison operators accept data of any of the basic data types as input (real, complex and integer numbers, strings) and return a boolean value. Comparison operators can also be used to compare matrices.

Table 2.1 - Logical and comparison operators.

Designation	Description
a & b	logical "AND", "True" if a and b is equal to "true"
a b	logical "OR", "True" if a and b is equal to "true"
~ a	(~) logical not
a == b	"True" if a is equal to b
a ~= b or a <> b	"True" if a and b are not equal
a < b	"True" if a is less than b
a > b	"True" if a is greater than b
a <= b	"True" if a is less than or equal to b
a >= b	"True" if a is greater than or equal to b

The following example illustrates performing operations on boolean types. Let's assign logical values to two variables in the text editor window and execute the logical "AND" operator.

```
// variable a is assigned the value "true"
a=%T
// variable b is assigned the value of the
//result of comparing two numbers "0" and "1"
b=(0==1)
a&b
```

After launching the program for execution by pressing the [Ctrl] + [L] keys in the same time , we will see in the console window:

```
--> // variable a is assigned the value "true"
--> a=%T
a =
```

```

      T
--> // variable b is assigned the value of the
--> //result of comparing two numbers "0" and "1"
--> b=(0==1)
    b =
      F
--> a&b
    ans =
      F

```

2.8 Functions of Scilab

All functions used in Scilab can be divided into two classes: built-in; user-defined.

In general, a call to a function in Scilab looks like this:

```
variable_name = function_name (variable1 [, variable2, ...])
```

where **variable_name** is the variable to which the results of the calculation in the function will be written; this parameter can be absent, then the value calculated by the function will be assigned to the system variable **ans**;

function_name is the name of a built-in or previously created by the user function;

variable1, variable2, ... is a list of function arguments.

2.8.1 Elementary mathematical functions.

The number of built-in functions supported by Scilab is quite large, there are more than 50 trigonometric functions alone, all of them can be found in the Scilab help system (fig. 2.2). Familiarization with the main of them will occur by you when you will refer to them.

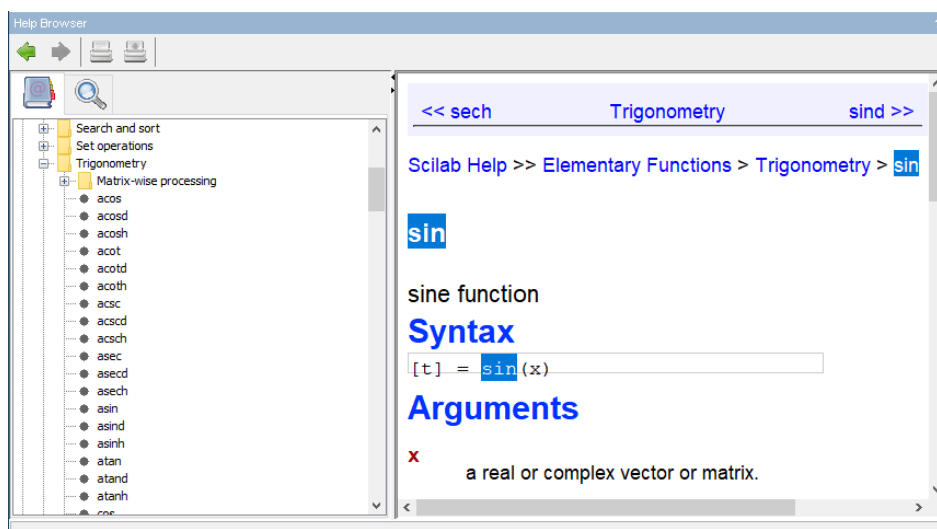


Figure 2.2 – Basic package functions in the Scilab help window.

Here we give only the elementary mathematical functions that are used most often (tab. 2.2).

Table 2.2 – Elementary mathematical functions

Function	Function description	Function	Function description
$\sin(x)$	sine of number x	$\text{atan}(x)$	arctangent of a number x
$\cos(x)$	cosine of number x	$\text{exp}(x)$	exponential of number x
$\tan(x)$	tangent of number x	$\log(x)$	Natural logarithm of x
$\text{cotg}(x)$	cotangent of number x	$\text{sqrt}(x)$	square root of number x
$\text{asin}(x)$	arcsine of number x	$\text{abs}(x)$	absolute value of number x
$\text{acos}(x)$	inverse cosine of x	$\log_{10}(x)$	base 10 logarithm of number x

Let's consider an example of calculating the value of an expression:

$$z = \sqrt{\left| \sin\left(\frac{x}{y}\right) \right|} \cdot e^{x^y}$$

```
-->x=1.2;y=0.3;
-->z=sqrt(abs(sin(x/y)))*exp(x**y)
z =
  2.5015073
```

Questions for self-examination for the second lecture:

1. What are text comments in the programs for?
2. How are text comments written in the Scilab programming language?
3. What operators are used to write mathematical expressions in the Scilab programming language?
4. What the priority level of operators for writing mathematical expressions in the Scilab programming language is?
5. How to determine the value of a variable in the Scilab programming language?

6. How to write the name of the variable?

7. What is the **clear** command for?

8. What is a system variable and how it can be used?

Questions for self-examination for the fourth lecture:

1. What is dynamic typing of Scilab variables?

2. How to enter a real number?

3. How displaying the results of calculations?

4. What is Boolean type of variables and where are they used?

5. What types of functions in Scilab do you know?

Lecture 3

The purpose of the lecture is to get acquainted with the type of Scilab functions, learn how to create user functions in two ways and with the definition of an array and a matrix, how they are formed, to study actions on vectors and matrices.

2.8.2 User-defined functions.

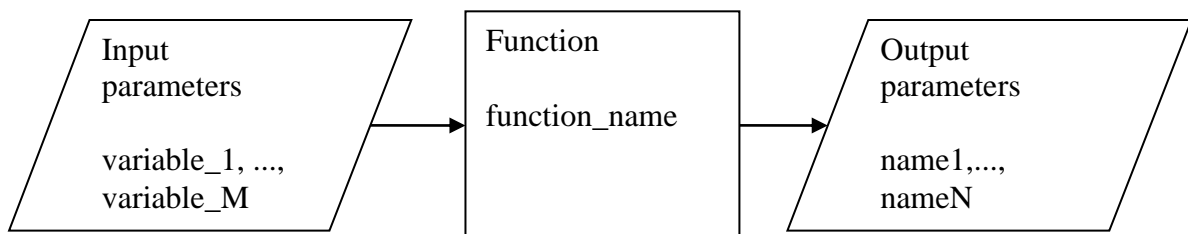
Which function, as a rule, is intended for repeated use; it has input parameters and cannot be executed without first specifying them. Let's look at several ways to create functions in Scilab.

The first way is to use the **deff** command (from the English define function - define a function), which in general can be written as follows:

```
deff(' [name1,...,nameN] = function_name (variable_1, ...,  
variable_M) ', ' name1 = expression1; ...; nameN =  
expressionN ')
```

where:

- **name1, ..., nameN** is a list of output parameters, that is, variables that will be assigned the final result of calculations;
- **function_name** - the name with which this function will be called;
- **variable_1, ..., variable_M** - input parameters.



The following is the simplest way to use the **deff** command. Let's create and use a function to evaluate an expression (the value of this expression has already been calculated in section 2.8.1).

$$z = \sqrt{\left| \sin\left(\frac{x}{y}\right) \right|} \cdot e^{x^y}$$

```
deff(' [z]=fun1(x,y) ', ' z=sqrt(abs(sin(x/y))) *exp(x**y) ');  
x=1.2; y=0.3; z=fun1(x,y)
```

After launching the program for execution by pressing the [Ctrl] + [L] keys in the same time, we get in the console window:

```
-->deff(' [z]=fun1(x,y) ', 'z=sqrt(abs(sin(x/y)))*exp(x**y) ');
-->x=1.2; y=0.3; z=fun1(x,y)
z =
    2.5015073
```

In the following example, we will create a function that can be used to find the roots of a quadratic equation of the form $ax^2 + bx + c = 0$ using the formulas $D = b^2 - 4ac$; $x_{1,2} = (-b \pm \sqrt{D}) / (2a)$

Let's type the program codes in the text editor window:

```
deff(' [x1,x2] = korni(a,b,c) ', ...
'd=b**2-4*a*c; x1=(-b+sqrt(d))/(2*a); x2=(-b-sqrt(d))/(2*a); ');
[x1,x2] = korni(-2,-3,5)
```

After started the program for execution by pressing the [Ctrl] + [L] keys, we will have in the console window:

```
--> deff(' [x1,x2] = korni(a,b,c) ', ...
-->'d=b**2-4*a*c; x1=(-b+sqrt(d))/(2*a); x2=(-b-sqrt(d))/(2*a); ');
--> [x1,x2] = korni(-2,-3,5)
    x2 =
        1.
    x1 =
    -2.5
```

The second way to create a function is to use a construction like this:

```
function[name1,...,nameN] = name_function(variable_1,
    ..., variable_M)
// function body
endfunction
```

here: **name1, ..., nameN** – a list of output parameters, that is, variables that will be assigned the final result of calculations;

name_function – the name with which this function will be called;

variable_1, ..., variable_M) – input parameters.

All variable names which was defined inside the function, as well as names from the list of input and output parameters are perceived by the system as local, that is, they are considered defined only inside the function.

Generally speaking, functions in Scilab play the role of subroutines. A subroutine is the same function, but it has no output parameters. For the convenience of using the functions, it is advisable to save their text as separate files.

Moreover, **the file name must necessarily coincide with the function name**. The extension for function files is usually sci or sce.

The function is accessed in the same way as any other built-in function of the system, that is, from the command line. However, the functions stored in separate files must be previously loaded into the system, for example, using the **exec**

(file_name) operator or the **File - Execute** . . . command of the main menu. ... , which is, in general, the same thing.

As an example, consider solving a cubic equation.

Cubic equation of the form

$$ax^3 + bx^2 + cx + d = 0,$$

after division by a equation takes the canonical form:

$$x^3 + rx^2 + sx + t = 0;$$

where $r = b/a$; $s = c/a$; $t = d/a$.

In the last equation, we make the change $x = y - r/3$ and we have the following reduced equation:

$$y^3 + py + q = 0;$$

where: $p = (3s - r^2)/3$, $q = 2r^2/27 - rs/3 + t$.

The number of real roots of the reduced equation depends on the sign of the discriminant $D = (p/3)^3 + (q/2)^3$

Table 2.2 – Number of roots of a cubic equation.

Discriminant	The number of real roots	Number of complex roots
$D > 0$	1	2
$D < 0$	3	–

The roots of the given equation can be calculated using the Cardano formula:

$$y_1 = u + v, \quad y_2 = \frac{-(u+v)}{2} + \frac{(u-v)}{2}i\sqrt{3}, \quad y_3 = \frac{-(u+v)}{2} - \frac{(u-v)}{2}i\sqrt{3}.$$

here:

$$u = \sqrt[3]{\frac{-q}{2} + \sqrt{D}}, \quad v = \sqrt[3]{\frac{-q}{2} - \sqrt{D}}.$$

Below is the text of the function that implements the above method for solving the cubic equation. Let's type the text of the function in the text editor and save it with the name cub.sce:

```
//файл cub.sce
function [x1,x2,x3]=cub(a,b,c,d)
r=b/a;
s=c/a;
t=d/a;
p=(3*s-r^2)/3;
q=2*r^2/27-r*s/3+t;
D=(p/3)^3+(q/2)^3;
u=(-q/2+sqrt(D))^(1/3);
v=(-q/2-sqrt(D))^(1/3);
```

```

y1=u+v;
y2=-(u+v)/2+(u-v)/2*%i*sqrt(3);
y3=-(u+v)/2-(u-v)/2*%i*sqrt(3);
x1=y1-r/3;
x2=y2-r/3;
x3=y3-r/3;
endfunction

```

In the editor window, type the command for calling the file and the command for calling the function:

```

exec('C:\Users\D\Desktop\Примеры\cub.sce')
[x1,x2,x3]=cub(3,-20,-3,4)

```

Let's start the program for execution by pressing the [Ctrl] + [L] keys in the same time, we will get the solution of the cubic equation in the console window:

```

-->exec('E:\Примеры\cub.sci');
-->[x1,x2,x3]=cub(3,-20,-3,4)
x3 =
    2.3745111
x2 =
   -1.8272
x1 =
    6.1193556

```

3 ARRAYS AND MATRICES IN Scilab.

If you want to work with a lot of homogeneous data, it is convenient to use arrays. For example, you can create an array to store numeric or character data. In this case, instead of creating a set of variables in which data is stored, it is enough to create one array, with many ordered elements, where each element will have a serial number and each element can be assigned its own value.

Thus, an **array** is a multiple data type with a fixed number of elements. Like any other variable, the array must be given a name. A variable that is simply a list of data is called a **one-dimensional array**, or **vector**. To access the data stored in a specific element of the array, you must specify the name of the array and the ordinal number of this element, which is called the **index**.

If it becomes necessary to store data in the form of tables, in the format of rows and columns, then it is necessary to use **two-dimensional arrays** or **matrices**. To access data stored in such an array, you must specify the name of the array and two indices, the first must correspond to the row number, and the second to the number of the column in which the required element is stored.

The **lower bound of indexing** in Scilab is equal to **one**. Indexes can only be positive integers.

3.1 Input and formation of arrays and matrices.

Scilab uses the following symbols to define matrices:

- square brackets "[" "]" denote the beginning and end of the enumeration of matrix elements,
- "space" or "," separates matrix elements that are in one line,
- semicolon ";" separates the rows of the matrix.

You can define a one-dimensional array in Scilab as follows:

name=Xn : dX : Xk

where: **name** is the name of the variable to which the generated array will be written,

xn - the value of the first element of the array,

dx - the value of the last element of the array,

xk - the step by which each next element of the array is formed, that is, the value of the second element will be equal to $Xn+dx$, the third - $Xn+ dx+dx$, and so on up to **Xk**.

If the **dx** parameter is absent in the construction, this means that by default it takes on a value equal to one, that is each next element of the array is equal to the value of the previous one plus one:

```
name=Xn:Xk
// let's determine the values of the variables Xn,dX и Xk
Xn=-3.5;dX=1.5;Xk=4.5;
// set the values of the array X
X=Xn:dX:Xk
// as a function argument sin() transmit the argument X
// each element of which is divided by two
Y=sin(X/2)
// Define array A with has six elements: from 0 to 5
A=0:5
0:5
// the system variable ans has the value of elements
// of the array A
ans/2+%pi
```

Running the program for execution by pressing the [Ctrl] + [L] keys at the same time and we will get:

```
--> // determine the values of the variables Xn,dX и Xk
-->Xn=-3.5;dX=1.5;Xk=4.5;
-->X=Xn:dX:Xk
X =
  - 3.5  - 2.  - 0.5    1.    2.5    4.
-->Y=sin(X/2)
```

```

Y =
- 0.9839859 - 0.8414710 - 0.2474040 0.4794255
0.9489846 0.9092974
-->A=0:5
A =

0. 1. 2. 3. 4. 5.
-->0:5
ans =
0. 1. 2. 3. 4. 5.
-->ans/2+%pi
ans =
3.1415927 3.6415927 4.1415927 4.6415927
5.1415927 5.6415927

```

Another way to set the structure and values of matrix elements in Scilab is to enter all the matrix data element by element.

So, to define a **row vector**, enter the name of the array, and then after the assignment sign, in square brackets separated by a space or comma, write down the values of the array elements:

name=[x1 x2 ... xn] or **name=[x1, x2, ..., xn]**

Example of inputting a row vector:

```

-->V=[1 2 3 4 5]
V =
1. 2. 3. 4. 5.
-->W=[6.1,4.56,-45.34,0.01]
W =
6.1 4.56 - 45.34 0.01

```

The values of the elements of the **column vector** are entered separated by semicolons:

name=[x1; x2; ...; xn]

Example of inputting of a column vector:

```

-->ST=[1;2;3]
ST =
1.
2.
3.

```

You can refer to an element of a vector by specifying the name of the array and

the ordinal number of the element in round brackets:

name (index)

For example:

```
-->W=[1.1,2.3,-0.1,5.88];
-->W(1)+2*W(3)
ans =
    0.9
```

Matrix elements are also entered in square brackets, with the line elements separated from each other by a space or comma, and the lines are separated by a semicolon:

```
name=[x11, x12, ..., x1n; x21, x22, ..., x2n; ...;
      xm1, xm2, ..., xmn;]
```

You can refer to an element of a two-dimensional matrix by specifying the row number and column number at the intersection of which the element is located after the name of the matrix. The index of the line number and the index of the column number are indicated in round brackets, where indexes are separated by commas:

name (index1, index2)

Let's consider an example of defining a two-dimensional matrix and accessing its elements:

```
-->Array=[1 2 3;4 5 6;7 8 9]
Array =
    1.    2.    3.
    4.    5.    6.
    7.    8.    9.
--> Array(1,2)^Array(2,2)/Array(3,3)
ans =  3.5556
```

The example which just is shown corresponds to this action:

```
-->2**5/9
ans =
    3.5555556
```

In addition, matrices and vectors you can be formed by composing them from previously specified matrices and vectors:

```

-->v1=[1 2 3]; v2=[4 5 6]; v3=[7 8 9];
-->// Horizontally conjunction of the row vectors:
-->V=[v1 v2 v3]
V =
    1.    2.    3.    4.    5.    6.    7.    8.    9.
-->// Vertical conjunction of the row vectors:
-->// the result is the matrix:
-->V=[v1; v2; v3]
V =
    1.    2.    3.
    4.    5.    6.
    7.    8.    9.
-->// Horizontally conjunction of the matrices:
-->M=[V V V]
M =
    1.    2.    3.    1.    2.    3.    1.    2.    3.
    4.    5.    6.    4.    5.    6.    4.    5.    6.
    7.    8.    9.    7.    8.    9.    7.    8.    9.
-->// Vertical conjunction of the matrices:
-->M=[V;V;V]
M =
    1.    2.    3.
    4.    5.    6.
    7.    8.    9.
    1.    2.    3.
    4.    5.    6.
    7.    8.    9.
    1.    2.    3.
    4.    5.    6.
    7.    8.    9.

```

An important role, when you are working with matrices, has the colon ":". By specifying it instead of an index when you are using an array, you can access groups of its elements, that is, rows or columns. Let's type the program in the text editor window:

```

//Let the matrix is given A
A=[5 7 6 5; 7 10 8 7;6 8 10 9;5 7 9 10]
//Let select the second column from matrix A
A(:,2)

```

We will receive the answer in the console:

```
ans =
```

```
8.  
7.  
10.  
9.  
9.  
10.
```

Let's add two lines to the console window:

```
//Select the third row from the matrix A  
A(3,:) 
```

After that in the console, we'll get the answer:

```
ans =  
6.    8.   10.    9.
```

And then we will carry out a number of operations, the meaning of which are indicated in the comment lines:

```
// Let select from the matrix A the submatrix M  
// which is located at the intersection of  
// the 3rd and 4th rows and the 2nd  
//and 3rd columns  
M=A(3:4,2:3)
```

Let's pressing the [Ctrl] + [L] keys at the same time and we get the answer in the console window:

```
M =  
8.    10.  
7.     9.
```

Consider the following example. After typing the program code into the editor window:

```
//Remove second column from matrix A  
A(:,2)=[]
```

Let's pressing the [Ctrl] + [L] keys at the same time and we get the answer in the console window:

```
A =  
5.    6.    5.  
7.    8.    7.
```

```
6.    10.   9.
5.    9.    10.
```

Next example. After typing the program code into the editor window:

```
//Remove the third row from the matrix A
A(3,:)=[]
```

Press the [Ctrl] + [L] keys at the same time and we get the answer in the console window:

```
A =
5.    6.    5.
7.    8.    7.
5.    9.   10.
```

Next example. After typing the program code into the editor window

```
//Represent the matrix M as a column vector
v=M(:)
```

Press the [Ctrl] + [L] keys at the same time and we get the answer in the console window:

```
v =
8.
7.
10.
9.
```

Next example. After typing the program code into the editor window:

```
//Extract from the column vector v the elements
//from the second to the fourth
b=v(2:4)
```

Press the [Ctrl] + [L] keys at the same time and we get the answer in the console window:

```
b =
7.
10.
9.
```

Next example. After typing the program code into the editor window:

```
//Let's remove second element from array b
```



```
b(2)=[ ]
```

Press the [Ctrl] + [L] keys at the same time and we get the answer in the console window:

```
b =  
7.  
9.
```

3.2 Actions with vectors and matrices.

When you are working with rows and columns in Scilab, the following mathematical operations are permissible for you:

- plus (+)

Numerical addition. Text concatenation (gluing).

Syntax:

```
X + Y
```

```
str1 + str2
```

Arguments:

X,Y – scalars, vectors, matrices of booleans, numbers, polynomials, or rationals.

str1, str2 – two texts, vectors or matrices of texts.

Description:

For numeric operands, the addition has its usual meaning.

For two texts, + concatenates (glues) them together.

If an operand is an array and the other one is a scalar, the scalar is applied (added or glued) to each component of the array.

Adding booleans together or to numbers of integer, performs the implicit conversions %F => 0 and %T => 1 before processing. The result has the type of the input numbers, or is decimal for booleans added together.

If an operand is the empty matrix [], the result is [].

Examples:

```
--> [1, 2] + 1  
ans =  
2. 3.  
  
--> [%f %f %t %t] + [%f %t %f %t]  
ans =  
0. 1. 1. 2.  
  
--> %f + [-1 0 2]  
ans =  
-1. 0. 2.  
  
--> %t + [-1 0 2]  
ans =  
0. 1. 3.
```

```

--> [] + 2
ans =
    []
--> "con" + ["catenate" "crete" "sole"]
ans =
!concatenate concrete console !

```

- minus (-)

Subtraction operator. Sign change.

Syntax:

X - Y

-X

Arguments:

X, Y – scalars, vectors, matrices of booleans, numbers.

Description:

Subtraction for numeric operands, the subtraction has its usual meaning. If one of the operands is a scalar, then the subtraction is performed with each component of the other operand.

As soon as a boolean is involved in a subtraction with a number (decimal or integer), it is automatically converted in the type (and integer type) of the number before performing the subtraction or the sign change, as %F => 0 and %T => 1.

Whatever is the (regular) type of X, then the empty matrix []-X, X-[], and -[] return [].

The addition and subtraction operations are defined for matrices of the same dimension or vectors of the same type, that is, you can sum (subtract) either column vectors or row vectors of the same length.

Examples:

```

--> [] - 2
ans =
    []
--> 1 - []
ans =
    []
--> [2, 5] - 1
ans =
    1.    4.
--> [2, 5] - [3 -2]
ans =
    -1.    7.
--> -[%f %t]
ans =
    0.   -1.
--> [%f %f %t %t] - [%f %t %f %t]
ans =
    0.   -1.    1.    0.

```

- quote (')

Transpose operator, string delimiter.

Description:

Quote (') is used for conjugate transpose of matrix.

Quote (.) is used for non-conjugate transpose of matrix.

If in some matrix the rows are replaced by the corresponding columns, then the transposed matrix will be obtained.

Single (') or double (") quotes are also used to define character strings. (Character strings are defined between two quotes). A quote within a character string is denoted by two quotes.

Examples:

```
--> [1, 2; 3,4]
ans =
    1.    2.
    3.    4.
--> [1, 2; 3,4] '
ans =
    1.    3.
    2.    4.
--> x='This is a character string'
x =
    "This is a character string"
--> x="This is another character string"
x =
    "This is another character string"
--> 'He said:""Very Good"'
ans =
    "He said:""Very Good""
```

- star (*)

Multiplication operator.

Description:

The operation of multiplying a matrix by a number. Each member in the matrix is multiplied by the number.

Examples:

```
--> A = [1 2;3 4]
A =
    1.    2.
    3.    4.
--> A*3
ans =
    3.    6.
    9.   12.
```

- dot star (.*)

Element-wise matrix multiplication;

Description:

The first member of the matrix is multiplied by the first member of the second matrix, then the second member of the matrix is multiplied by the second member of the second matrix, and so on until the last members of the matrix will be multiplied.

Examples:

```
--> A = [1 2;3 4]
A =
    1.    2.
    3.    4.
--> B = [5 6;7 8]
B =
    5.    6.
    7.    8.
--> A.*B
ans =
    5.    12.
   21.    32.
```

- power (.^)

Power operation.

Syntax

t = A.^ b

Arguments:

A, t – a scalar, vector, or matrix, decimal or complex numbers or polynomials.

b – a scalar, vector or matrix, decimal or complex numbers.

Examples:

```
--> A = [1 2 ; 3 4]
A =
    1.    2.
    3.    4.
--> A.^3
ans =
    1.    8.
   27.   64.
--> B=[1 2;3 4]
B =
    1.    2.
    3.    4.
--> A.^B
ans =
```

1. 4.
27. 256.

- element-wise left division (./)
- element-wise right division (./)

Description:

$(A \setminus B) \Rightarrow (A^{-1} \cdot B)$, the operation can be applied to solve a matrix equation of the form $A \cdot X = B$, where X is an unknown vector.

$(B / A) \Rightarrow (B \cdot A^{-1})$ are used to solve matrix equations of the form $X \cdot A = B$.

(

Examples:

```
--> // Solve matrix equations A*X=B and X*A=B.
--> A=[3 2;4 3];
--> B=[-1 7;3 5];
--> // Solving the matrix equation AX=B:
--> X=A \ B
X =
    - 9.    11.
    13.   - 13.
--> // Checking of the equation solution
--> A*X-B
ans =
    0.    0.
    0.    0.
```

In addition, if any function is applied to some given vector or matrix then the result will be a new vector or matrix of the same dimension, but the members of the original matrix will be transformed in accordance with the applied function:

```
--> x=[0.1 -2.2 3.14 0 -1];
--> sin(x)
ans =
    0.0998334   - 0.8084964    0.0015927    0.   - 0.8414710
```

Questions for self-examination for the third lecture:

1. Explain how to create a user-defined function in the first way?
2. Explain how to create a user-defined function in the second way?
3. What is a vector and a matrix?
4. How to form a vector?
4. How to form a matrix?
6. What actions can be performed on vectors in Scilab?
7. What actions on matrices can be performed in Scilab?

Lecture 4

The purpose of the lecture is to get acquainted with the functions of determining matrices, creating a matrix of ones, creating a matrix of zeros, creating an identity matrix, creating a matrix of random numbers, ordering an array, determining the size of an array and to get acquainted with the functions of determining the length of a matrix, calculating the sum and product of matrix elements, calculating the determinant of a matrix, calculating the largest and smallest elements in a matrix, calculating the average value of matrix elements, obtaining an inverse matrix. Get acquainted with symbolic matrices and operations of them.

3.3 Special matrix functions in Scilab.

There are special functions for working with matrices and vectors in Scilab. Let's consider the most commonly used ones.

3.3.1 Functions for definition of matrix.

3.3.1.1 The function of converting a matrix v to a matrix of a different size:

`y=matrix(v,n,m)`

here v is a vector or matrix;

n, m - integers of the number of rows and columns;

y is a vector or matrix.

For a vector or matrix with $n \times m$ elements, command **`y = matrix(v,n,m)`** or the equivalent command **`y = matrix(v,[n,m])`**, converts the vector or matrix v to an $n \times m$ matrix with column-wise v .

To get acquainted with the operation of the **`matrix`** function, let us set the text matrix **`D`** and 4 variants of the **`matrix`** function in the editor window:

```
D=[1 2;3 4;5 6]
matrix(D,2,3)
matrix(D,1,6)
matrix(D,6,1)
matrix(D,3,2)
```

Let's run the program by pressing the [Ctrl] + [L] keys. In the Scilab console window we will get:

```
-->D=[1 2;3 4;5 6]
D =
  1.    2.
  3.    4.
```

```

    5.    6.
-->matrix(D,2,3)
ans =
    1.    5.    4.
    3.    2.    6.
-->matrix(D,1,6)
ans =
    1.    3.    5.    2.    4.    6.
-->matrix(D,6,1)
ans =
    1.
    3.
    5.
    2.
    4.
    6.
-->matrix(D,3,2)
ans =
    1.    2.
    3.    4.
    5.    6.

```

3.3.1.2 The function creating a matrix which is composed of ones:

```

y=ones(m1,m2,...)
y=ones(x)
y=ones()

```

where \mathbf{x} , \mathbf{y} are matrices;

$\mathbf{m1}$, $\mathbf{m2}$, ... are integers defining the dimension of the matrix \mathbf{v} .

The `ones(m1,m2)` function returns a matrix of dimensions ($\mathbf{m1}$, $\mathbf{m2}$) filled with ones.

The `ones(m1,m2,...,mn)` function creates a multidimensional matrix of size ($\mathbf{m1}$, $\mathbf{m2}$, ..., \mathbf{mn}) which is filled with ones.

The `ones(x)` function returns a matrix of the same size as the ones-filled matrix \mathbf{x} .

Let's look at some examples of using the ones function. For this, in the text editor window, we will type:

```

ones(3)
ones(3,3)
ones()
m=3; n=2;
//A matrix of dimensions (mxn)will be formed
//An (m× n) matrix will be formed

```

```

X=ones (m,n)
//A two-dimensional matrix M will be formed
M=[1 2 3;4 5 6]
//A matrix Y consisting of ones will be formed,
//with same dimension as matrix M
Y=ones (M)

```

Let's run the program by pressing the [Ctrl] + [L] keys. In the Scilab console window we will get:

```

--> ones (3)
ans =
  1.
--> ones (3,3)
ans =
  1.  1.  1.
  1.  1.  1.
  1.  1.  1.
--> ones ()
ans =
  1.
--> m=3; n=2;
--> //A matrix of dimensions (mxn)will be formed
--> //An (mx n) matrix will be formed
--> X=ones (m,n)
X =
  1.  1.
  1.  1.
  1.  1.
--> // A two-dimensional matrix M will be formed
--> M=[1 2 3;4 5 6]
M =
  1.  2.  3.
  4.  5.  6.
--> // A matrix Y consisting of ones will be
formed,
--> //with same dimension as matrix M
--> Y=ones (M)
Y =
  1.  1.  1.

```


1. 1. 1.

3.3.1.3 The function creating a matrix which is composed of zero:

```
y=zeros ()  
y=zeros (x)  
y=zeros (m1 ,m2 , . . )
```

here **x**, **y** are matrices;

m1 , **m2** , . . . are integers.

The functions for creating a matrix which are consisting of zeros:

zeros (m1 ,m2) function returns a matrix of dimensions (**m1** ,**m2**);

zeros (m1 ,m2 , . . ,mn) function creates a multidimensional matrix of size (**m1** ,**m2** , . . . ,**mn**) , which is filled with zeros;

zeros (A) function returns a matrix of the same size as the matrix **A**;

zeros () returns a single zero, that is, a 1×1 matrix equal to zero.

Let's look examples of using the zeros function. For this, in the text editor window we will type:

```
zeros ()  
zeros (3 ,3)  
zeros (2 ,3 ,2) // Two matrices of dimensions 2x3  
// or matrix of dimensions 3D  
M=[1 2 3 4 5];  
Z=zeros (M)
```

Let's run the program by pressing the [Ctrl] + [L] keys. In the Scilab console window we will get:

```
--> zeros ()  
ans =  
0.  
--> zeros (3 ,3)  
ans =  
0. 0. 0.  
0. 0. 0.  
0. 0. 0.  
--> zeros (2 ,3 ,2)  
// Two matrices of dimensions 2x3  
ans =  
(: , : , 1)
```

```

    0.    0.    0.
    0.    0.    0.
(:, :, 2)
    0.    0.    0.
    0.    0.    0.
--> // or matrix of dimensions 3D
--> M=[1 2 3 4 5];
--> Z=zeros(M)
Z =
    0.    0.    0.    0.    0.

```

3.3.1.4 The function creating the identity matrix with undefined dimensions.

```

X=eye(m,n)
X=eye(A)
X=eye()

```

where: **A, X** – matrices, hypermatrices, or syslin lists;

m x n – integer values: numbers of rows and columns for **X**.

The **X = eye(m,n)** defines the $(m \times n)$ identity matrix (in the identity matrix, the elements of the main diagonal are equal to one, and all the rest are zero).

The function **X = eye(A)** defines the identity matrix of the same dimension as the matrix **A**.

The **X = eye()** function forms an identity matrix of undefined sizes. The dimensions will be determined when the given identity matrix will be summed with the matrix whose dimensions are determined.

Let's consider some examples of using the **eye** function. For this, in the text editor window, we will type the following code:

```

eye(3,4)
A=[1,2,3;4,5,6;7,8,9]
eye(A)
m=3; n=4;
E=eye(m,n)
M=[0 1;2 3]
// Forms a matrix E of the same dimension
// as the matrix M
E=eye(M)
M=[1 2;3 4;5 6]
E=eye()
A=E+M

```

M-E

Let's run the program by pressing the [Ctrl] + [L] keys. In the Scilab console window we will get:

```
--> eye(3,4)
ans =
  1.  0.  0.  0.
  0.  1.  0.  0.
  0.  0.  1.  0.
--> A=[1,2,3;4,5,6;7,8,9]
A =
  1.  2.  3.
  4.  5.  6.
  7.  8.  9.
--> eye(A)
ans =
  1.  0.  0.
  0.  1.  0.
  0.  0.  1.
--> m=3; n=4;
--> E=eye(m,n)
E =
  1.  0.  0.  0.
  0.  1.  0.  0.
  0.  0.  1.  0.
--> M=[0 1;2 3]
M =
  0.  1.
  2.  3.
--> // Forms a matrix E of the same dimension
--> // as the matrix M
--> E=eye(M)
E =
  1.  0.
  0.  1.
--> M=[1 2;3 4;5 6]
M =
  1.  2.
  3.  4.
```

```

    5.    6.
--> E=eye()
    E =
eye *
    1.
--> A=E+M
    A =
    2.    2.
    3.    5.
    5.    6.
--> M-E
    ans =
    0.    2.
    3.    3.
    5.    6.

```

3.3.1.5 The function to create a matrix of random numbers.

```

r = rand()
r = rand(m1,m2,...)
r = rand(m1,m2,...,key)
r = rand(x)
r = rand(x,key)
rand(key)
key = rand("info")

```

here: **m1**, **m2**, ... - integers defining the dimension of the matrix of random numbers **r**;

key - a variable of the "string" type that determines the law of distribution of random numbers. The available values for the variable are "**uniform**" for an equiprobable distribution and "**normal**" for a normal distribution. By default, the variable key is defined as "**uniform**";

x is a real or complex matrix;

r is a real array of double precision numbers of size **m1** × **m2** × ... with random elements.

For an equiprobable distribution, random numbers are evenly distributed in the range from 0 to 1. For a normal distribution, the characteristic of the distribution of random numbers will be the mathematical expectation equal to zero and the variance equal to one.

It should be noted that these sequences are called random by convention, since the sequences will be repeated during repeated calculations. This sequence is called pseudo-random. In what follows, for simplicity of presentation, pseudo-random

sequences will be called random. To get a truly random sequence, you need to randomly determine the generator seed for the **rand()** function.

rand("seed", s)

sets the seed of the random number generator to **s** (by default **s** is = 0 on first call).

The functions **rand** return:

r=rand() returns a 1-by-1 matrix of doubles, with one random value;

r=rand(m1, m2) is a random matrix with dimension m1-by- m2;

r=rand(m1, m2, . . . , mn) returns a random matrix with dimension m1-by-m2-by-... -by-mn;.


r=rand(a) returns the random matrix with the same size as the matrix **a**.

The matrix **r** is real if **a** is a real matrix and **r** is complex if **a** is a complex matrix.

The function **rand("uniform")** or **rand("u")** sets the generator to a uniform random number generator. Random numbers are uniformly distributed in the interval [0,1].

The function **rand("normal")** or **rand("n")** sets the generator to a normal (Gauss-Laplace) random number generator, with mean which is equal to 0 and variance which is equal to 1.

The function **key=rand("info")** return the current distribution of the random generator ("**uniform**" or "**normal**")

In order to get acquainted with the capabilities of the rand function, type the command «**rand**» in the command line of the console window, highlight this word, and right-click on it (RMB). In the context menu that opens, select the option "Help about 'rand'". In the help service window that opens, read the description of the function «**rand**». Find the description section "Examples" and click LMB on the text editor icon  which is located on the lilac field of the example description (fig. 3.1). The example text will be transferred to the window of the text editor.

```
// Get one random double (based on the current distribution)  
r=rand()  
// Get one 4-by-6 matrix of doubles (based on the current distribution)  
r=rand(4,6)  
// Get one 4-by-6 matrix of doubles with uniform entries  
r=rand(4,6,"uniform")  
// Produce a matrix of random doubles with the same size as x  
x=rand(4,4);  
r=rand(x,"normal")  
// Produce a 2-by-2-by-2 array of random doubles  
r=rand(2,2,2)
```

Figure 3.1 - Text of an example of using the function **rand**.

Let's run the program by pressing the [Ctrl] + [L] keys and in the Scilab console window we will get:

```

--> // Get one random double (based on the current distribution)
--> r=rand()
r =
    0.2113249
--> // Get one 4-by-6 matrix of doubles (based on the current distribution)
--> r=rand(4,6)
r =
    0.7560439  0.6283918  0.068374  0.1985144  0.2164633  0.9329616
    0.0002211  0.8497452  0.5608486  0.5442573  0.8833888  0.2146008
    0.3303271  0.685731  0.6623569  0.2320748  0.6525135  0.312642
    0.6653811  0.8782165  0.7263507  0.2312237  0.3076091  0.3616361
--> // Get one 4-by-6 matrix of doubles with uniform entries
--> r=rand(4,6,"uniform")
r =
    0.2922267  0.5935095  0.6325745  0.4818509  0.1280058  0.6856896
    0.5664249  0.5015342  0.4051954  0.2639556  0.7783129  0.1531217
    0.4826472  0.4368588  0.9184708  0.4148104  0.211903  0.6970851
    0.3321719  0.2693125  0.0437334  0.2806498  0.1121355  0.8415518
--> // Produce a matrix of random doubles with the same size as x
--> x=rand(4,4);
--> r=rand(x,"normal")
r =
    0.8895818  1.7315364  0.6472466 -1.2660345
   -0.9060355  2.5692052  1.3250986 -0.4619728
   -0.0047727 -0.4122445 -0.3449928  1.1932161
   -1.7322409 -0.2976452  0.1317441  0.3176977
--> // Produce a 2-by-2-by-2 array of random doubles
--> r=rand(2,2,2)
r =
(:, :, 1)
    0.4829179  0.8400886
    0.2232865  0.1205996
(:, :, 2)
    0.2855364  0.8494102
    0.8607515  0.5257061

```

3.3.1.6 Array ordering function:

gsort (X)

The function performs the ordering of the vector X; if X is a matrix, sorting will be done by columns.

Let's consider an example of how the function works.

```

-->b=[2 0 1]
b =
    2.    0.    1.

```

```

-->gsort(b) //Sort descending
ans =
    2.    1.    0.
-->-gsort(-b) //Sort Ascending
ans =
    0.    1.    2.
-->A=[1 2 0;-1 3 1;4 -2 5];
-->gsort(A) //Sort a matrix
ans =
    5.    2.    0.
    4.    1.   - 1.
    3.    1.   - 2.

```

3.3.2 Functions for calculating some numeric characteristics of matrices.

3.3.2.1 Function for determining the size of an array:

```

[n1, n2, n3, ...] = size(x)
n = size(x, sel)

```

here: **x** is two-dimensional or n-dimensional array of any type;

sel is a positive scalar with integer value or one of the character strings 'r', 'c' or '*'

n1, n2, ... are numbers with integer values;

n is a number with integer value.

The **size(x)** function returns in each argument [**n1, n2, ...**], the value of the corresponding dimension.

The expression **n = size(x, sel)** can be used to determine the dimension of array **x**:

- set **sel** to **1** or 'r' to get the number of rows;
- set **sel** to **2** or 'c' to get the number of columns.
- Set **sel** to **m**, where **m** is a positive integer to get the **m**th dimension. If **m** is greater than dimension of the array **x**, then **size(x,m)** returns **1**.
- Set **sel** to '*' to get the product of the dimensions.

Let's consider an example of how the function works.

```

x=rand(3, 2)
[n, m] = size(x)
// Function returns the number of rows
n = size(x, "r")
// Function returns the number of columns
m = size(x, "c")
// Function returns the the product of the

```

dimensions

```
nm = size(x, "*")
```

Let's run the program by pressing the [Ctrl] + [L] keys. In the Scilab console window we will get:

```
--> x=rand(3, 2)
x =
  0.993121    0.050042
  0.6488563  0.7485507
  0.9923191  0.4104059
--> [n, m] = size(x)
n =
  3.
m =
  2.
--> // Function returns the number of rows
--> n = size(x, "r")
n =
  3.
--> // Function returns the number of columns
--> m = size(x, "c")
m =
  2.
--> // Function returns the the product of the
dimensions
--> nm = size(x, "*")
nm =
  6.
```

3.3.2.2 Function for determining the length of the matrix:

n = length(M)

where: **M** - matrix;

n is an integer or a matrix of integer values.

For an ordinary matrix, **n** is an integer equal to the product of the number of rows and columns of the matrix **M**. It is also correct for a matrix of booleans.

For a matrix whose element is a character variable, the **length()** function returns the length of that variable in characters, including spaces. For a matrix composed of several rows, the **length()** function returns the length of all elements of the matrix, in other words, the length of all rows.

For a sparse matrix, the function does not work correctly, so it is recommended to use the `size(..., '*')` function for it.

Let's consider an example of using the function:

```
-->length([123,456 ; 789,101 ])// Matrix
ans =
    4.
-->length(['hello world'])// Row
ans =
    11.
```

3.3.2.3 Function for calculating the sum of array elements:

```
y=sum(x)
y=sum(,orientation)
```

where: **x** is array of real, complex, logical values, polynomials or rational fractions;
orientation - (orientation) can be either a string with possible values **"*"**, **"r"**, **"c"**;;

y is a scalar or array.

For an array **x** function `y=sum(x)` returns the number **y**, which is the sum of all the elements of **x**.

The function `y=sum(x,orientation)` returns in **y** the sum of **x** along the dimension given by orientation:

- if orientation is equal to **1** or **"r"**, then the function will return a string equal to the elementwise sum of the columns of the array **x**;
- if **orientation** is equal to **2** or **"c"**, then the result of the function will be a column vector, each element of which is equal to the sum of the elements of the rows of the array **x**.

Let's consider an example of using the function:

```
-->M=[1 2 3;4 5 6;7 8 9]
M =

    1.    2.    3.
    4.    5.    6.
    7.    8.    9.
-->Y=sum(M) //The sum of the matrix elements.
Y =
    45.
-->S1=sum(M,1) //Sum of matrix elements by columns
S1 =
    12.    15.    18.
-->S2=sum(M,'c') // Sum of matrix elements by rows
```

```

S2 =
    6.
   15.
   24.
--> V=[-1 0 3 -2 1 -1 1];
-->sum(V) //Sum of the elements of vector
ans =
    1.

```

3.3.2.3 Function for calculating the product of array elements:

```

y=prod(x)
y=prod(x,orientation)

```

The function works similarly to the **sum** function.

An example of how the function works:

```

-->M=[1 2 3;4 5 6;7 8 9]
M =

    1.    2.    3.
    4.    5.    6.
    7.    8.    9.
-->Y=prod(M) //Product of matrix elements
Y =
   362880.
-->P1=prod(M,1) //Product of matrix elements by
// columns
P1 =
    28.    80.   162.
-->P2=prod(M,'c') // Product of matrix elements by
//rows
P2 =
    6.
   120.
   504.
-->V=[-1 2 3 -2 1 -1 1];
-->prod(V) //Product of matrix elements
ans =
   - 12.

```

3.3.2.4 Function for calculating the determinant of a square matrix.

```

det(M)

```

where M is a square matrix.

An example of how the function works:

```
-->M=[1 0 2;3 2 1;0 3 1];
-->det(M)
ans =
    17.
-->Z=[1 2 2;0 1 3;2 4 4];
-->det(Z)
ans =
     0.
-->M=[1 3;4 2]
M =
     1.     3.
     4.     2.
-->det(M)
ans =
    -10.
```

3.3.2.5 Function for calculating the largest element in an array.

```
max(M)
max(M, 'c')
max(M, 'r')
```

A function can return a value in one or more variables

```
m = max(M)
[mx[,n]] = max(M)
```

where: **M** is a real vector or matrix;

'**c**' is pointer to work by rows;

'**r**' is pointer to work by columns;

m - the return value of the element with the maximum value;

n the return value of the index of the maximum element.

The **max(M)** function returns the value of the largest element in the array **M**.

The **max(M, 'c')** function returns the largest element in each row.

The **max(M, 'r')** function returns the largest element in each column.

The **[m,n] = max(M)** function returns the index value of the largest element **n** and the value of the largest element **m**.

An example of how the function works:

```
-->M=[5 0 3 5;2 7 1 4;0 4 9 10]
M =
```

```

    5.    0.    3.    5.
    2.    7.    1.    4.
    0.    4.    9.   10.
-->max(M)
ans =
    10.
-->max(M, 'c')
ans =
    5.
    7.
    10.
-->max(M, 'r')
ans =
    5.    7.    9.    10.
--> [m,n] = max(M)
n =
    3.    4.
m =
    10.
--> // the largest element in each row.
--> [m,n] = max(M, 'c')
n =
    1.
    2.
    4.
m =
    5.
    7.
    10.
--> // the largest element in each column
--> [m,n] = max(M, 'r')
n =
    1.    2.    3.    3.
m =
    5.    7.    9.    10.

```

3.3.2.6 Function for calculating the smallest element in an array.

```

    min (M)
    min (M, 'c')
    min (M, 'r')
    [m,n] = min (M)
    [m,n] = min (M, 'c')
    [m,n] = min (M, 'r')

```

The **min** function works in the same way as the **max** function works.
An example of how the function works:

```
-->M=[5 0 3 5;2 7 1 4;0 4 9 10]
M =
    5.    0.    3.    5.
    2.    7.    1.    4.
    0.    4.    9.   10.
// smallest element in each row
-->[m,n] = min (M,'c')
n =
    2.
    3.
    1.
m =
    0.
    1.
    0.
// smallest element in each column
-->[m,n] = min (M,'r')
n =
    3.    1.    2.    2.
m =
    0.    0.    1.    4.
```

3.3.2.7 Function for calculating the average value of array elements.

```
Y = mean(x)
Y = mean(x,'r')
Y = mean(x,'c')
```

where: **Y** is average value of array elements;

x is a real vector or matrix;

'**c**' is pointer to work by rows;

'**r**' is pointer to work by columns;

The **mean(x)** function calculates the average of the elements in the array **x**.

The **mean(x,'r')** function calculates the average value of the elements in each column. As a result in the console window the line will be shown. The number of elements in a line will be equal to the number of columns in the array.

The **mean(x,'c')** function calculates the average value of the elements in each row. A column will be shown in the console window. The number of elements in a column will be equal to the number of rows in the array.

An example of how the function works:

```
-->x=[1,2,10;7,7.1,7.01]
```

```

x =
    1.    2.   10.
    7.    7.1  7.01
-->Y = mean(x)
Y =
    5.685
// The average value of the elements in each column
-->Y = mean(x, 'r')
Y =
    4.    4.55    8.505
// The average value of the elements in each row
-->Y = mean(x, 'c')
Y =
    4.3333333
    7.0366667

```

3.3.2.8 Function of inverse matrix calculation.

$$\mathbf{B} = \text{inv}(\mathbf{A})$$

where \mathbf{A} is a square matrix;

\mathbf{B} is square matrix inverse of matrix \mathbf{A} .

The inverse matrix, in relation to a given one, is a matrix of the same type, which, being multiplied both on the left and on the right by the given matrix, will result in the identity matrix. Those multiplying \mathbf{A} by the inverse matrix (\mathbf{B}) on the left should result in the identity matrix.

Let's consider an example of obtaining an inverse matrix.

```

--> A=rand(3,3)
A =
    0.993121    0.050042    0.6084526
    0.6488563    0.7485507    0.8544211
    0.9923191    0.4104059    0.0642647
--> inv(A)*A
ans =
    1.    0.    0.
    0.    1.    0.
    0.    0.    1.

```

3.3.2.9 The function of converting a matrix to a triangular form.

$$\mathbf{B} = \text{rref}(\mathbf{A})$$

where \mathbf{A} is a matrix;

\mathbf{B} is matrix \mathbf{A} reduced to triangular form using the Gaussian elimination method.

An example of how the function works:

```

-->A=[3 -2 1 5;6 -4 2 7;9 -6 3 12]
A =
    3.  - 2.    1.    5.
    6.  - 4.    2.    7.
    9.  - 6.    3.   12.
-->rref(A)
ans =
    1.  - 0.6666667    0.3333333    0.
    0.    0.          0.          1.
    0.    0.          0.          0.

```

3.4 Symbolic matrices and operations on them.

In Scilab it is possible to define symbolic matrices, that is, matrices whose elements are of string type. It should be remembered that string elements must be enclosed in double or single quotes.

```

--> M=['a' 'b';'c' 'd']
M =
!a  b  !
!           !
!c  d  !
--> P = ['1' '2';'3' '4']
P =
!1  2  !
!           !
!3  4  !

```

Symbolic matrices can be added (the result of addition is the concatenation of the corresponding strings) and transposed:

```

--> M+P
ans =
!a1  b2  !
!           !
!c3  d4  !

```

In addition, addition operations can be performed on individual elements of symbolic matrices:

```

--> M(1,1)+P(2,2)
ans =
a4

```

The above list of functions is far from complete, for example, we have not considered matrix functions that implement numerical algorithms for solving linear algebra problems. The description of functions for working with vectors, arrays and matrices is widely presented in the Scilab help system.

Questions for self-examination for the fourth lecture:

- 1. How to convert a matrix to a matrix of a different size?*
- 2. How to create a matrix of ones?*
- 3. How to create a matrix of zeros?*
- 4. How to create an identity matrix?*
- 5. How to create a matrix of random numbers?*
- 6. How to perform the ordering of the the elements of the matrix?*
- 7. How to determine the size of the matrix?*
- 8. How to determine the length of the matrix?*
- 9. How to calculate the sum of array elements?*
- 10. How to calculate the product of array elements?*
- 11. How to calculate the determinant of a square matrix?*
- 12. How to calculate the largest element in an array?*
- 13. How to calculate the smallest element in an array?*
- 14. How to calculate the average of the elements?*
- 15. What are symbolic matrices?*
- 16. What is an inverse matrix and how to get it?*
- 17. What operations can be performed on symbolic matrices?*

Lecture 5

The purpose of the lecture is to learn how to solve systems of linear algebraic equations in various ways, to learn how to plot graphs using the plot function.

3.5 Solving systems of linear algebraic equations.

A system of m equations with n unknowns of the form:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \dots + a_{1n}x_n - b_1 &= 0 \\ a_{21}x_1 + a_{22}x_2 + \dots + a_{2n}x_n - b_2 &= 0 \\ \dots & \\ a_{m1}x_1 + a_{m2}x_2 + \dots + a_{mn}x_n - b_m &= 0 \end{aligned}$$

is called a system of linear algebraic equations (SLAE), where x_j are unknowns, a_{ij} are coefficients of unknowns, b_i are free coefficients ($i= 1 \dots m, j= 1 \dots n$). A system of m linear equations with n unknowns can be described using matrices: $A \cdot x = b$, where x is the vector of unknowns, A is the matrix of coefficients for unknowns or the matrix of the system, b is the vector of free members of the system or the vector of the right-hand sides. The set of all solutions of the system (x_1, x_2, \dots, x_n) is called the set

of solutions or simply the solution of the system.

Let's solve the SLAE using Cramer's rule:

$$2x_1 + x_2 - 5x_3 + x_4 = 8,$$

$$x_1 - 3x_2 - 6x_4 = 9,$$

$$2x_2 - x_3 + 2x_4 = -5,$$

$$x_1 + 4x_2 - 7x_3 + 6x_4 = 0.$$

Cramer's rule is as follows. If the determinant $\Delta = \det A$ of the matrix of a system of n equations with n unknowns $Ax = b$ is nonzero, then the system has a unique solution x_1, x_2, \dots, x_n , determined by Cramer's formulas: $x_i = \Delta_i / \Delta$, where Δ_i is the determinant of the matrix obtained from the matrix of system A by replacing the i th column with the column of free members b .

Let's write down the program code with the solution of the system of equations using Cramer's formulas in the text editor:

```
//Matrix and vector of free coefficients
// of the system:
A=[2 1 -5 1;1 -3 0 -6;0 2 -1 2;1 4 -7 6];
b=[8;9;-5;0]; //Vector of free coefficients
A1=A;A1(:,1)=b; //First auxiliary matrix
A2=A;A2(:,2)=b; //Second auxiliary matrix
A3=A;A3(:,3)=b; //Third auxiliary matrix
A4=A;A4(:,4)=b; //Fourth auxiliary matrix
D=det(A); //Main determinant
//Determinants of auxiliary matrices:
d(1)=det(A1);d(2)=det(A2);d(3)=det(A3);d(4)=det(A4);
x=d/D //Vector of unknowns
P=A*x-b //Verification
```

Let's start the program for execution by pressing the [Ctrl] + [L] keys and view the results of solving the system of equations in the Scilab console window.

```
x =
    3.
   -4.
   -1.
    1.
-->P=A*x-b // Verification
P =
    0.
    0.
   -8.882D-16
    2.665D-15
```

Let us solve the same SLAE by the inverse matrix method.

For a system of n linear equations with n unknowns $A \cdot x = b$, provided that the determinant of the matrix A is not equal to zero, the only solution can be represented as $x = A^{-1} \cdot b$.

Let's type the program code in the SciNotes text editor:

```
//Matrix and vector of free coefficients
// of the system:
A=[2 1 -5 1;1 -3 0 -6;0 2 -1 2;1 4 -7 6];
b=[8;9;-5;0];//Vector of free coefficients
x=inv(A)*b //System solution
```

Let's start the program for execution by pressing the [Ctrl] + [L] keys and view the results of solving the system of equations in the Scilab console window.

```
x =
  3.
 - 4.
 - 1.
  1.
```

As you can see, the result is the same.

Now we will solve the system of linear equations by the Gauss method:

The solution of a system of linear equations using the Gauss method is based on the fact that from a given system we go to an equivalent system, which is easier to solve than the original system.

Gauss's method consists of two steps. The first step is a direct move, as a result of which the expanded matrix of the system is reduced to a stepwise form by means of elementary transformations (rearrangement of the equations of the system, multiplication of equations by a number other than zero, and addition of equations). At the second step (backward move), the stepped matrix is transformed so that the identity matrix is obtained in the first n columns. The last, $n + 1$ column of this matrix contains the solution of a system of linear equations.

Let's type the program code in the SciNotes text editor:

```
//Matrix and vector of free coefficients
// of the system:
A=[2 1 -5 1;1 -3 0 -6;0 2 -1 2;1 4 -7 6];
b=[8;9;-5;0];//Vector of free coefficients.
//Reducing the extended matrix to a triangular form:
C=rref([A b]);
//Determination of the dimension of
//the extended matrix:
[n,m]=size(C); //m- the number of the last column
// of the matrix C
```

```
//Extracting the last column from matrix C:  
x=C(:,m) //x - System solution
```

Let's run the program by pressing the [Ctrl] + [L] keys and view the results of solving the system of equations in the Scilab console window.

```
x =  
 3.  
- 4.  
- 1.  
 1.
```

4 PLOTTING A SET OF 2D CURVES.

In the Scilab software environment, graphic means of displaying the results of various calculations and transformations are widely presented. First, let's get acquainted with the tools for plotting two-dimensional graphs. Two-dimensional plots are those in which the position of each point is set by two values.

4.1 **plot** function.

To plot two-dimensional graphs of a function of one variable of the form $y=f(x)$ in Scilab there is a **plot** function, which can be accessed in the following way:

plot(x,y)

where: **x** is real matrix or vector of abscissa values;

y is a real matrix or a vector of ordinates of the function f at these points given **x**.

At the same time, with the **plot** function, it is convenient to use the **xtitle** function to plot explanatory labels on the chart and its axes. For two-dimensional plots, the function is:

xtitle(title,x_label,y_label)

here: **title** is a string type variable in which will be added title on a graphic window;

x_label is a string type variable into which the label located next to the abscissa axis is placed;

y_label is a string type variable into which the label located next to the ordinate axis is placed.

Consider an example of plotting the change in the size of a part in time during the day. Let the change in the size of the part depend on the temperature, which in turn changes in a sinusoidal manner. Suppose that measurements were made every

half hour, then the data array of time changes can be represented by the vector `0:0.5:24`. The nominal size of the part is 40 mm and its change occurred in the range of ± 0.01 mm. Considering all of the above, the function of resizing over time will look like:

$$y=40+0.01*\sin((2*\%pi*x)/24).$$

Let's type the code of the program in the window of the SciNotes text editor:

```
x=0:0.5:24;  
y=40+0.01*sin((2*%pi*x)/24);  
plot(x,y)  
xtitle('Changing the size of a part over time',...  
'Time, hour', 'Diameter, mm');
```

Let's start the program for execution by pressing the [Ctrl] + [L] keys and in the graphic window that appears on the computer screen we will see the change graph of the part size over time (fig. 4.1).

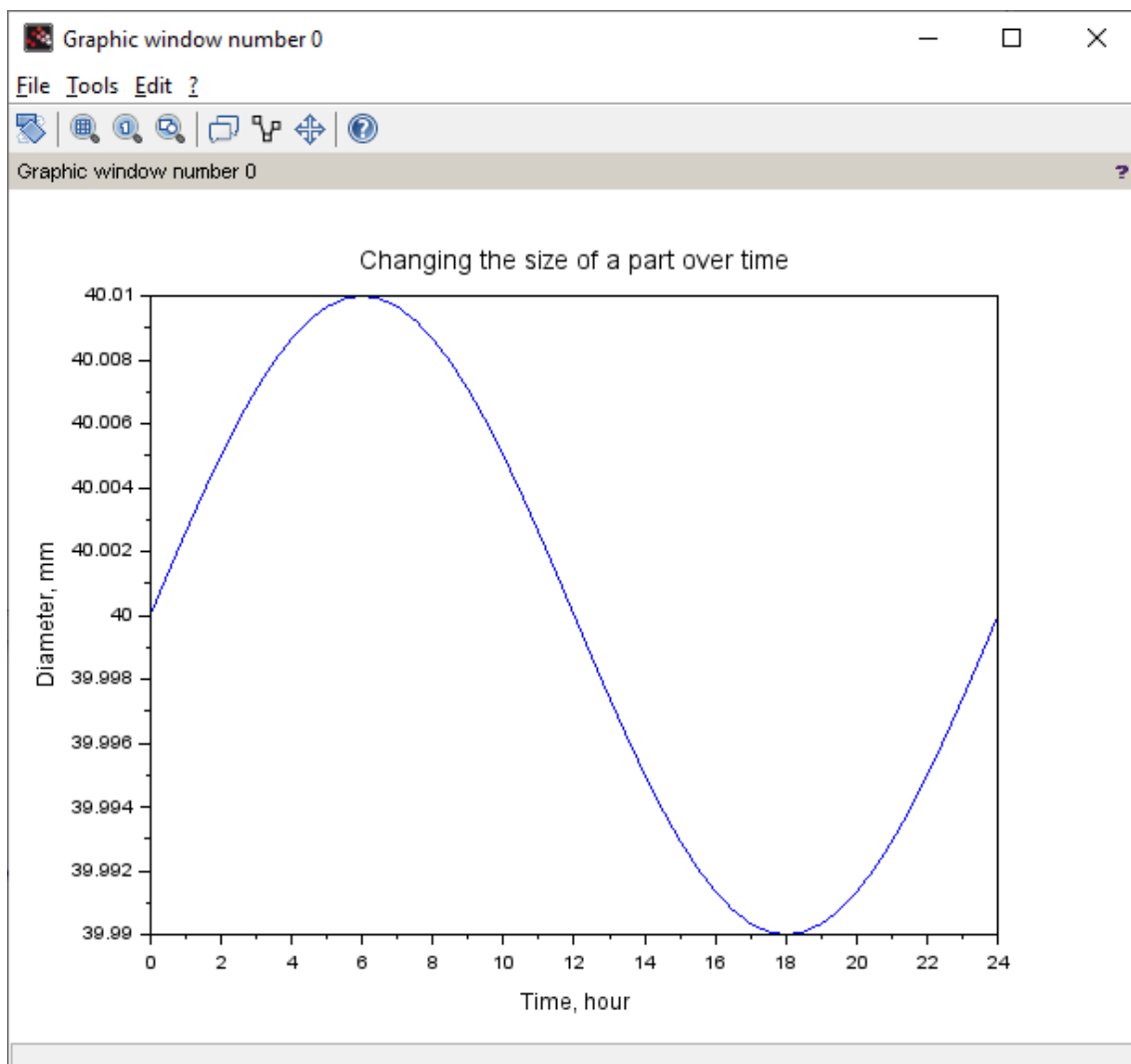


Figure 4.1 – Change graph of the part size over time.

To increase the clarity of the graph, it is useful to use the function for drawing a grid on the graph:

```
xgrid(c[,t] [,p])
```

where: **c** is a prime number defining the colour of the grid;

t is the thickness of the grid line [optional];

p is grid line type [optional].

In the simplest case, the call to the function has the form **plot(y)**, the array of points numbers of the array **y** acts as an array **x** as in the previous example. Let's consider an example of plotting a function of the form $y=f(i)$, where i is the number of a point in the **y** array.

```
y=[1 2 3 -2 4 5 -1 6 9 11 0 -2 5];  
xgrid(5)  
plot(y);
```

After starting the program, we get the graph which is shown in fig. 4.2.
рис 4.2.

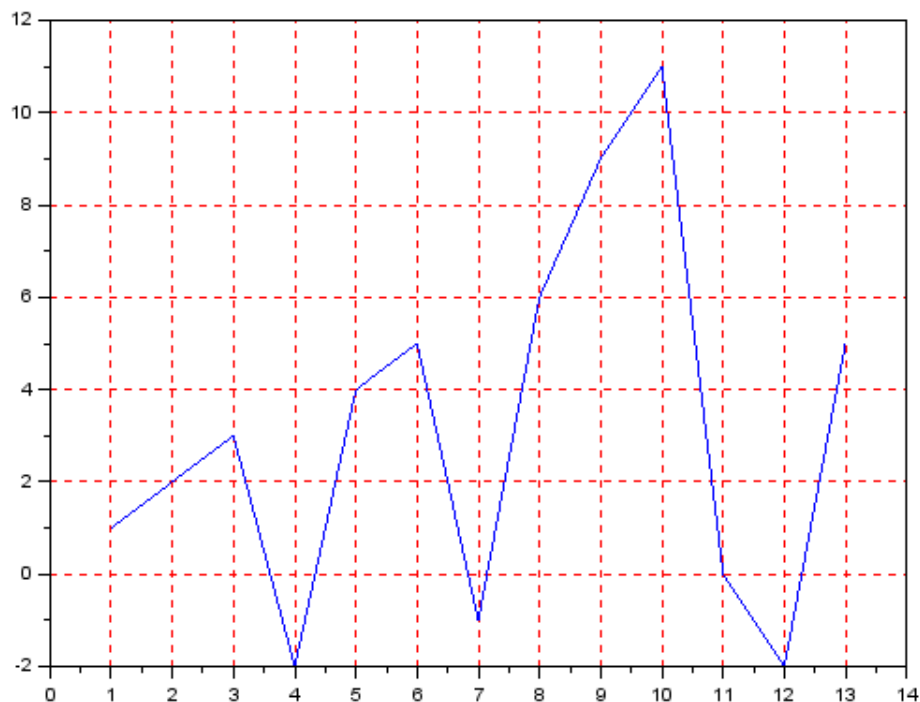


Figure 4.2 – Graph of the function $y=f(i)$.

When plotting several graphs, it is enough to simply set several **plot()** functions in the current program, that is, without closing the graphics window. However, in this case, the graphs will be of the same color.

```
x=0:0.5:24;  
y=40+0.01*sin((2*pi*x)/24);  
plot(x,y)
```

```

y1=39.99+0.01*sin((2*pi*x)/24);
plot(x,y1)
xlabel('Changing the size of a part over time',...
'Time, hour', 'Diameter, mm');

```

It is more convenient (and more correct) to use the `plot()` function in the following way:

```

plot(x,[f1(x) f2(x) f3(x) ...])

```

Let x belong to the interval $[0...2\pi]$. Since x is an argument for all functions, so x can be absent in the `plot()` functions. It is also not necessary to form its own array of values for each function. It is enough to indicate their mathematical expressions in square brackets separated by a space, and these arrays will automatically be created as an intermediate stage in plotting the curves of functions.

Example. Plot the function `sin(x)`, `cos(x)` and `exp(sin(x))` on the interval $[0; 2\pi]$ in one window.

```

x = [0:0.1:2*pi]';
xgrid(5)
plot([sin(x) cos(x) exp(sin(x))])

```

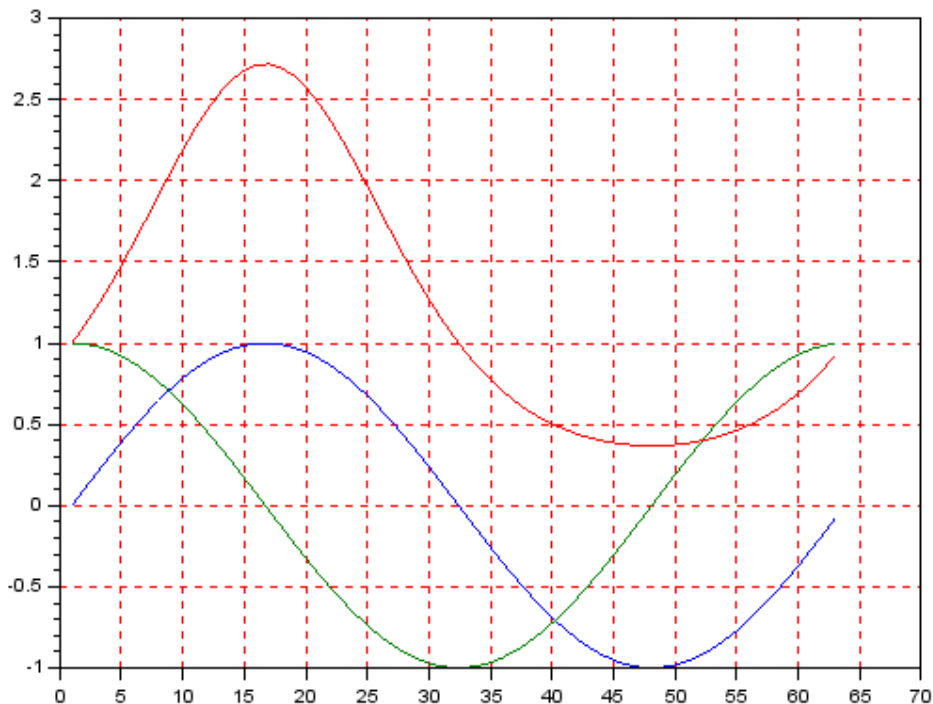


Figure 4.3 – An example of graphing multiple functions in one window.

Let's consider plotting a graph of a two-dimensional array using an example:

```

t=[1    1    1    1
   2    3    4    5]

```

```

    3    4    5    6
    4    5    6    7];
xgrid(1)
plot(t)

```

As a result of the program execution, we get the graph which is shown in fig. 4.4.

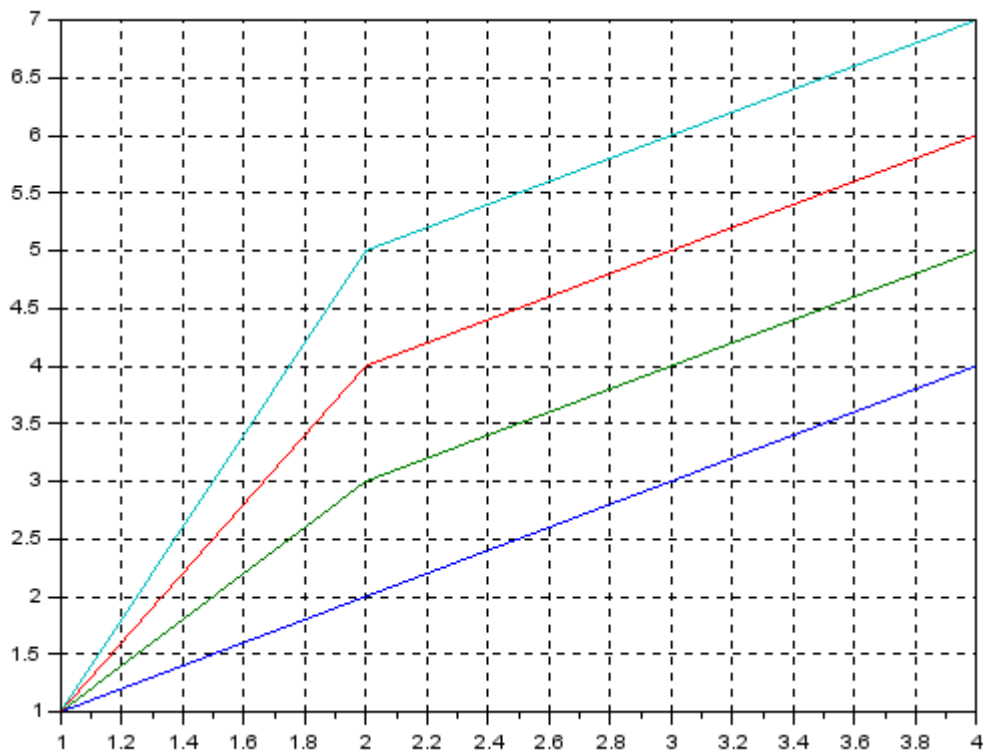


Figure 4.4 – An example the graph of the two-dimensional array.

4.2 Modification of graphs.

The type of the graph can be changed when using the plot function, for that in addition to the main arguments, one more argument - a string of three symbols that will determine the color of the line, the type of symbol which will be used for the graph and the type of the line. A call to the plot () function will look like this:

```
plot(x1,y1,string1,x2,y2,string2,...)
```

here: **string** looks like this:

```
'parameter1parameter2parameter3'.
```

Parameters are written one after the other without separators.

parameter1 defines the color of the graph:

Symbol	Description
y	yellow
m	pink
c	blue
r	red
g	green
b	blue
w	white
k	black

parameter2 sets the symbol for drawing the graph:

Symbol	Description
.	Point
o	Circle
x	Cross
+	Plus sign
*	Star
'square' or 's'	Square
'diamond' or 'd'	Rhombus
v	Down-pointing triangle
^	Up-pointing triangle
<	Left-pointing triangle
>	Right-pointing triangle
'pentagram' or 'p'	Five pointed star
	No marker (default)

parameter3 install plot line type:

Symbol	Description
-	Solid line (default)
:	Dotted line
--	Dashed line
-.	Dash-dotted line

If one of the symbols for line type is not specified, so it means that its value is selected by default (as a rule, a blue solid line), if the symbol for drawing the graph is not specified, it will be absent.

Let's consider examples of plotting three graphs:

- $f_1 = |3x|$ on the interval $[-3; 3]$ - solid blue line;
- $f_2 = 8\sin(x)$ on the interval $[-\pi; \pi]$ - dashed red line;
- $f_3 = e^x/10$ on the interval $[-1; 4]$ the graph drawn by green circles.

x1=-3:0.2:3; f1=abs(3*x1);


```

x2=-%pi:0.2:%pi; f2=8*sin(x2);
x3=-1:0.2:4; f3=exp(x3/10);
plot(x1,f1,'b-',x2,f2,'r--',x3,f3,'go')
xgrid(1)

```

The result of the program working is shown in fig. 4.5.

For more convenient use of graphs, the description of lines (the so-called legend) is used with help of the command:

```

legend(line1,line2,...,lineN,place,frame)

```

here: **line1** and others - the names of the graphs;

place - the location of the description:

1 - upper right corner of the graph's window;

2 - top left corner of the graph's window;

3 - bottom left corner of the graph's window;

4 - lower right corner of the graph's window,;

5 - defined by the user after displaying the graph.

frame sets the frame for the description:

%t - the description is framed,

%f - no frame.

As an example, let's plot a graph of the functions $f(x)=\sin(x)$ and $g(x)=\cos(x)$ on the interval $[-2\pi; 2\pi]$ and design it by signing the graph itself, axes and displaying the description of the lines and the grid.

```

clf()
x=-2*%pi:0.2:2*%pi; f1=sin(x); f2=cos(x);
xgrid(2)
plot(x,f1,'b.--',x,f2,'r*:')
legend('f1(x)', 'f2(x)', 3, %t)
xlabel('Draph f1(x) and f2(x)', 'Axis X', 'Axis Y')

```

As a result of the program execution, we get the graph which is shown in fig. 4.6.

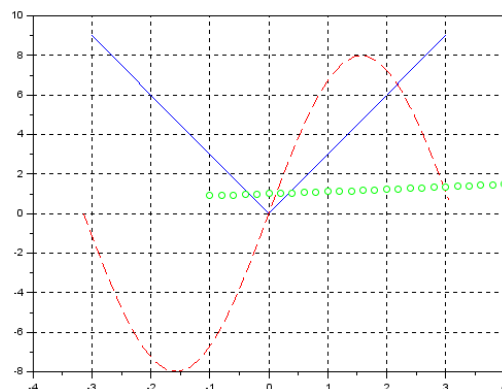


Figure 4.5 – An example of the graph with different types of lines.

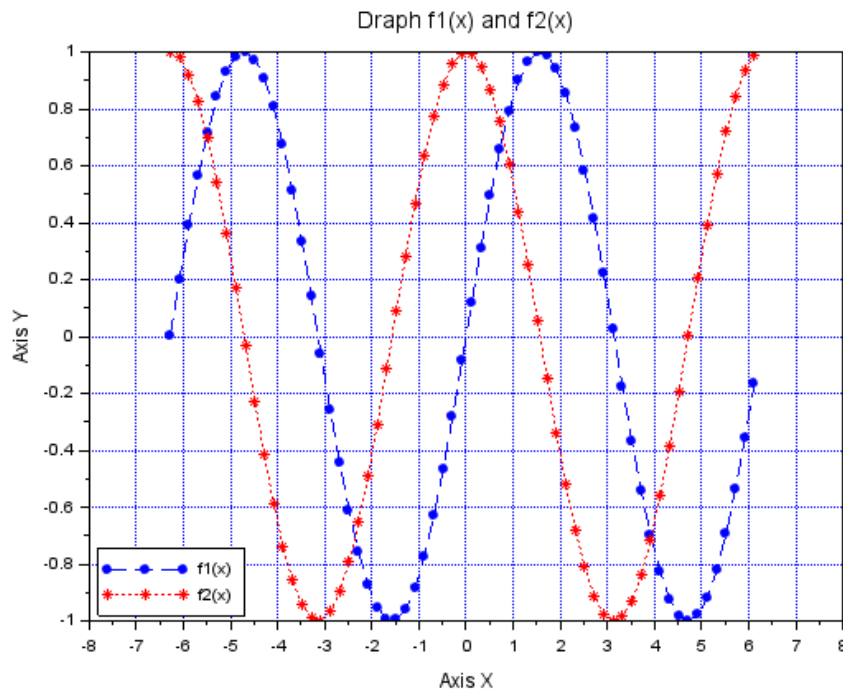


Figure 4.6 – An example of a complete graph formatting in Scilab.

Questions for self-examination for the fifth lecture:

1. *What is a system of linear algebraic equations?*
2. *How to solve SLAE using Cramer's formulas?*
3. *How to solve SLAE by the inverse matrix method?*
4. *How to solve SLAE by the Gaussian method?*
5. *What is the plot function for?*
6. *How to display graphs of several functions in one window?*
7. *How to display the graph of the matrix?*
8. *What parameters of the plot function control the appearance of the lines?*
9. *How to place labels on the axes and the title of the graphs?*
10. *How do I plot a grid of the graph?*

Lecture 6

The purpose of the lecture is to learn how to build several graphs in one graphic window, study the plot2d function and its parameters, and learn how to draw up graphs using it.

4.3 Plotting several graphs in one graphic window.

In Scilab, it is possible to plot several graphs in one window without aligning them in the same coordinate axes. For example, if a graphic window should contain 4 independent graphs, then it is divided into 4 areas, and then a function graph is displayed in each of them.

To form several areas in the graphic window, use the command

subplot (m,n,p)

here m, n, p are integers defining the position of the sub-window in the area of the graphic window.

The **subplot** function divides the area of the graphic window into a matrix consisting of $m \times n$ subwindows and selects the subwindow with the ordinal number p to display the graf. The subwindow number is counted starting from the top left, from left to right, along the rows of the $m \times n$ matrix. After selecting an area, you can enter a graph into it, for example, using the **plot** function.

As an example, let's build graphs of the functions $y = \sin(2x)$, $z = \cos(3x)$, $u = \cos(\sin(2x))$, $v = \sin(\cos(3x))$ in one graphics window, each graph in its own coordinate system.

Let's assume that x changes in the interval $[-3: 0.1: 30]$. Let's form arrays of values of the function Y, Z, U, V .

Using the **subplot** function, divide the graphic window into 4 subwindows. Let's type the program in the SciNotes text editor:

```
x=[-3:0.1:3];
y=sin(2*x);      z=cos(3*x);      u=cos(sin(2*x));
v=sin(cos(3*x));
subplot(221) // Subwindow 1
xgrid(2)
xlabel('Graph y=sin(2*x)', 'Axis X', 'Axis Y')
plot(x,y,'b--'); legend('f1(x)',3,%t)
subplot(222) // Subwindow 2
xgrid(3)
plot(x,z,'r--*');
subplot(223) // Subwindow 3
xgrid(4)
plot(x,u,'c--<');
subplot(224) // Subwindow 4
xgrid(5)
plot(x,v,'k--o');
```

Let's start the program for execution by pressing the [Ctrl] + [L] keys and in the graphic window that appears on the computer screen we will see four subwindows with graphs of functions (fig. 4.7).

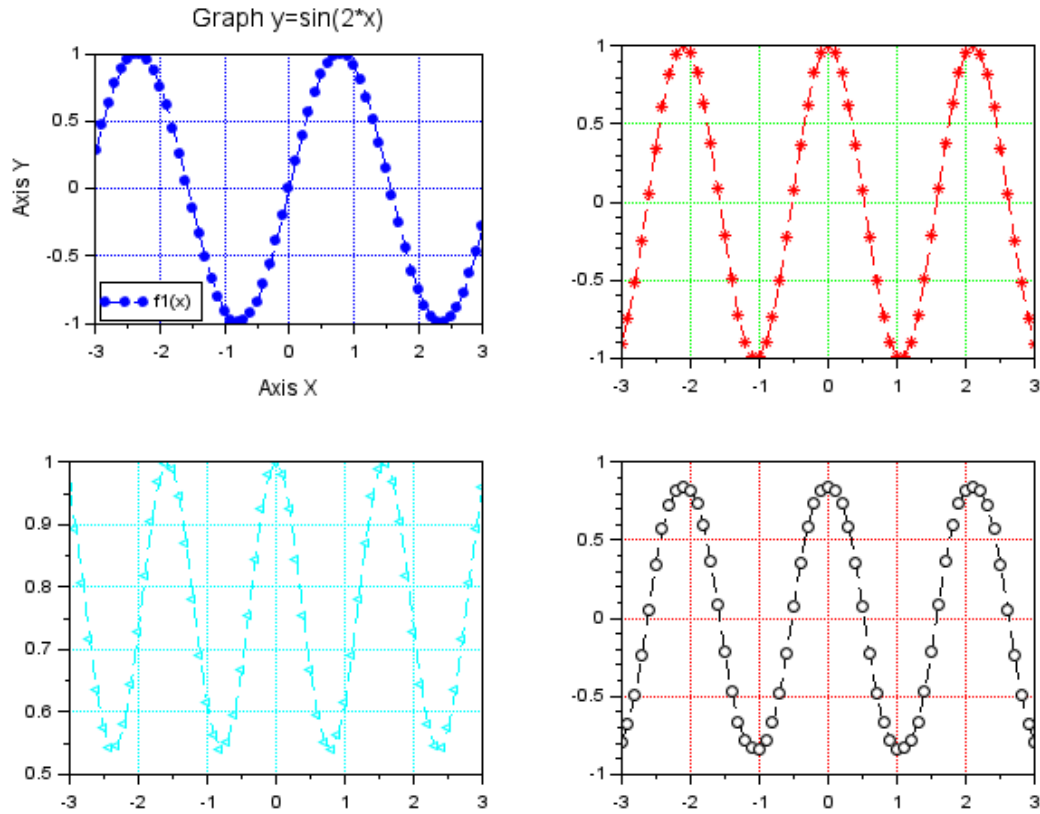


Figure 4.7 – An example of splitting a graphic window into four parts.

4.4 The plot2d function.

The next function that can be used to plot 2D graphs is the `plot2d` function. In general, a function call looks like this:

```
plot2d([logflag], x, y', [key1=value1, key2=value2, ...  
keyn=valuen])
```

where: **logflag** is a string of two characters, each of which defines the type of axes (**n** - normal axis, **l** - logarithmic axis), by default - (**nn**);

x is array of abscissas;

y is an array of ordinates or a matrix, each column of which contains an array of ordinates of the next graph - in case it is necessary to plot graphs of several functions y_1, y_2, \dots, y_n , when they all depend on the same variable **x**. In this case, the number of elements in the array **x** and **y** must be the same. If **x** and **y** are matrices of the same size, then each column of **y** is mapped relative to the corresponding column of **x**;

keyi=valuei is a sequence of values of the graph **key1=value1,**

key2=value2, ..., keyn=valuen, which determine its appearance. Possible values of the graph properties will be described in detail below.

It should be noted that it is not at all necessary to use the full notation of the **plot2d** function with all its parameters. In the simplest case, you can refer to it briefly, as well as to the **plot** function.

Let's consider an example of using the **plot2d** function. Let the variable **x** is in the range from -2π to $+2\pi$ with a step of 0.1. Let's form an array **x**. It is not necessary to create an array **y**, you only need to specify the mathematical expression of the function as an argument to the **plot2d** function, then the program will look like this:

```
x=[-2*%pi:0.1:2*%pi];  
plot2d(sin(x));
```

In the graphic window, we get the graph which is shown in figure 4.8.

Using the **plot2d** function, you can also plot multiple graphs in the same coordinate axes. For example, let's build the graphs of the functions $y = \sin(x)$, $y1 = \sin(2x)$, $y2 = \sin(3x)$ in the same coordinate axes. To do this, we will form an array **x**, with values of x and varying in the range from 0 to $+2\pi$ with a step of 0.1. To plot curves in the same coordinate axes, we will use the **plot2d(x,y)** function. However, as an array **y** in square brackets, we alternately indicate the mathematical expressions of the given functions, separating them with spaces:

```
x=[0:0.1:2*%pi]';  
plot2d(x,[sin(x) sin(2*x) sin(3*x)])
```

As a result, we get the graph which is shown in figure 4.9.

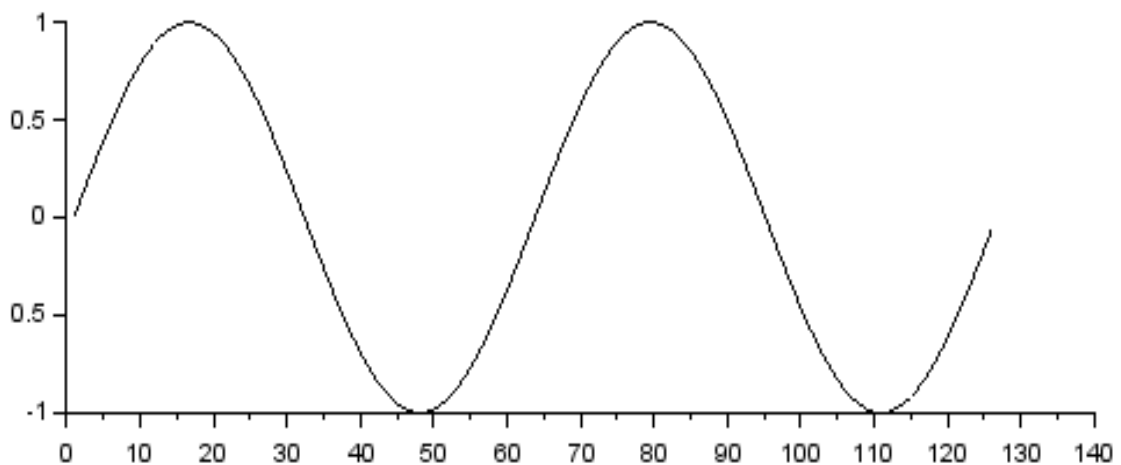


Figure 4.8 – Graph of the function $y = \sin(x)$.

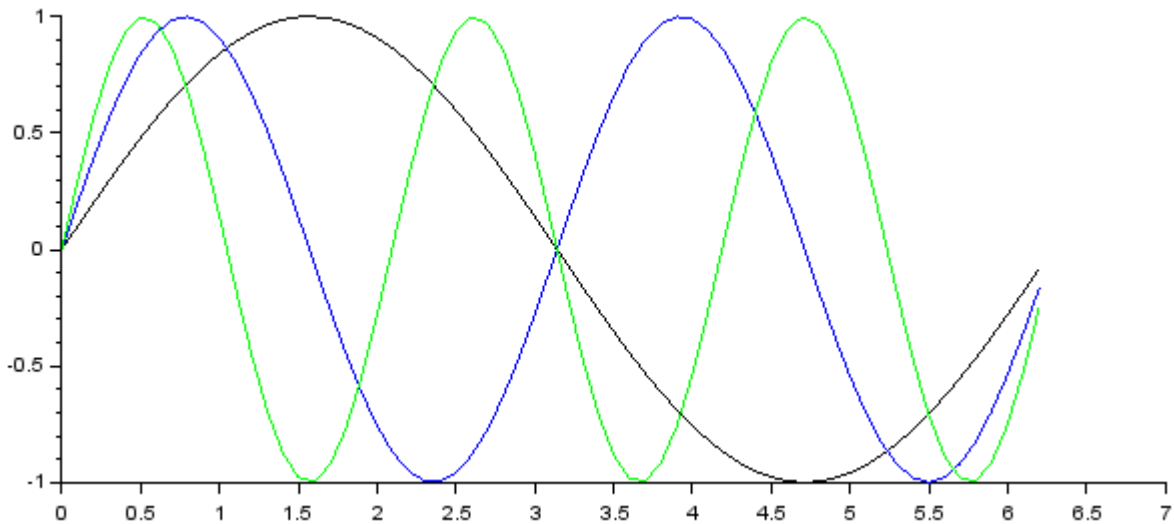


Figure 4.9 – Graphs of the functions $y = \sin(x)$, $y_1 = \sin(2x)$, $y_2 = \sin(3x)$.

In chapter 4.4 it was mentioned that the full form of calling the **plot2d** function allows of the user to independently determine the appearance of the graph - the parameter **keyn=valuen** is responsible for this.

The full form of calling the plot2d function looks like:

```
plot2d( [logflag] , x , y' , [key1=value1 , key2=value2 , ...
, keyn=valuen] )
```

4.5 Figuration of the graphs using the plot2 function.

The following values are available for the **keyn=valuen** parameter.

The style value defines an array of numeric values for the graph colors. The number of array elements coincides with the number of displayed graphs. You can use the **color** function, which, by the name (**color ("color name")**) or the rgb code (**color (r,g,b)**) of the color, forms the desired color **id** (code). A complete list of all shades available for formatting with their RGB-id can be found in the article in the built-in Scilab help system **color_list** (Scilab Help >> Graphics > Color management > color_list). As an example, we will plot the graphs of the functions $y = \sin(x)$ and $y = \cos(x)$ in the same coordinate axes, for the sinusoid using the style parameter we define the color name - red ('red'), and for the cosine curve - the green **id** (0,176, 0). The corresponding program will look like:

```
x=[-2*%pi:0.1:2*%pi];
y=[sin(x);cos(x)] ;
plot2d(x,y' ,style=[color("red") , color(0,176,0)] ) ;
```

The graph corresponding to this program is shown in fig. 4.10.

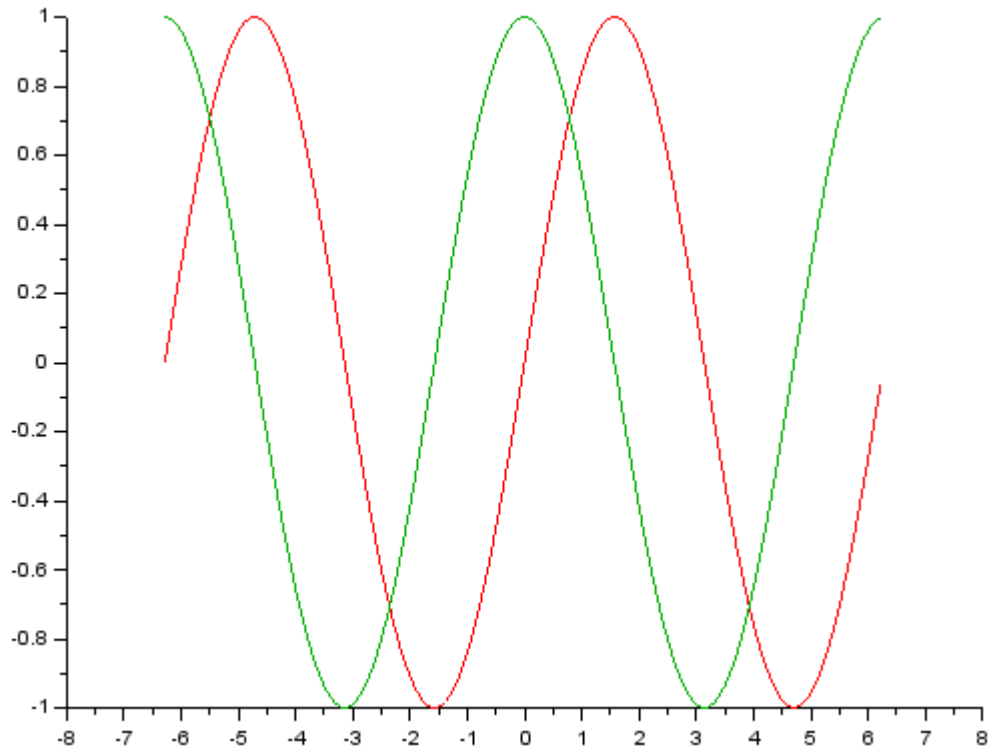


Figure 4.10 – The **style** parameter of the **plot2d** function.

The **rect** value of the **keyn=valuen** parameter of the **plot2d** function is a **[xmin, ymin, xmax, ymax]**, that determines the size of the window around the graph. Here **xmin, ymin** is position of the lower left corner of the window; **xmax ymax** is position of the upper right corner of the window.

Consider an example of using the **rect** value for the previous example, setting it to the following values **[-8, -2, 8, 2]** (fig. 4.11), the program in this case will look like this:

```
x=[-2*%pi:0.1:2*%pi];  
y=[sin(x);cos(x)];  
plot2d(x,y',style=[color("red"),...  
color(0,176,0)],rect=[-8,-2,8,2]);
```

Due to the fact that the **Y**-axis has extended from **[-1: 1]** to **[-2: 2]**, and the **X**-axis has remained unchanged, it visually appears as if the graph has shrunk along the **Y**-axis.

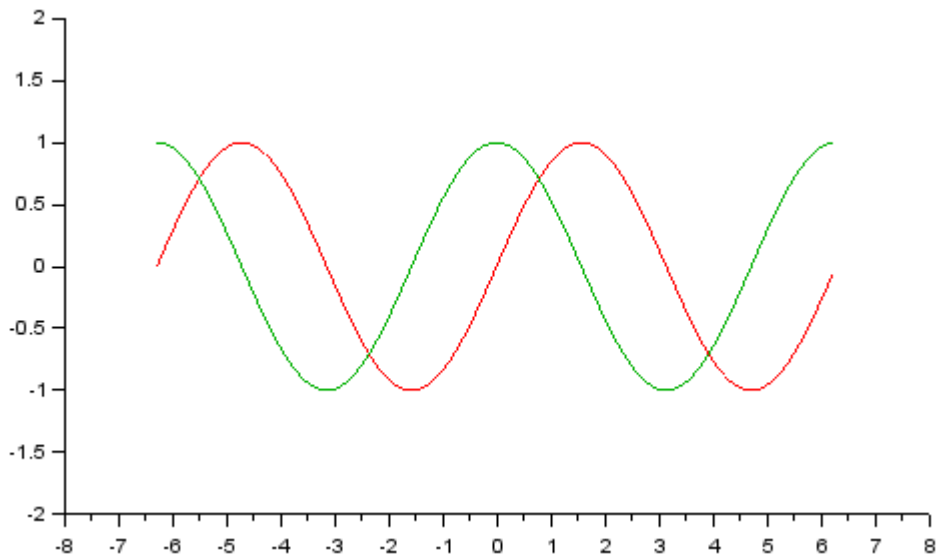


Figure 4.11 – The **rect** value of the **keyn=valuen** parameter of the **plot2d** function.

The **axesflag** value of the **keyn=valuen** parameter of the **plot2d** function determines the presence of a frame around the graph. It is necessary to highlight the following basic values of this parameter:

- 0 - no frame;
- 1 - image of the frame, Y-axis to the left (by default);
- 3 - image of the frame, Y-axis to the right;
- 5 - the image of the axes which are passed through the point (0,0).

Let's plot graphs of the functions $y = \sin(x)$ and $y1 = \cos(x)$, using 4 basic values of the **axesflag** parameter, in one graphic window using the **subplot** function. Divide the graphic window into 4 subwindows. The action of the parameter **axesflag=0** will be displaying the subwindow in the upper left corner of the graphic window, **axesflag=1** - in the upper right corner, **axesflag=3** - in the lower left corner, and **axesflag=5** - in the lower right corner of the graphic window. The program will take the form:

```
clf
x=[-2*%pi:0.1:2*%pi];
y=[sin(x); cos(x)];
subplot(2,2,1)
plot2d(x,y',style=[color("red"), color("blue")],...
axesflag=0);
subplot(2,2,2)
plot2d(x,y',style=[color("red"), color("blue")],...
axesflag=1);
subplot(2,2,3)
```



```

plot2d(x,y',style=[color("red"), color("blue")],...
axesflag=3);
subplot(2,2,4)
plot2d(x,y',style=[color("red"), color("blue")],...
axesflag=5);

```

And the graph corresponding to this program is shown in fig. 4.12.

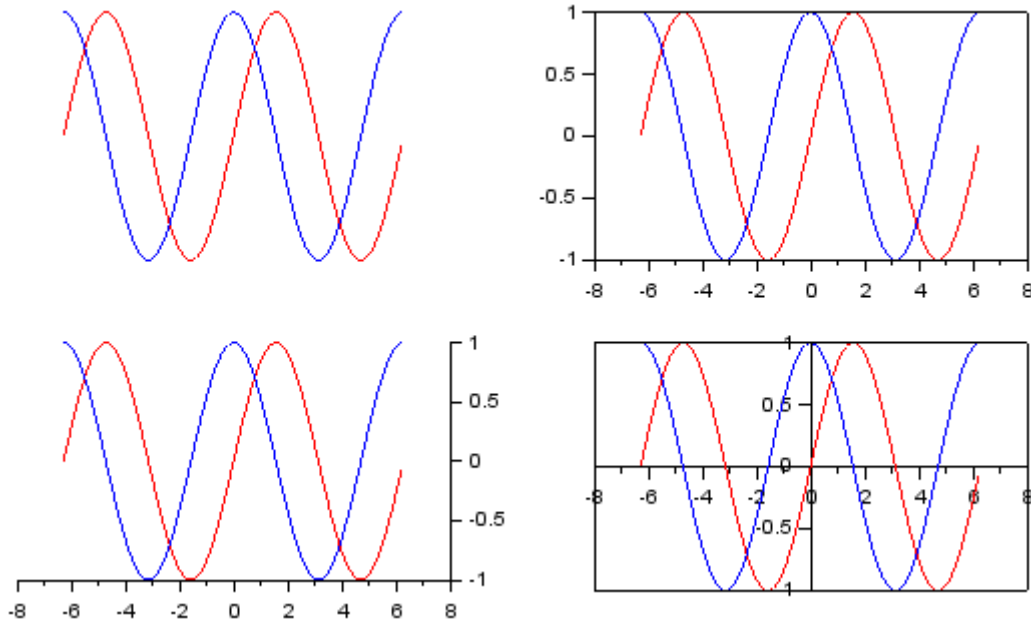


Figure 4.12 – The **axesflag** value of the **keyn=valuen** parameter of the **plot2d** function.

To determine the number of major and intermediate divisions of the coordinate axes, Scilab has the **nax** parameter. If **axesflag=1** (default), then **nax** is an array of four values: **[nx,Nx,ny,Ny]**. Here **Nx (Ny)** is the number of major divisions with labels under the *X (Y)* axis; **nx (ny)** is the number of intermediate divisions.

For example, let's plot graphs of the functions $y = \sin(x)$ and $y1 = \cos(x)$ by changing the scale of the coordinate axes of the graph. Let's form an array **x**, its values will be varying in the range $[-8: 8]$ with a step of 0.1, then we'll form arrays of values of the specified functions using the notation **y=[sin(x); cos(x)]**.

Using the **plot2d** function, plot the curves of the functions $y = \sin(x)$ and $y1 = \cos(x)$, setting the value of the parameter **nax=[4,9,3,6]**. Thus, the *X*-axis will be divided by 9 main divisions and each main division will be divided by 4 intermediate divisions, and the *Y*-axis - respectively by 6 and 3. Then the program will look like:

```

x=[-8:0.1:8];

```

```

y=[sin(x) ; cos(x)] ;
plot2d(x,y',style=[color("red"),color("blue")],...
axesflag=1,nax=[4,9,3,6]) ;

```

And the graph which is corresponded to the program is shown in fig. 4.13.

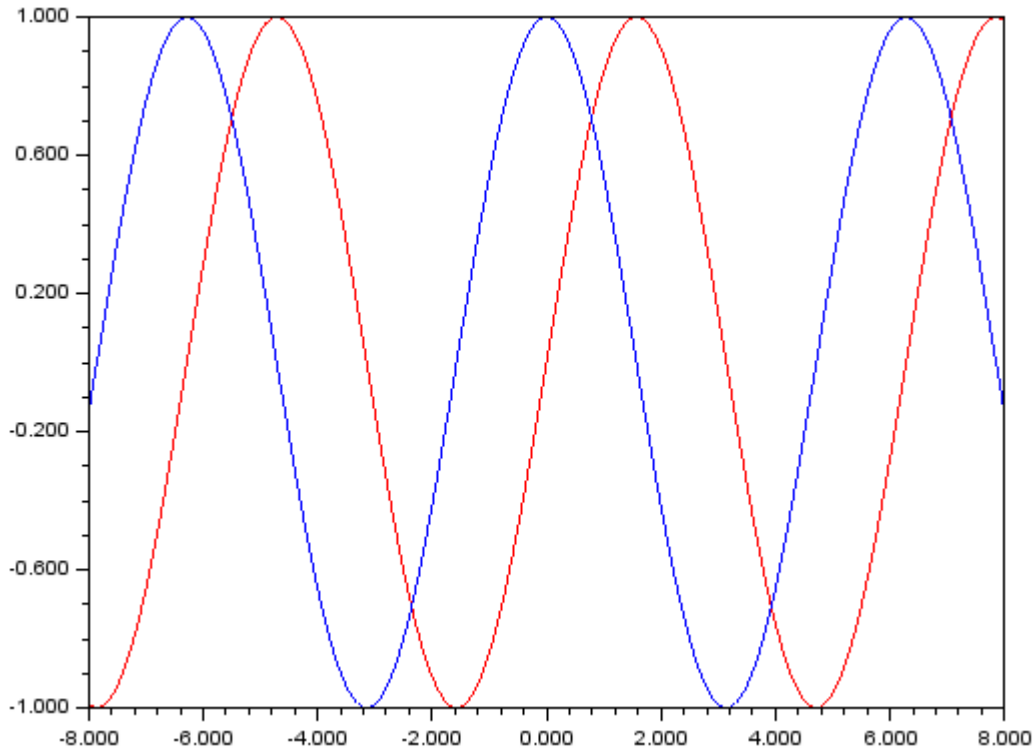


Figure 4.13 – The **nax** value of the **keyn=valuen** parameter of the **plot2d** function.

The value **leg** of the **keyn=valuen** parameter of the **plot2d** function is a string that defines the legends for each plots: "**leg1@leg2@leg3@ ...@legn**", where **leg1** is the legend of the first plot, ..., **legn** is the legend of the **n**-th plot.

Let's use the previous task as an example. Let's plot the graphs of the functions $y = \sin(x)$ and $y1 = \cos(x)$, with the intersection of the X and Y axes at the point (0,0) - the value of the parameter **axesflag = 4**, display a legend with legend for both curves. In this case, the program will look like:

```

x=[-8:0.1:8] ;
y=[sin(x) ; cos(x)] ;
plot2d(x,y',style=[color("red"),color("blue")],...
axesflag=4,nax=[4,9,3,6],leg="sin(x)@cos(x)") ;

```

And the graph which is corresponded to the program is shown in fig. 4.14.

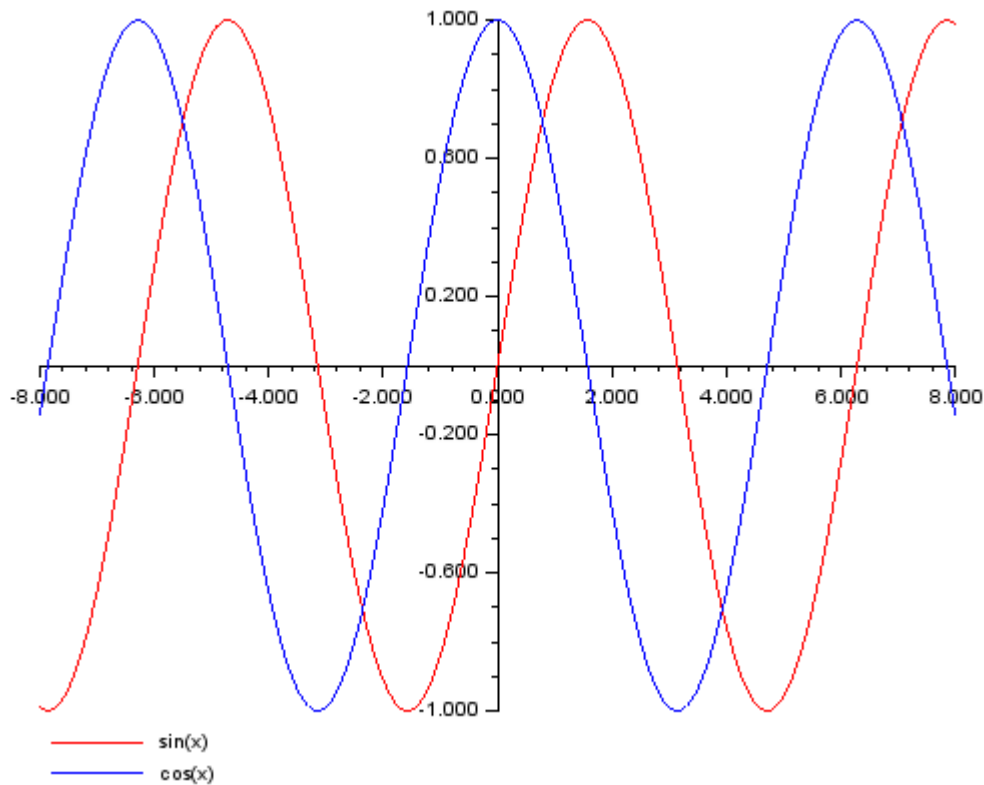


Figure 4.14 – The **leg** value of the **keyn=valuen** parameter of the **plot2d** function.

Questions for self-examination for the sixth lecture:

1. What is the subplot function used for?
2. What is the difference between the plot2d function and the subplot function?
3. How to set the color of the curve on the graph in the plot2d function?
4. How to set the defining window size on the chart in the plot2d function?
5. How to set the position of the axes and the presence of the frame on the graph in the plot2d function?
6. How to determine the number of main and intermediate divisions on the coordinate axes using the plot2d function?
7. How to set a legend for each curve on the graph using the plot2d function?

Lecture 7

The purpose of the lecture is to learn how to plot point graphs and graphs in the form of a stepped line, in a polar coordinate system and plots of functions specified in parametric form, get acquainted with the modes of plot formatting.

4.6 Plotting dot graphs using the plot2d function.

The **plot2d** function can be used to plot point graphs. In this case, the call to the function looks like:

plot2d(x,y,d)

here **d** is a negative number that defines the type of marker.

Number	Type of marker
- 0	point
- 1	plus
- 2	cross
- 3	plus inscribed in a circle
- 4	filled rhombus
- 5	no filled rhombus
- 6	upward triangle
- 7	downward triangle
- 8	plus inscribed in a rhombus
- 9	circle
- 10	star
- 11	square
- 12	right-pointing triangle
- 13	left-pointing triangle
-14	five-pointed star

To get acquainted with the possibilities of plotting pointed graphs, consider the following example of plotting a pointed graph of the function $y = \sin(x)$ with the marker type «plus inscribed in a rhombus».

Have determined the range of x variation, we will form arrays of **x** and **y** values. When plotting a curve using the **plot2d** function, we will specify the argument as number 8, which determines the type of marker «plus inscribed in a rhombus». In this case, the program will look like this:

```
x=[-2*%pi:0.25:2*%pi];
y=sin(x);
plot2d(x,y,-8)
```

The graph which was obtained as a result of the execution of the program will be obtained in the form shown in fig. 4.14.

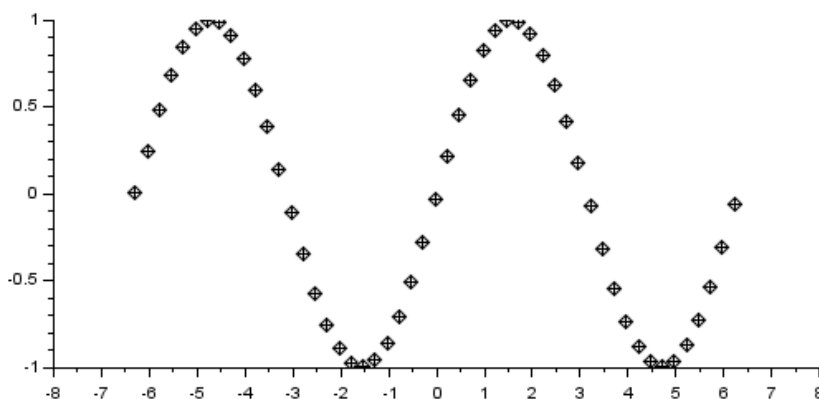


Figure 4.14 - Point graph of the function $y = \sin(x)$.

4.7 Plotting graphs in the form of a stepped line.

To display a graph as a stepped line there is a `plot2d2(x,y)` function in Scilab. It completely coincides in syntax with the `plot2d` function. The main difference is that `x` and `y` can be functions independent of each other, it is only important that the arrays `x` and `y` are split into the same number of intervals.

Let's consider the following example, we have detailed observations of population growth on the planet from 1947 to 2021 in billions of people. Let's plot a graph reflecting the dynamics of the process based on data from 1947, 1958, 1970, 1980, 1999, 2006 and 2021.

We will introduce the arrays `x` and `y` element by element and use the function `plot2d2(x,y)`:

```
x=[1947 1958 1970 1980 1999 2006 2021];  
y=[2.003 3.1 3.6 4.7 5.2 5.4 7.1];  
plot2d2(x,y,axesflag=1);
```

The graph will be obtained as a result of the execution of the program and obtained in the form which will be shown in fig. 4.15.

4.8 Plotting graphs in polar coordinate system.

The polar coordinate system consists of a given fixed point 0 – the pole of the system, concentric circles which are centred at the pole and rays emanating from the point 0, one of which is 0X – the polar axis.

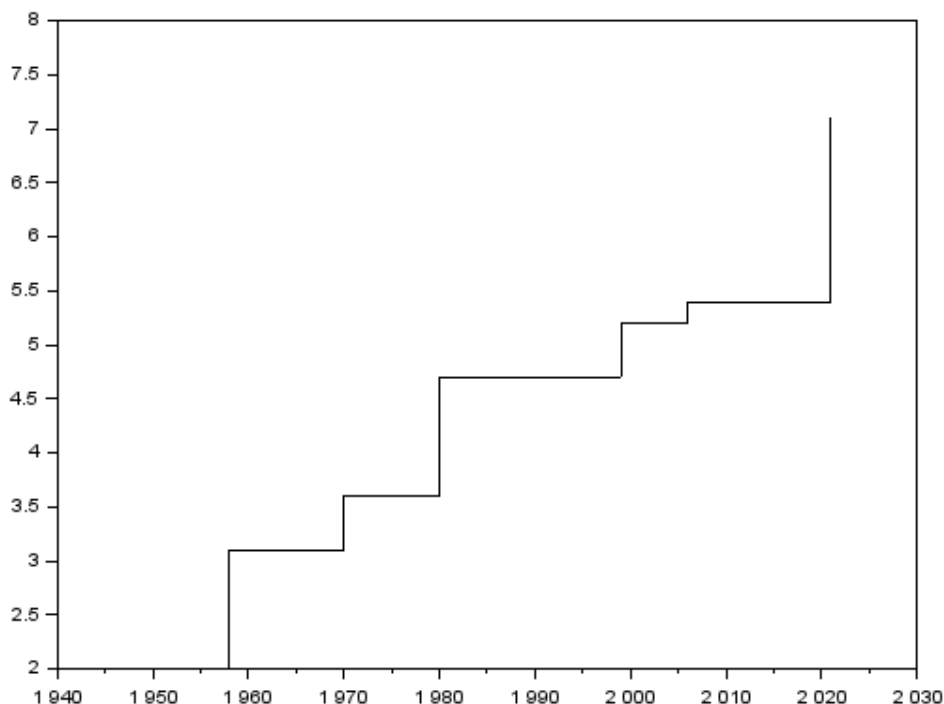


Figure 4.15 – Stepped graph which is obtained using the `plot2d2` function.

The location of any point M in polar coordinates can be specified by a positive number $\rho = OM$ (polar radius), and a number φ which is equal to the value of the XOM angle (polar angle).

In Scilab, to generate a plot in a polar coordinate system, you need to generate arrays of polar angle and polar radius values, and then call the **polarplot** function:

```
polarplot(fi, ro, [key1=value1, key2=value2, . . . , keyn=valuen])
```

Let's consider an example of plotting a polar graph of the functions $\rho = 3\cos(5\varphi)$ and $\rho_1 = 3\cos(3\varphi)$.

Have determined the range and step of changing the polar angle, we will form the arrays **fi** and **ro**.

Let's plotting the given curves one by one using the **polarplot** function, while for the line of the graph of the **ro** function we will set the colour to red, and for the function **ro1** - blue. Let's type a program in the SciNotes text editor that will look like this:

```
fi=0:0.01:2*%pi;  
ro=3*cos(5*fi);
```

```
ro1=3*cos(3*fi);  
polarplot(fi, ro, ...  
style=color("red"));  
polarplot(fi, ro1, ...  
style=color("blue"));
```

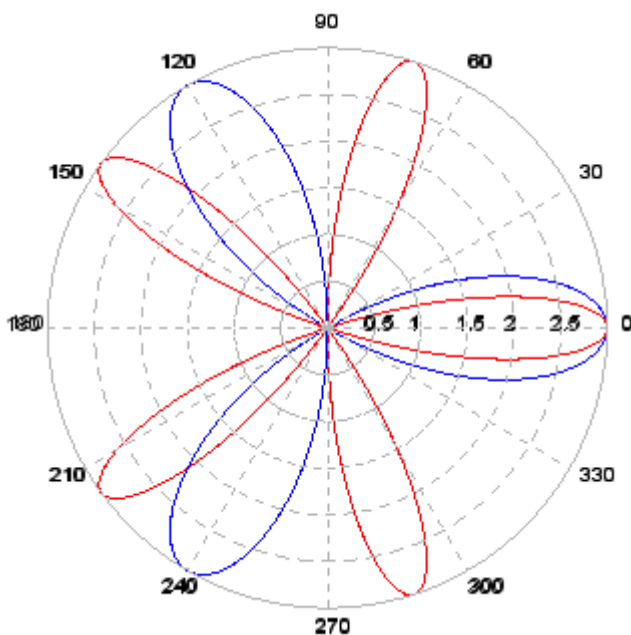


Figure 4.16 - Plotting the graph by **polarplot** function.

Let's run the program and get the graph which is shown in fig. 4.16

4.9 Graphs of functions specified in parametric form.

The setting of the function $y(x)$ using the equalities $x = f(t)$ and $y = g(t)$ is called parametric, and the auxiliary value t is called a parameter.

To plot the graph of a function specified parametrically, it is necessary to define the array t , define the arrays $x = f(t)$, $y = g(t)$ and plot the function $y(x)$ using the functions **plot(x, y)** or **plot2d(x, y)**.

For example, let's plot a graph of a strophoid, which is an algebraic curve of the third order and is generally given by the equation:

$$x^2(a + x) = y^2(a - x)$$

We represent this equation using the parameter t :

$$\begin{cases} x(t) = \frac{(t^2 - 1)}{(t^2 + 1)} \\ y(t) = \frac{t \cdot (t^2 - 1)}{(t^2 + 1)} \end{cases}$$

Let's set the arrays t , x and y and plot a graph using the function `plot(x,y)`:

```
t=-5:0.01:5;
x=(t.^2-1)./(t.^2+1); y=t.*(t.^2-1)./(t.^2+1);
plot(x,y);
```

The function graph is shown in fig. 4.17.

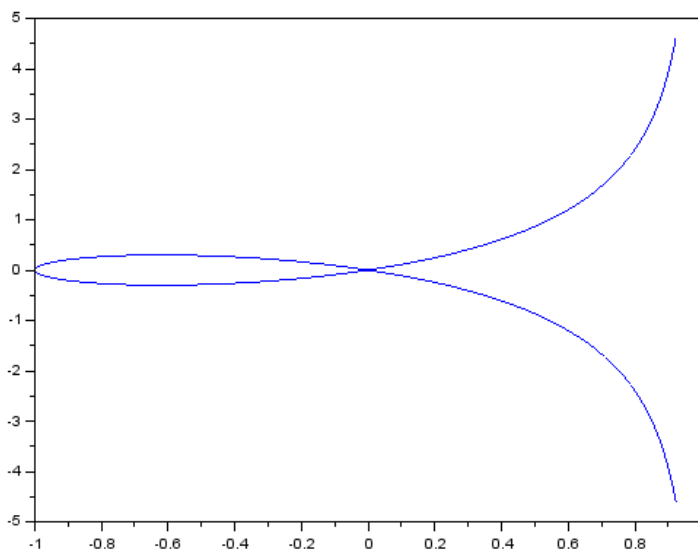


Figure 4.17 – The graph of a strophoid.

Let's consider another example of plotting a graph of a function given in a parametric form. Let's plot a semi-cube parabola.

A semi-cubic parabola is a second-order algebraic curve, which in general form can be described by the following equation:

$$y^m = A + Bx + Cx^2 + \dots + Nx^n.$$

Let's bring this equation to a parametric form:

$$\begin{cases} x(t) = 0,5 \cdot t^2 \\ y(t) = 0,3 \cdot t^3 \end{cases}$$

As in the example with the strophoid, the **t** is a parameter which is defined as an array, and **x** and **y** as dependent values. For diversity, let's plot a graph with help of the **plot2d(x,y)** function:

```
t=-3:0.01:3;  
x=0.5*t.^2;  
y=0.3*t.^3;  
plot2d(x,y);
```

The function graph is shown in fig. 4.18.

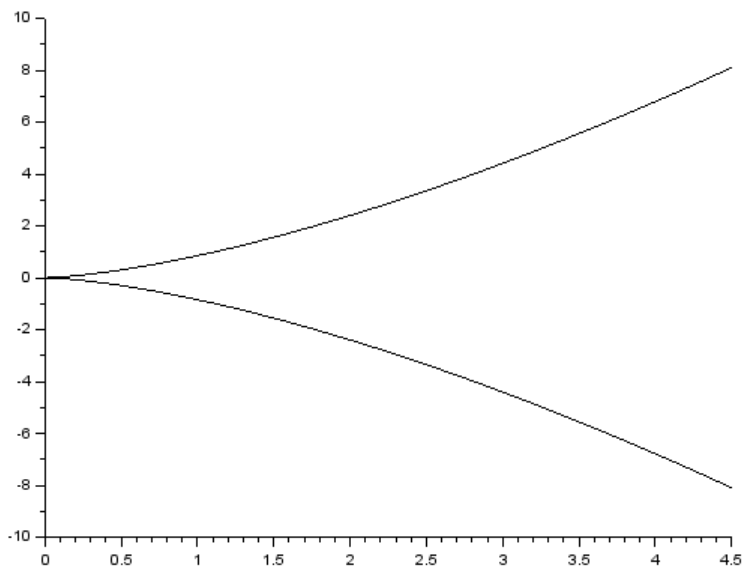


Figure 4.18 – The graph of a semi-cube parabola.

4.10 Graph formatting mode.

In Scilab, the appearance of a graph can be changed using the capabilities of the graphic window in which it is displayed. The transition to the formatting mode is carried out by the command «Edit» - «Figure properties» of the graphical window menu.

We will consider formatting possibilities by the example of plotting the graphs of the functions $y_1 = \sin(2x)$ and $y_2 = \sin(3x)$ in the interval $[0; 2\pi]$ with a step of 0.1. Let's form an array **x** and use the **plot2d** function:


```
x=[0:0.1:2*%pi]';
plot2d(x,[sin(2*x) sin(3*x)]);
```

The function graph is shown in fig. 4.19.

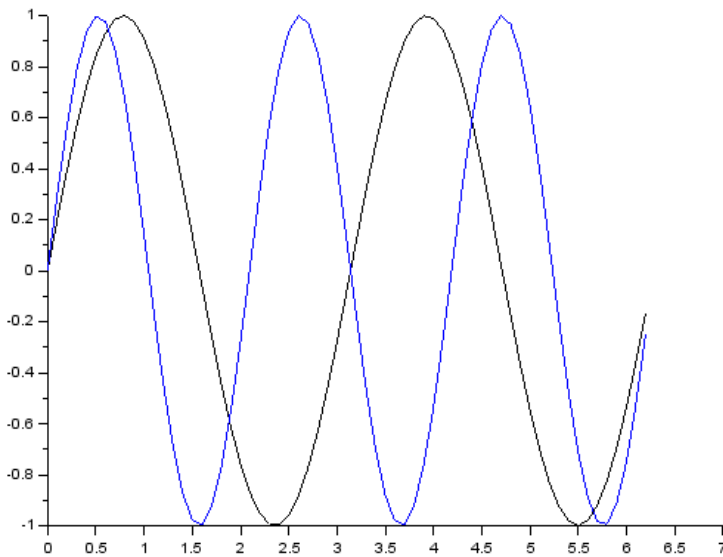


Figure 4.19 – Graphs of the functions $y_1 = \sin(2x)$ and $y_2 = \sin(3x)$.

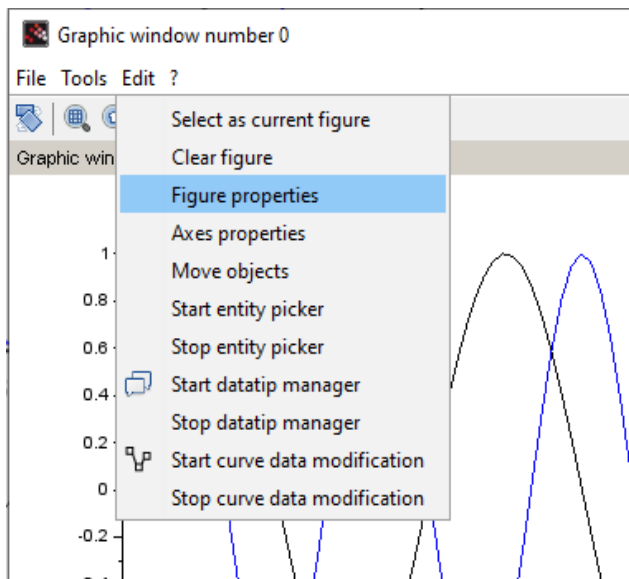


Figure 4.20 - The sequence of calling the properties of the graphic window.

Let's click on the «Edit» option of the graphic window menu, in the pop-up window click on the «Figure properties» option (fig. 4.20), as a result of these actions we call the «Figure Editor» window fig. 4.21.

The left side of the «Object Browser» is the viewport for objects available for formatting. Clicking on the Figure (1) object makes it active (highlighted in blue), and the properties of the active object appear in the right panel of the «Object Properties» window, which can be changed (fig 4.21).

Initially, the «Object Browser» always displays two objects: the «Figure (1)» and its child «Axes(1)». A plus sign next to an object indicates that it contains

objects of a lower order.

When you click the plus sign next to the «Axes(1)» object, the «Compound(1)» object appears, also with a plus sign. The «Compound(1)» object contains the graphs of the functions $y_1 = \sin(2x)$ and $y_2 = \sin(3x)$ plot in the same coordinate axes – «Polyline(1)» and «Polyline(2)», respectively (fig. 4.22).

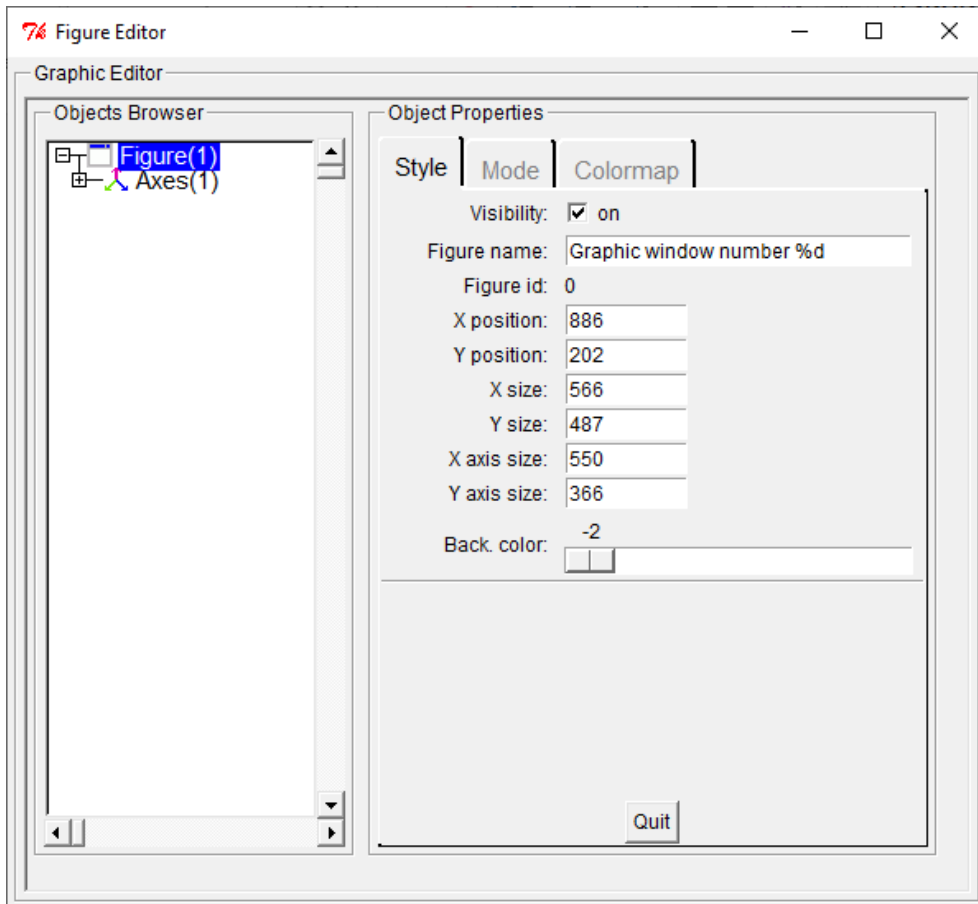


Figure 4.21 – Window of the «Figure Editor»

4.10.1 Formatting object «Figure(1)».

Recall that the Figure object is a graphic window and the actual graph which is displayed in it. To change the properties of the graphic window, select «Figure (1)» in the «Object Browser» window (fig 4.21).

Figure 4.22 shows the «Style» tab of the panel «Object Properties» for formatting properties of the «Polyline(1)» object, which is corresponded to the graph of the function $y_1 = \sin(2x)$ for our example. Here you can change the values of the following properties:

- «Visibility:» (graph display) is a switch that takes on values (a check mark in the box next to it). By default, the state is set to «on» - the graph is displayed on the screen.
- «Figure name:» is a sequence of characters that appear in the title bar of the graphic window. By default, the graphic window is assigned «Graphic window number %d», where %d is the ordinal number of the graph. However, you can enter any name which you want. For example, replace «Graphic window number %d» with «My first graph» and press the [Enter] key. The title of the window will be changed.
- «Figure id:». For the first graphic window, the «Figure id» is 0, for the

second - 1, for the third - 2, etc.

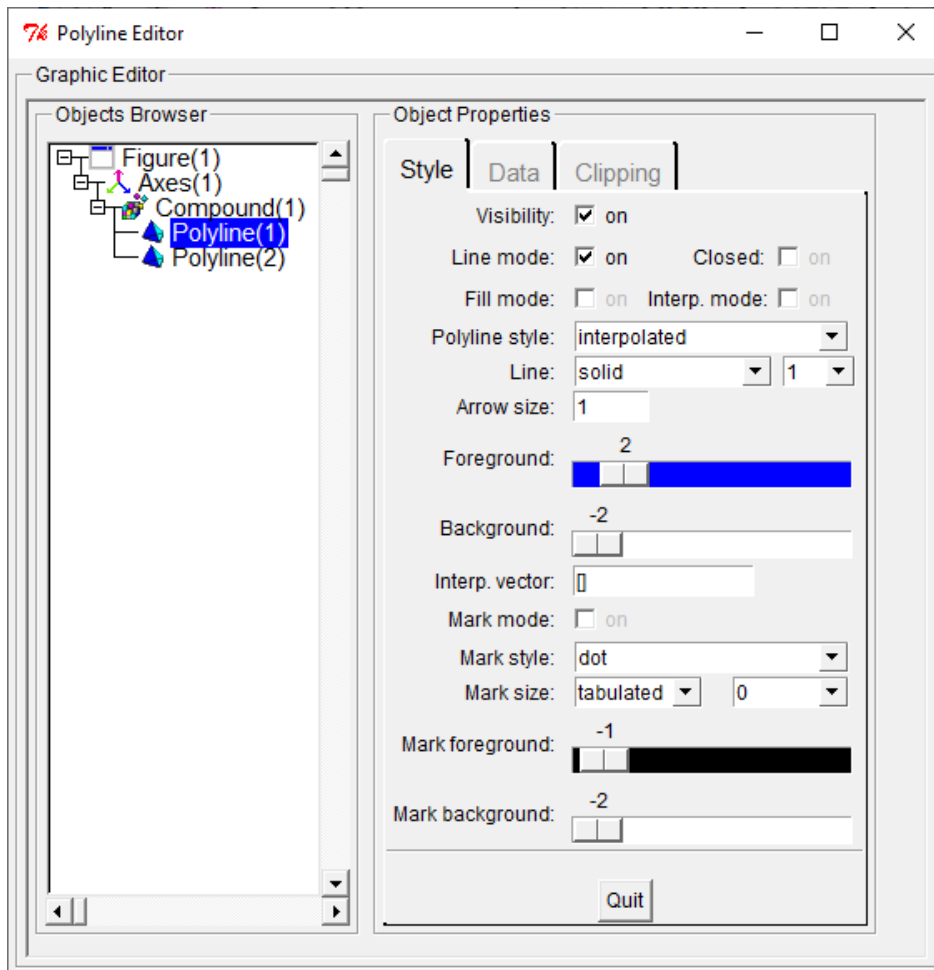


Figure 4.22 - Window for formatting graph properties.

- «X position:», «Y position:» - these fields define the position of the graphic window on the monitor in pixels horizontally and vertically, respectively. Point with coordinates [0; 0] is the upper left corner of the screen.
- «X size:», «Y size:» are respectively the width and height of the graphic window in pixels.
- «X axis size:», «Y axis size:» - these values determine the size of the axes.
- «Back. Color:» - each position of the slider has its own color number (RGB-id). 35 shades are available (from 2 - white to 32 - hot yellow). If you set the slider to position 30. The background color will turn pale pink (fg.4.23).

On the «Mode» tab in the «Object Properties» area for the «Figure(1)» object (fig. 4.24), you can set the following properties:

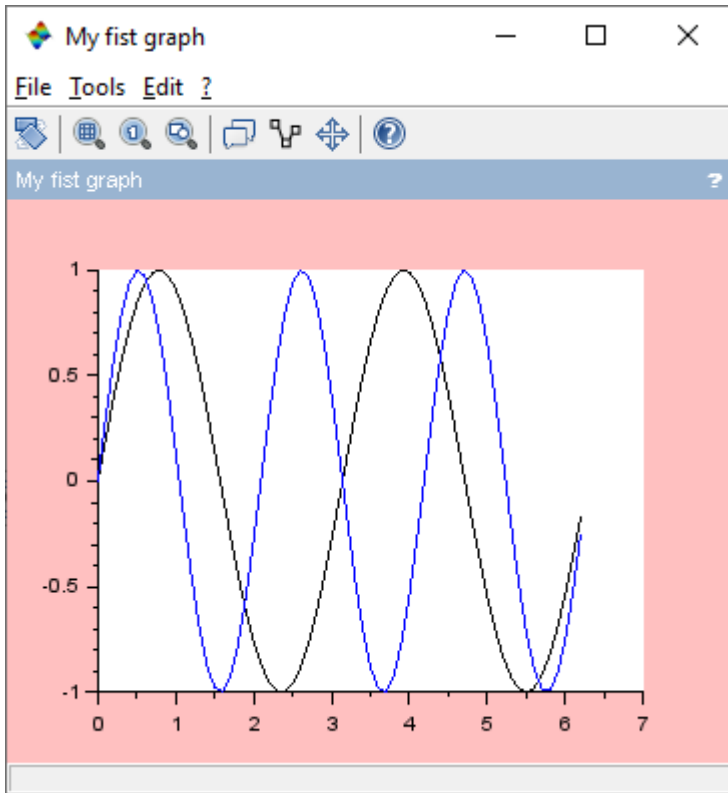


Figure 4.23 – Changing the properties of the Graphic window.

is performed exactly. However, it is often necessary to apply an image to an existing one, while the color of the newly built graphic should clearly stand out. There is a set of modes for this: «clear», «and», «andReverse», «nInverted», «noop», «xor», «nor», «equiv», «invert», «orReverse», «copyInverted», «orInverted», «nand», «set».

- «Rotation style:» - this property applies only to 3D graphs. The default «unary» mode is designed to rotate the selected graphs; when the «multiple» mode is on, all three-dimensional graphs are rotated.

- «Auto resize:» - a property that allows you to resize the graph. When this mode is enabled (the switch position is "on" - by default), we can resize the graphic window by dragging its borders with the mouse, and the size of the graph displayed in the window will automatically resize. In the off position, the graph will retain its size (fig. 4.24).

- «Pixel drawing mode:» - property that determines how the image is rendered on the screen. The default is “copy” mode. In this case, the required plotting operation

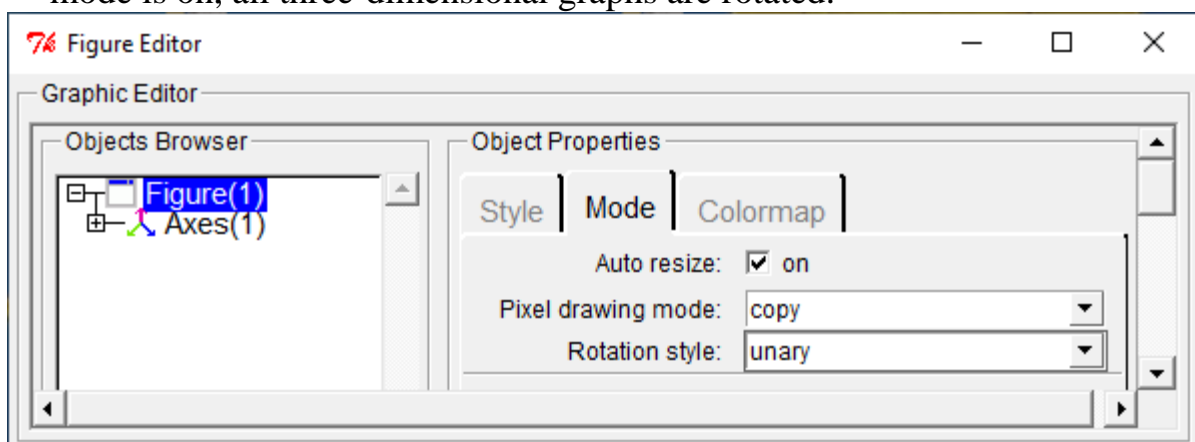


Figure 4.24 – «Mode» tab of the formatting the window of the «Figure Editor».

Questions for self-examination for the seventh lecture:

1. How to plot point graphs? on
2. How to plot graphs in the form of a stepped line?
3. How to plot graphs in a polar coordinate system?
4. How to plot graphs of functions which are specified in parametric form?
5. How to format graphs?

Lecture 8

The purpose of the lecture is to study the parameters of the `plot2d` function and learn how to plot graphs using it.

4.10.2 Formatting the Axes object.

To change the properties of the «Axes(1)» object, select it in the «Object Browser» field of the formatting window (fig. 4.25). In the «Object Properties» area, the properties available for modification will be grouped into several tabs. The «X», «Y» and «Z» tabs are identical, with the only difference that they allow you to set the desired appearance for the «X», «Y» and «Z» axes, respectively. Therefore, we will only consider the «X» tab.

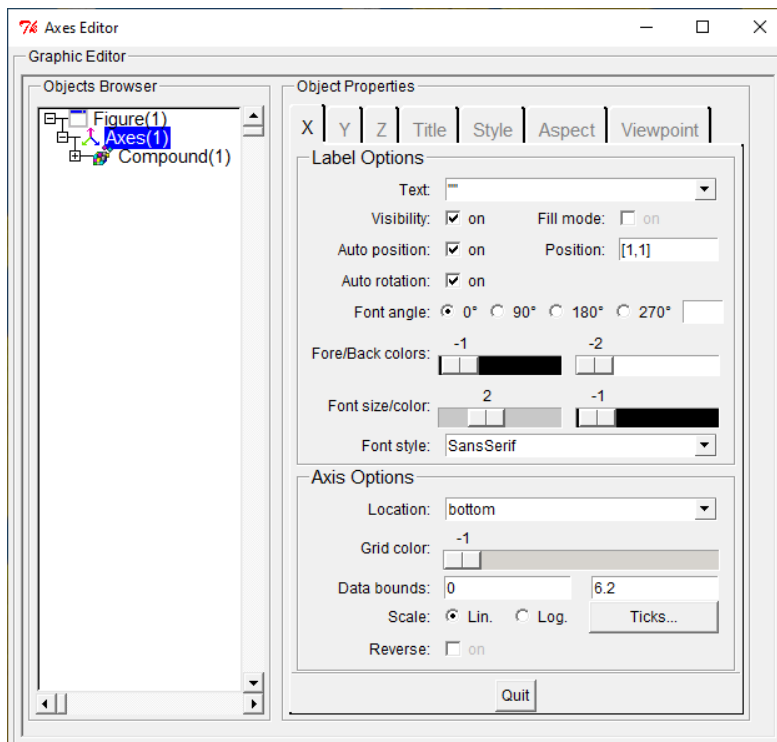


Figure 4.25. The «X» tab of the formatting the properties of the axis X

On the «X» tab, all properties are divided into two areas: «Label Options» and «Axis Options». In the «Label Options» area you can set:

- «Text:» – axis label is any sequence of characters which is located next to the corresponding axis;
- «Visibility:» determines the visibility of the «Text:» caption. The switch is set

to the «on» position, which is indicated by a mark in the window on. By default, the axis label of the graph is displayed (position «on»).

- «Fill mode:» – is a switch that accepts values «on» or «off» (default is «off»). In order to define the background color around the axis label, it is necessary to set the state «on».

- «Auto position:» – automatic determination of the position of the graph axis label. By default, the value is set to «on» - the label is displayed at the bottom, in the center of the axis. However, the position of the label can be determined independently, for this, coordinates are specified in the «Position:» field in the form of a vector [x; y]. In this case, the «Auto position:» switch will automatically take the value «off».

- «Auto Rotation:» – the mode of automatic rotation of the axis label. By default, this mode is disabled (switch state «off»)

- «Font angle:» – angle of rotation of the axis label. You can set one of the suggested values: 0, 90, 180 and 270 degrees, as well as any arbitrary angle of rotation of the inscription in the last field (fig. 4.25).

- «Fore/Back colors:» – the color of the frame around the label and the background color of the axis label, respectively - set using the slider, each position of which corresponds to a certain color. A total of 35 colors are available. .

- «Font size/color:» – The first slider sets the size of the axis label symbols, possible values are from 0 to 6, the default size for the font is 1. The second slider sets the color of the text symbols, 35 colors are available.

- «Font style:» – style of the axis label symbols. The default style is «Sans Serif».

In the «Axis Options» area, you can change:

«Location:» – the location of the axis of the graph. For the «X» axis, the following values of this property are possible: bottom, top, middle and origin. For the «Y» axis, the following values of this property are possible: left, middle, right and origin.

- «Grid color:» – the color of the graph grid lines, set using the slider. At position «-1» there are no graph grid lines, at position «0» black lines are displayed, in addition, 32 more colors are available. In order to display grid lines for the «X» and «Y» axes tabs, you must set the «Grid color:» property on both the «X» and «Y» tabs.

- «Data bounds:» – data limitation. For each axis, you can reduce the range of data that is plotted by stretching or shrinking the axis.

- «Scale:» – scale of the axis of the graph. There are two automatic modes: «Lin» (linear) and «Log» (logarithmic). Pressing the [Ticks...] button brings up the «Edit Axes Ticks» window for modifying the tick of the axis. This window is shown in fig. 4.26.

It can be used to set the following properties of the axis ticks:

- «Visibility:» – a switch that takes on the value «on», which is signaled by a check mark in the box on. By default, the tick marks on the axis of the graph are displayed (position «on»).

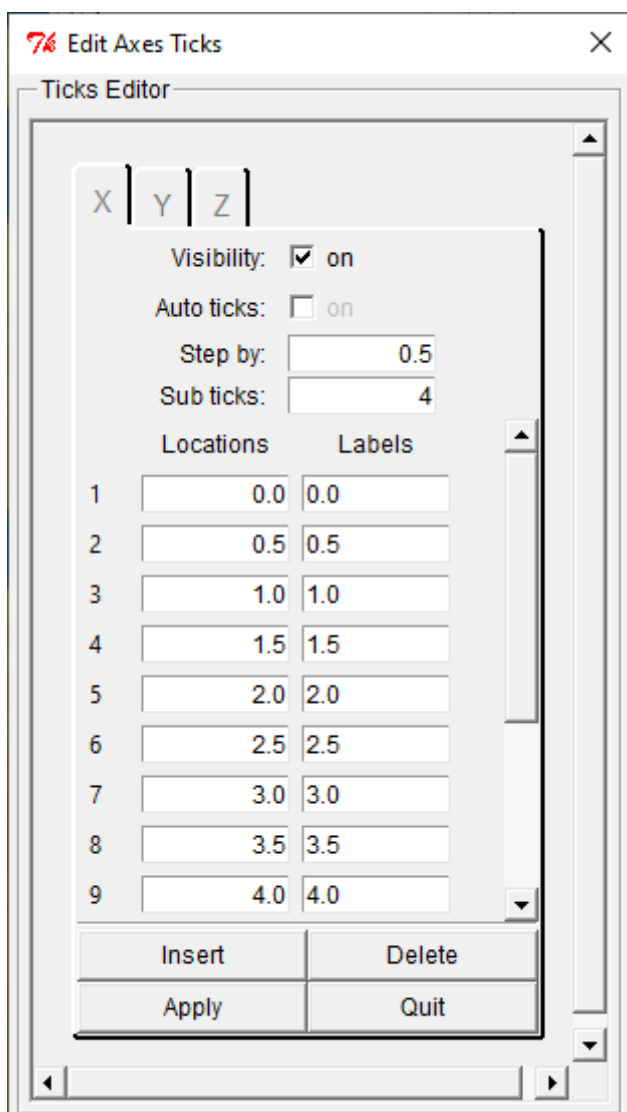


Figure 4.26 – Window Edit Axes Ticks.

starting from the position of the active cell. The [Delete] button allows you to delete not only the active cell, but the entire row to which it belongs. The [Apply] button confirms the changes, and [Quit] exits the «Edit Axes Ticks» window.

The last option on the «X» tab is the «Reverse:» switch. If you set it to «on», the graph will be mirrored about the Y axis. If you enable this mode on the «Y» tab, the graph will be mirrored about the X axis.

The «Title» tab of the formatting window is intended for changing the properties of the graph title. It contains only one area – «LabelOptions», which is identical to the «Label Options» area of the «X», «Y» and «Z» tabs (fig. 4.27)

Let's illustrate the possibilities of changing the properties of the graph coordinate axes using the «X», «Y» and «Title» tabs of the «Axes Editor» window.

Plotting the functions $y_1 = \sin(2x)$ and $y_2 = \sin(3x)$ in the interval $[0; 2\pi]$ with a step of 0.1. Let's form an array **x** and use the **plot2d** function:

```
x=[0:0.1:2*%pi]';
plot2d(x, [sin(2*x) sin(3*x) ] );
```

- «Auto ticks:» – the mode of automatic division of the axis is also enabled by default (the value of the switch is «on»). However, it is possible to independently determine the step with which the axis will be staked, it must be entered in the «Step by:» field and press [Enter] key. The «Auto ticks:» switch will automatically uncheck in the box «on».

- «Sub ticks:» – intermediate ticks. In this field you need to enter the number of tick marks that will be displayed between the major tick marks on the axis. It should be noted that intermediate ticks are not signed.

A table of ticks is generated in the «Ticks Editor:» window. The first column, «Locations», specifies the position of the ticks, and the second, «Labels», the label of the ticks.

For the convenience of editing the table, the window is equipped with buttons [Insert], [Delete], [Apply], [Quit]. The [Insert] button allows you to insert a ready-made ticks table (or its fragment) into the window using the clipboard. The insertion is performed

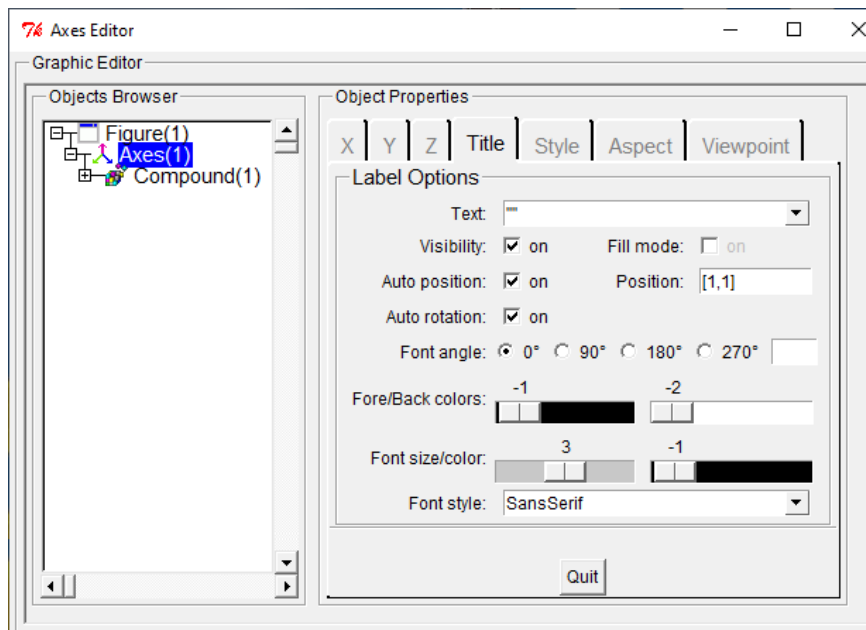


Figure 4.27 – Tab «Title» of the window «Axes Editor».

Let's display the labels for the X-axis «Abscissa Axis» and for the Y-axis «Ordinate Axis», set the «Serif Bold» font style and size for the labels to 3. For both axes, set the «Grid Color:» slider to position 1. Define the middle position for the «X» axis, and on the tab «Y» turn «on» the «Reverse:» mode. Let's display the «Title» of the graph "Graphs of functions y1 and y2)", defining the «Serif Bold» font style, the «Font size:» is 4, and the «Font color:» is red (the position of the slider is 5). Turn «on» the «Fill mode:» and set its «Back colors:» to yellow (the position of the slider 7).

As a result which is shown in figure 4.28 the form of the graphic window will be formed.

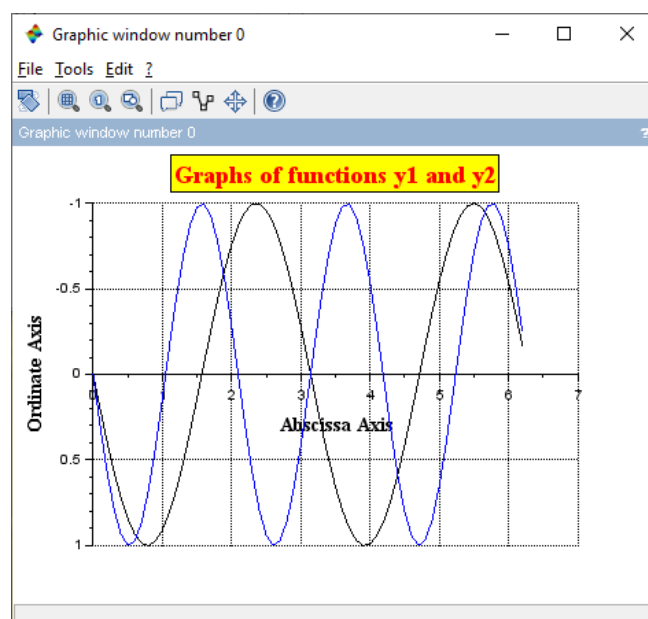


Figure 4.28 – The graph which is formed by the «Axes Editor» window tabs.

The «Style» tab of the «Axes Editor» graph axes formatting window (see fig. 4.29) provides the ability to change the following properties of the axis line and tick marks:

- «Visibility:» is a switch that accepts values «on» (by default). If there is no check mark in the box, then the chart is not displayed in the graphic window at all.
- «Font style:» is the character style of the tick labels on the axis. The default style is SansSerif.
- «Font color:» is a slider, each position of which determines the color of the symbols for the tick labels. The default is set to «-1» - black.
- «Font size:» – is the size of the characters of the tick labels on the axis, the possible values are from «0» to «6». By default, the font size is set to «1».

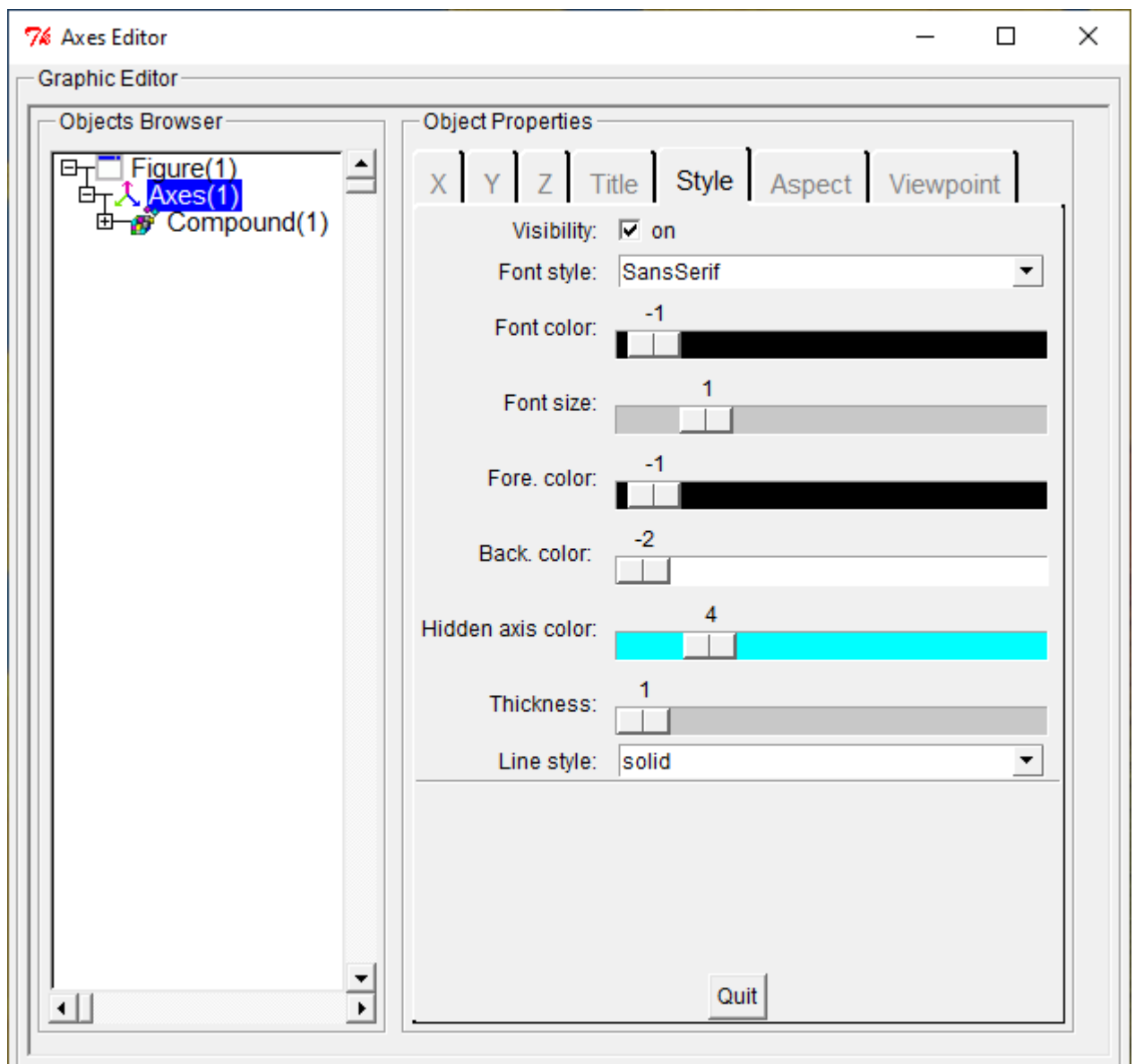


Figure 4.29 – The Style tab of the Axes Editor formatting window.

- «Fore.color:» – is a slider, each position of which determines the color of the

actual coordinate axis. The default is set to «-1» - black.

- «Back.color:» is a slider, each position of which determines the fill color of the chart background. The default is set to «-2» - white.

- «Thickness:» is the line width of the coordinate axis, defined by a slider with positions from «1» to «30». By default, the line width is set to «1».

- «Line style:» – line style. There are 6 possible modes, solid - solid line, other modes - dotted line variations.

As an example, let's format the graphs of the functions of the previous example using the «Style» tab. Let's set the font style of the caption on the «Serif Bold» axes, the font size to 2, set the blue background color of the graph (the slider position «Back.color» 12) and the line width of the coordinate axes 3. The graphical window shown in fig. 30 will be displayed on the screen.

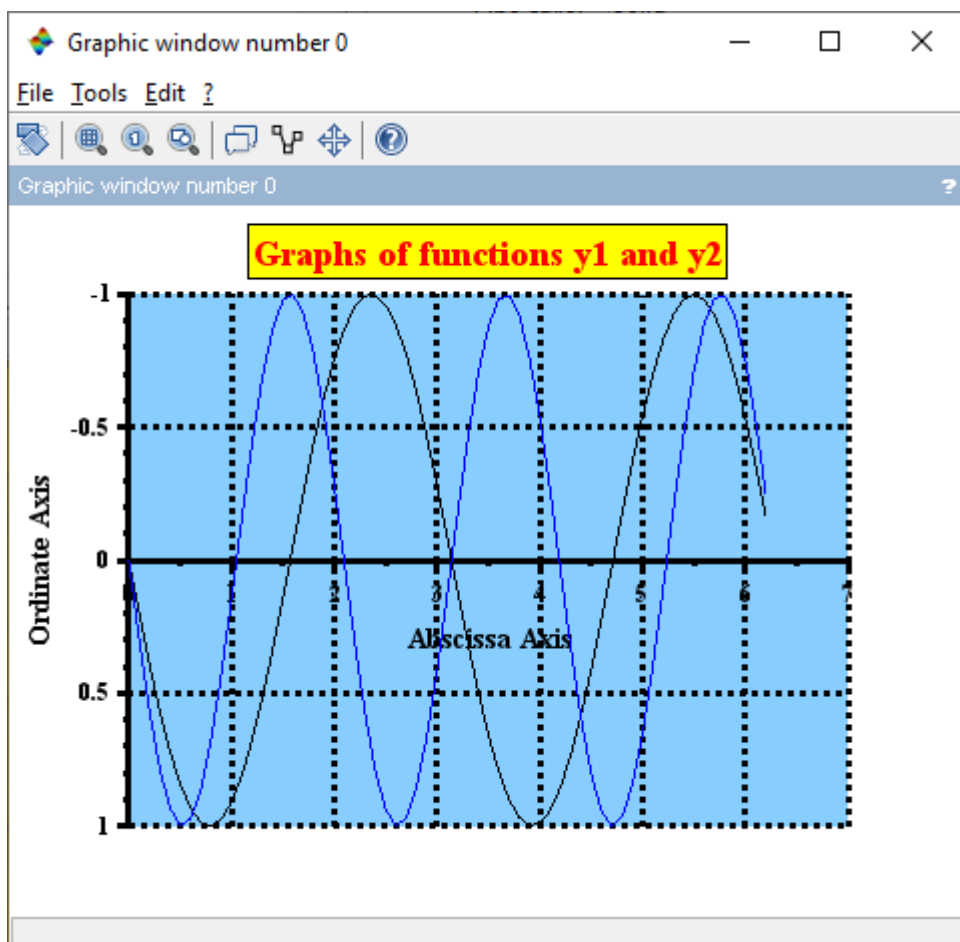


Figure 4.30 – Formatting the graph axes using the «Style» tab.

«Aspect» tab of the «Axes Editor» (fig. 4.30) allows you to change the following properties:

- «Auto clear:» if the switch is set to «on» (the checkbox is checked), the graphic window will be automatically cleared every time before a new graph is drawn. If this mode is disabled (by default), graphs will be superimposed in the same coordinate axes in accordance with the «Auto scale» mode.

- «Auto scale:» – is the mode of updating the boundaries of the coordinate axes

of the graph. In the state of the switch «on» (the checkbox is checked by default), the new graph will change the boundaries of the previous graph to form over the entire specified interval, but at the same scale as the previous graph. When the «Auto scale:» mode is disabled, the new graph will be drawn within the axes of the previous graph and, possibly, will reflect only a part of the specified interval.

- «Isoview:» – is the property which is used to set the same scale for all graph axes. The property is disabled by default (there is no check mark in the box).
- «Tight limits:» if this mode is enabled, the axes of the graph are changed in such a way as to exactly correspond to the value of the «Data bounds:» property of the X, Y and Z tabs. In the disabled state (by default), the axes can increase the initial interval to make it easier to select the scale of the axis and apply ticks to it.
- «Cube scaling:» – this option only applies to 3D graphs. When the switch is «on», the initial data is limited so that the surface fits into a cube of size 1. This allows you to more clearly depict the 3D graph in cases where the scale of the coordinate axes is too different from one axis to another. By default, the switch state is set to «off» (there is no check mark in the box).

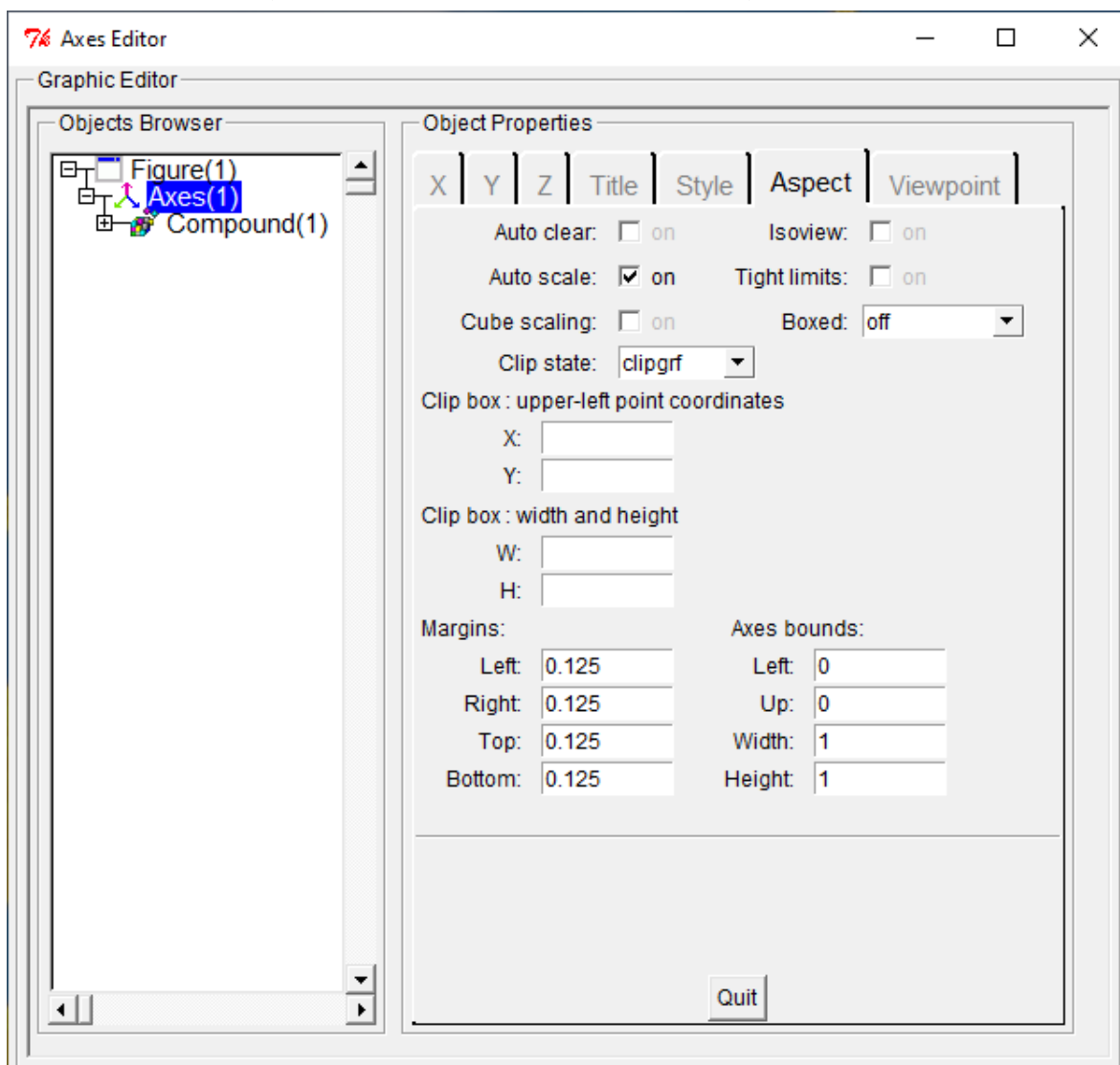


Figure 4.31 – The «Aspect» tab of the «Axes Editor» formatting window.

- «Clip state:» is cropping mode. One of the following switch states is possible: «off» means that the generated image is not cropped; «Clipgrf» (default) - the area outside the axes is cropped from the created image; «On» - the area that is outside the boundaries set by the «Clip box:» property is cropped from the created image.
- «Clip box:» is a rectangular area that will be displayed after cropping the image. First, the upper-left point coordinates are set in the «X:» and «Y:» fields, then the width and height - the «W:» and «H:» fields.
- «Margins:» this property sets the distance from the border of the graphic window to the graph area: «Left:», «Right:», «Top:», «Bottom:». The value must be in the range [0: 1]. By default, each field is assigned a value of 0.125.
- Axes bounds this property sets the part of the graph that will be displayed in the coordinate axes. Left and Up define the position of the upper left corner, Width and Height - the width and height of the graph fragment. The value must be in the range [0: 1]. By default, the displayed fragment is set by the matrix [0 0 1 1].

Let's illustrate the action of the Auto scale mode by plotting the graphs of the functions $y = \sin(x)$ and $y_1 = \cos(x)$ on the interval $(-2\pi..2\pi)$, and then the graph of the function $y_2 = \sin(3x)$ on the interval $(0..2\pi)$. Note that the interval of the third graph is much narrower. Since the «Auto scale:» mode is on by default, the axes will be changed so that both graphs are formed completely at the specified intervals:

```
x=[-2*%pi:0.1:2*%pi]; y=[sin(x); cos(x)];  
plot2d(x,y');  
x=[0:0.1:2*%pi];  
plot2d(sin(3*x));
```

The resulting graph is shown in Figure 4.32.

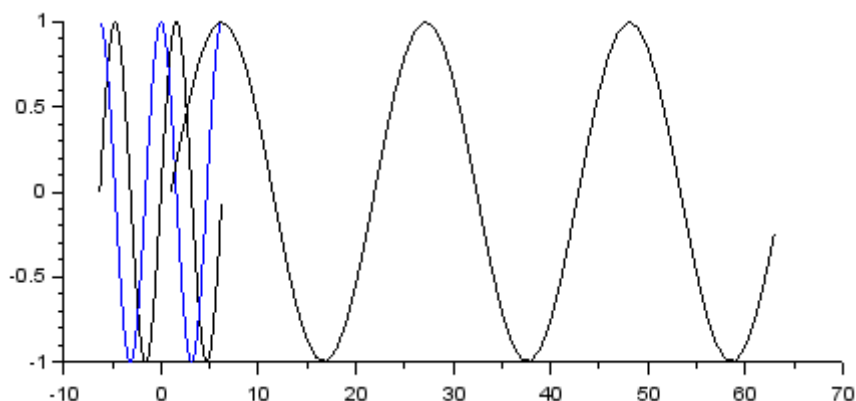


Figure 4.32 – Plotting of the graph with Autoscale enabled.

The «Viewpoint» tab of the «Axes Editor» formatting window (fig. 4.33) allows you to set only one property - the angle at which the observer sees the graph. By default, the «Rotation angles:» are set to 0 and 270 degrees.

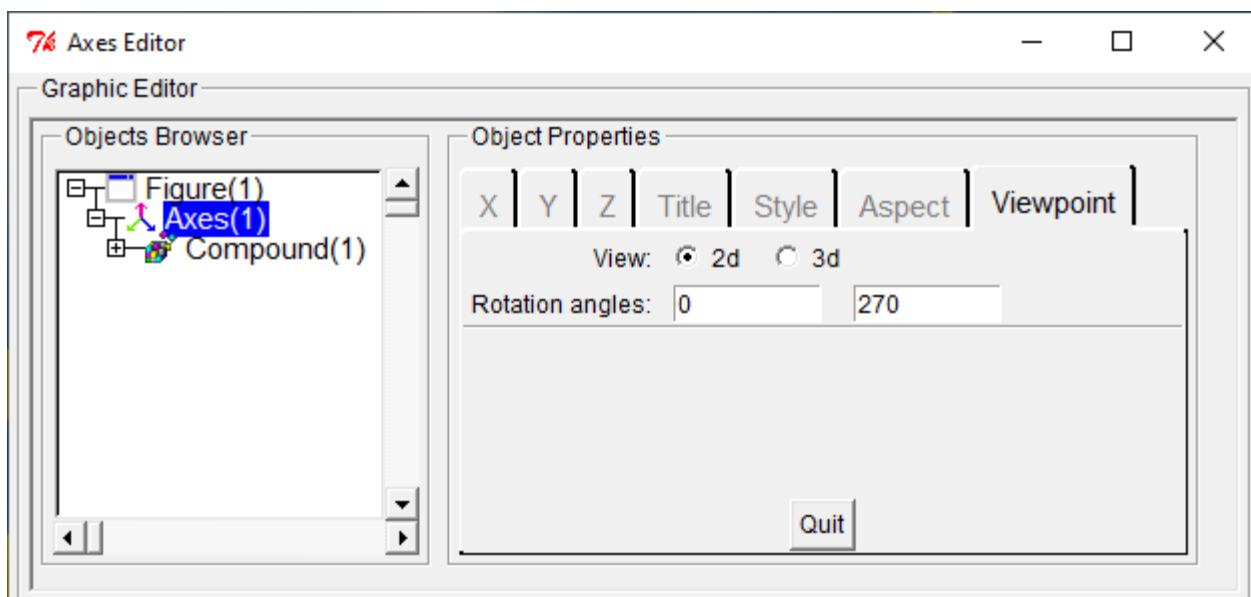


Figure 4.33 – The View Point tab of the Axes Editor format window.

4.10.3 Polyline object formatting.

For formatting the graph line, select the «Polyline(1)» object in the «Object Browser» window. The properties available for editing in the «Object Properties» area are grouped on three tabs: «Style», «Data», «Clipping».

The «Style» tab of the «Polyline Editor» formatting window (see fig. 4.34) allows you to set the values of the following properties:

- «Visibility:» – a switch that takes values «on» (default). In the off position (there is no mark in the box), the graph line is not displayed in the graphic window.
- «Line_mode:» – This value should be «on» (line drawn) or «off» (no line drawn).
- «Fill mode:» – fill mode (disabled by default). In order to determine the background color of the area that the curve limits, the switch must be set to «on».
- «Closed:» – if you turn on this property, the graph line will become closed.
- «Polyline style:» – graph display style. Possible values are: «interpolated» - solid smooth line; «staircase» - stepped line; «barplot» - striped areas; «arrowed» - a line consisting of a sequence of arrows, the size of the arrow can be set in the «Arrow size:» field; «filled» - filled areas; «bar» - banded areas bounded by a solid smooth line (fig. 4.45).
- «Line:» – line styles. There are 8 styles available: solid, dot, the rest - variations of the dashed line. Here you can also select the desired curve thickness from the list: from 0 to 10.
- «Foreground:» and «Background:» – properties that set, respectively, the color of the graph line and the fill of the area that is limited by the curve, in this time the «Fill mode:» switch should be set to «on».

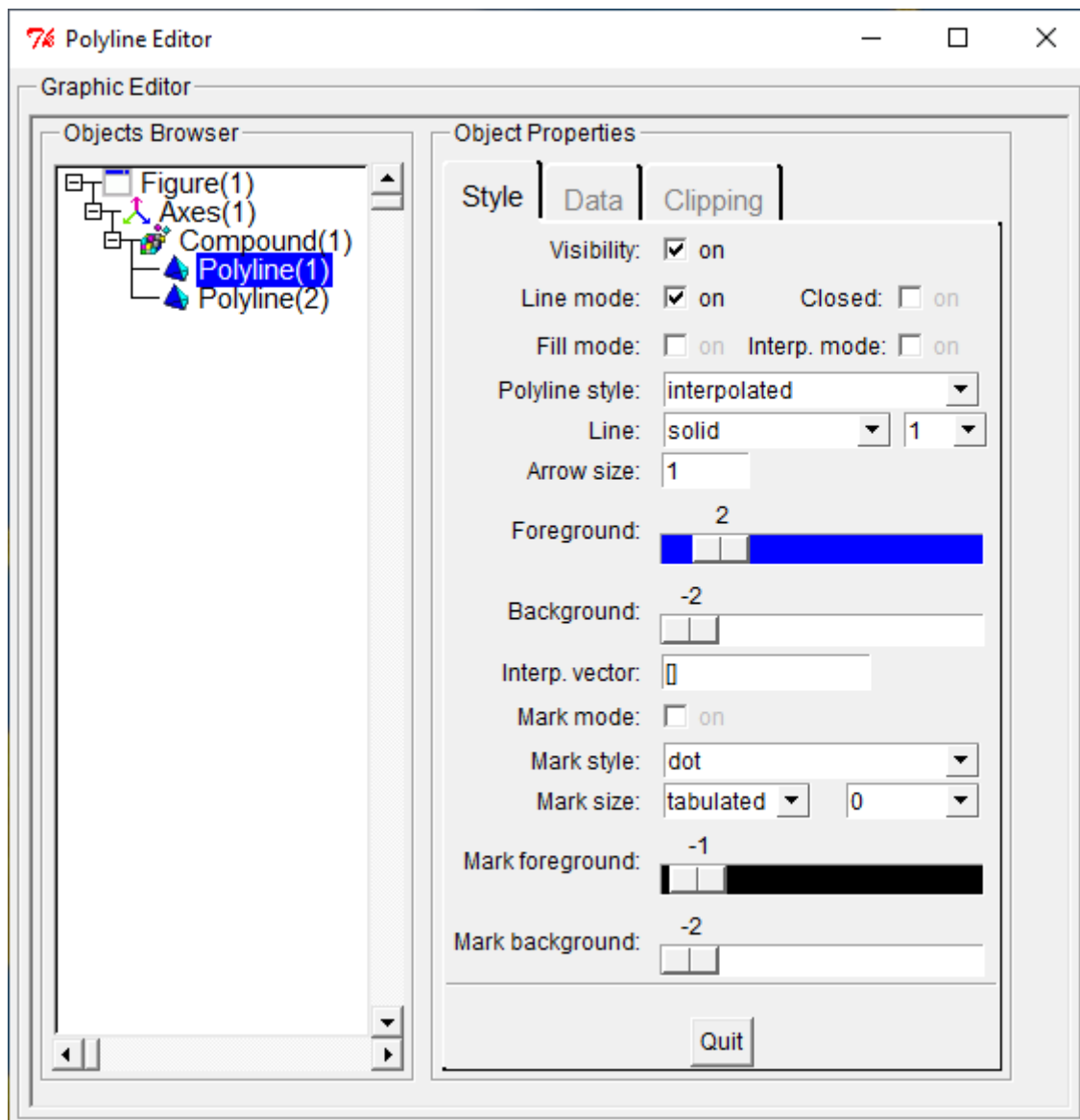


Figure 4.34 – The «Style» tab of the «Polyline Editor» formatting window.

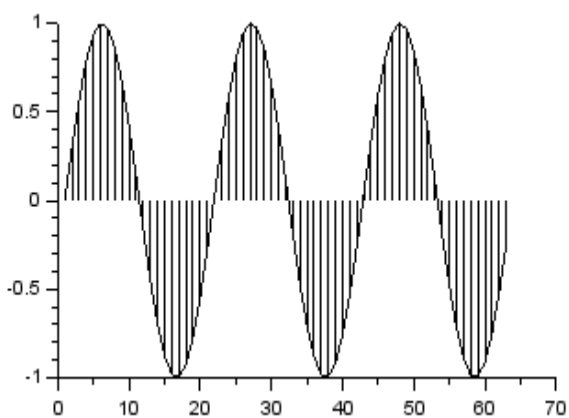


Рис. 4.45. Стиль отображения линии графика bar.

- «Interp.vector:» – a vector that defines the fill for each segment of the graph.
- «Mark mode:» – a mode that allows you to build pointed graphs (switch position «on»). This property is disabled by default.
- «Mark style:» – marker style, the following values are possible: «dot» - point; «plus» - plus sign; «cross» - cross; «star» - plus inscribed in a circle; «filled diamond» - filled diamond; «diamond» - rhombus; «triangle up»

- triangle with top up; «triangle down» - triangle with top down; «diamond plus»
- a plus inscribed in a rhombus; «circle» - circle; «asterisk» - little star; «square»
- square; «triangle right» - triangle with top to the right; «triangle left» - triangle with top to the left; «pentagram» is a five-pointed star.

- «Mark size:» – marker size settable values can vary from 0 to 30 pt.
- «Mark foreground:» – a slider, each position of which determines the fill color of the marker.

The «Data» tab of the «Polyline Editor» formatting window allows you to specify the data area on which the graphs are drawn. The field «Data field:» initially indicates the current range, in our case these are 2 arrays of type Double, each of them has 63 values - [63x2 double array] (fig. 4.46). However, in «Data field:» you can select the «Edit data...» line and edit the source data table.

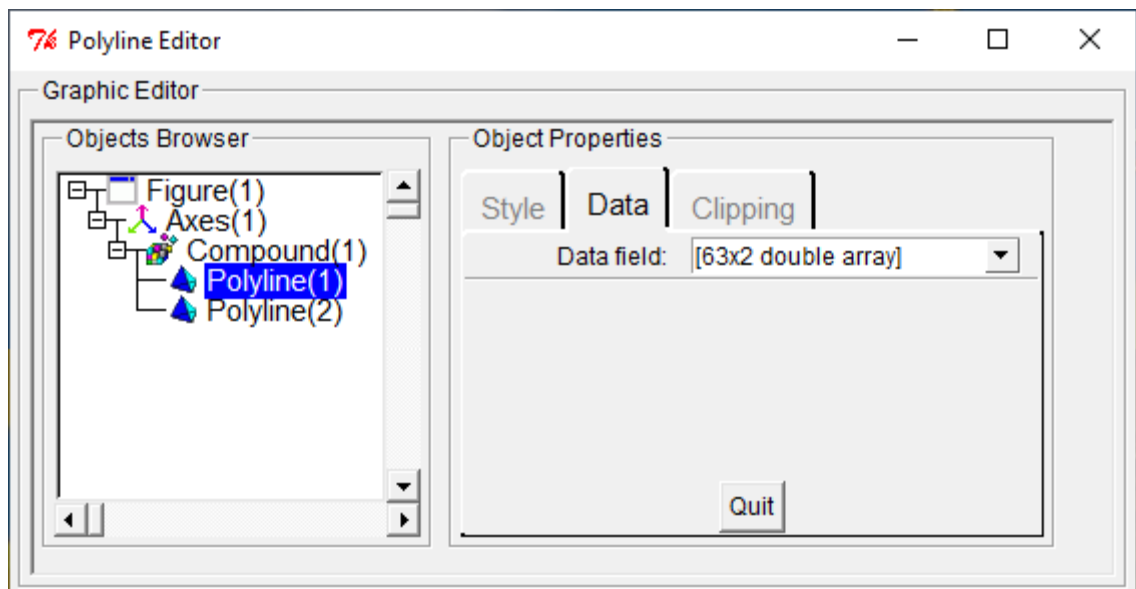


Figure 4.46 – The «Date» tab of the «Polyline Editor» formatting window.

The «Clipping:» tab of the «Polyline Editor» formatting window allows you to set the borders of a rectangular area – «Clip box:», which will remain visible after cropping the image (fig. 4.47). In the «X:» and «Y:» fields you should specify the x ; y coordinates of the upper left corner of the frame, and in the «W:», «H:» fields - its width and height. The «Clip state:» mode can also take one of the following values: «off» - means that the generated graph are not cropped; «Clipgrf» - (default) means that the area outside the bounds of the axes is cropped from the generated graph; «On» - the area that is outside the boundaries set by the «Clip box:» property is cropped from the generated graph.

Questions for self-examination for the eighth lecture:

1. How to label the axes of the graph and style its title?
2. How to change the font style of the axis labels and the thickness of the lines of the coordinate axes?
3. How to plot graphs of several functions at different intervals in one graphics window?

4. How do change the display style of the graph lines?

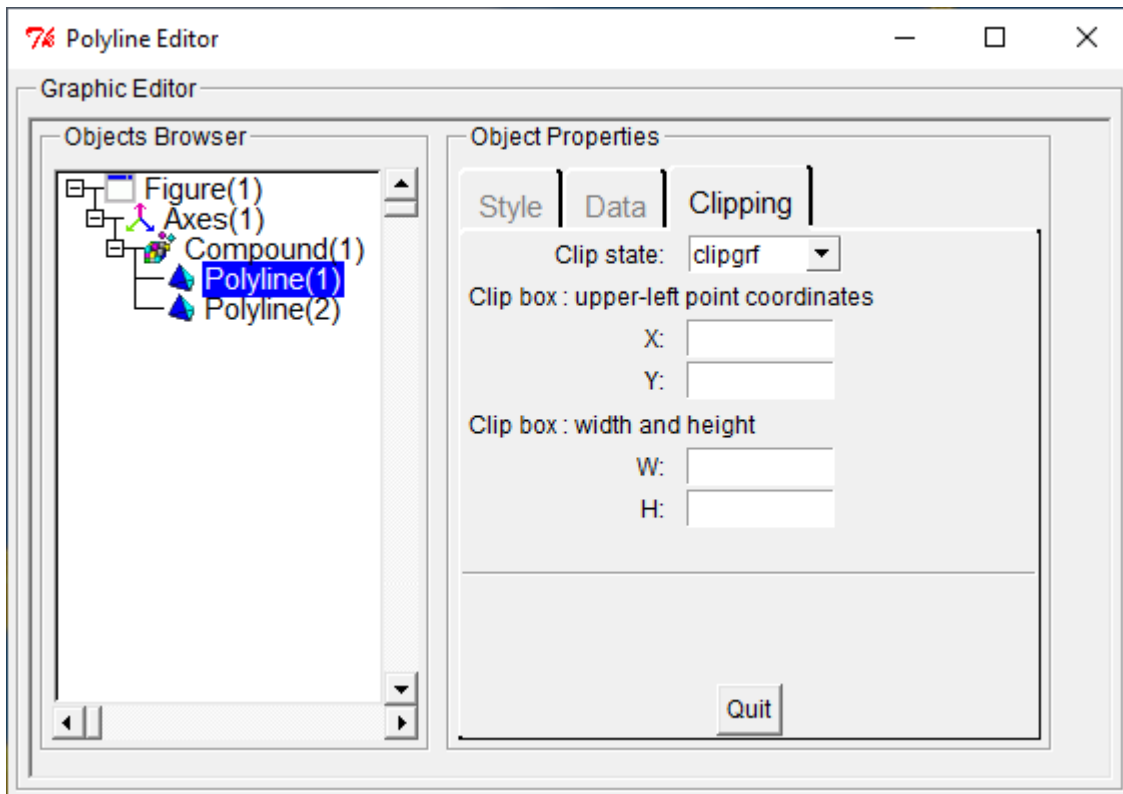


Figure 4.47 – The «Clipping» tab of the «Polyline Editor» formatting window.

Lecture 9

The purpose of the lecture is to learn how to plot volumetric graphs using the `plot3d`, `plot3d1`, `meshgrid`, `surf` and `mesh` functions, using the `genfac3d` and `eval3dp` commands.

5 PLOTTING THREE-DIMENSIONAL CHARTS IN Scilab.

The Scilab environment allows you to build 3D graphs. In this case, all graphs belong to three-dimensional, the position of each point of which is set by three values.

In general, the process of plotting a function of the form $Z(x;y)$ can be divided into three stages:

1. Creation of a rectangular grid in the plotting area. For this, straight lines are formed parallel to the coordinate axes x_i and y_j , j ,

$$x_i = x_0 + ih_x, h_x = \frac{x_n - x_0}{n}, i = 0,1,\dots,n,$$

$$y_j = y_0 + ih_y, h_y = \frac{y_k - y_0}{k}, j = 0,1,\dots,k.$$

here: h is the step of plotting the graph along the corresponding axis;

n is the number of intervals on the x -axis;

k is the number of intervals on the y -axis.

2. Calculation of the values of the function $z_{ij} = f(x_i, y_j)$ at all grid nodes.

3. Calling the function of building three-dimensional graph.

5.1 Functions plot3d and plot3d1.

In Scilab, a surface can be plotted using the plot3d or plot3d1 functions. Their difference is that plot3d builds a surface and fills it with one color, while plot3d1 builds a surface, each cell of which has a color depending on the value of the function at each corresponding grid point (fig. 5.2).

The function calls are as follows:

```
plot3d(x,y,z,[theta,alpha,leg,flag,ebox][keyn=valuen]),  
plot3d1(x,y,z,[theta,alpha,leg,flag,ebox][keyn=valuen]),
```

here: \mathbf{x} is a column vector of abscissa values;

\mathbf{y} - column vector of ordinate values;

\mathbf{z} - matrix of function values;

theta, alpha - real numbers that define in degrees the spherical coordinates of the angle of view on the graph. Simply put, it is the angle at which the viewer sees the displayed surface;

leg - labels of the coordinate axes of the graph, symbols separated by the «@» sign. For example, ' $\mathbf{x} @ \mathbf{y} @ \mathbf{z}$ ' ;

flag - an array consisting of three integer parameters: [**mode, type, box**].

Here:

mode - sets the color of the surface (table 5.1). The default is 2 - the fill color is blue, the rectangular grid is drawn.

type - allows you to control the scale of the graph (table 5.2), by default it has a value of 2;

box - defines the presence of a frame around the displayed graph (Table 5.3). The default is 4.

Table 5.1 – **mode** parameter values.

Values	Description
>0	the surface has the color of mode , a rectangular grid is displayed
0	a rectangular grid is displayed, there is no fill (white)
<0	the surface has the color mode , there is no rectangular grid

Table 5.2 – **type** parameter values.

Values	Description
0	the scaling method is applied, as for the previously created graph
1	graph boundaries are specified manually using the parameter ebox
2	the boundaries of the graph define the original data

Table 5.3 – **box** parameter values.

Values	Description
0 and 1	no frame
2	only axes behind the surface
3	the frame and axis labels are displayed
4	the frame, axes and their labels are displayed

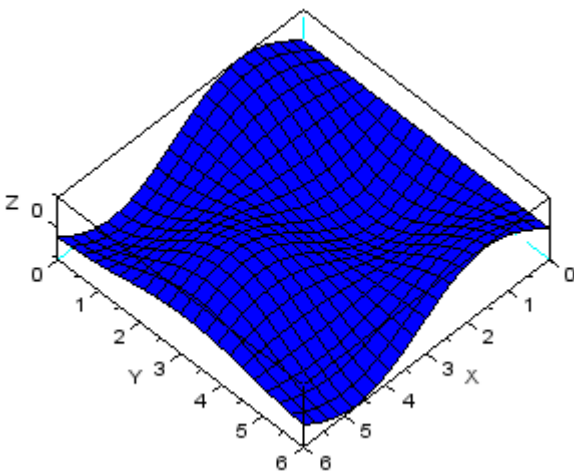


Figure 5.1 – 3D graph of the function $Z = \sin(t) \cdot \cos(t)$.

Let us plot the function $Z = \sin(t) \cdot \cos(t)$ (fig.5.1). For this, we will create an array of values for the argument t . Let's calculate the values of the function and write them to the array Z .

Note that when calling the **plot3d** function, the parameter t is specified twice as parameters X and Y that define a rectangular grid, since both functions **sin** and **cos** depend on the same variable t :

```
t=[0:0.3:2*%pi]';
Z=sin(t)*cos(t');
plot3d(t,t,Z);
```

ebox – defines the boundaries of the area into which the surface will be displayed, as a vector **[xmin, xmax, ymin, ymax, zmin, zmax]**. This parameter can only be used if the parameter **type=1**.

keyn = valuen – a sequence of values of the graph properties **key1 = value1, key2 = value2, ..., keyn = valuen**, such as line thickness, line color, background fill color of the graphic window, presence of a marker, etc.

Thus, to the function **plot3d** (**plot3d1**) must be passed a rectangular grid and a matrix of values at the grid nodes as parameters.

Having launched the program for execution, we will receive a three-dimensional graph of the function $Z = \sin(t) \cdot \cos(t)$ which is shown in fig.5.1.

Now let's complicate the task a little bit. We will plot a surface whose equation is given by two independent variables $Z = 5y^2 - x^2$ (fig. 5.2).

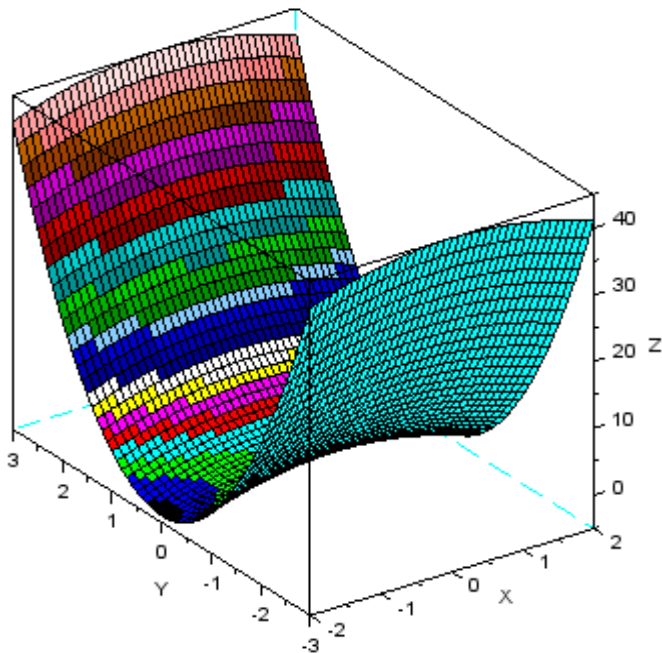


Figure 5.2 – 3D graph of the function $Z = 5y^2 - x^2$.

First, we define the arrays X and Y . Then we form the matrix of values of the function $Z(x_i; y_j)$, using the loop operator **for**. Here i is the parameter of the loop, which will iterate over all the values of the array X , and j is the parameter of the loop, which will iterate each value of the array Y which corresponds to each value of the array X .

So, first, all the values of the Z function will be calculated with changing Y (from the first to the last value in the array) and the first value of the array X . Then, with the second value of the array X , and so on. Recall that here **length** determines the number of elements in the array X (Y). Finally, to plot the surface, turn to the

plot3d1 function:

```
x=[-2:0.1:2];
y=[-3:0.1:3];
for i=1:length(x)
for j=1:length(y)
z(i,j)=5*y(j)^2-x(i)^2;
end
end
plot3d1(x',y',z,-125,10);
colorbar(-3,3)
```

Having launched the program for execution, we will receive a three-dimensional graph of the function $Z = 5y^2 - x^2$ shown in fig.5.2.

As you can see from the example, using only the **plot3d** function to graphically display indicators that depend on two independent variables is quite difficult. Scilab provides several commands to make it easier to create a rectangular grid - these are the commands **genfac3d** and **eval3dp**.

The function **genfac3d** has simplest syntax:

```
[xx,yy,zz]=genfac3d(x,y,z)
```

here: **xx**, **yy**, **zz** – the resulting matrix of size $(4, n - 1 \times m - 1)$, and **xx(:,1)**, **yy(:,1)** and **zz(:,1)** are the coordinates of each of the cells of the rectangular grid;

x – vector of x -coordinates of size m ;

y – vector of y -coordinates of size n ;

z – matrix of size $(m$ by $n)$ of values of the function $Z(x_i; y_j)$.

Let's plot the function $Z = \sin(t) \cdot \cos(t)$. For this we define the array of the parameter **t** and calculate the values of the function $Z = \sin(t) \cdot \cos(t)$. Then we will create a rectangular grid using the **genfac3d** command:

```
t=[0:0.3:2*%pi]';
z=sin(t)*cos(t);
[xx,yy,zz]=genfac3d(t,t,z);
plot3d(xx,yy,zz);
```

We will receive a graph which similar to that was obtained earlier (fig. 5.1).

The disadvantage of the **genfac3d** command is that it still does not simplify the work with the **plot3d** function, if the surface is set as a function of two variables. In this case, use the **eval3dp** command:

```
[Xf,Yf,Zf]=eval3dp(fun,p1,p2)
```

here: **Xf**, **Yf**, **Zf** – the resulting matrix of size $(4, n - 1 \times m - 1)$, **xx(:,1)**, **yy(:,1)** and **zz(:,1)** – coordinates of each of the cells of the rectangular grid;

fun – a user-defined function which defines a 3D plot;

p1 – vector of size m ;

p2 – vector of size n .

Let's illustrate the effect of the **eval3dp** command with the following example.

We will plot a graph given by the following equations $x = p_1 \cdot \sin(p_1) \cdot \cos(p_2)$, $y = p_1 \cdot \cos(p_1) \cdot \cos(p_2)$, $z = p_1 \cdot \sin(p_2)$.

First of all, let's define arrays of values for the parameters p_1 and p_2 . Next, will create a **scp** function that sets the graph.

Recall that functions in Scilab are created using the **deff** command:

```
deff([s1,s2,...]=newfunction(e1,e2,...)'
```

here: **s1**, **s2**, ... is a list of output parameters, i.e. variables to which the final result of calculations will be assigned;

newfunction – the name of the function to be created, it will be used to call this function;

e1, **e2**, ... – input parameters.

For the convenience of specifying the input matrices, we will use the function

linspace.

[v]=linspace(x1,x2 [,n])

here: **x1, x2** – real or complex numbers or column vectors;

n – integer: the number of requested values. It must be greater than or equal to two (by default, it is 100);

v – real or complex row vector.

The **linspace(x1, x2)** function generates a row vector from **n** (by default **n** = 100) equally spaced points between **x1** and **x2**. If **x1** or **x2** are complex, then **linspace(x1, x2)** returns a row vector of **n** complex values. The real part (and the imaginary part) of these **n** complex values is evenly distributed between the real parts (or the imaginary parts) **x1** and **x2**.

We will form a rectangular network using the **eval3dp** command, note that the **def** command is written in four lines only for the convenience of reading the program, and draw a graph using the **plot3d** function.

The program will take the form:

```
p1=linspace(0,2*pi,10);
p2=linspace(0,2*pi,10);
def(["x,y,z]=scp(p1,p2)",...
["x=p1.*sin(p1).*cos(p2)";
"y=p1.*cos(p1).*cos(p2)";
"z=p1.*sin(p2)"]);
[Xf,Yf,Zf]=eval3dp(scp,p1,p2);
plot3d(Xf,Yf,Zf);
```

Let's run the program for execution and get a three-dimensional graph which is shown in fig. 5.3.

Scilab also has several other functions for constructing surfaces. They have a simpler syntax, although they cannot always replace the **plot3d** function.

5.2 **meshgrid, surf** end **mesh** functions.

To form a rectangular grid, Scilab uses the **meshgrid** function, which creates matrices or three-dimensional arrays. The call to function looks like:

```
[X, Y] = meshgrid(x)
[X, Y] = meshgrid(x,y)
[X, Y, Z] = meshgrid(x,y,z)
```

here: $\mathbf{x}, \mathbf{y}, \mathbf{z}$ are vectors; $\mathbf{X}, \mathbf{Y}, \mathbf{Z}$ – matrices, if there are two input arguments, and if there are three input arguments, then three-dimensional arrays.

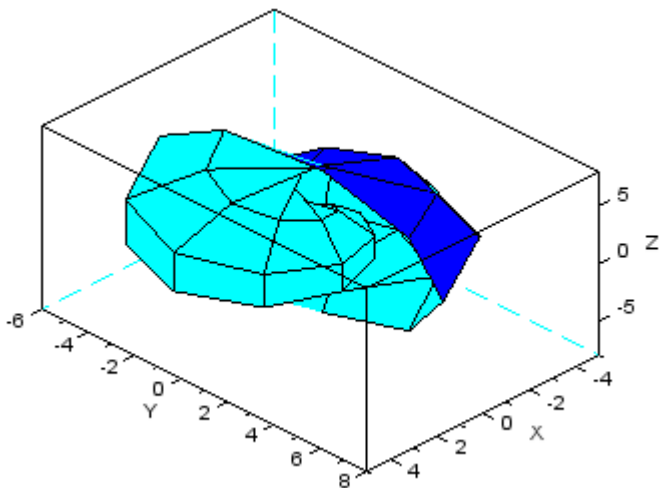


Figure 5.3 – Function graph plot using the **eval3dp** command.

After forming the grid, you can display the graph in it using the **surf** or **mesh** functions. As with the **plot3d** and **plot3d1** functions, **surf** plots the surface by filling each cell with a color that depends on the specific value of the function at the grid node, and **mesh** fills it with one color.

Thus, **mesh** function is a complete analog of the **surf** function with the value of the parameters **Color mode = index** of white color in the current color palette and **Color flag=0**.

The call to function looks like:

```
surf([X,Y],Z,[color,keyn=valuen])
mesh([X,Y],Z,[color,])
```

here: \mathbf{X}, \mathbf{Y} - arrays defining a rectangular grid;

\mathbf{Z} - matrix of function values;

color - a matrix of real numbers that set the color for each knot of the network;

keyn=valuen - a sequence of values of the graph properties **key1=value1, key2=value2, ..., keyn=valuen**, which determine its appearance.

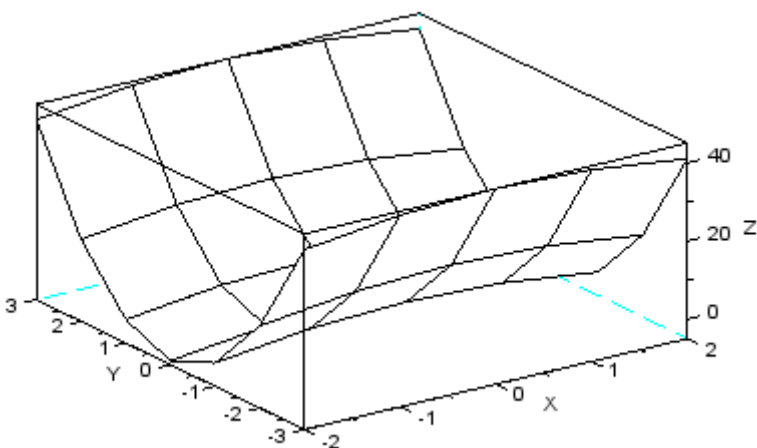


Figure 5.4 - Function graph plot with help of the **mesh** command.

Of course, in the event that the rectangular grid was built with the **meshgrid** command, there is no need to specify the \mathbf{X}, \mathbf{Y} parameters. In the simplest case, the **surf** function can be accessed like this - **surf(z)**.

To get acquainted with the functions **meshgrid** and **mesh**, we will plot a graph of dependence $Z = 5y^2 - x^2$ shown in fig. 5.4, the program for this graph will look like:

```
[x y]=meshgrid(-2:2,-3:3)
```

```

z=5*y.^2-x.^2;
mesh(x,y,z)

```

As you can see from figure 5.4, the grid, built with a step of 1, is too rare, and the calculated values of the function at the nodes are not enough to display a smooth graph.

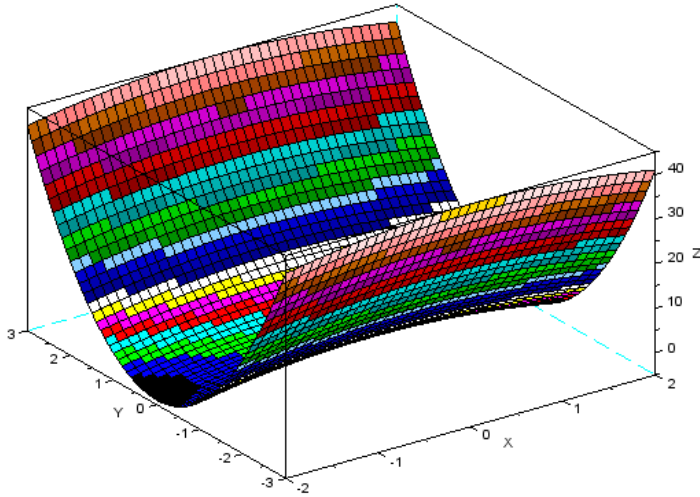


Figure 5.5 – Function graph plotted by the **surf** command.

Therefore, it is often better to independently specify the step of forming a rectangular grid when calling the **meshgrid** command. To do this, let's plot the same graph using the **meshgrid** command with a rectangular grid, specifying a step of 0.1, through which lines parallel to both coordinate axes will be drawn. In this case, the grid will be denser and the graph smoother than in the previous example. Next, we compute the values of the Z function and call **surf** to plot the surface:

```

[x y]=meshgrid(-2:0.1:2,-3:0.1:3);
z=5*y.^2-x.^2;
surf(x,y,z)

```

We will get the graph which is shown in figure 5.5. At first view, it may seem that for the function $Z = 5y^2 - x^2$ **plot3d1** and **surf** have plotted different surfaces (see fig. 5.2 and fig. 5.5). However, it is not true. The difference is due to mode «Cube scaling:». If you disable it, **surf** will display an image identical to that which was plotted using the **plot3d** function (figure 5.6).

In Scilab, you can plot two surfaces in the same coordinate system, for this, as for two-dimensional plots, use **mtlb_hold('on')** command, which blocks the creation of a new graphic window when the **surf** or **mesh** commands are executed.

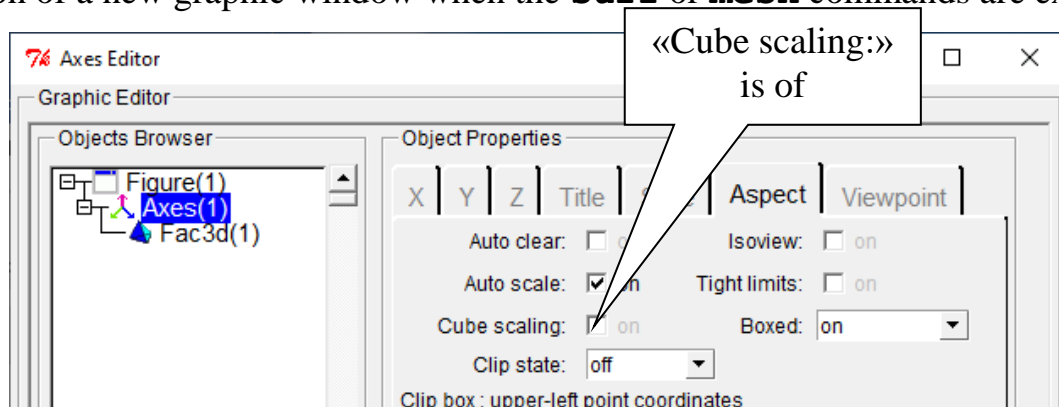


Figure 5.6 – «Cube scaling:» mode of the Axes Editor format window.

To illustrate this possibility, let's plot a function graph. Create a dense rectangular grid using the **meshgrid** command. Let $z(x,y) = (3x^2+4y^2) - 1$ and $z_1(x,y) = -(3x^2+4y^2) - 1$. Calculate the values of the functions at all grid nodes. The surface $z(x,y) = (3x^2+4y^2) - 1$ is constructed using the **surf** command. Next, we call the **mtlb_hold('on')** command, which will block the creation of a new graphic window, and using the **mesh** command, plot a surface $z_1(x,y) = -(3x^2+4y^2) - 1$ in the same coordinate axes with $z(x,y) = (3x^2+4y^2) - 1$ (fig. 5.7) this will display a rectangular grid. The program will look like:

```
[x y]=meshgrid(-2:0.2:2,-2:0.2:2);
```

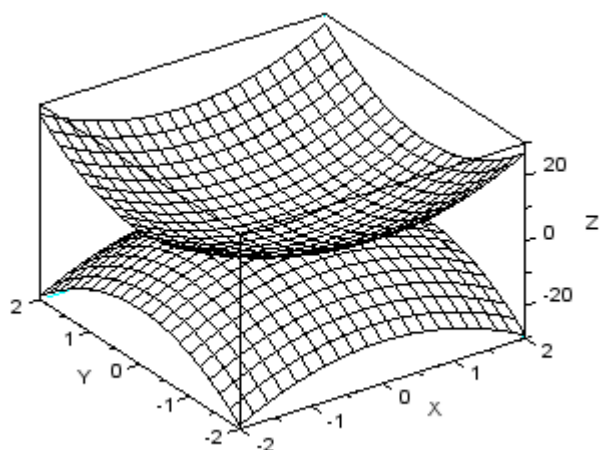


Figure 5.7 – A 3D graph of the function $z(x,y) = (3x^2+4y^2) - 1$

```
z=3*x.^2+4*y.^2-1;
```

```
z1=-3*x.^2-4*y.^2-1;
```

```
mesh(x,y,z);
```

```
mtlb_hold('on');
```

```
mesh(x,y,z1);
```

5.3 **plot3d2** и **plot3d3** functions.

plot3d2 and **plot3d3** functions are analogous to the **plot3** function, so they have the same syntax:

```
plot3d2(x,y,z,[theta,alpha,leg,flag,ebox][keyn=valuen]),  
plot3d3(x,y,z,[theta,alpha,leg,flag,ebox][keyn=valuen])
```

These functions are designed to plot a surface that is defined by a set of faces. That is, if the **plot3d** function, based on the input data, is able to plot only flat faces that are separately standing from each other, then **plot3d2** (**plot3d3**) will interpret the relative position of these faces in the form of a solid geometric body.

The difference between the functions **plot3d2** and **plot3d3** is similar to the difference between the actions of the functions **plot3d** and **plot3d1**, as well as **surf** and **mesh**.

The **plot3d2** function plots a surface, displays a grid and fills all cells with one of the colors, blue by default. The **plot3d3** function also displays the grid, but leaves all cells unfilled (that is white).

For example, let's plot a sphere (fig.5.8) using the **plot3d2** function whose equation is presented in the form:

$$\begin{cases} x(u,v) = \cos(u) \cdot \cos(v), \\ y(u,v) = \cos(u) \cdot \sin(v), \\ z(u,v) = \sin(u). \end{cases}$$

When plotting graphs of surfaces specified parametrically - $x(u,v)$, $y(u,v)$ and $z(u,v)$ – it is necessary to form matrices X , Y and Z of the same size. For this, the arrays u and v must be the same size. After that, two main types of representation x , y and z should be distinguished in the case of parametric definition of surfaces:

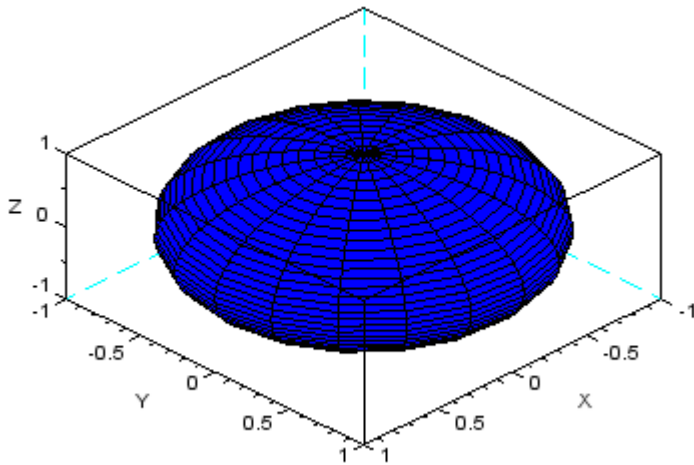


Figure 5.8 – A graph of the sphere which was plotted with the **plot3d2** function.

1. If x , y and z can be represented in the form $f(u) \cdot g(v)$, then the corresponding matrices X , Y and Z should be formed in the form of matrix multiplication $f(u)$ by $g(v)$.

2. If x , y and z can be represented in the form $f(u)$ or $g(v)$, then in this case the matrices X , Y and Z should be written in the

$f(u) \cdot \text{ones}(\text{size}(v))$ or $g(v) \cdot \text{ones}(\text{size}(u))$ respectively.

We define the arrays x , y and z using the **linspace** function, then the program will look like this:

```
u = linspace(-%pi/2,%pi/2,40);
v = linspace(0,2*%pi,20);
X = cos(u)'*cos(v);
Y = cos(u)'*sin(v);
Z = sin(u)'*ones(v);
plot3d2(X,Y,Z);
```

The sphere plotted by the **plot3d2** function is shown in Figure 5.8. Now let's see how the **plot3d** function will perform the same task.

Let's define the parameters u and v , then calculate the values of the functions x , y and z , as in the previous example. However, to plot the graph, let's call the **plot3d** function. We get an image (fig.5.9, a) and the program corresponding to it:

```
u = linspace(-%pi/2,%pi/2,40);
v = linspace(0,2*%pi,20);
X = cos(u)'*cos(v);
Y = cos(u)'*sin(v);
```

```

Z = sin(u)'*ones(v);
plot3d(X,Y,Z);

```

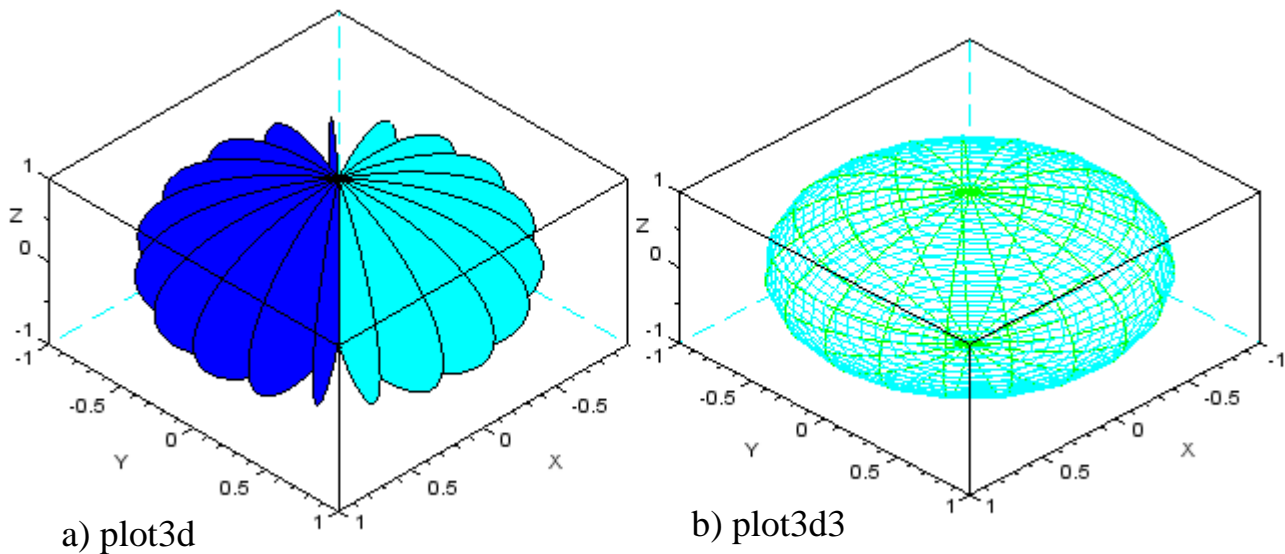


Figure 5.9 – Plots of the spherewhich was plotted by the functions **plot3d** and **plot3d3**.

Let's illustrate the action of the **plot3d3** function using the same example. We leave everything as in the previous example for building a graph (fig.5.9, b), the program will look like this:

```

u = linspace(-%pi/2,%pi/2,40);
v = linspace(0,2*%pi,20);
X = cos(u)'*cos(v);
Y = cos(u)'*sin(v);
Z = sin(u)'*ones(v);
plot3d3(X,Y,Z);

```

Questions for self-examination for the ninth lecture:

1. What are the functions *plot3d* and *plot3d1* used for?
2. What parameters do the functions *plot3d* and *plot3d1* have and what are they used for?
3. What are the *genfac3d* and *eval3dp* commands used for?
4. What is the *linspace* function used for?
5. What are *meshgrid*, *surf*, and *mesh* functions for?
6. What is the difference between the functions *plot3d2* and *plot3d3*?

Lecture 10

*The purpose of the lecture is to learn how to plot three-dimensional graphs using the *param3d* and *param3d1* functions, explore the capabilities of the *contour* and *contourf* functions and how to use them, learn how to plot three-dimensional*

histograms.

5.4 `param3d` and `param3d1` commands.

To plot a parametric curve in Scilab, there is the `param3d` command.

The `param3d` command is used to plot three dimensional lines defined by their x, y, z coordinate. The command looks like:

```
param3d(x,y,z,[theta,alpha,leg,flag,ebox])
```

here: $\mathbf{x}, \mathbf{y}, \mathbf{z}$ are three vectors of the same size that define the points of the parametric curve;

$\mathbf{theta}, \mathbf{alpha}$ are integers defining the values of the viewing angles in a spherical coordinate system, by default they are 35 and 45 degrees;

\mathbf{leg} is a string that defines the label of each axis, with the @ sign as a field separator, for example "X @ Y @ Z".

Let's illustrate the capabilities of the `param3d` function with the following examples.

Let's plot a line, specified parametrically:

$$\begin{cases} y = \sin(t), \\ y_1 = \cos(t), \\ y_2 = t/7. \end{cases}$$

First, we define the range and step of the parameter t . Then we turn to the `param3d` command, passing to it the values of the vectors $\mathbf{x}, \mathbf{y}, \mathbf{z}$ expressed in the

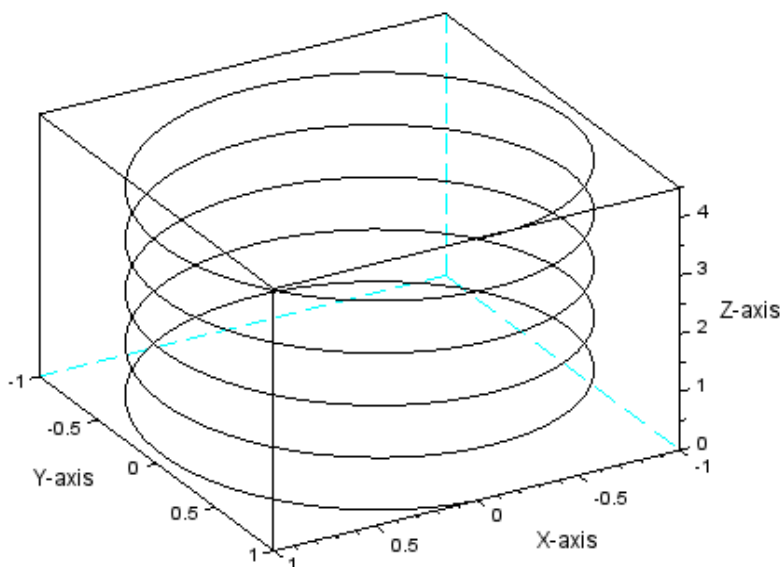


Figure 5.10 - Graph of a parametric line, plot

form of mathematical functions y, y_1 and y_2 , as well as the angles in degrees at which the observer will see the generated graph equal to 60 and 30 degrees. We will sign the axes in the form: "X-axis", "Y-axis" and "Z-axis". The graph is shown in fig. 5.10, and the program will look like:

```
t=[0:0.1:10*%pi];  
param3d(sin(t),cos(t),t/7,60,30,'X-axis@Y-axis@Z-axis');
```

As a second example of using the `param3d` command, let's plot a

parametrically defined line:

$$\begin{cases} x = t \cdot \sin(t), \\ y = t \cdot \cos(t), \\ z = \frac{(t \cdot |t|)}{(50 \cdot \pi)}. \end{cases}$$

Have determined the array of values of the parameter t , we will calculate the values of x, y, z coordinate of the curve. To plot the graph, we use the **param3d** command, setting the viewing angles to 45 and 60 degrees. The graph is shown in fig. 5.11, and the program will look like:

```
t=-50*%pi:0.1:50*%pi;  
x=t.*sin(t);  
y=t.*cos(t);  
z=t.*abs(t)/(50*%pi);  
param3d(x,y,z,45,60,'X-axis@Y-axis@Z-axis');
```

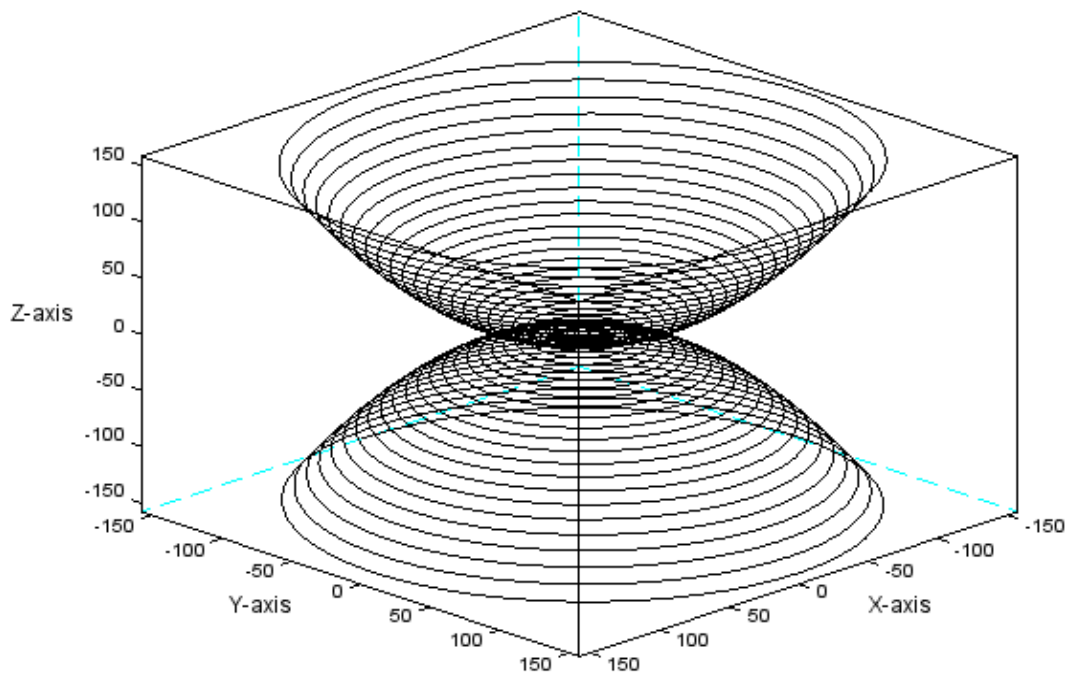


Figure 5.11 – Graph of a parametric line, plot by the **param3d** command.

To display several parametrically defined curves in the same coordinate in Scilab, use the **param3d1** command. It has a slightly different syntax.

```
param3d1(x,y,list(z,colors),[theta,alpha,leg,flag,ebox])
```

Here it becomes necessary to use the **list(z,colors)** construction, which allows not only to set the Z-coordinate for each of the curves, but also to set the

desired color for them. Let's look at an example. It is necessary to plot graphs of two parametrically defined lines:

$$\left\{ \begin{array}{l} x = \sin(t), \\ y = \sin(2t), \text{ and} \\ z = t/10. \end{array} \right. \quad \left\{ \begin{array}{l} x = \cos(t), \\ y = \cos(2t), \\ z = \sin(t). \end{array} \right.$$

Let's set an array of values of the t parameter. To plot lines in one coordinate system, let's call the **param3d1** command. As

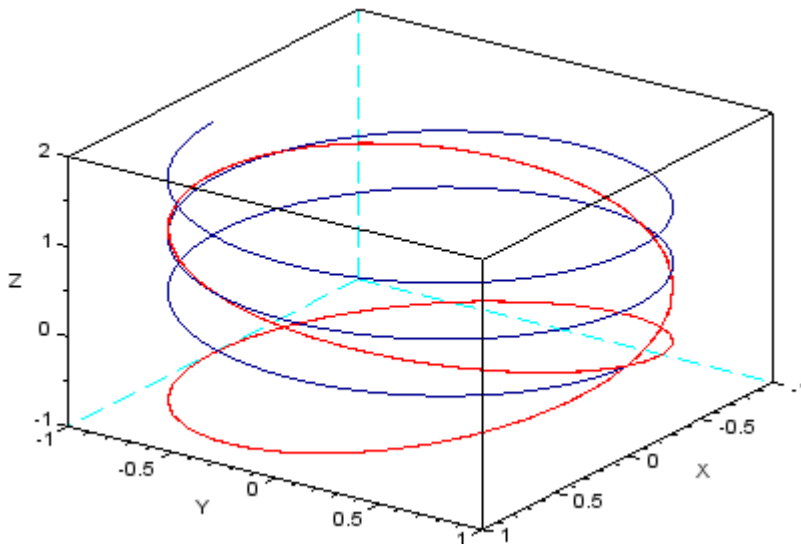


Figure 5.12 - Graphs of curves plotted by the **param3d1** command.

parameters in the first square brackets, we pass it the X and Y coordinate of the first curve, and in the second square brackets - of the second curve. Using the **list** property, define the Z-coordinate and set the line color to dark blue (9) for the first curve, and red for the second (5). Numbers 35 and 45 viewing angles. The '**X@Y@Z**' parameters are responsible for displaying

the labels of the graph axes. The graph is shown in fig. 5.12, and the program will look like:

```
t=[0:0.1:5*pi]';
param3d1([sin(t),sin(2*t)],[cos(t),cos(2*t)],...
list([t/10,sin(t)],[9,5]),35,45,'X@Y@Z');
```

5.5 **contour** function.

Scilab, in addition to plotting volumetric graphs, also implements the ability to create spatial models of objects. In practice, it is often necessary to build maps in the isolines of the indicator values, where X, Y – coordinates set the position of a particular point under study on the plane, and Z is the coordinate of the fixed value of this point height. Points with the same height values are connected by so-called isolines. These are lines of the same levels of values of the investigated quantity.

To plot contour lines (isolines) in Scilab there is a **contour** function. The appeal to it looks like:

```
contour(x,y,z,nz[theta,alpha,leg,flag,ebox,zlev])
```

here: **x**, **y** are arrays of real numbers;

z is matrix of real numbers of values of the function describing the surface $Z(x,y)$;

nz is a parameter that sets the number of contour lines. If **nz** is an integer, then **nz**'s isolines will be drawn at equal intervals in the range between the minimum and maximum values of the function $Z(x,y)$. If **nz** is set as an array, then isolines will be drawn through all the values specified in the array;

theta, **alpha** are real numbers which define the spherical coordinate of the viewing angle in degrees;

leg are labels of the graph coordinate axes, symbols separated by the @ sign. For example, 'X@Y@Z';

flag is an array consisting of three integer parameters [**mode**, **type**, **box**], here:

mode - sets the method and place of drawing the level lines (Table 5.4);

type – allows you to control the graph scale (table 5.2), by default it has a value of 2;

box – defines the presence of a frame around the displayed graph (Table 5.3). The default is 4;

ebox –defines the boundaries of the area into which the surface will be displayed, as a vector [**xmin**, **xmax**, **ymin**, **ymax**, **zmin**, **zmax**]. This parameter can only be used if the parameter **type=1**;

zlev is a mathematical expression that specifies a plan (horizontal projection of a given surface) for creating contours. By default, it is the same as the equation describing the plane and in this case can be omitted.

It should be noted that it is more convenient to pass the equation of the surface $Z(x,y)$ to the **contour** function as a parameter as a user-defined function. Recall that functions in Scilab are created using the **deff** or **function** command (see point 2.8.2).

Table 5.4 – **mode** parameter values.

Value	Description
0	Isolines applied to the surface $Z = (x,y)$.
1	Isolines are plotted on the surface and on the plan, which is given by the equation $z=zlev$
2	Isolines are plotted on a 2D graph

As an example of using the **contour** function, plot the level lines of the surface $Z = x \cdot \sin(x)^2 \cdot \cos(y)$.

Let's introduce the parameter t and define an array of its values. Using the **function** command, we will plot a function **my_surface** with inputs x , y and output z parameters. In the body of the function, we calculate the values of the

mathematical expression that defines the surface.

To plot contour lines, let's turn to the **contour** function, composing the appropriate program:

```
t=linspace(-%pi,%pi,30);
function z=my_surface(x,y)
z=x*sin(x)^2*cos(y)
endfunction
contour(t,t,my_surface,10)
```

After starting the program, we will receive a graph of isolines which are shown in figure 5.12.

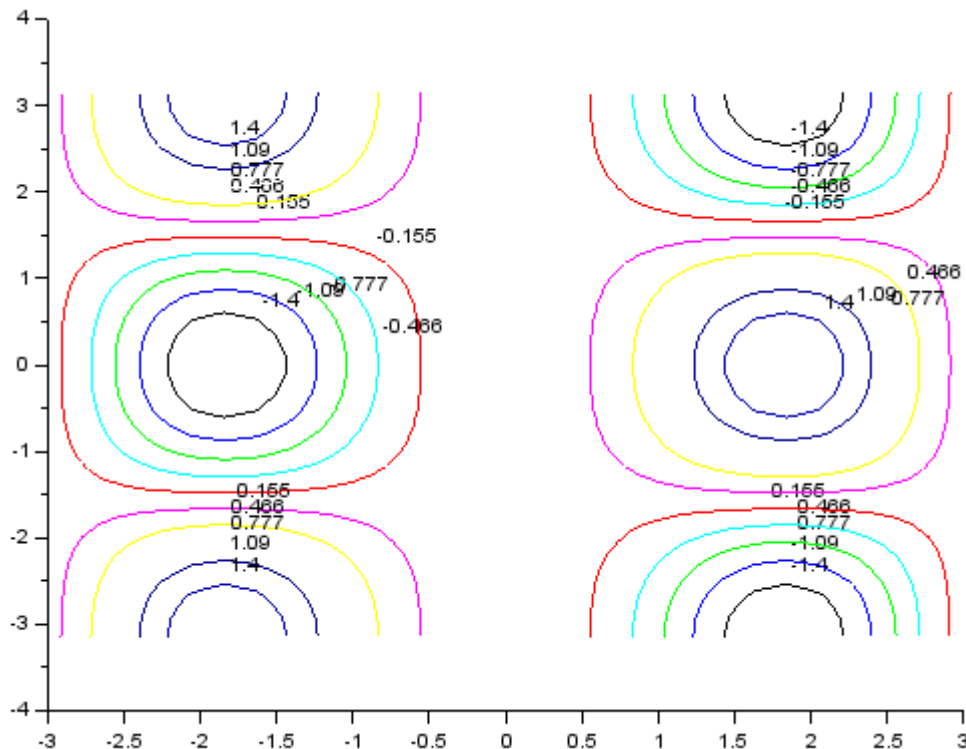


Figure 5.12 - Isolines of the surface $Z = x \cdot \sin(x)^2 \cdot \cos(y)$.

This example shows that the execution of the **contour** function results in the formation of lines of the same indicator values and their projection onto the horizontal plane. Obviously, such a presentation of data is not very informative. Much clearer is the image of surface contours and the surface itself in one graphic window. To illustrate this situation, consider an example of plotting a surface $Z = \sin(x) \cdot \cos(y)$ and displaying isolines in one graphic window (fig. 5.13).

The plotted graph is shown in figure 5.13, and the program looks like this:

```
t=%pi*(-10:10)/10;
deff(' [z]=Surf(x,y) ','z=sin(x)*cos(y) ');
rect=[-%pi,%pi,-%pi,%pi,-5,1];
z=feval(t,t,Surf);
plot3d(t,t,z,35,45,'X@Y@Z',[2,1,4],rect);
```

```

contour(t,t,z,10,35,45,'X@Y@Z',[1,1,4],rect,-5);
xlabel('plot3d and contour');

```

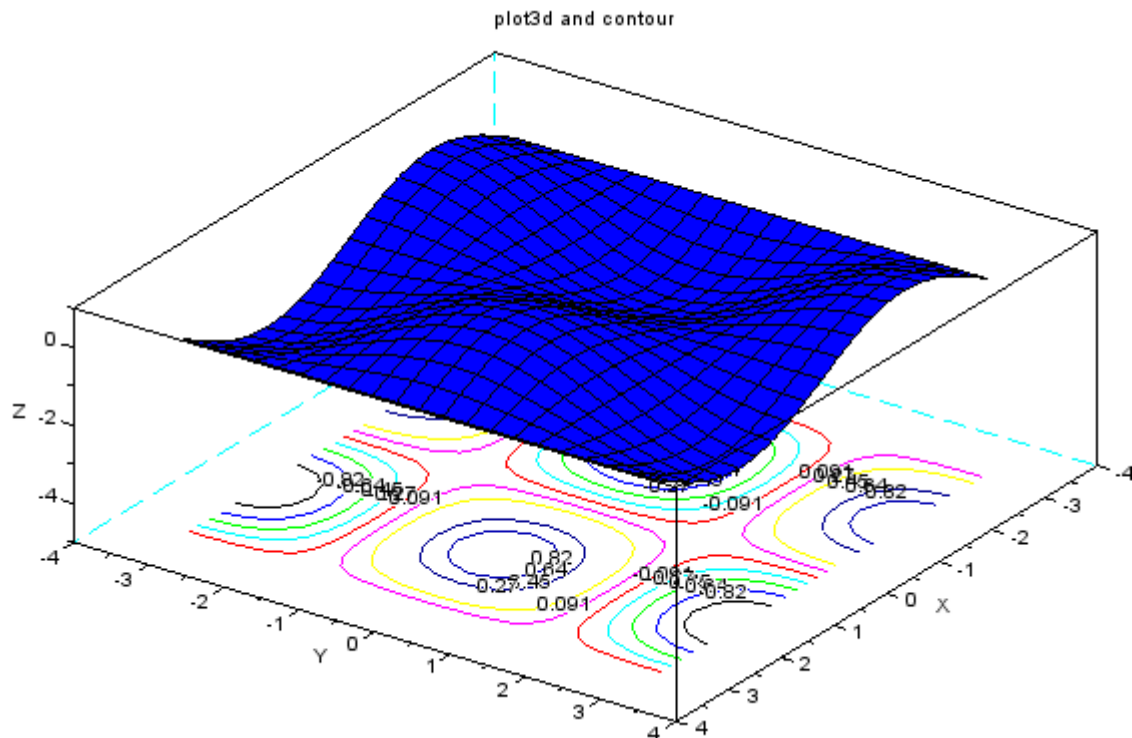


Figure 5.13 - Graph of the surface and its isolines in one window.

First of all, we introduce the parameter t and form an array of its values. Let's plot **Surf** function by calling the **def** command. Using the **rect** command, we set the boundaries of the plotting area in the graphic window in order to make it possible to match both the surface and the contours of the surface projected onto the horizontal plane.

Recall that when plot a graph of a function of the form $Z(x,y)$ using the **plot3d** function, it is necessary to use the cycle operator **For**, form a matrix of values of the function $z_{ij} = f(x_i, y_j)$. To avoid this, we will use the **feval** command.

Next, using the **plot3d** function, we plot a graph of the surface $Z = \sin(x) \cdot \cos(y)$, setting the viewing angles of the observer and labels for the coordinate axes. We also define the flag **[2,1,4]**: array: **2** – the color of the graph is blue, **1** – the boundaries of the plotting area are defined manually (further, the **rect** parameter specified above is indicated), **4** – all axes and a frame around the graph are displayed. Then we will form isolines, referring to the **contour** function, we also set the viewing angles, the labels of the coordinate axes the number of isolines will be equal and the values f the flag array is **[1,1,4]**: **1** – the mode of outputting isolines on a separately plotted plan, which is set by the same equation as the surface $Z = \sin(x) \cdot \cos(y)$; **1** – the boundaries of the plotting area are defined manually (the **rect** parameter specified above is specified below); **4** – all axes and a frame around the graph are displayed. The number **-5** sets the position of the horizontal plane with

isolines of 5 units below the surface plot. With help of **xtitle** command will be displayed the caption for the graph.

However, even such an image of the surface and its isolines is not always convenient. Let's try to combine the graph of the same surface $Z = \sin(x) \cdot \cos(y)$ with its level lines.

As in the previous example, let's set an array of values for the t parameter, create a **Surf** function, constrain the area for displaying the graph inside the graphic window using the **rect** command, and calculate the values of the function $Z = \sin(x) \cdot \cos(y)$, by executing the **feval** command.

When plotting the surface, we do't change all the parameters, except for the viewing angles of the observer (set 45 and 70) and the graph fill color (set the value of the mode parameter in the flag array to 19 - brown).

When calling the **contour** function to align the surface and its isolines, let delete the value of the location parameter , -5 and set for "mode" in the flag array to 0 ([0,1,4]) – contour lines are applied directly to the surface $Z = \sin(x) \cdot \cos(y)$. Use the **xtitle** command to display the caption for the graph.

With help of the **xtitle** command we displayed the caption for the graph.

The plotted graph is shown in Figure 5.14, and the program looks like this:

```
t=%pi*(-10:10)/10;
deff(' [z]=Surf(x,y) ','z=sin(x)*cos(y) ');
rect=[-%pi,%pi,-%pi,%pi,-1,1];
z=feval(t,t,Surf);
plot3d(t,t,z,45,45,'X@Y@Z',[-19,1,4],rect);
contour(t,t,z+0.1,10,45,70,'X@Y@Z',[0,1,4],rect);
xtitle('plot3d and contour');
```

5.6 **contourf** function.

Scilab has a **contourf** function that not only plots a surface on a horizontal plane as isolines, but also fills the intervals between them with color, depending on the specific level of values.

The function call looks like:

```
contourf (x,y,z,nz,[style,strf,leg,rect,nax])
```

here: **x,y** are arrays of real numbers;

z is the matrix of real numbers of the value of the function describing the surface $Z(x,y)$;

nz is a parameter that sets the number of contour lines. If **nz** is an integer, then nz isolines will be drawn at equal intervals in the range between the minimum and maximum values of the function $Z(x,y)$. If **nz** is set as an array, then isolines will be drawn through all the values specified in this array;

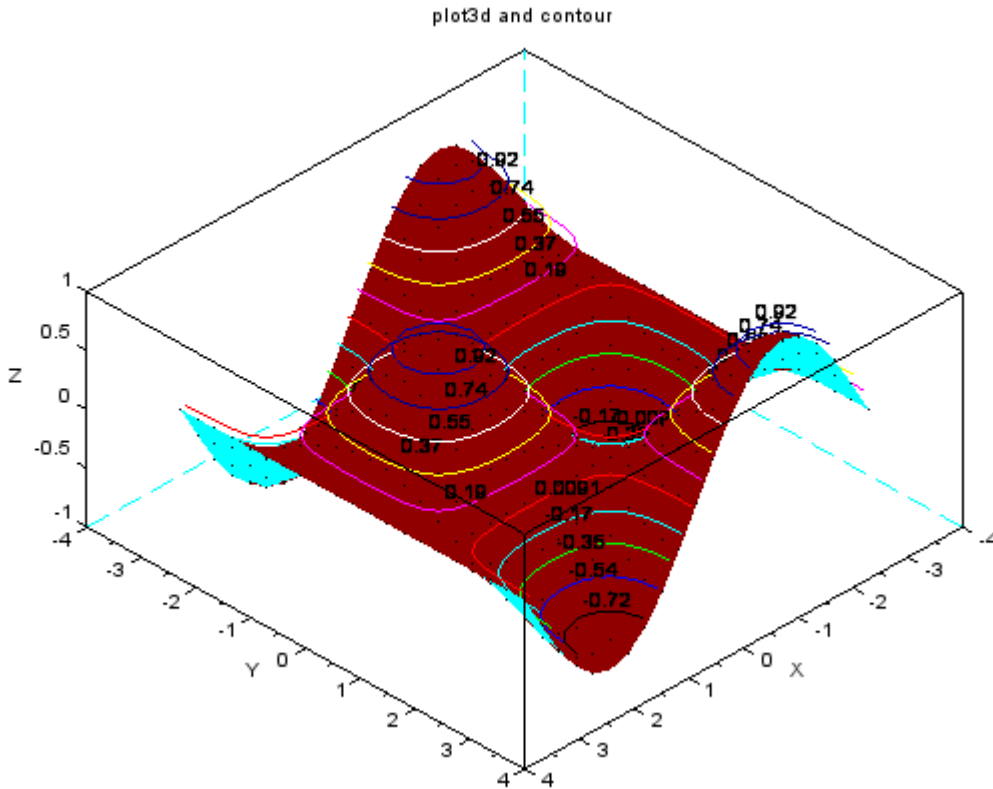


Figure 5.14 – Imposition of the isolines on surface.

style is an array of the same size as **nz**, it sets the color for each interval of value levels;

strf is a string consisting of three numbers «**csa**». Here **c** (Captions) sets the display mode of graph captions (see Table 5.5); **s** (Scaling) - scaling mode (see table 5.6); **a** (Axes) – defines the position of the graph axes (see Table 5.7);

leg is the legend of the graph, the signature of each of the curves, characters are separated by @. **leg** is the legend of the graph, the signature of each of the curves, characters are separated by @ sign. By default, the string is empty «»;

rect is vector [**xmin**, **ymin**, **xmax**, **ymax**], which defines the boundaries of the x and y coordinate of the graphic area of the window;

nax is an array of four values [**nx**, **Nx**, **ny**, **Ny**], defining the number of major and intermediate divisions of the graph coordinate axes. Here **Nx** (**Ny**) is the number of major ticks with labels under the X (Y) axis; **nx** (**ny**) is the number of intermediate ticks.

Table 5.5 – The value of the **c** parameter (Captions) of the **strf** string.

Value	Description
0	no signatures
1	the captions given by the leg parameter are displayed

Table 5.6 – The value of the **s** parameter (Scaling) of the **strf** string.

Value	Description
0	default scaling
1	set by the rect parameter
2	the scale depends on the minimum and maximum values of the input data
3	isometric axes are plotted based on the values of the rect parameter
4	isometric axes are displayed based on input data
5	expanding the axes for the best view based on the values of the rect parameter
6	expanding the axes for the best view based on the input data

Table 5.7 – The value of the **a** parameter (Axes) of the **strf** string.

Value	Description
0	no axes
1	the axes are displayed, the Y-axis is on the left
2	a frame is displayed around the graph without ticks
3	the axes are displayed, the Y-axis is on the right
4	the axes are centered in the graphic window
5	the axes are drawn in such a way that they intersect at the point (0; 0)

To get acquainted with the operation of the **contourf** function, we will plot an image of the surface $Z = \sin(x) \cdot \cos(y)$.

Let's introduce the t parameter and create an array of its values, define them using the **def** command and the **surf** function.

For clarity, we present a graph of the surface $Z = \sin(x) \cdot \cos(y)$, plot by the **plot3d1** function, and its image on the horizontal plane, formed by the **contourf** function in one graphics window.

For this purpose, let's use the **subplot** function, which will split the graphic window into two areas for displaying graphs.

Using **feval**, we calculate the values of the function $Z = \sin(x) \cdot \cos(y)$ and plot its graph using **plot3d1**, specifying the viewing angles of 80 and 15 degrees, and also, by calling the **xtitle** command, display the graph caption "plot3d1".

After that, we will form a projection of the surface onto a horizontal plane using the **contourf** function. As parameters, we pass X , Y and Z - coordinate, the number of isolines (**10**): **10:20** is an array defining the color of each interval between isolines, as well as the values of the string **strf="121"** (**1** – signature display mode; **2** – the choice of scale depends on the minimum and maximum values of the input data; **1** – the axes are displayed, the Y-axis is on the left).

For this graph, we will also display the labels of the axes and the graph as a whole (**'contourf', 'X', 'Y'**); using the **xtitle** command, the graph will take the form shown in fig. 5.15, and the program will be like this:

```

t=-%pi:0.2:%pi;
deff(' [z]=Surf(x,y) ','z=sin(x)*cos(y) ');
subplot(121);
z=feval(t,t,Surf);
plot3d1(t,t,z,80,15);
xlabel('plot3d1');
subplot(122);
contourf(t,t,z,10,10:20,strf="121");
xlabel('contourf','X','Y');

```

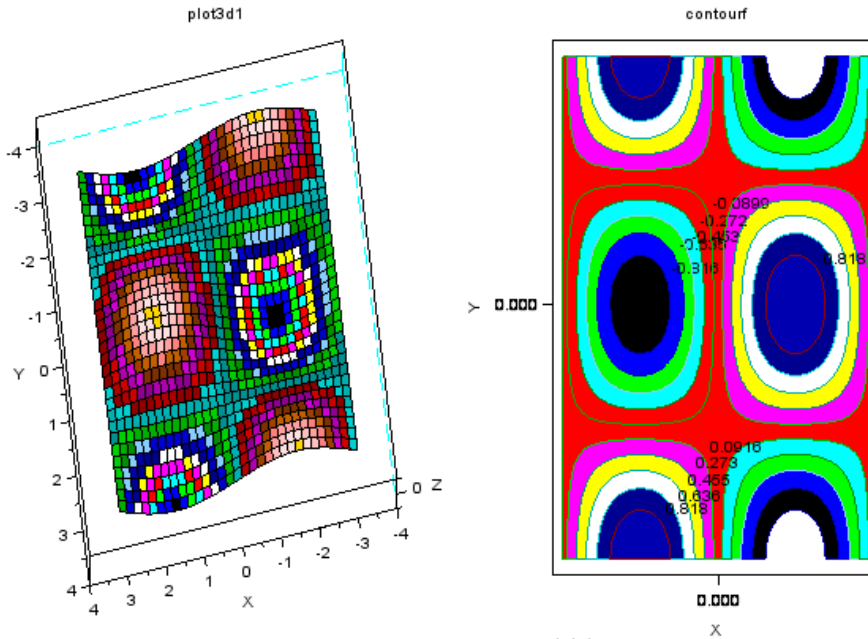


Figure 5.15 - The graphs plotting by the functions **plot3d1** and **contourf**.

5.7 **hist3d** function.

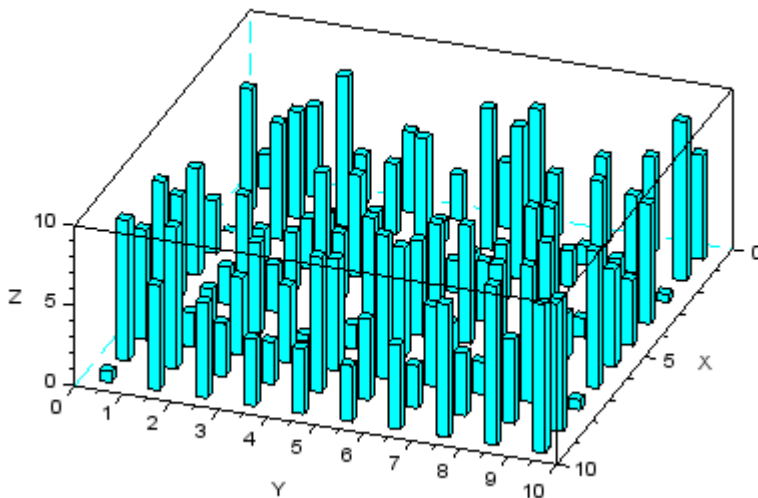


Рис. 5.8. Трехмерная гистограмма, построенная функцией

Scilab uses the **hist3d** function to plot 3D histograms:

```

hist3d(f, [theta
a, alpha, leg, flag, e
box])

```

here: **f** is ($m:n$), matrix defining the histogram of the function $f(i,j) = F(x(i),y(j))$;

theta, **alpha**, **le**

g, flag, ebox are parameters that control the same properties as for the `. plot3d` function.

To get acquainted with the work of the **hist3d** function, let's plot a graph of a three-dimensional histogram.

To form the input data matrix, we will use the **rand** command. Recall, to create a matrix of size (m, n), you need to use the construction **rand(m,n)**. The resulting graph is shown in fig. 5.16, and the program will look like this:

```
hist3d(10*rand(10,10),20,35);
```

Using this function, you can also plot a two-dimensional histogram:

```
hist3d(10*rand(1,10),0,90);
```

Questions for self-examination for the tenth lecture:

1. *What is the param3d function for?*
2. *What is the param3d1 function for?*
3. *What is the contour function for?*
4. *How to plot the isolines of the surface?*
5. *How do imposition of the isolines on surface?*
- 6.. *What is the contourf function for?*
7. *How to plot a 3D histogram?*

Lecture 11

The purpose of the lecture is to learn how to solve algebraic equations of any degree, transcendental equations and systems of equations and integrate functions which are given both in the form of an equation and in the form of a data set, how to integrate and calculate derivatives.

6 NONLINEAR EQUATIONS AND SYSTEMS IN SCILAB.

If a nonlinear equation is complex enough, then finding its roots is a very difficult task. Let's consider what tools Scilab has for solving this problem.

6.1 Algebraic equations.

Any equation of the form $P(x) = 0$ where $P(x)$ is a nonzero polynomial, is called an algebraic equation or polynomial. Any algebraic equation for x can be written in the form $a_0x^n + a_1x^{n-1} + \dots + a_{n-1}x + a_n = 0$, where $a_0 \neq 0$ $n \geq 1$ and a_i are the coefficients of an algebraic equation of the n -th degree. For example, a linear equation is an algebraic equation of the first degree, quadratic - second, cubic - third, and so on.

Algebraic equations are separated into a separate class for two reasons. First,

when working with such equations, you don't have to think about the domain of definition. The x argument can be any valid value. Second, polynomials have as many roots as the order of the polynomial, but some of the solutions can be complex.

Solving an algebraic equation in Scilab has two steps. First, you define the polynomial $P(x)$ using the **poly** function, and second – find its roots using the **roots** function.

So, the definition of polynomials in Scilab is carried out by the function:

```
poly(a, "x" ["f1"])
```

here: **a** is a number or a matrix of numbers;

x is a string, the name of a symbolic variable, if the string is more than 4 characters, then only the first 4 of them are taken into account;

f1 is an optional symbolic variable that defines how the polynomial is specified.

The symbolic variable **f1** can only have two values «**roots**» or «**coeff**» (**r** or **c**, respectively). If **f1 = c**, then a polynomial with the coefficients stored in the **a** parameter will be generated. If **f1 = r**, then the values of the parameters **a** are perceived by the function as roots, for which it is necessary to calculate the coefficients of the corresponding polynomial. By default **f1 = r**.

The following example shows the creation of a polynomial **p** with a three as a root and a polynomial **f** with a coefficient which is equal to three.

In the console window, type sequentially: `-->p=poly(3, 'x', 'r')` and `->f=poly(3, 'x', 'c')`, and accordingly we will get:

```
p =  
- 3 + x and
```

```
f =  
3
```

The below are examples of how to create more complex polynomials.

An example using the **poly** function:

```
//Polynomial with roots 1, 0 and 2  
poly([1 0 2], 'x')  
ans =  
          2    3  
2x - 3x + x
```

In this example, we got a polynomial of the form: $2x - 3x^2 + x^3$.

```
//Polynomial with coefficients 1, 0 and 2  
poly([1 0 2], 'x', 'c')
```

```
ans =
      2
    1 + 2x
```

In this example, we got a polynomial of the form: $1 + 2x^2$.

Let's consider some examples of symbolic operations with polynomials:

```
p1=poly([-1 2], 'x', 'c')
p1 =
    - 1 + 2x
p2=poly([3 -7 2], 'x', 'c')
p2 =
```

```
      2
    3 - 7x + 2x
p1+p2 //Addition
ans =
```

```
      2
    2 - 5x + 2x
p1-p2 //Subtraction
ans =
```

```
      2
    - 4 + 9x - 2x
p1*p2 //Multiplication
ans =
```

```
      2      3
    - 3 + 13x - 16x + 4x
p1/p2 //Division
ans =
```

```
      1
    -----
    - 3 + x
p1^2 //Exponentiation
ans =
```

```
      2
    1 - 4x + 4x
p2^(-1) //Negative exponentiation
ans =
```

```
      1
    -----
      2
    3 - 7x + 2x
```

The **roots(p)** function is designed to solve an algebraic equation. Here **p** is the polynomial created by the **poly** function and represents the left side of the equation $P(x) = 0$.

Let`s solve several algebraic equations.

First we will find the roots of the polynomial $2x^4 - 8x^3 + 8x^2 - 1 = 0$.

To solve this task, it is necessary to specify a polynomial **p**. We will do this using the **poly** function, having previously defined the vector of coefficients **V**.

Please note that the equation does not contain the variable **x** in the first degree, this means that the corresponding coefficient is equal to zero and that when we will be forming the vector of coefficients **V**, first the coefficient at x^0 is written, then at x^1 and the last is written the coefficient at x^n :

```
V=[-1 0 8 -8 2];  
p=poly(V, 'x', 'c')  
p =  
      2      3      4  
- 1 + 8x - 8x + 2x
```

Now let's find the roots of the polynomial:

```
X=roots(p)  
X =  
 2.306563  
 1.5411961  
- 0.3065630  
 0.4588039
```

To check the correctness of the solution of the task, let's plot a graph of the function $y=2x^4 - 8x^3 + 8x^2 - 1 = 0$ by running the program:

```
x=-1:0.1:3;  
y=2.*x.^4-8.*x.^3+8.*x.^2-1;  
plot2d(x,y)
```

In order for the X axis to cross zero in the «Axes Editor» window, set the value of the «Location» parameter equal to «origin», for the X axis, and set the value of the «Grid color» parameter to 0 to display the grid for both axes. The graphic solution of the task shown in fig. 6.1 allows you to make sure that the roots are found correctly.

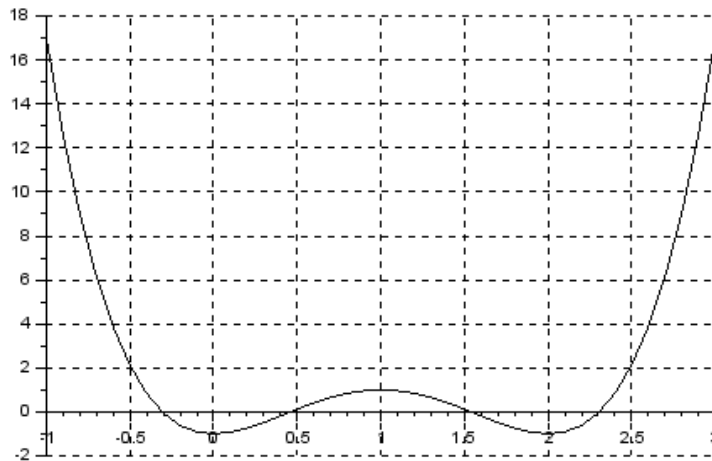


Figure 6.1 – Graphic solution of the equation $2x^4 - 8x^3 + 8x^2 - 1 = 0$.

Let's solve once more equation by finding the roots of the polynomial $x^3 + 0,4x^2 + 0,8x - 1 = 0$. The solution of this task is similar to the solution to the previous one, the difference lies in the way of calling the function which needs for this:

```
roots(poly([-1 0.8 0.4 1], 'x', 'c'))
ans =
    - 0.5319410 + 1.1060428i
    - 0.5319410 - 1.1060428i
    0.6638819
```

As you can see from the answer, the polynomial has one real and two complex roots. The graph of the function $y=x^3 + 0,4x^2 + 0,8x - 1$ is shown in fig. 6.2.

```
x=-1:0.1:1;
y=x.^3+0.4.*x.^2+0.8.*x-1;
plot2d(x,y)
```

Let us find another solution to the equation $x^4 - 18x^2 + 0,6 = 0$. The solution to this task is presented in a different way from the previous examples only in the way of determining the polynomial:

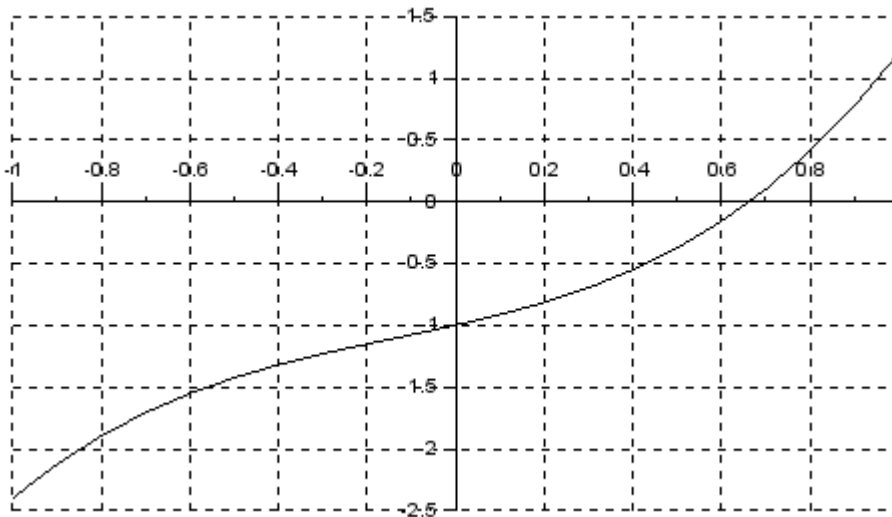


Figure 6.2 – Graphic solution of the equation $x^3 + 0,4x^2 + 0,8x - 1 = 0$.

```

x=poly(0,'x');
y=x.^4-18.*x.^2+.6;
roots(y)
ans =
  - 4.2387032
    4.2387032
  - 0.1827438
    0.1827438

```

As you can see from the answer, the polynomial has four roots. The graph of this function $y = x^4 - 18x^2 + 0,6$ is shown in fig. 6.3.

```

x=-6:0.1:6; y=x.^4-18.*x.^2+0.6;
plot2d(x,y)

```

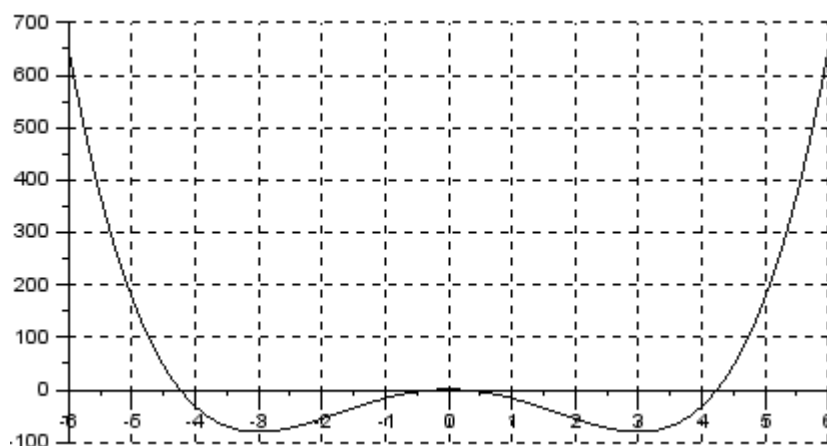


Figure 6.3 – Graphic solution of the equation $x^4 - 18x^2 + 0,6 = 0$.

6.2 Transcendental equations.

The equation $f(x) = 0$, in which the unknown is included in the argument of transcendental functions, is called a transcendental equation. Transcendental equations include exponential, logarithmic and trigonometric. In the general case, an analytical solution to the equation $f(x) = 0$ can be found only for a narrow class of functions. Most often, this equation has to be solved by numerical methods.

The numerical solution of the nonlinear equation is carried out in two stages. At the beginning, the roots of the equation are separated, that is find rather close intervals, which contain only one root. These intervals are called root isolation intervals and can be determined by plotting the function $f(x)$ or by any other method. Methods for determining the root isolation interval are based on the following property: if a continuous function $f(x)$ on the interval $[a, b]$ has changed sign, that is, $f(a) \cdot f(b) < 0$, then it has at least one root. At the second stage, the separated roots are clarified, or, in other words, the roots are found with a given accuracy.

To solve transcendental equations in Scilab use the function:

fsolve (x0 , f)

here: **x0** is initial approximation;

f is the function describing the left side of the equation $f(x) = 0$.

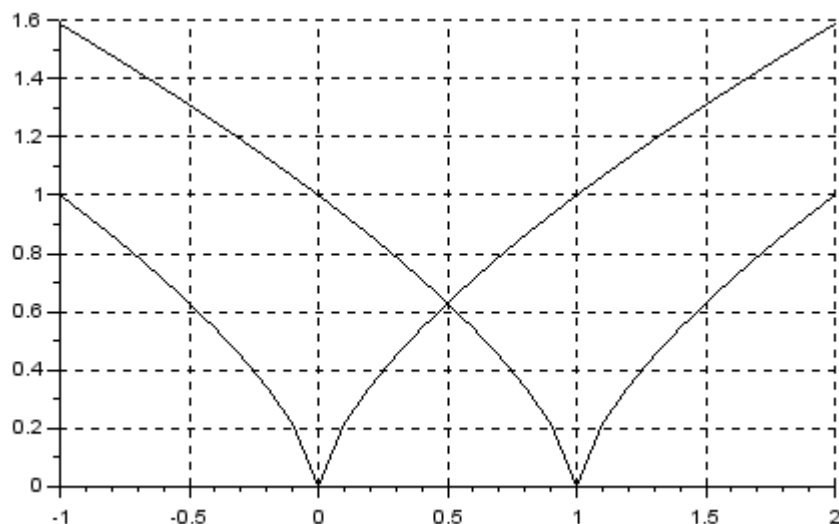
Let's look at the application of this function by examples.

Let's find a solution to the equation:

$$\sqrt[3]{(x-1)^2} - \sqrt[3]{x^2} = 0.$$

Let us determine the isolation interval of the root of the given equation. We will use the graphic method of separating the roots. If the expression on the right side of the equation is represented as the difference of two functions $f(x) - g(x) = 0$ then the abscissa of the point of intersection of the lines $f(x)$ and $g(x)$ is the root of this equation. In our case, this are $f(x) = ((x-1)^2)^{1/3}$ and $g(x) = (x^2)^{1/3}$. In fig. 6.4 it can be seen that the root of this equation lies in the interval $[0; 1]$.

Figure 6.4 – Graphic solution of the task.



We choose zero as the initial approximation, define a function describing the equation, and solve it:

```
deff(' [y]=f1(x) ', 'y1=((x-1)^2)^(1/3), y2=(x^2)^(1/3), y=y1-y2')
    fsolve(0, f1)
    ans =
        0.5000000
```

Let's solve one more equation $e^x/5 - 2(x - 1)^2 = 0$. The graphic solution is shown in fig.6.5. It shows that the graph of the function $f(x)$ crosses the abscissa axis three times, that is the equation has three roots.

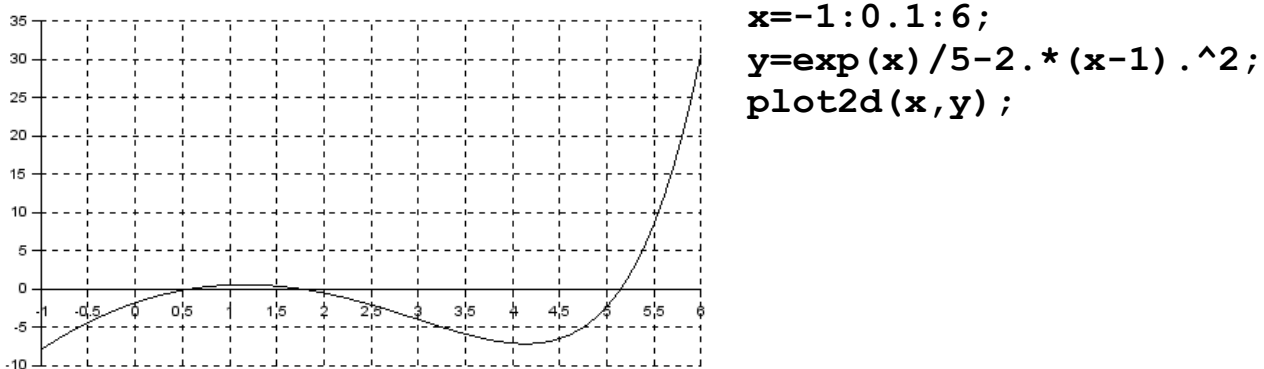


Figure 6.5 – Graphic solution of the equation $e^x/5 - 2(x - 1)^2 = 0$.

By sequentially calling the **fsolve** function with different initial approximations, we get all the solutions of the given equation:

```
clear
deff(' [y]=f(x) ', 'y=exp(x)/5-2*(x-1)^2')
x(1)=fsolve(0, f); x(2)=fsolve(2, f); x(3)=fsolve(5, f)
x =
    0.5778406
    1.7638701
    5.1476865
```

In addition, the initial approximations can be specified as a vector, and then the function can be called once:

```
deff(' [y]=f(x) ', 'y=exp(x)/5-2.*(x-1).^2')
x=fsolve([0;2;5], f)
x =
    0.5778406
    1.7638701
    5.1476865
```

Let's calculate the roots of the equation $\sin(x) - 0,4 = 0$ in the range $[-5\pi; 5\pi]$.

The equation graph is shown in fig. 6.6.

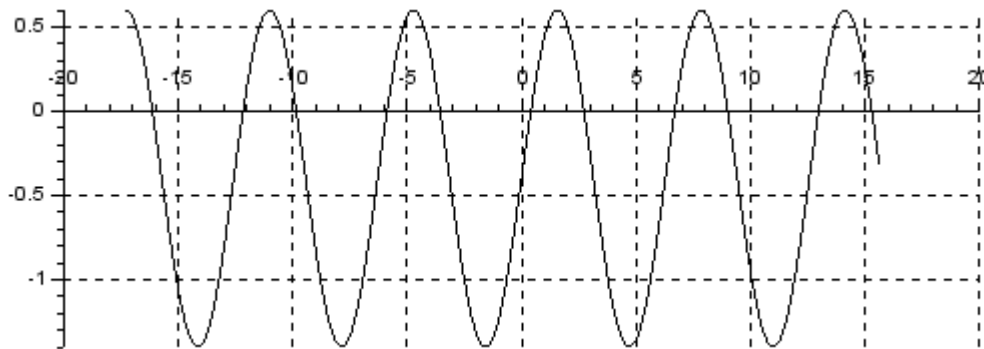


Figure 6.6 – Graphic solution of the equation $\sin(x) - 0,4 = 0$.

```
x=-5*pi:0.1:5*pi;
y=-0.4+sin(x)
plot(x,y)
```

And the program for solving the equation is:

```
def(' [y]=fff(x) ', 'y=-0.4+sin(x) ');
V=[-5*pi:pi:5*pi]; X=fsolve(V,fff)
x =
  - 16.11948   - 12.154854   - 9.8362948
  - 5.8716685   - 3.5531095     0.4115168
   2.7300758    6.6947022     9.0132611
  12.977887    15.296446
```

Let's find the solution to the transcendental equation $x^5 - x^3 + 1 = 0$ for which we will plot a graph, the program for its plotting has the form:

```
x=[-1.5:0.1:1.5];
y=x.^5-x.^3+1;
plot2d(x,y)
```

From the graph of the function $y = x^5 - x^3 + 1$ (fig. 6.7) it can be seen that the equation has one solution in the interval from $-1,5$ to -1

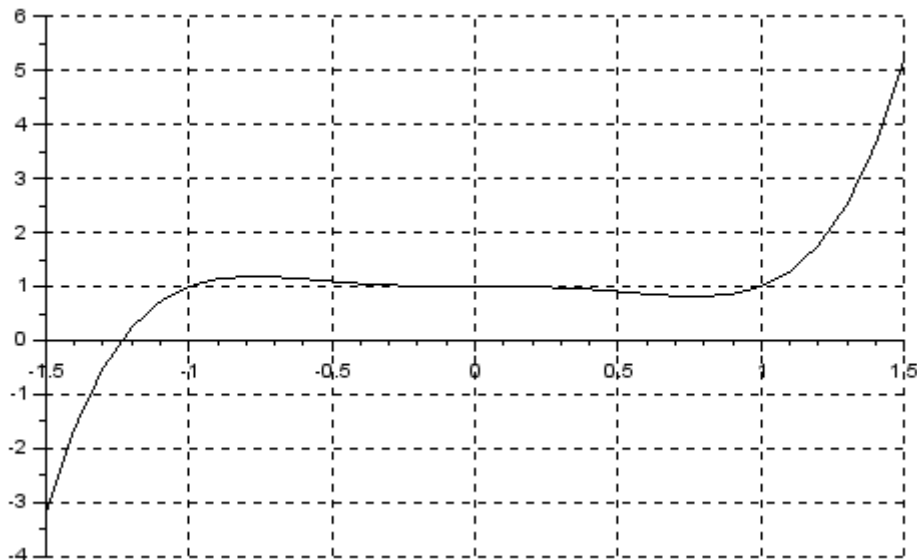


Figure 6.7 – Graphic solution of the equation $x^5 - x^3 + 1 = 0$.

Let's solve this problem using the **fsolve** function. Let's type the program in the SciNotes editor:

```
def f(' [f]=y(x) ', ' f=x.^5-x.^3+1 ')
X=fsolve(-1.5,y)
```

After starting the program in the console window, we get the answer:

```
X =
- 1.2365057
```

This could be limited, but the equation may have imaginary roots. Let's check this using the **roots** function by adding one more line at the end of the last program:

```
roots(poly([1 0 0 -1 0 1], 'x', 'c'))
```

From the answer it is clear that the transcendental equation has 5 roots, one of which is real and four are imaginary:

```
ans =
- 1.2365057
 0.9590477 + 0.4283660i
 0.9590477 - 0.4283660i
- 0.3407949 + 0.7854231i
- 0.3407949 - 0.7854231i
```

Thus, we can conclude that the way to solve transcendental equations using the **roots** function gives a more reliable result than using the **fsolve** function.

7 NUMERICAL INTEGRATION AND DIFFERENTIATION.

Various numerical algorithms are implemented in the integration and differentiation functions in Scilab.

7.1 Integration by the method of trapezoids.

In Scilab, numerical integration by the method of trapezoids is implemented using the `inttrap([x,]s)` function. This function calculates the area of a shape under the graph of the function $y(x)$, which is described by a set of points (x, y) .

Function call sequence:

$$\mathbf{v} = \text{inttrap}([\mathbf{x},] \mathbf{s})$$

here: \mathbf{x} is the data vector along the X coordinate in ascending order. The default is 1:

`size(s, '*')`;

\mathbf{s} is a vector of data along the Y coordinate.

\mathbf{v} is the value of the integral.

The function calculates: $s(i)=f(x(i))$, $x_0=x(1)$, and $x_1=x(n)$, here f is the function described by a set of experimental values. The function is interpolated linearly between grid points.

As an example of using the function, let calculate a definite integral:

$$\int_5^{13} \sqrt{2x-1} dx.$$

This integral can be easily reduced to a tabular one:

$$\int_5^{13} \sqrt{2x-1} dx = \frac{\sqrt{(2x-1)^3}}{3},$$

and is easily calculated using the Newton – Leibniz formula:

$$\int_a^b f(x) = F(b) - F(a).$$

The program for calculating the integral will look like:

```
a=5;b=13;
```

```
I=1/3*(2*b-1)^(3/2)-1/3*(2*a-1)^(3/2)
```

```
I =
```

```
32.666667
```

Now we will apply the method of trapezoids to find a given definite integral. In accordance with this method, to calculate the integral by method of trapezoids, the

integration section is divided into a certain number of equal segments, each of the obtained curvilinear trapezoids is replaced by a rectilinear one, and the approximate value of the integral is calculated as the sum of the areas of these trapezoids.

Let's consider several options for solving this problem. In the first case, the integration interval is divided into segments with a step of one, in the second 0.5 and in the third 0.1. It is easy to see that the more points of the partition, the more accurate the value of the required integral. The program for these variants is as follows:

```
a=5;b=13;
x=a:b;y=sqrt(2*x-1);
inttrap(x,y)
ans =
    32.655571

h=0.5; x=a:h:b; y=sqrt(2*x-1);
inttrap(x,y)
ans =
    32.66389

h=0.1; x=a:h:b; y=sqrt(2*x-1);
inttrap(x,y)
ans =
    32.666556
```

Below is an example of using the **inttrap** function with one argument. As you can see, in the first case, the value of the integral calculated using this function is inaccurate and coincides with the value obtained by the **inttrap(x,y)** function on the interval [5; 13] with step 1. That is, we found the sum of the areas of eight rectilinear trapezoids with base $h = 1$ and sides given by the vector y .

In the second case, when trying to increase the accuracy of integration, the value of the integral increases significantly. The fact is that, having reduced the step of dividing the integration interval to 0.1, we increased the number of elements of the vectors x and y , and the use of the **inttrap(y)** function will lead to the calculation of the sum of the areas of eighty trapezoids with base $h = 1$ and lateral sides given by the vector y . Thus, in the first and second examples, the areas of completely different figures are calculated.

```
a=5;b=13;
x=a:b; y=sqrt(2*x-1);
inttrap(y)
ans =
    32.655571

h=0.1; x=a:h:b; y=sqrt(2*x-1);
```



```

inttrap(y)
ans =
326.66556

```

Using the **inttrap** function, you can process experimental data. Suppose we need to determine the integral of some dependence, the formula for which is unknown, but the results of changing the values of the function from its parameter are known.

<i>x</i>	0.000	0.314	0.628	0.442	1.256	1.571	1.885	2.199	2.513	2.827	3.141
<i>y</i>	0.000	0.309	0.583	0.809	0.951	1.000	0.951	0.809	0.588	0.309	0.000

The program and the result are as follows:

```

x=[0.000 0.314 0.628 0.442 1.256 1.571...
1.885 2.199 2.513 2.827 3.141];
y=[0.000 0.309 0.583 0.809 0.951 1.000...
0.951 0.809 0.588 0.309 0.000];
inttrap(x,y)
ans = 2.0740015

```

7.2 Integration by quadrature.

The trapezoid methods are special cases of Newton – Cotes quadrature formulas, which have the form:

$$\int_a^b y dy = (b - a) \sum_{i=0}^n H_i y_i,$$

here: H_i are some constants called Newton – Cotes constants.

If for this integral we take $n = 1$, then we get the method of trapezoids, and for $n = 2$ – the Simpson method. These methods are called lower order quadrature methods. For $n > 2$, higher order Newton – Cotes quadrature formulas are obtained. The computational algorithm of quadrature formulas is implemented in Scilab by the function:

```

integrate(fun, x, a, b, [,er1 [,er2]])

```

here: **fun** is a function that sets the integrand in symbolic form;

x is variable of integration, also specified as a symbol;

a and **b** are limits of integration, real numbers;

er1 is real number (absolute error limit), default value $1e^{-8}$;

er2 is real number (relative error limit), default value: $1e^{-14}$.

Now calculate the integral:

$$\int_5^{13} \sqrt{2x-1} dx.$$

The program and answer will look like:

```
integrate(' (2*x-1)^0.5 ', 'x', 5, 13)
ans =
    32.666667
```

7.3 Integration of an external function.

The most universal integration command in Scilab is:

```
[I, err]=intg(a, b, name [,er1 [,er2]])
```

here: **name** is the name of the function that defines the integrand (here the function can be specified as a set of discrete points (like a table) or using an external function);

a and **b** are limits of integration;

er1 and **er2** - absolute and relative error of calculations (dispensable parameters).

Let's calculate the integral from the previous task, the solution will have the form:

```
deff('y=G(x)', 'y=sqrt(2*x-1)');
intg(5, 13, G)
ans =
    32.666667
```

Let's look at another example by calculating the integral:

$$\int_0^1 \frac{t^2}{\sqrt{3+\sin(t)}} dt.$$

The solution will look like:

```
function y=f(t)
    y=t^2/sqrt(3+sin(t))
endfunction;
[I, er]=intg(0, 1, f)
er =
    1.933D-15
I =
    0.1741192
```

7.4 Calculation of the derivative.

The operation of calculating the derivative of a function at a point, which is very common in mathematical analysis, is solved in Scilab using the command:

```
numderivative(f, x)
```

here **f** is the name of the function, **x** is the point at which the derivative is calculated.

For example, we will calculate the derivative $f'(x)$ of the function $f(x) = (x+3)^2+2x$ at the point $x = 2$.

The program and the result will look like:

```
def ('y=f(x)', 'y=(x+3).^2+2*x');  
pr_ot_y=numderivative(f,2)  
pr_ot_y =  
12.000000
```

To check the calculation of the derivative, we take the first derivative of the function $f(x)$, which will be equal to $f'(x) = 2(x+3)+2$. Substitute the value $x = 2$ into this equation and get the value $f'(x) = 12$, which coincides with the one found above.

When calculating a function of one variable at several points at once, the result of the **numderivative** command will be a matrix on the main diagonal of which the desired values are located.

Example. Calculate the derivative of the function $f'(x)=(x+3)^2+2x$ at the points [1; 1.8; 2.5]. To solve this task, add two lines to the program which was typed above and we will get the following result and immediately we will check the calculation of the derivative:

```
x0=[1 1.8 2.5];  
pr_ot_y=numderivative(f,x0)  
pr_ot_y =  
  
10.    0.    0.  
0.    11.6  0.  
0.    0.    13.  
  
// Check  
def ('y=f1(x)', 'y=2*(x+3)+2');  
f1(x0)  
ans =  
10.    11.6  13.
```

Although, to be honest, this check is useful only for educational purposes when you are mastering Scilab. When taking derivatives of complex functions, when taking a derivative is quite difficult, checking is completely unnecessary and even harmful, since it requires large intellectual costs.

7.5 Calculation of the partial derivative.

The **numderivative** command can handle partial derivatives easily as well. For example, let the function $f(x_1, x_2, x_3) = x_1^2 x_2 + x_2^2 x_3 + x_3^2 x_1$ be given. Calculate the partial derivatives $\partial f / \partial x_1$, $\partial f / \partial x_2$, $\partial f / \partial x_3$. The program will look like this:

```
function f=fCHP(x)
f=x(1).^2*x(2)...
+x(2).^2*x(3)...
+x(3).^2*x(1)
endfunction
x=[1 2 3];
numderivative(fCHP,x)

ans =
    13.    13.    10.
```

Questions for self-examination for the eleventh lecture:

1. What is called an algebraic equation and a polynomial?
2. What is the poly function for?
3. What is the roots(p) function used for?
4. What is called a transcendental equation?
5. What is the fsolve function for?
6. What is called a system of equations?
7. What is the intrtrap function for?
8. How can the experimental data be processed using the intrtrap function?
9. What is the integrate function for?
10. What is the intg command for?
11. What is the numderivative command for?
12. How to calculate the partial derivative?

Lecture 12

The purpose of the lecture is to learn how to write down the text of a program and subsequently launch it for execution, master the input-output functions, learn how to use the assignment operator, conditional operator, selection operator and the loop operator while.

8 PROGRAMMING IN Scilab.

Scilab has a powerful object-aware programming language built in. We will look at the possibilities of structured programming, then visual programming in the Scilab environment will be discussed.

As noted earlier, work in Scilab can be carried out not only in the command line mode, but also in the so-called program mode. Recall that to create a program (a program in Scilab is sometimes called a script) you need:

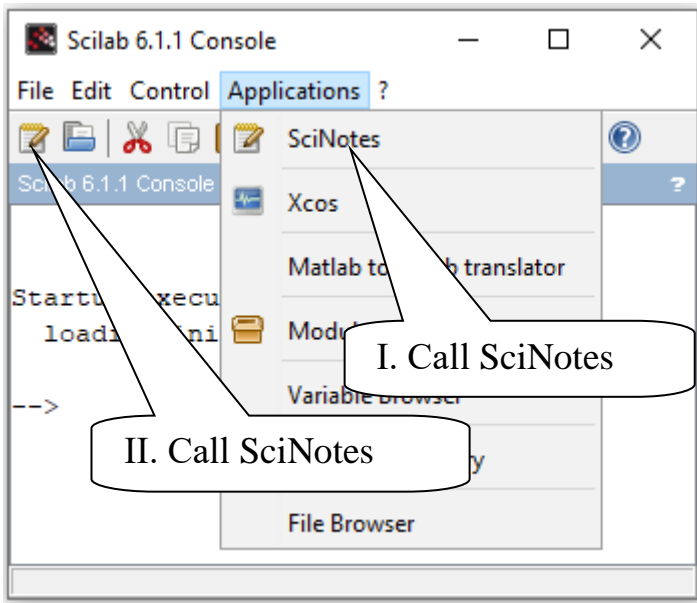



Figure 8.1 – To call the SciNotes.

1. To call the SciNotes text editor window. This can be done in two ways (fig. 8.1):

I - Select the "Applications" option in the main menu of the Scilab console and select the "SciNotes" option from the drop-down menu;
 II – On the Scilab console toolbar, click LMB on the icon .

2. In the SciNotes text editor window, to type the program text (fig. 8.2).

3. Save the text of the program, for which in the main menu of the SciNotes window select the option «File» and in the drop-down window select the option

«Save» or «Save as» and in the opened standard save window in its window «File name: » write the name of the file which will be saved, and in the «Files type:» window - the «sce» file extension.

4. After that, the program can be called by selecting the «File» option in the main menu of the SciNotes window and selecting the «Open» option in the drop-down window. Or select the «File» option in the main menu of the Scilab console and select the «Open a file» option in the drop-down window.

8.1 Basic operators of SciLab language.

8.1.1 Input and Output functions in Scilab.

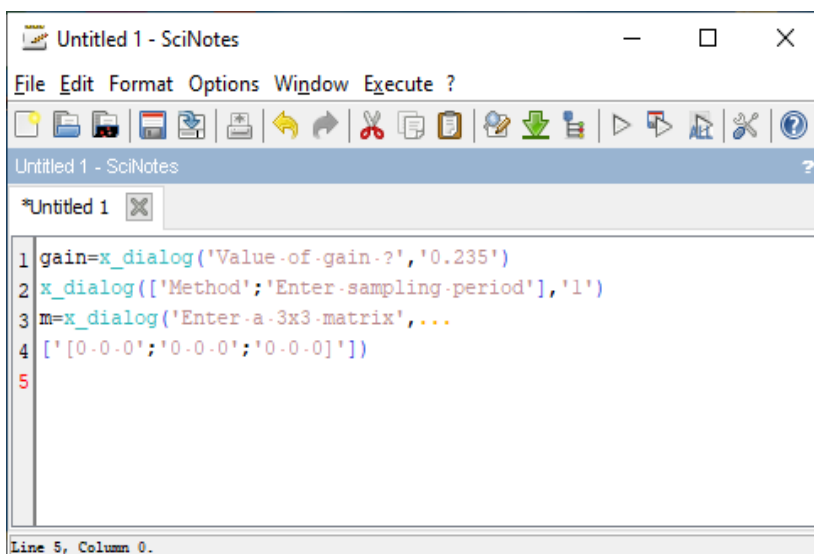


Figure 8.2 – SciNotes window with program

To organize the simplest input in Scilab, you can use the **x=input('title')** or **result=x_dialog(label s, val)** functions.

The **input** function is necessary if, during the execution of the program, you need to enter different values of variables, for example, depending on the results of previous calculations. The **input** function prints on the

command line Scilab prints out the hint title and waits until the user to enter a value,

which is returned as a result in the variable x .

For example: Let's enter the following string into the command line:

```
-->x=input("Number of iterations = ")
```

and press the [Enter] key. The following line appears in the console window: «**Number of iterations = |**» and after the «**=**» sign, will be a blinking cursor. If you now enter, for example, the number 3 and press the [Enter] key, then into the console window will be displayed:

```
x =  
3.
```

This value of the variable x can then be used in the further work of the program.

The function for entering a multi-string value is:

```
result=x_dialog(labels, val)
```

here: **labels** is dialog line;

val is a number, vector, or matrix which are entered using the keyboard in the pop-up dialog box;

result is a number, vector or matrix used in further calculations. If you click

the [OK] button in the pop-up window, the program will use the **val** value. If you click on the [Cancel] button, the function will return an empty set [] to the program.

Let's look at an example of how the **x_dialog** function works. Let the program contain three different commands. When it is launched, three windows will be displayed sequentially on the computer screen (fig. 8.3).

```
gain=x_dialog('Value of gain ?', '0.235')  
x_dialog(['Method'; 'Enter sampling period'], '1')  
m=x_dialog('Enter a 3x3 matrix', ...  
['[0 0 0'; '0 0 0'; '0 0 0']'])
```

The results of each of the three commands, respectively, will look like:

```
gain =  
"0.235"
```

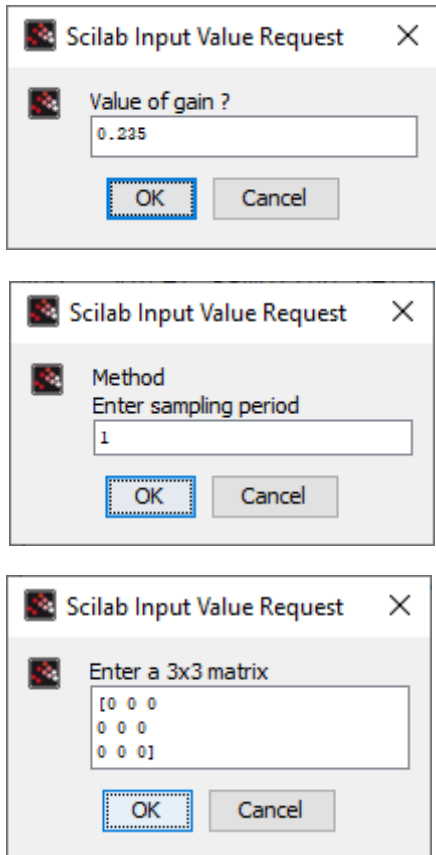


Figure 8.3 – Data entry windows.

```

ans =
    "1"
m =
    "[0 0 0"
    "0 0 0"
    "0 0 0]"

```

The **input** function converts the input value to a numeric data type, and the **x_dialog** function returns a string value. Therefore, when using the **x_dialog** function to enter numeric values, the returned string must be converted to a number using the **evstr** function. You can suggest the following form of using the **x_dialog** function to enter numeric values:

```
x = evstr(x_dialog(labels,val)).
```

If you enter a program like this:

```

m=evstr(x_dialog('Enter a 3x3 matrix',...
    '['[0.10 0.13 10.0';'150 230 450';'15 0.12 0]']),

```

then we will get a matrix which is consisting of numbers:

```

m =
    0.1      0.13     10.
    150.     230.     450.
    15.      0.12     0.

```

For text output into console window, you can use the **disp** function of the following structure:

```
disp(x1, [x2, ... xn])
```

here: **x1**, **[x2, ... xn]** is a number, vector or string.

For example:

```

-->disp([1 2],3)
    1.    2.
    3.
-->disp("a",1,"c")
"a"
    1.
    "c"

```

8.1.2 Assignment operator.

The assignment operator has the following structure:

```
a = b,
```

here **a** is the name of a variable or array element, **b** is a value or expression.

As a result of the execution of the assignment statement to the variable **a**, the value of the expression **b** is assigned.

8.1.3 Loop conditional operator **if**.

One of the main operators for organizing a loop and implementing branching is the loop conditional operator **if**. There are common and extended forms of the **if** operator in Scilab. The loop conditional operator **if** looks like:

```
if expr1 then  
statements 1  
else  
statements  
end
```

here: **expr1** is boolean expression;

statements 1, **statements** are Scilab statements or built-in functions;
else is a function word denoting the end of the action of the condition **expr1**;
end is a function word denoting the end of the loop conditional operator **if**.

The **if** operator works according to the following algorithm: if the **expr1** is equal to **true**, then the **statements 1** are executed, if it is **false**, the **statements** are executed.

Scilab can use next conditional operators to build logical expressions: **&** (boolean and), **|** (logical or), **~** (logical negation), and relational operators: **<** (less than), **>** (greater than), **==** (equal), **~=** or **<>** (not equal), **<=** (less than or equal), **>=** (more than or equal).

As a rule, when solving practical tasks, it is not enough of the choose when one condition is fulfilled or not. In this case, you can, create a new **if** operator on the **else** branch, but it is better to use the extended form of the **if** operator:

```
if expr1 then  
statements 1  
elseif expr1 then  
statements i  
....  
else
```



```
statements
end
```

In this case, the **elseif** check of the **expri** condition can be executed almost an unlimited number of times.

For an example of using the capabilities of the **if** operator, we will create a program for solving the binary quadratic equation $ax^4 + bx^2 + c = 0$.

To solve the biquadratic equation, it is necessary to reduce it to a quadratic equation by replacing $y = x^2$ and solve it. Then, to find the roots of the biquadratic equation, it will be necessary to extract the roots from the founded values of y . The input data for this task are the coefficients of the biquadratic equation a, b, c . The output is the roots of the equation x_1, x_2, x_3, x_4 , or the message that there are no valid roots.

The algorithm consists of the following steps:

1. Entering the coefficients of the equation a, b, c ;
2. Calculation of the discriminant of the equation d ;
3. If $d < 0$, then y_1 and y_2 are determined, otherwise the message «No roots» is displayed.
4. If $y_1 < 0$ and $y_2 < 0$, then the message «No roots» is displayed.
5. If $y_1 > 0$ and $y_2 > 0$, then four roots are calculated by the formulas $\pm (y_1)^{1/2}$, formulas $\pm (y_2)^{1/2}$, and the values of the roots are displayed.
6. If conditions 4) and 5) are not performed, then it is necessary to check the sign of y_1 .
7. If y_1 is non-negative, then two roots are calculated by the formula $\pm (y_1)^{1/2}$, otherwise both roots are calculated by the formula $\pm (y_2)^{1/2}$.

Open SciNotes and type the program:

```
// Entering the values of the coefficients
// of the biquadratic equation.
a=evstr(x_dialog('a=', '-6'))
b=evstr(x_dialog('b=', '9'))
c=evstr(x_dialog('c=', '-1'))
// Calculation of the discriminant.
d=b*b-4*a*c
///// If the discriminant is negative,
if d<0 then
// then output the message,
disp('Real roots are not present');
else
// otherwise, calculate the roots
//of the corresponding quadratic
//equation.
disp('Real roots are x1 and x2');
x1=(-b+d^(1/2))/2/a
x2=(-b-d^(1/2))/2/a
```

```

if (x1<0)&(x2<0) then
// display a message about the
//absence of valid roots.
disp('Real roots are not present');
else
if (x1>=0)&(x2>=0) then
    // calculating of the four roots.
disp('Four real roots');
y1=x1^(1/2)
y2=-y1
y3=x2^(1/2)
y4=-y3
disp(y1,y2,y3,y4);
// Otherwise, if both conditions
// (x1<0)&(x2<0)
// and (x1>=0)&(x2>=0)
// are not performed,
else
//then the message displayed
disp('Two real roots');
//Checking the sign of the x1.
    if x1>=0 then
// If x1 is positive, then the calculation
// of two roots of the biquadratic equation
// by extracting the root
y1=x1^(1/2);
y2=-y1;
disp(y1);
disp(y2);
// otherwise (there is only one option
// left - x2 is positive), calculating
//two roots of the biquadratic equation
// by extracting the root from x2 and from x2.
else
y1=x2^(1/2); y2=-y1;
disp(y1); disp(y2);
    end //if x1>=0 then
    end //if (x1<0)&(x2<0) then
    end // if (x1>=0)&(x2>=0) then
end //if d<0 then

```

Let's start the program for execution and select the results of entering the coefficients and the result of the solution from the listing of the program in the Scilab console.

```

a =
  - 6.
b =
  9.
c =
  - 1.
"Four real roots"
  0.3476307
 -0.3476307
  1.1743734
 -1.1743734

```

If this program is saved, then later, it can be repeatedly called by the `exec` command from the Scilab console window, specifying the path and name of the file to be called. For example like this:

```

exec('C:\Users\D\Desktop\Примеры\BQE1.sce');

```

It is possible to find all the roots of a biquadratic equation without an `if` operator, taking advantage of the fact that Scilab defines operations on complex numbers:

```

a=evstr(x_dialog('a=', '-6'));
b=evstr(x_dialog('b=', '9'));
c=evstr(x_dialog('c=', '-1'));
d=b*b-4*a*c;
x1=(-b+sqrt(d))/2/a;
x2=(-b-sqrt(d))/2/a;
y1=sqrt(x1);
y2=-y1;
y3=sqrt(x2);
y4=-y3;
disp(y1,y2,y3,y4)

```

The result of the program is as follows:

```

  0.3476307
 -0.3476307
  1.1743734
 -1.1743734

```

8.1.4 Alternative **select** operator.

Another way of organizing branching is the alternate **select** operator of the following structure:

```

select variable
case value1 then
instructions 1
case value2 then
instructions 2
...
case valuen then
instructions n
else
instructions
end

```

here: **variable** is a variable which value to be analyzed;

value1, **value2** are values of the **variable** for which the corresponding set of instructions **instructions 1**, ..., **instructions n** is provided;

instructions – block of valid instructions.

The **select** operator works as follows: if **variable** is equal to **value1**, then **instructions 1** are executed, if **variable** is equal to **value2**, then **instructions 2** are executed, and so on. If there is no match, then the instructions following the **else** are executed.

The only restriction is that each **then** keyword must be on the same line as the corresponding **case** keyword (three dots can be used for indication a continuation).

According to the Code Conventions for the Scilab Programming Language it is recommended:

- Start each statement on a new line.
- Write no more than one simple statement per line.
- Break compound statements over multiple lines.

Of course, any algorithm can be programmed without using **select** operator, using only **if** operator, but using the alternate **select** operator makes the program more compact.

Let's look at the use of the **select** operator using the example of solving the following task.

It is necessary to print the name of the day of the week corresponding to the given number D of the day, provided that there are 31 days in the month and the 1st day is Monday.

To solve the task, let's use the condition that the 1st day is Monday. If, as a result, the rest of dividing a given number D by seven is equal to one, then this is Monday, two is Tuesday, three is Wednesday, and so on. You can calculate the rest of dividing x by k using the formula $x - \text{int}(x/k) \cdot k$. Therefore, when constructing the algorithm, seven **select** operators must be used.

The solution of the task will become much easier if, when writing a program, you use the **select** operator:

```
D=evstr(x_dialog('Enter a number from 1 to 31','1'));
//Calculate the rest after dividing D
//by 7, comparing it with numbers from 0 to 6.
select D-int(D/7)*7;
case 1 then disp('Monday');
case 2 then disp('Tuesday');
case 3 then disp('Wednesday');
case 4 then disp('Thursday');
case 5 then disp('Friday');
case 6 then disp('Saturday');
else
disp('Sunday');
end
```

Let's save the text of the program, typed in SciNotes, for example, under the name "ДН.sce" and call the program for execution:

```
exec('C:\Users\D\Desktop\Примеры\ДН.sce');
```

In the pop-up window, type, for example, the number 23. In the console window, we get the answer:

```
-->exec('C:\Users\D\Desktop\Примеры\ДН.sce');
Tuesday
```

8.1.5 The **while** loop operator.

The **while** loop operator looks like this:

```
while expr
instructions
end
```

here: **while** is keyword defining the name of the cycle;

expr is a boolean expression, as long as it remains true, the loop will continue to execute;

instructions is a sequence of commands;

end is a keyword which is indicating the end of the loop.

Let's consider the work of the loop operator **while** using the following example:

```
clear
i=1;
while i<3
i = i + 1
end
i = i^2;
disp(i, 'End of cycle work');
```

After writing the program to a file and calling it for execution, we will get:

```
i =
  2.
i =
  3.
9.
" End of cycle work"
```

The loop operator **while** is very flexible, but not very convenient for organizing a sequence of actions that must be performed a given number of times.

Questions for self-examination for the twelfth lecture:

1. How can the SciNotes text editor window be invoked?
2. What is the input function for?
3. What is the `x_dialog` function for?
4. What is the `evstr` function for?
5. What is the difference between the `=` and `==` operators?
6. What is the `if` operator for?
7. What logical operators do you know?
8. What is the conditional select operator for?
9. What the conditional while operator is for.

Lecture 13

The purpose of the lecture is to learn how to use the for loop operator, functions for working with files and get the simplest skills in writing SciLab programs.

8.1.6 The non-conditional loop operator **for**.

The loop operator **for** looks like this:

```
for variable=expression [do]
```

```
instruction;  
instruction;  
...  
instruction;  
end
```

here: **variable** is the variable through which the loop is conducted;

expression is an expression describing the law of variation of a variable within the specified limits; can be a vector, matrix, list;

instruction is any valid Scilab instructions;

do is a keyword that separates the definition block of the loop variable from the block of instructions;

end is a keyword denoting the end of the loop.

It is used to organize loops on a given variable. The change of the loop variable **variable** is described by **expression**, however, **expression** can be a vector or a matrix. In this case, the variable **variable** sequentially from iteration to iteration takes the values of the elements of the vector or matrix from the first to the last, column by column.

As an example, consider the program:

```
for i = 1:5  
disp(i);  
end
```

As a result of executing the program, we will get:

```
1.  
2.  
3.  
4.  
5.
```

It should be taken in mind that the number of characters used to define the body of any condition statement (**if**, **while**, **for** or **select**) must be limited to 16K.

We have not considered all the operators of the control logic, but those that we have already considered are enough to create programs of significant complexity.

8.2 An example of working with arrays.

Control logic operators allow you to create almost any program for processing data or calculating various functions. Earlier, in section 3, we looked at many of the functions for working with arrays. Let's create another program, which is essentially a function for processing array elements. For example, when analyzing the obtained

experimental data, we found that one of the values is a slip, that is, a gross error and must be removed from the data array. Using the operators of the control logic, we will create a program that allows us to do this. To simplify the task, let us have a small data array that we enter into the program, defining its values programmatically. So we have an array x , consisting of n numbers, it is necessary to remove the element with the number m .

To do this, it is enough to write element number $(m + 1)$ in place of element with number m , $(m + 2)$ - in place $(m + 1)$, ... n - in place $(n - 1)$, and then delete the last n -th element. The program performing these actions will be looked like:

```
x=[3 4 4 5 9 3 5 4];
disp('x = ',x);
n=length(x);
//Enter the number of the element
//which will be deleted.
m=evstr(x_dialog('m=', '5'));
//Shift all elements, starting from
//the m-th one by one to the left.
for i=m:n-1
x(i)=x(i+1);
end;
// Removing the n-th element from an array.
x(:,n)=[];
//Decrease n by 1.
n=n-1;
//Outputting the transformed array.
disp('x(corrected) = ',x);
```

Let's write the program to a file, call it for execution. After launch, a record of the values of the x array will appear in the console window. Let's analyze its values and come to the conclusion that the fifth value is a miss. Let's enter the number 5 into the pop-up window and get a new corrected data array:

```
-->exec('C:\Users\D\Desktop\Примеры\ГрОш.sce');
"x = "
  3.   4.   4.   5.   9.   3.   5.   4.
"x(corrected) = "
  3.   4.   4.   5.   3.   5.   4.
```

8.3 Scilab functions for working with files.

8.3.1 File opening function **mopen**

As in any other programming language, working with a file begins with opening it. To open a file in sci-language, the **mopen** function is intended, which looks like:

[fd,err]=mopen (file ,mode)

here: **file** is a character string containing the name of the file which will be opened;
mode is a character string that defines the required mode of access to the file, this parameter can have one of the following values:

'**r**' – the text file is opened in read mode, **the file must exist, otherwise nothing happens;**

'**t**' – text file;

'**b**' – a binary file;

'**rb**' – the binary file is opened in read mode;

'**w**' – an empty text file is opened, which is intended only for writing information, **if this file already exists, then its contents will be destroyed;**

'**wb**' – an empty binary file is opened, which is intended only for writing information;

'**a**' – opens a text file that will be used to add data to the end of the file; if the file does not exist, it will be created;

'**ab**' – opens a binary file that will be used to append data to the end of the file; if the file does not exist, it will be created;

'**r+**' – opens a text file that will be used in read and write mode, **the file must already exist, otherwise nothing will work;**

'**rb+**' – opens a binary file that will be used in read and write mode;

'**w+**' – the file is opened both for reading and for writing, **if the file exists, then its contents will be destroyed;**

'**wb+**' – the created empty binary file is intended for reading and writing information;

'**a+**' – the opened text file will be used to add data to the end of the file and read data, if there is no file, it will be created;

'**ab+**' – the binary file being opened will be used to append data to the end of the file and read the data; if the file does not exist, then it will be created;

err – scalar, error indicator:

error value:	error message:
0	no mistake
-1	no more logical modules
-2	can not open the file
-3	no more memory
-4	incorrect name
-5	incorrect status

fd – open file identifier, name (code) by which the functions described below will refer to a real file on disk.

8.3.2 The **mfprintf** function of writing a text to file.

The function of writing a text to file looks like this:

```
mfprintf(f, s1, s2)
```

here: **f** – file identifier (the identifier value is returned by the **mopen** function);

s1 – output string;

s2 – list of output variables.

In the output line, instead of the output variables, a conversion line of the following form is indicated:

```
% [flag][width][. precision][modifier]type
```

The values of the conversion line parameters are shown in table 8.1.

Some of the special characters shown in table 8.2. can be used in the output line.

Table 8.1 – Value of the conversion line parameters

Parameter	Assignment
flag	
-	Left alignment of a number. The right side is filled with spaces. Right alignment by default
+	A «+» or «-» sign is displayed before of the number
Space sign	A space sign is displayed before of a positive number, a «-» sign before a negative number
#	The number system code is displayed: 0 - before the octal number, 0x (0X) - before the hexadecimal number.
width	
n	Output field width. If n positions are not enough, then the output field is expanded to the minimum required. Unfilled positions are filled with space signs.
0n	Same as n , but blanks are filled with zeros.
. precision	
nothing	Default precision
n	For types e , E , f displaying n numbers after the decimal point
type	
c	On input the character type char , on output one byte
d, i	Decimal number with sign
i	Decimal number with sign

o	Octal int unsigned
u	Decimal number without sign
x, X	Hexadecimal int unsigned , along with x uses a – f symbols, along with X uses A – F symbols,
f	A value with sing of the form [-] dddd.dddd
e	A value with sing of the form [-] d.dddde [+ -] ddd
E	A value with sing of the form [-] d.ddddE [+ -] ddd
g	A value with sing of the type e or f depending on value and accuracy
G	A value with sing of the type E or F depending on value and accuracy
s	Character string
modifier	
h	For d, i, o, u, x, X using short integer
l	For d, i, o, u, x, X long using integer

8.3.3 Function **mfscanf** for reading data from a text file.

While reading data from a file, you can use a function of the following form:

A=mfscanf(f, s1)

here: **f** is the file identifier which is returned by the **mopen** function;

s1 – format string of the form:

%[width] [. precision] type

The **mfscanf** function works as follows: values from the file with the **f** identifier are read into the variable **A** in accordance with the **s1** format. When reading numeric values from a text file, remember that two numbers are considered separated if there is at least one space, tab or line break symbol between them.

While reading data from a text file, the user can keep track of whether the end of the file has been reached by using the function **feof(f)** (**f** is the file identifier), which returns **1** if the end of the file has been reached, and **0** otherwise.

Table 8.2 – Some special characters.

Symbol	Assignment
\b	Shift the current position to the left
\n	Moving to a new line
\r	Moving to the beginning of a line without moving to a new line
\t	Horizontal tabulation
\'	Single quote character
\"	Double quote character
\?	Symbol «?»

8.3.4 File close function `mclose`.

After performing all operations with the file, it should be closed using the `mclose` function of the following structure:

```
mclose(f)
```

here `f` is the identifier of the file which will be closed.

The `mclose` function should be used to close a file opened by the `mopen` function. If identifier `f` is omitted, then `mclose` function closes the last open file.

Using the `mclose('all')` function, you can close all open files at once, except for the standard system files. The `mclose('all')` function closes all files opened with `file('open',...)` or `mopen`. Be careful when using `mclose` because when you use it inside a Scilab program it will also close the program itself and Scilab will not execute commands written after `mclose('all')` function.

Let's consider an example of creating a test file. Open the SciNotes text editor window and write the program into it:

```
// The text file File_write.txt stores:
//the size of the matrix N by M;
//the matrix A (N, M).
//Here N is the number of matrix rows.
N=3;
//and M- the number of columns in the matrix.
M=4;
//the matrix A (N, M).
A=[2 4 6 7; 6 3 2 1; 11 12 34 10];
// Open an empty file W_file.txt in write mode.
f=mopen('C:\Users\D\Desktop\Примеры\W_file.txt','w');
// Write the N and M values,
//separated by «Horizontal tabulation» sing,
// to the file W_file.txt.
mfprintf(f,'%d\t%d\n',N,M);
// Write the next element of the matrix A
// to the file W_file.txt.
for i=1:N
for j=1:M
mfprintf(f,'%g\t',A(i,j));
end// for j=1:M
// At the end of the writing line,
// write the << Moving to a new line >> sing
```

```

// to the file.
mfprintf(f, '\n');
end// for i=1:N
fclose(f);

```

Open the «Примеры» folder on the desktop and click on the file name «W_file»

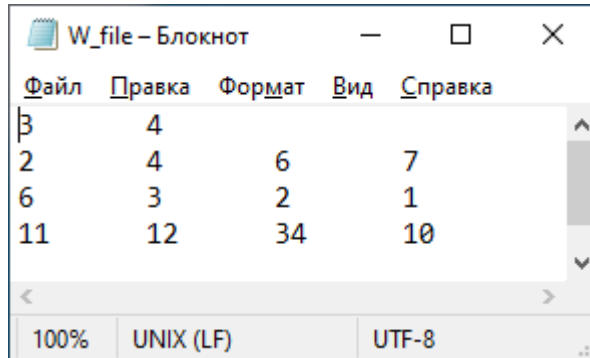


Figure 8.4 – Text file which was created.

Now let's create a program which will read data from this file:

```

//Reading the data from the file:
f=fopen('C:\Users\User\Desktop\Примеры\W_file.txt','r');
N=mfscanf(f, '%d');
M=mfscanf(f, '%d');
for i=1:N
for j=1:M
A(i,j)=mfscanf(f, '%g');
end// for j=1:M
end// for i=1:N
//Output the reading result on the display
disp('N = ',N);
disp('M = ',M);
disp('A = ',A);
//Close the file
fclose(f);

```

Let's write a program file named «Reading_a_File.sce» to the «Примеры» folder on the desktop.

Let's start the program from SciNotes by typing the command:

```

exec('C:\Users\User\Desktop\Примеры\Reading_a_File.sce');

```

and pressed the [Ctrl] + [L] keys, we will have on the SciLab console:

```
"N = "  
  3.  
"M = "  
  4.  
"A = "  
  2.    4.    6.    7.  
  6.    3.    2.    1.  
 11.   12.   34.   10.
```

8.4 Sample program in Scilab.

As an example of a sci-language program, consider the following task.

Rewrite positive numbers from array Y into array X, and remove elements from array X that are less than the average.

Below is a program for solving a task in which the use of the `min1` variable in the program is due to the fact that Scilab has a built-in `min` function, using a variable named `min` will not allow using the built-in `min` function:

```
//Enter the number of elements in an array Y.  
N=input('N=');  
disp('Vvod massiva Y');  
//Loop to input items in Y array.  
for i=1:N  
//Displaying the number of the input element  
disp('I=',i);  
//Input of the i-th element of the Y array.  
y(i)=input('Y=');  
end//for i=1:N  
// Displaying an array Y.  
disp(' Array y = ',y);  
// Variable k contains the number of positive  
//elements in array y, and as a result,  
//the number of elements in array x.  
// Initially k = 0.  
k=0;  
//Iterate over all the elements in the Y array.  
for i=1:N do  
//If the current element is positive then  
if y(i)>0 then  
//the value of the variable k is increased by 1,  
k=k+1;  
//and the element of the array Y is rewritten  
//into array X.
```

```

x(k)=y(i);
end;//if y(i)>0 then
end;//for i=1:N do
//output of array X.
disp(' Array X = ',x);
// In the variable s we will store the sum
// of the elements of the array x, in the
//variable min1 - the minimum the value
//in the x array, in Nmin - the number of
//the minimum element in the x array.
s=sum(x);
xmean=s/k;
[min1,Nmin]=min(x);
//Starting from the minimum, iterate over all
// the elements of the array, and if
i=Nmin;
while i<=k do
//the current element is less than the average,
if x(i)<xmean then
//then remove the current element. Please note
//that when deleting, the elements are shifted
// by 1 to the left, and therefore
//it is not necessary to increase i
//by 1 and go to the next element after deletion.
for j=i:k-1 do
x(j)=x(j+1);
end;//for j=i:k-1 do
//Reducing the number of elements in an array by 1.
x(k)=[];
k=k-1;
else
// If the deletion did not occur, then go
//to the next element of the array.
i=i+1;
end;//if x(i)<s/k then
end;//while i<=k do
disp('Transformed array X = ',x);

```

Let's write the program file under the name «Array_converter.sce» to the «Примеры» folder on the computer desktop.

Start the program from the Scilab console window by typing the command:

```
exec('C:\Users\D\Desktop\Примеры\Array_converter.sce');
```

Press the [Ctrl] key and enter the number of elements in the array in the SciLab console window (for example) :

N = 7

Sequentially, element by element, enter the array.

```

    "Vvod massiva Y"
    "I="
    1.
Y=5
    "I="
    2.
Y=6
    "I="
    3.
Y=-5
    "I="
    4.
Y=6
    "I="
    5.
Y=6
    "I="
    6.
Y=3
    "I="
    7.
Y=6

```

After entering the value of the last seventh element of the Y array, the program will display the values of the elements of the Y array:

```

" Array y = "
    5.
    6.
   -5.
    6.
    6.
    3.
    6.

```

As well as the values of array X elements with the excluded negative value and the values of the elements of the transformed array with the excluded number less than the average value of the elements:

```

" Array X = "
    5.
    6.
    6.
    6.
    3.
    6.
"Transformed array X = "
    5.
    6.
    6.
    6.

```


6.

In this part of the course, the basic possibilities of programming in Scilab were discussed. The next chapter will consider the possibilities of building visual programs.

Questions for self-examination for the thirteenth lecture:

1. *What is the for loop operator intended for?*
2. *How can be opened a file using the mopen function?*
3. *How can be written a text file using the mfprintf function?*
4. *How can data be read from a file using the mfprintf function?*
5. *How can the file be closed?*

Lecture 14

The purpose of the lecture is to learn how to work with graphic windows using interface controls: buttons, labels, checkboxes.

9 CREATING GRAPHIC APPLICATIONS IN THE Scilab

Scilab allows you to create not only ordinary programs for automating calculations, but also visual applications that will run in the Scilab environment.

Scilab allows you to create not only ordinary programs for automating calculations, but also visual applications that will run in the Scilab environment.

The main object in the Scilab environment is the graphic window.

9.1. Working with the graphic window.

To create an empty graphic window, use the function:

F=figure ()

As a result of executing this command, a graphic window named **objfigure1** will be created. By default, the first window is named **objfigure1**, the second is named **objfigure2**, and so on. A handle to a graphic window (a handle is understood as a variable that stores the address of a window or other object) is written to the variable **F**. The size and position of the window on the computer screen can be set using the parameter:

'position',[x y dx dy]

here: **x y** is position of the upper left corner of the window (horizontally and vertically, respectively) relative to the upper left corner of the screen;

dx is horizontal size of the window (window width) in pixels;

dy is vertical size of the window (window height) in pixels.

Window parameters can be set in one of two ways.

Directly when creating a graphic window, its parameters are set. In this case, the call to the **figure** function looks like this:

```
F=figure('Property1', 'Value1',  
..., ..., 'Propertyn', 'Valuen')
```

here: ' **Property1**' is the name of the first parameter;
'**Value1**' is the value of the first parameter;
'**Propertyn**' is the name of the n-th parameter;
'**Valuen**' is the value of the n-th parameter.

The **valuen** will be used in quotes if the parameter value is a string, if the parameter value is a number, then quotes should not be used.

For example, using the command:

```
F=figure('position', [10 100 300 200]);
```

will be plotted the window which is shown in fig. 9.1.

2. After plotting a graphic window we will use the function:

```
set(f, 'Property', 'Value')
```

which sets the value of the parameters; here **f** is a handle of a graphic window, '**Property**' is the name of the parameter, '**Value**' is its value.

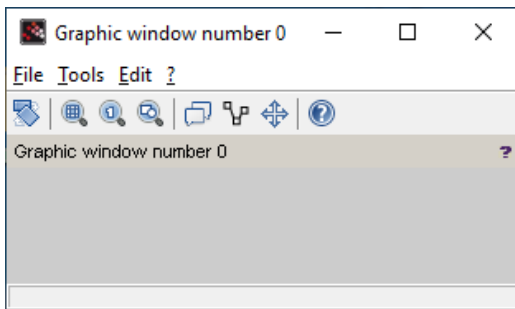


Figure 9.1 – First graphic window.

The next two lines define the location and size of the window. The window (fig. 9.2) will be located approximately in the middle of the display (620,440) and width will be greater than the height (350,100).

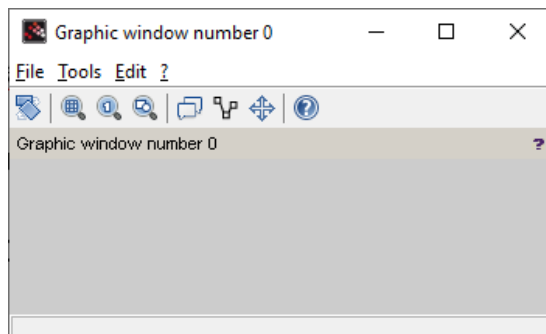


Figure 9.2 – The graphic window of a specific size and location.

```
f=figure();
set(f, 'position', [620,440,350,100])
```

To change the title of the window, we can use the '**figure_name**' parameter which defines the title of the window. The program below shows an example of creating a window which is named **FIRST WINDOW** (fig. 9.3).

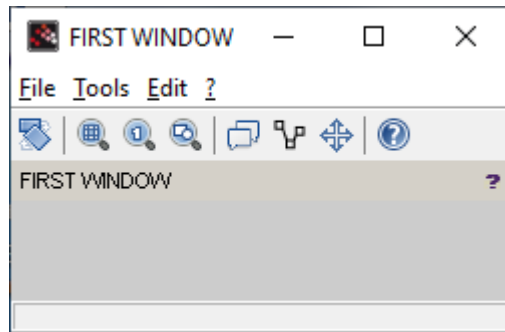


Figure 9.3 – Window is titled FIRST WINDOW

```
f=figure();
set(f, 'position', [20,40,250,50]);
set(f, 'figure_name', 'FIRST WINDOW');
```

The same window can be obtained using the **figure()** function, referring to it once:

```
f=figure('position', [20,40,250,50], ...
'figure_name', 'FIRST WINDOW');
```

The graphic window can be closed using the function:

```
close(f)
```

here: **f** is a handle of a window.

The window is removed with help of the function:

```
delete(f)
```

here: **f** is a handle of a window.

9.2 Dynamic creation of interface elements and description of the main functions.

Scilab uses a dynamic way to create interface components. It consists in the fact that at the stage of program execution, certain controls (buttons, labels, checkboxes, etc.) can be created (and deleted) and their properties are assigned corresponding values.

To create any interface component with specified properties, use the **uicontrol** function, which returns a handle of the component which is formed:

```
C=uicontrol(F, 'Style',  
'component_type','Property_1',Value_1,'Property_2',  
Value_2,..., 'Property_k', Value_k);
```

here: **C** –is a handle of the created component;

F is a handle of the object inside which the component will be created (most often this component will be a window); The first argument **F** of the **uicontrol** function is optional, and if it is absent, the parent (owner) of the component being created is the current graphic object, that is the current graphic window;

'**Style**' is a service string which is indicating the style of the component which will be created (symbolic name);

'**component_type**' defines to which class the created component belongs, it can be **PushButton, Radiobutton, Edit, StaticText, Slider, Panel, Button Group, Listbox** or other components, this property will be specified for each of the components;

'**Property_k, Value_k**' – define the properties and values of individual components, they will be described below specifically for each component.

You can change certain properties of an existing interface object using the **set** function:

```
set(C,'Property_1',Value_1,  
'Property_2',Value_2, ...,  
'Property_k',Value_k)
```

here: **C** is a handle of a dynamic component whose parameters will be changed. **C** can also be a vector of dynamic components, in this case the set function will set property values for all objects **C(i)**;

'**Property_k, Value_k**' – define the parameters which will be changed and their values.

You can get the value of the component parameter using the **get** function of the following structure:

```
get(C,'Property')
```

here: **C** is a handle of a dynamic interface component, the parameter value of which needs to be found;

'**Property**' is the name of the parameter, the value of which you want to know.

The function returns the value of the parameter.

Next, we will talk about the features of creating various components.

9.2.1 Command button.

A command button of the **PushButton** type is created using the **uicontrol** function, in which the '**Style**' parameter must be set to the value '**pushbutton**'. By default, it is not provided with any caption, it has a gray color and is located in the lower left corner of the window. The caption on the button (fig. 9.4) can be set using the **String** property. For example:

```
//Plot a window
d=figure();
set(d,"position",[20,40,300,110]);
set(d,"figure_name","Window With Button");
//Create a button and define a property Style.
dbt=uicontrol(d,"Style","pushbutton","position",...
[5,5,70,20]);
```

There is no caption on the button. Let's create a YES caption on the button

```
//Create a YES caption on the button
set(dbt,"string","YES");
```

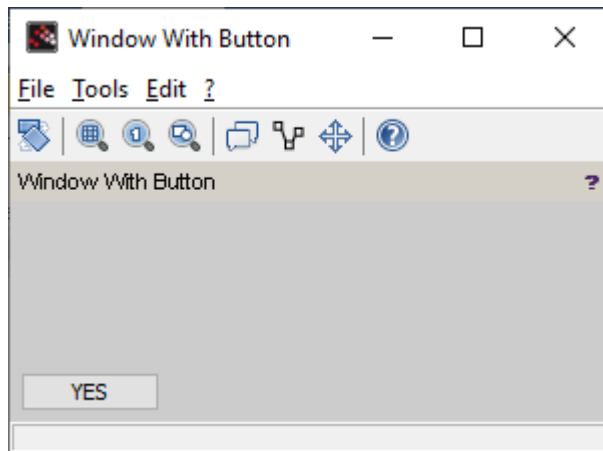


Figure 9.4 – The button with YES caption.

Now we will modify the program for creating a button by setting additional values of some properties:

- location and title of the window;
- a caption on the button;
- location of the button.

The text of the program is shown below, and in fig. 9.5 shows the window that turned out as a result of the work of this program:

```
//Create the window.
f=figure();
```

```

//Determine the location of the window.
set(f,"position",[0,0,250,100])
//Determine the name (title) of the window.
set(f,"figure_name","Window with control button");
//Create a button (style - pushbutton),
// the caption on the button is «Button»,
// the position of the button is determined
// by the position parameter.
Button=icontrol("style","pushbutton","string",...
" Button","position",[50,50,70,20])

```

When you click on a button, a dotted rectangle appears around its caption, indicating that the button is in focus. The main purpose of a command button is to call a function that responds to a click on the button.

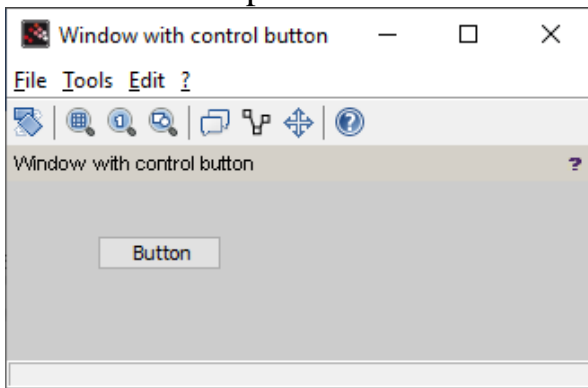


Figure 9.5 – Window with control button.

Clicking the button generates a **Callback** event, which is specified as a parameter to the **icontrol** function. The value of the **Callback** parameter is a string with the name of the function called when the button is clicked. In this case, the **icontrol** function becomes like this:

```

Button=icontrol('style','pushbutton','string',
'Button','Callback','Function');

```

Here **'Function'** is the name of the function called when the **Callback** event occurs.

As an example, consider a window with a button, when you click on which a window appears with a graph of the function $y = \sin(x)$, the program looks like this:

```

f=figure();
set(f,"position",[0,0,250,100])
set(f,"figure_name","Grafik");
// Create a button that, when you click on it
// with the mouse, calls the gr_sin function.

```

```

Button=icontrol("style","pushbutton",...
"string","Button","position",[50,50,100,20],...
"Callback","gr_sin");
function y=gr_sin()
f2=figure();
set(f2,"position",[300,300,350,200]);
x=-5:0.2:5;
y=sin(x);
plot(x,y);
xgrid();
endfunction

```

After starting this program, a window with a button will appear, which is shown in fig. 9.6, when the [Button] is clicked, the event handler and the `gr_sin` function are called, as a result a window with a chart appears.

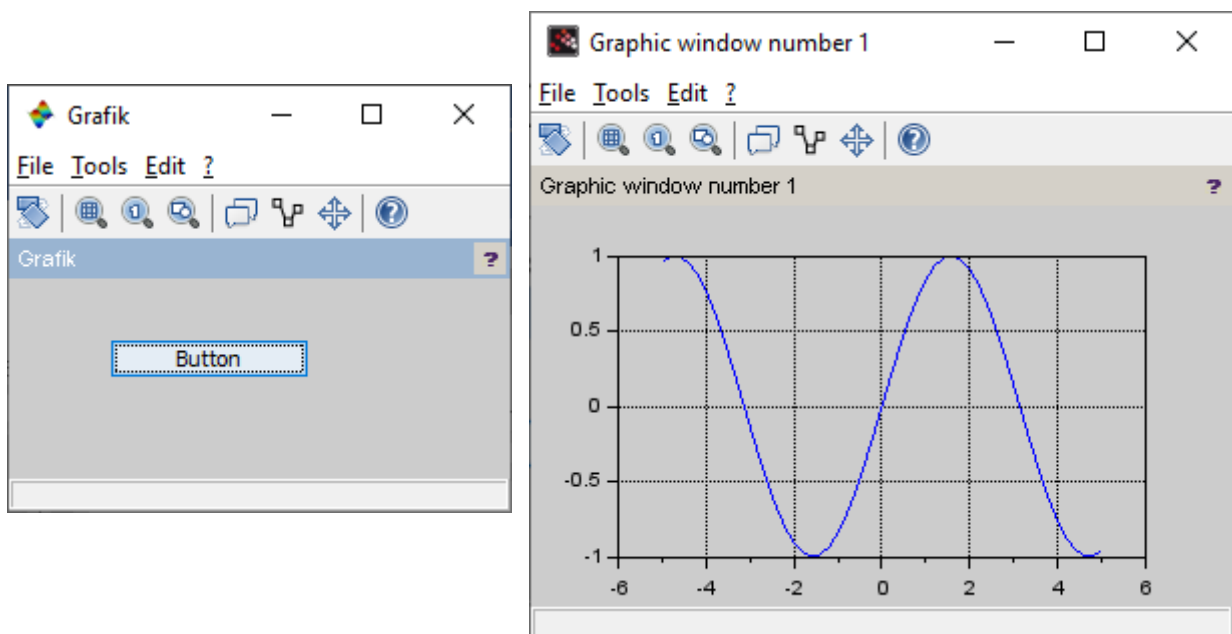


Figure 9.6 – The window with the button and the graphic window which is called.

9.2.2 Label.

The next most commonly used component is the label, this is a text field for displaying symbolic information. To define a label, the value of the '**Style**' parameter in the `icontrol` function must be set to '**text**'. The component is intended for displaying a character string (or several lines). The text displayed on the label - the value of the '**String**' parameter - can be changed only from the program.

Let's consider an example of creating a text field (label) using the `icontrol` function (Figure 9.7). It should take in mind that the position of the window in which the label text will appear (the "**Position**" parameter) differs from (the

"**position**" parameter) defining the position of the graphic window. The "**Position**" parameter is set as follows $[x, y, l, h]$, where x and y are the coordinates of the lower left corner of the window for the location of the label, the origin is at the lower left corner of the graphic window, l is the length of the window, h is the height of the window the program for this example looks like:

```
f=figure();
set(f,"position",[0,0,250,100])
set(f,"figure_name","Window with label");
uicontrol("Style","text","Position",...
[100,80,40,15],"String","Metka");
```

One of the main properties of a label is the horizontal alignment of the text, which is determined by the **HorizontalAlignment** property. This property can take one of the following values:

- left** – alignment text to the left (default);
- center** – center alignment of text;
- right** – alignment text to the right.

As an example, consider a window containing 4 text boxes with different values for the **HorizontalAlignment** property. The corresponding window is shown in fig. 9.8, the program text looks

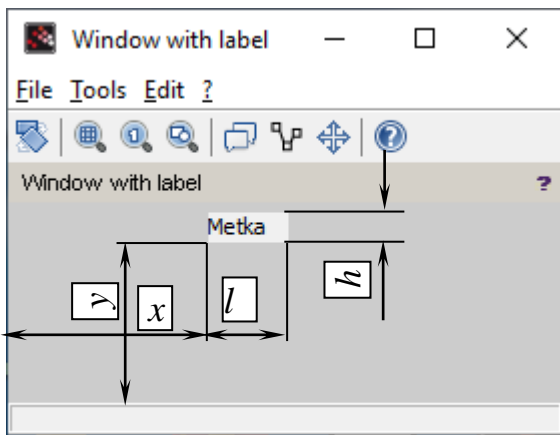


Figure 9.7 – Window with label.

like:

```
hFig=figure();
set(hFig,"Position",[50,50,300,200]);
set(hFig,"figure_name","Different labels");
//First label
hSt1=uicontrol("Style","text","Position",[30,30,150,20],"String",
"Label 1");
set(hSt1,"BackgroundColor",[0 0.9 1]);
set(hSt1,"HorizontalAlignment","left");
// Second label
hSt2=uicontrol("Style","text",
"Position",[30,60,150,20],"HorizontalAlignment","center",
"BackgroundColor",[1 0 1],"String","Label 2");
// Third label
hSt3=uicontrol("Style","text","Position",[30,90,150,20],"H
orizontalAlignment","right","BackgroundColor",[1 0.9
0],"String","Label 3");
// Fourth label
```



```
hSt4=uicontrol("Style","text","Position",[30,120,150,20],"
BackgroundColor",[1 0.75 0.75],"String","Label 4");
```

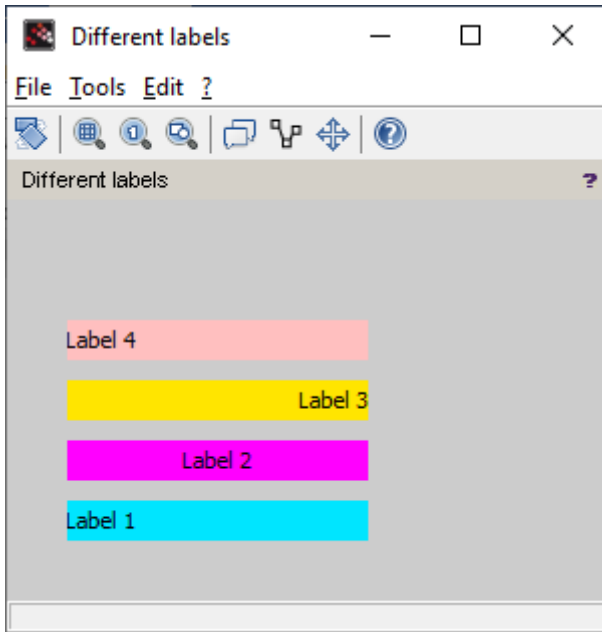


Figure 9.8 – Window with different labels.

9.2.3 The Switch and Checkbox components.

Let's look at two more components: a radio button and a check box that allow you to toggle between states or turn off one of the properties.

For a checkbox, the "**Style**", property takes the value '**checkbox**', for a radio button, the "**Style**" property must be set to '**radiobutton**'.

9.2.3 Checkbox and radiobutton components.

Let's look at two more components: a radio button and a check box that allow you to switch between states or turn on one of the properties.

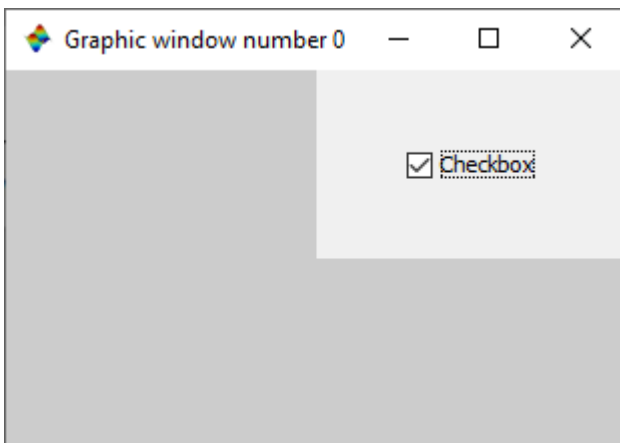


Figure 9.9 – Window with state component.

For a component to enable or disable a state, the **Style**' property takes the value '**checkbox**', for a component to select one of a number of states, the **Style**' property must be set to '**radiobutton**'.

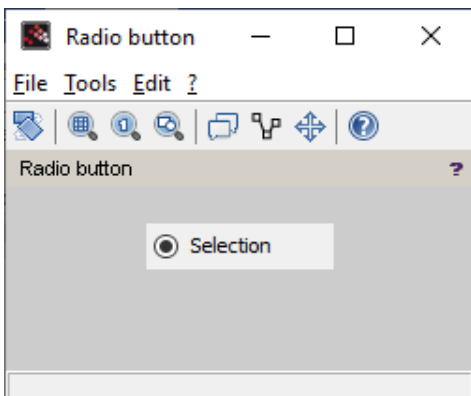
An example of creating a **checkbox** state component is shown in figure 9.9 and the corresponding program looks like this:

```
// Create a window
```

```
f = createWindow()
f.axes_size = [400 400];
uicontrol(f, "style", "checkbox",...
    "units", "normalized", "position",...
    [0.5 0.5 0.5 0.5], "string",...
    "Checkbox", "horizontalalignment",...
    "center", "groupname", "grouptest");
```

An example of creating a component for selecting one of several states **"radiobutton"** is shown in figure 9.10, and the corresponding program looks like this:

```
hFig=figure();
set(hFig,"Position",[50,50,250,100]);
set(hFig,"figure_name"," Radio button ");
R=uicontrol("Style","radiobutton","String",...
    " Selection","value",1,"Position",[75,55,100,25]);
```



During the creation of a switch, its state must be set (parameter **'value'**), the switch can be active (value **'value'** is 1) or not (value **'value'** is 0).

The switch can respond to the **'Callback'** event and call a specific function to be executed. In this case, you can create a button by calling the following **uicontrol** function:

Figure 9.10 – Window with component for selection.

```
hFig=figure();
set(hFig,"Position",...
    [50,50,250,100]);
set(hFig,"figure_name"," Radio button ");
R=uicontrol("Style","radiobutton","String",...
    " Selection","value",1,"Callback","gr_sin",...
    "Position",[75,55,100,25]);
function y=gr_sin()
f2=figure();
set(f2,"position",[300,300,350,200]);
x=-5:0.2:5;
y=sin(x);
plot(x,y);
```

```

xgrid() ;
endfunction

```

Here **"gr_sin"** is the name of the function that will be called when a switch is clicked. However, when writing a function, remember that when you click on a switch, its state automatically changes.

After starting the program for execution, first a window with a switch will appear (fig. 9.10), and then a window with a graph (fig. 9.6).

Now we use the switches in another program, in which using the switch you can select the function, the graph of which will be reproduced in the graphic window (fig. 9.11), when you click on the [Plot] button:

```

//Create a graphics window.
hFig=figure("Position",[50,50,600,400]);
//Draw a panel for controls
uicontrol("Style","text","BackgroundColor",[0 0.95
0.75],"Position",[10,15,100,160]);
//Creating radio buttons
hRb1=uicontrol("Style","radiobutton","String","sin(x)","
value",1,"BackgroundColor",[1 0.75 0.75],
"Position",[25,100,60,20]);
hRb2=uicontrol("Style","radiobutton","String","cos(x)","
value",1,"BackgroundColor",[0.5 0.75 1],
"Position",[25,140,60,20]);
// Create a button named «Plot», which uses the «Radio»
//handler to plot the function according to the position
// of the radio buttons.
Button=uicontrol("style","pushbutton","string","Plot","p
osition",[20,50,80,20],"CallBack","Radio");
// Create a button named «Close», which closes the
//window using the «Final» handler.
Button1=uicontrol("style","pushbutton","string","Close",
"position",[20,25,80,20],"CallBack","Final");
// «Radio» function that responds to a click on the
// button
function Radio()
//Select the area where the graph axes are located
a1=newaxes();
a1.axes_bounds=[0.15,0,0.9,1];
//Determining the range of variation of the variable x
x=-2*%pi:0.1:2*%pi;
if get(hRb1,"value")==1 //If the first button is active,

```

```

y=sin(x) ;
plot(x,y,"-r"); //then plotting the sinusoid
xgrid();//Plotting a grid
end;
if get(hRb2,"value")==1// If the second button is active,
y=cos(x) ;
plot(x,y,"-b"); //then plotting of the cosine curve.
xgrid(); //Plotting a grid
end;
endfunction;
//The function responsible for the [Close] button
// and closing the window.
function Final()
close(hFig) ;
endfunction;

```

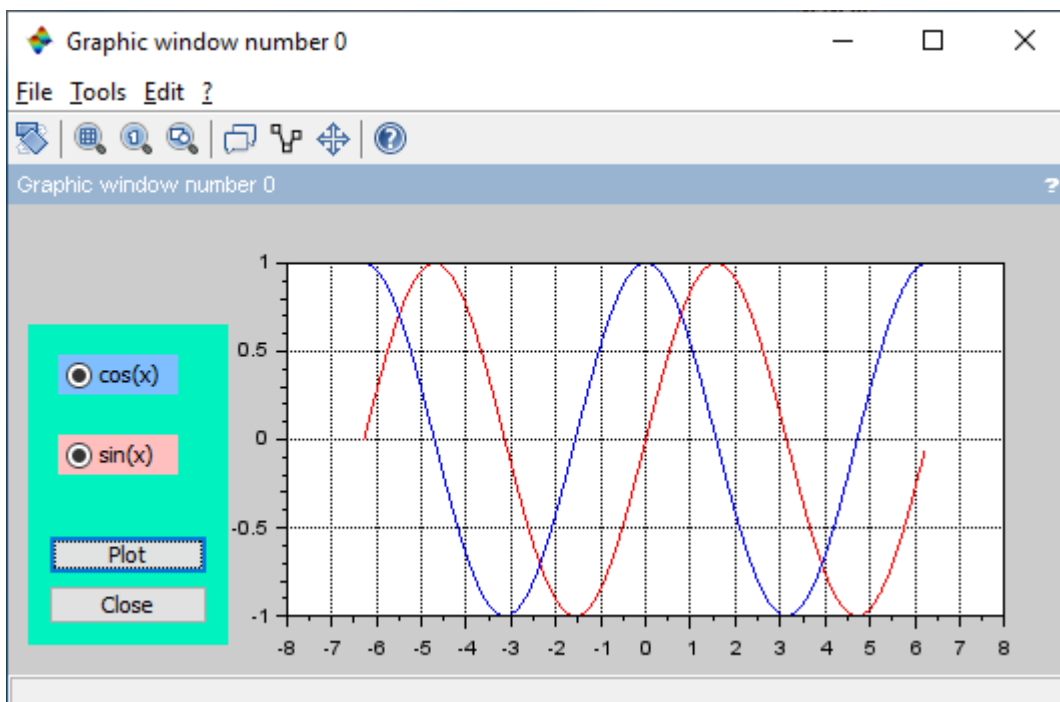


Figure 9.11 – Application window.

In fig. 9.11 the application window is presented. When the [Plot] button is clicked, the area for the plots selected with the function switches will appear in the graphic window. If no function is selected, a blank area will be displayed in the graphic window when the [Plot] button is clicked. The [Close] button closes the application. The state of the switches changes automatically when you click on them.

The checkbox component) is used to indicate non-alternative combinations. The "Callback" event is generated and the button is automatically selected when you click on the square or its accompanying caption. If the checkbox is enabled, the

value of the "**value**" property is 1. Click on the checkbox automatically changes the state to the opposite. The use of a check box is the same as a radio button.

Questions for self-examination for the fourteenth lecture:

1. What is the function $F = \text{figure}()$ for?
2. What is the **uicontrol** function for?
3. How can be created a button in the graphic window?
4. How to use the button to control the program?
5. What is the label for?
6. How to create radio button and checkbox control components?

Lecture 15

The purpose of the lecture is to learn how to use the mechanism of interaction of standard Scilab components and event handlers when writing programs and use the least squares method when processing experimental data and solve optimization tasks.

9.2.4 Editing window component.

The interface element editing window (for that component, the '**Style**' property must take the value '**edit**') can be used for input and output of symbolic information. The text typed in the editing window can be corrected. When working with a component, you can use clipboard operations. An input procedure terminated by pressing the [Enter] key generates a **CallBack** event.

The input string is defined by the '**String**' parameter, which defines the text in the component. For the component to function normally, this parameter must be set when defining a component using the **uicontrol** function. You can change the value of this property using the **set** function, and read its value using the **get** function.

The entered text can be aligned to the left or right edge of the input window if the corresponding value of the **HorizontalAlignment** property is set (by analogy with the **Label** component). If the entered text is a numerical value that should be used in the program, then the contents of the '**String**' property are converted to a numerical format using the **eval** function (you could also use the **evstr** function) (it will be discussed further on the example of a quadratic equation).

As an example of working with several components, consider the following task. Write a program for solving a quadratic or biquadratic equation.

The choice of the type of equation will be carried out using the **radiobutton** component.

```
f=figure(); //Creation of a graphic object.
//Setting the size of the window.
set(f,"position",[0,0,700,300])
//Setting the title of the window.
```

```

set(f,"figure_name","THE EQUATION");
// Creatting text fields for labels of input fields
// for coefficients.
//Coefficient A =.
lab_a=icontrol(f,"style","text","string","A=","position"
,[50, 250, 100, 20]);
// Coefficient B=.
lab_b=icontrol(f,"style","text","string","B=","position"
,[150, 250, 100, 20]);
// Coefficient C=.
lab_c=icontrol(f,"style","text","string","C=","position"
,[250, 250, 100, 20]);
//Editing field for entering the coefficient a.
edit_a=icontrol(f,"style","edit","string","1","position"
,[50, 230, 100, 20]);
//Editing field for entering the coefficient b.
edit_b=icontrol(f,"style","edit","string","-
2","position",[150, 230, 100, 20]);
//Editing field for entering the coefficient c.
edit_c=icontrol(f,"style","edit","string","-
1","position",[250, 230, 100, 20]);
//A text field that determines the output of the results.
textresult=icontrol(f,"style","text","string","",""positi
on",[5, 80, 650, 20]);
//The radio button which is responsible for the
//choice of the type of equation.
radio_bikv=icontrol("style","radiobutton","string","
Biquadratic equation?",
"value",1,"position",[100,100,300,20]);
BtSolve=icontrol("style","pushbutton","string","Solve","
Callback", "Solve","position",[50,50,120,20]);
BtClose=icontrol("style","pushbutton","string","Close","
Callback", "_Close","position",[300,50,120,20]);
//Equation solution function.
function Solve()
// Read the value of the variables from the
//text fields and convert them to a numeric type.
a=evstr(get(edit_a,"string"));
b=evstr(get(edit_b,"string"));
c=evstr(get(edit_c,"string"));
d=b*b-4*a*c;
//Check the value of the radio button
//if the radio button is disabled,
if get(radio_bikv,"value")==0
//then we solve the quadratic equation,
if d<0
set(textresult,"string","No quadratic equation

```

```

solution");
    else
        x1=(-b+sqrt(d))/2/a;
        x2=(-b-sqrt(d))/2/a;
        set(textresult,"string",sprintf("2      quadratic      roots\t
x1=%1.2f\t x2=%1.2f",x1,x2));
    end;
    //if the radio button is on,
    else
        //then we solve the biquadratic equation.
        if d<0
            set(textresult,"string","No solution of the iquadratic
equation");
        else
            y1=(-b+sqrt(d))/2/a;
            y2=(-b-sqrt(d))/2/a;
            if(y1<0)&(y2<0)
                set(textresult,"string","No solution of the biquadratic
equation");
            elseif (y1>=0)&(y2>=0)
                x1=sqrt(y1);x2=-x1;x3=sqrt(y2);x4=-x3;
                set(textresult,"string",sprintf("4      roots      of      the
biquadratic equation\t x1=%1.2f\t x2=%1.2f\t x3=%1.2f\t
x4=%1.2f",x1,x2,x3,x4));
            else
                if y1>=0
                    x1=sqrt(y1);x2=-x1;
                else
                    x1=sqrt(y2);x2=-x1;
                end;
                set(textresult,"string",sprintf("2      roots      of      the
biquadratic equation\t x1=%1.2f\t x2=%1.2f",x1,x2));
            end;
        end;
    end;
endfunction
// Window close function.
function _Close()
close(f)
endfunction

```

This program helps you understand the interaction mechanism of standard Scilab components and event handlers, so that you can use this knowledge when developing your own visual applications.

9.2.5 Lists of strings.

The interface component 'listbox' in the simplest case can be viewed as a window with an array of strings in it. If the length of the list more than the height of the window, then a vertical scroll bar that is generated automatically can be used to move through the list.

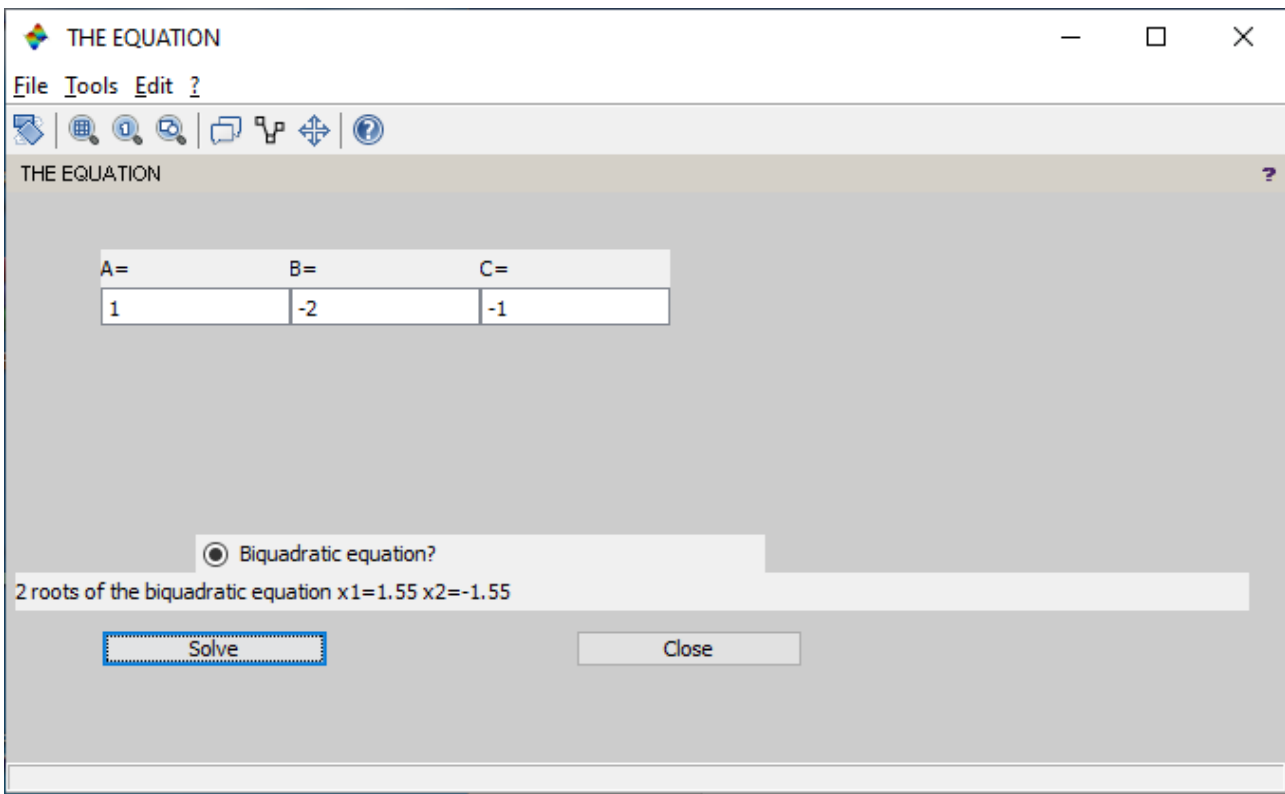


Figure 9.11 - Window of the program for solving the equations.

The list of strings is created using the `uicontrol` function when the 'Style' parameter is set equal to 'listbox'. Let's consider this with a simple example, the window for which is shown in fig. 9.12, and the program looks like this:

```
//Creation of a graphic window.
f=figure();
//Set the size of the window.
set(f,"position",[50,50,280,100])
// Creation of listbox
h=uicontrol(f,"style","listbox","position",[10 10 100
70]);
//Filling out the list.
set(h, "string", "string 1|string 2|string 3|string
4|string 5|long string 6");
//Selecting 1 and 6 lines in a list
set(h, "value", [1 6]);
```

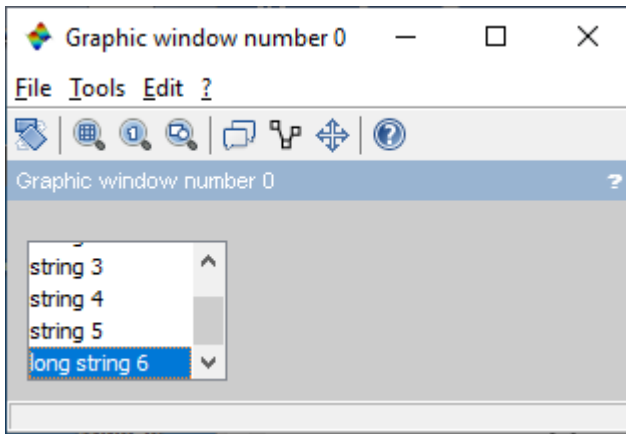



Figure 9.12 – List with selected item.

The list allows the user to select one or several lines and, depending on the choice, perform one or another action.

The selection of a row is carried out by clicking the left mouse button at the moment when the cursor points to the selected row. Simultaneously with the highlighting of the line, its number is entered into the '**value**' property and the '**Callback**' event is generated. Lines in the list are numbered from 1.

For example, let's consider the program

for determining the day of the week by date:

```
// Specifying the name of the graphics window
h = figure('position', [50 50 400 400],
'backgroundcolor', [0.7 0.9 1], "figure_name", 'What day were
you born?');
// Placing a label with prompting you to enter the year
T1 = uicontrol(h, 'style', 'text', 'string', 'Enter the
year:', 'position', [20 340 200 50], 'backgroundcolor', [0.7
0.9 1]);
// Defining the box for entering the year
E1 = uicontrol(h, 'style', 'edit',
'string', '1997', 'position', [150 350 70 30], 'fontsize',
15, 'backgroundcolor', [1 1 1]);
// Placing a label with prompting to you enter the month
T2 = uicontrol(h, 'style', 'text', 'string', 'Select the
month:', 'position', [20 260 200 50], 'backgroundcolor', [0.7
0.9 1]);
// Creating and filling the days of the week
L1 = uicontrol(h, 'style', 'listbox', 'position', [150 130
120 170], 'fontsize', 15, 'backgroundcolor', [1 1 1]);
set(L1, 'string', ' January | February | March | April |
May | June | July | August | September | October | November
| December ');
//set(L1, 'value', [1:12]);
// Placing a label with prompting to you enter the day
T3 = uicontrol(h, 'style', 'text', 'string', 'Enter the
day:', 'position', [20 60 200 50], 'backgroundcolor', [0.7
0.9 1]);
// Defining the box for entering the day
E2 = uicontrol(h, 'style', 'edit', 'string', '15',
'position', [150 70 70 30], 'fontsize', 15, ...
'backgroundcolor', [1 1 1]);
```

```

// Create a birthday announcement window
PrintBD = uicontrol(h, 'style', 'text', 'string', ' Day
of the week of your birthday:', 'position', [20 30 200 30],
'backgroundcolor', [0.7 0.9 1]);
//Day of the week calculation function
function birthday(guientries)
    y = evstr(get(E1, 'string'))
    m = get(L1, 'value')
    d = evstr(get(E2, 'string'))
    num = datenum(y, m, d);
    [n, s] = weekday(num);
// Outputting the result to the console
    disp(y,m,d,'Day of the week of your birthday '+s)
//Displaying a specific birthday in the graphic window
DWBD = uicontrol(h, 'style', 'text', 'string', '
'+s,'position', [210 30 40 30],'backgroundcolor', [0.7 0.9
1]);
endfunction

// Creating a birthday button
P1 = uicontrol(h, 'position', [250 70 120 30], 'style',
'pushbutton', 'string', 'Define!', 'callback', 'birthday',
'backgroundcolor', [1 1 0]);

```

The process of creating a program can be traced by the comments provided in the text of the program. The program window is shown in fig. 9.13. The disadvantage of the program is that the day of the week is given only in English. This drawback can be easily eliminated using the **case** selection operator, which is proposed to be done independently.

10. PROCESSING OF EXPERIMENTAL DATA

10.1 Least square method

The least squares method makes it possible, from experimental data, to select an analytical function that passes as close to the experimental points as possible.

Suppose that as a result of the experiment, some data were obtained, which are displayed in the form of a table:

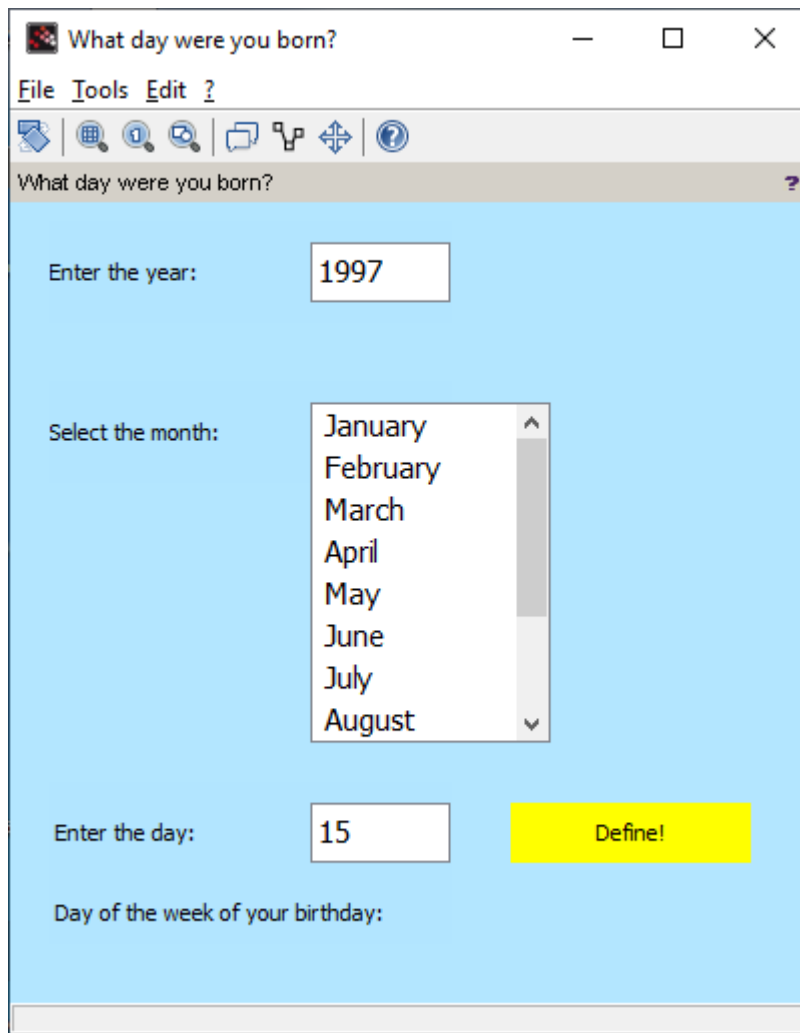


Figure 9.13 – The program for determining the day of the week by the date.

x_i	x_1	x_2	x_3	x_4	x_5	x_6	...	x_n
y_i	y_1	y_n

It is required to construct an analytical relationship that most accurately describes the results of the experiment.

The idea behind the least squares method is that the function $Y = f(x, a_0, a_1, \dots, a_k)$ must be selected in such a way that the sum of the squares of the deviations of the measured values y_i from the calculated ones Y_i have to be the least:

$$S = \sum_{i=1}^n (y_i - f(x_i, a_0, \dots, a_k))^2 \rightarrow \min \quad (10.1)$$

The task is reduced to determining the coefficients a_i from condition (10.1). To solve this task, Scilab provides the function

$$[\mathbf{a}, \mathbf{S}] = \text{datafit}(\mathbf{F}, \mathbf{z}, \mathbf{c})$$

here \mathbf{F} is the function, the parameters of which must be selected;

z - matrix of initial data;
c is the vector of initial approximations;
a is vector of coefficients;
S is the sum of the squares of the deviations of the measured values from the calculated ones.

Let's look at the use of the **datafit** function using an example.

As a result of the idling operation of the electric motor, the dependence of the power consumed from the network (P , in W) on the input voltage (U , in V) was determined (table 10.1).

Table 10.1 – Experimental data.

U , in V	132	140	150	162	170	180	190	200	211	220	232	240	251
P , in W	330	350	385	425	450	485	540	600	660	730	920	1020	1350

Using the least squares method, choose a dependence of the form $P = a_1 + a_2U + a_3U^2 + a_4U^3$.

For a more visual representation of the graph of parameter changes, we will reduce the parameters by a factor of one hundred, then the solution to the task with comments is given:

```

//A function that calculates the difference
//between experimental and theoretical values.
// Before starting calculations, it is needed to define:
//z=[x;y] - the initial data matrix - and
//c - vector of initial values of the coefficients.
function [zr]=G(c,z)
zr=z(2)-c(1)-c(2)*z(1)-c(3)*z(1)^2-c(4)*z(1)^3
endfunction
// The initial data:
x=[1.32 1.40 1.50 1.62 1.70 1.80 1.90 2.00 2.11 2.20 2.32
2.40 2.51];
y=[3.30 3.50 3.85 4.25 4.50 4.85 5.40 6.00 6.60 7.30 9.20
10.20 13.50];
//Formation of a matrix of initial data
z=[x;y];
//Initial approximations vector
c=[0;0;0;0];
//The solution of the task
[a,S]=datafit(G,z,c)
//Plotting the graph which is based on experimental data
plot2d(x,y,-1);
//Plotting the graph of the approximation function
t=1.32:0.01:2.51;
Ptc=a(1)+a(2)*t+a(3)*t.^2+a(4)*t.^3;
plot2d(t,Ptc);
  
```

After starting the program in the console window, we will get:

```

a =
- 51.576664
  95.594671
- 55.695312
  11.111453
S =
  0.5287901

```

So, as a result of the program operation, an analytical dependence of the form $P = -51.57 + 95.59U - 55.69U^2 + 11.11U^3$ was selected, and the sum of the squares of the deviations of the measured values from the calculated S is equal to 0.53.

And in the graphic window we get the location of the experimental points marked with crosses and the theoretical curve constructed according to the equation with the calculated coefficients (fig. 10.1).

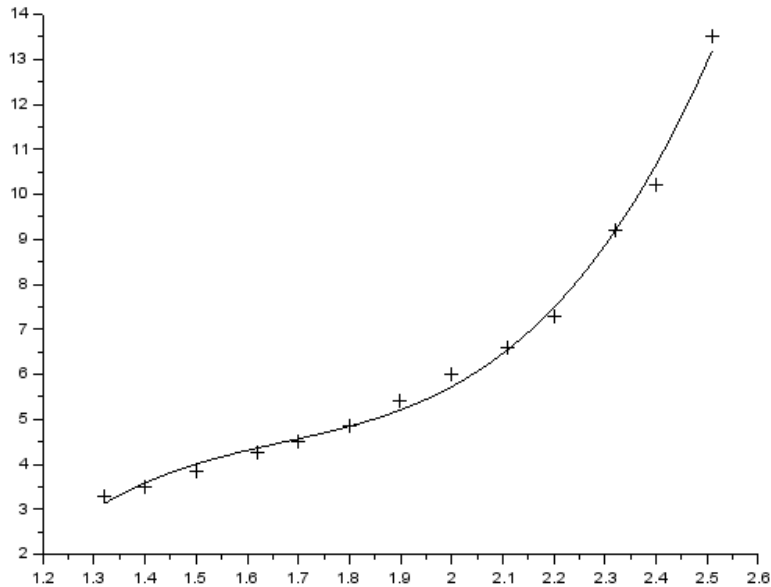


Figure 10.1 – Graphical representation of the task.

In practice, quite often there is a mathematical dependence describing the change in a function depending on the initial data of the following form: $Y = a_1 \cdot x^{a_2} + a_3$. Consider the data processing program for this case and the corresponding graph (fig.10.2):

```

function [zr]=F(c,z)
zr=z(2)-c(1)*z(1).^c(2)-c(3);
endfunction
x=[10.1,10.2,10.3,10.8,10.9,11,11.1,11.4,12.2,13.3,13.
8,14,14.4,14.5,15,15.6,15.8,17,18.1,19];
y=[24,36,26,45,34,37,55,51,75,84,74,91,85,87,94,92,96,

```

```

97,98,99];
z=[x;y];
c=[0;0;0];
[a,S]=datafit(F,z,c)
plot2d(x,y,-3);
t=10:0.01:19;
Yt=a(1)*t.^a(2) + a(3);
plot2d(t,Yt);

```

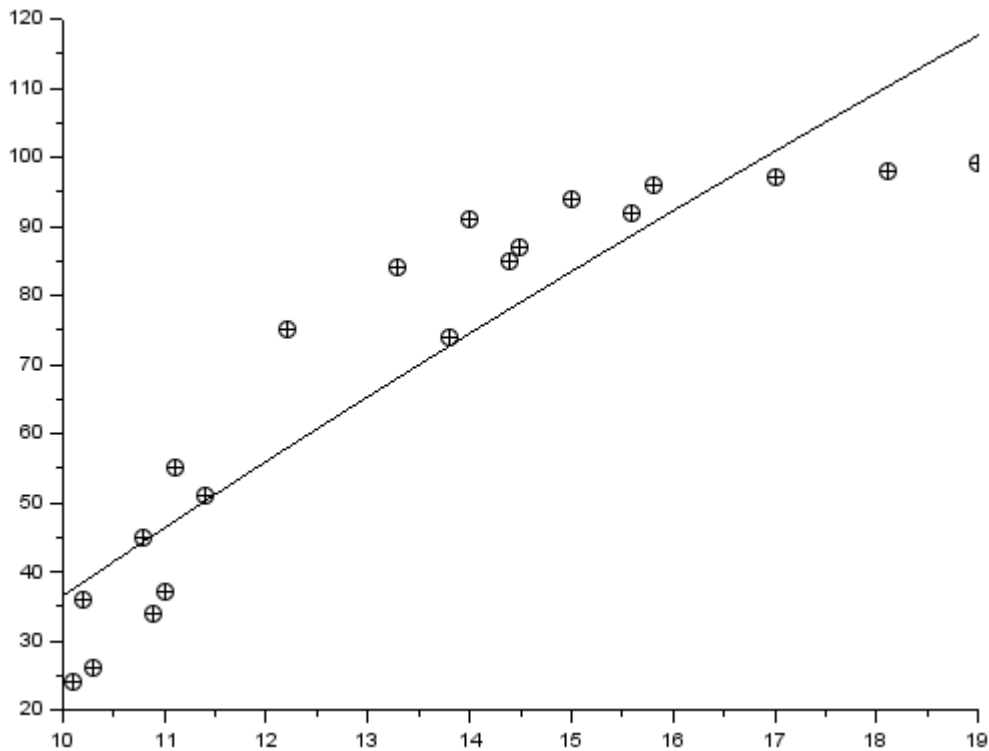


Figure 10.2 – Graphic solution of the task.

One of the most frequently used functions in the least squares method is a straight line described by an equation of the form $y = a_1 + x a_2$, which is called the y-x regression line. Parameters a_1 and a_2 are regression coefficients. The indicator characterizing the linear relationship between x and y is called the correlation coefficient and is calculated by the formula:

$$R = \sqrt{1 - \frac{\sum_{i=1}^n (y_i - Y_i)^2}{\sum_{i=1}^n (y_i - \bar{y})^2}}, \bar{y} = \frac{\sum_{i=1}^n y_i}{n}, Y_i = f(x_i).$$

The correlation coefficient satisfies the relationship $-1 \leq R \leq 1$. The less the absolute value of R differs from unity, the closer the experimental points are to the

regression line. If the correlation coefficient is close to zero, it means that there is no linear relationship between x and y , but a non-linear relationship may exist.

The analogue of the correlation coefficient r for nonlinear dependencies is the correlation index calculated by the formula:

$$r = \frac{\sum_{i=1}^n (x_i - \bar{x}) \cdot (y_i - \bar{y})}{\sqrt{\sum_{i=1}^n (x_i - \bar{x})^2 \cdot \sum_{i=1}^n (y_i - \bar{y})^2}}, \bar{x} = \frac{\sum_{i=1}^n x_i}{n}, \bar{y} = \frac{\sum_{i=1}^n y_i}{n}.$$

The correlation index in its absolute value ranges from 0 to 1. With a functional dependence, the correlation index is 1. In the absence of a relationship, $r = 0$. If the correlation coefficient r is a measure of the relationship only for a linear form, then the correlation index R - and for a linear, and for curved. In the case of a straight-line connection, the correlation coefficient in its absolute value is equal to the correlation index.

To calculate the regression coefficients in Scilab, the function is used:

[a,b,sig]=reglin(x,y)

here **x** and **y** are experimental data;

a – a_1 vector of the coefficient of the regression line $y = a_1 + x a_2$;

b – a_2 vector of the coefficient of the regression line $y = a_1 + x a_2$;

sig is the standard residual deviation.

Let's look at how this function works using an example. D.I. Mendeleev's «Fundamentals of Chemistry» provides data on the solubility of sodium nitrate NaNO_3 depending on the temperature of the water. It is required to determine the solubility of sodium nitrate at a temperature of 32 degrees in the case of a linear relationship and find the coefficient and index of correlation, if the number of conventional parts of NaNO_3 dissolving in 100 parts of water at the appropriate temperatures is shown in the table:

0	4	10	15	21	29	36	51	68
66.7	71.0	76.3	80.6	85.7	92.9	99.4	113.6	125.1

The program for this example and the results of its execution are as follows:

```
// Experimental data
x=[0 4 10 15 21 29 36 51 68];
y=[66.7 71 76.3 80.6 85.7 92.9 99.4 113.6 125.1];
// Calculation of the regression coefficients
[a2,a1,sig]=reglin(x,y)
// Solubility at 32 degrees
t=32;
a1+a2*t
// Correlation coefficient
```

```

r=sum((x-mean(x)).*(y-mean(y)))/...
sqrt(sum((x-mean(x)).^2).*sum((y-mean(y)).^2))
// correlation index
R=sqrt(1-sum((y-(a1+a2.*x)).^2)/sum((y-mean(y)).^2))
t=0:70; Yt=a1+a2.*t;
plot2d(x,y,-3); plot2d(t,Yt)

```

Calculation results:

```

a2 =
    0.8706404
a1 =
    67.507794
sig =
    0.8460731
ans =
    95.368287
r =
    0.0002826
R =
    0.9989549

```

The data graph and the regression equation graph are shown in figure. 10.3.

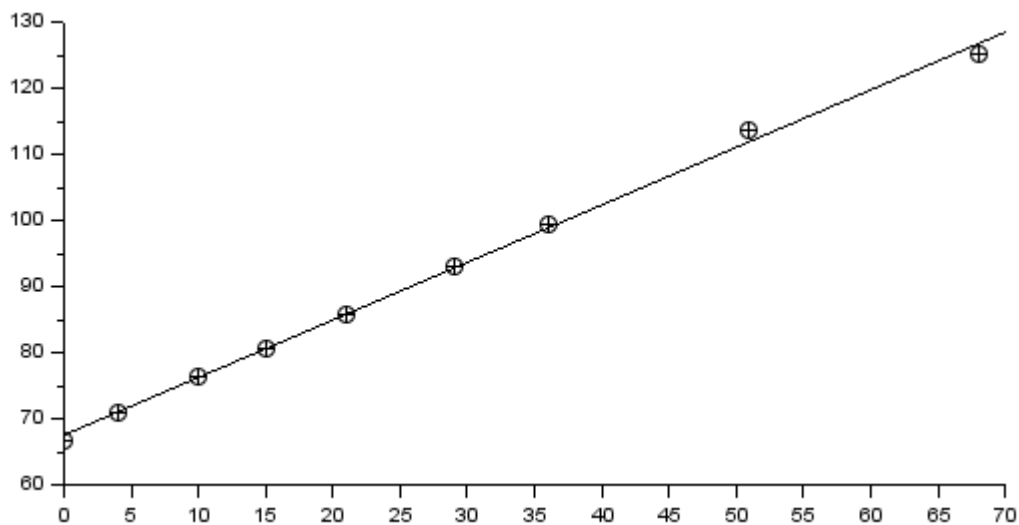


Figure 10.3 – The graphs of data and regression equation.

10.2 Function interpolation.

The simplest interpolation task is as follows. On the segment $[a, b]$, $x_0, x_1, x_2, \dots, x_n$ ($n + 1$ points in total) are given, which are called interpolation nodes, and the values of some function $f(x)$ at these points:

$$f(x_0) = y_0, f(x_1) = y_1, f(x_2) = y_2, f(x_n) = y_n.$$

It is required to construct an interpolating function $F(x)$ which is belonging to a known class and taking the same values at the interpolation nodes as $f(x)$:

$$F(x_0) = y_0, F(x_1) = y_1, F(x_2) = y_2, F(x_n) = y_n.$$

To solve such a task, spline interpolation is often used (from the English word spline - rake, ruler). One of the most common interpolation options is cubic spline interpolation.

In addition, there are quadratic and linear splines.

In Scilab, for linear interpolation is used **interpln** function

$$[\mathbf{y}] = \text{interpln}(\mathbf{xyd}, \mathbf{x})$$

here: **xyd** is matrix of original data;

x is the abscissa vector;

[y] is vector of linear spline values in points.

The following is an example of using the **interpln** function. Here a linear spline is used to solve the task of the power which is consumed from the network (P, W):

```
x=[132 140 150 162 170 180 190 200 211 220 232 240 251];
y=[330 350 385 425 450 485 540 600 660 730 920 1020 1350];
plot2d(x,y,-4);
z=[x;y];
t=132:5:252; ptd=interpln(z,t);
plot2d(t,ptd);
```

A graphic solution to the task is shown in figure 10.4.

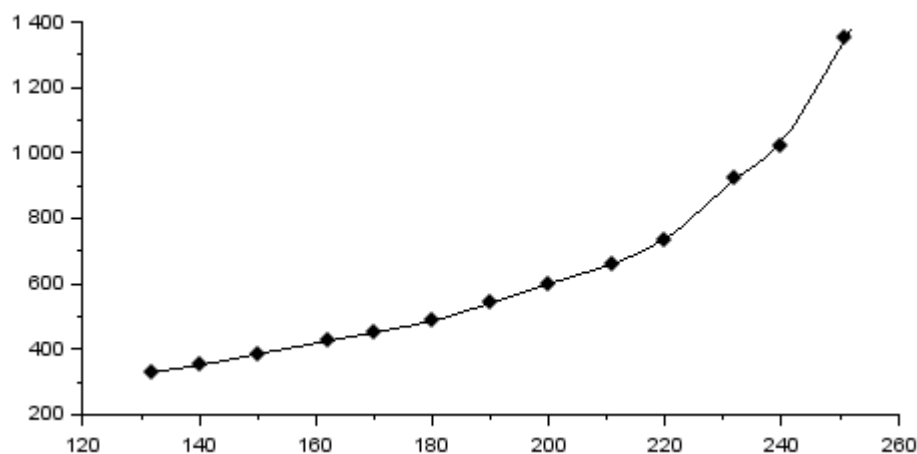


Figure 10.4 – Applying a linear spline to solve a task.

The construction of a cubic spline in Scilab consists of two stages: first, the coefficients of the spline are calculated using the function **d=splin(x,y)**, and then

the values of the interpolation polynomial at the point $\mathbf{y}=\text{interp}(\mathbf{t},\mathbf{x},\mathbf{y},\mathbf{d})$ are calculated.

The function for calculating the spline coefficients look like this:

$$\mathbf{d}=\text{splin}(\mathbf{x},\mathbf{y})$$

here: \mathbf{x} is a strictly increasing vector consisting of at least two components;

\mathbf{y} is a vector of the same format as \mathbf{x} ;

\mathbf{d} - the result of the function, the coefficients of the cubic spline.

The function which is calculating the values of the interpolation polynomial has the form:

$$\mathbf{Y}=\text{interp}(\mathbf{t},\mathbf{x},\mathbf{y},\mathbf{k})$$

here: \mathbf{x} is a strictly increasing vector consisting of at least two components;

\mathbf{y} is a vector of the same format as \mathbf{x} ;

\mathbf{Y} is the result of the function, the coefficients of the cubic spline;

\mathbf{t} - is a vector or matrix: abscissa, on which \mathbf{y} is unknown and must be estimated;

\mathbf{k} are the coefficients of the cubic spline.

Let's find the approximate value of the function for a given value of the argument using cubic spline interpolation at the points $x_1 = 0,702$, $x_2 = 0,512$, $x_3 = 0,608$. The values of function is set in a table:

0.43	0.48	0.55	0.62	0.7	0.75
1.63597	1.73234	1.87686	2.03345	2.22846	2.35973

The program for solving the task has the form:

```
x=[0.43 0.48 0.55 0.62 0.7 0.75];  
y=[1.63597 1.73234 1.87686 2.03345 2.22846 2.35973];  
plot2d(x,y,-4); // Experimental data graph.  
coeff=splin(x,y);  
x=[0.702 0.512 0.608]  
// Function value at given points.  
Y=interp(X,x,y,coeff)  
plot2d(X,Y,-3); // Plotting points on a graph.  
// Constructing a Cubic Spline.  
t=0.43:0.01:0.75;  
ptd=interp(t,x,y,coeff);  
plot2d(t,ptd);  
xgrid();
```

A graphic illustration of the task solution is shown in fig. 10.5.

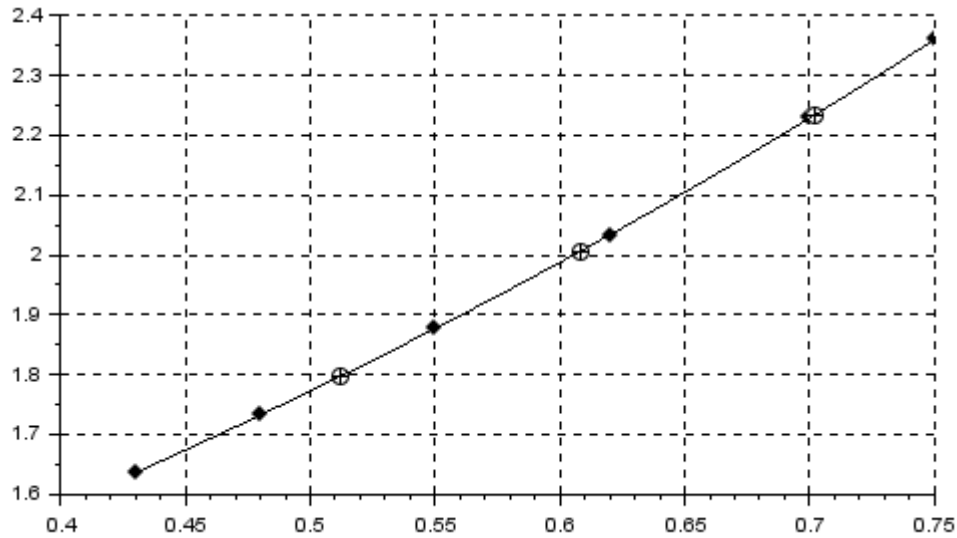


Figure 10.5 – Cubic spline interpolation.

Questions for self-examination for the fifteenth lecture:

1. How to create an edit field for entering a number?
2. How to create lists of strings?
3. What function implements the least squares method?
4. How to determine the coefficients of the linear regression equation?
5. What is the `interp` function used for?
- 6 How to implement cubic spline interpolation?

LITERATURE

1. Струтинський В.Б. Математичне моделювання процесів та систем механіки / Підручник .- Житомир : ЖІТІ, 2001.- 612 с.- ISBN 966-7570-94-0.
2. Учебное пособие «SciLab. Система компьютерной математики». <http://izido.ru/mod/book/tool/print/index.php?id=158>
3. Internet resource: <http://www.scilab.org/>
4. Internet resource: <https://help.scilab.org/>
5. Internet resource: <https://ru.wikibooks.org/wiki/Scilab>.

CONTENTS

PREFACE	3
1 FAMILIARIZATION WITH SCILAB PACKAGE	4
1.1. Installing the Scilab package on a computer.....	4
1.2 Start Scilab.....	6
1.3 The destination of the program windows.....	9
1.3.1 Command window or Console window (Scilab 6.1.1 Console).....	9
1.3.2 Variable Browser.....	10
1.3.5 Scilab help system.	12
1.3.6. Built-in SciNotes text editor.....	13
2 SCILAB PROGRAMMING LANGUAGE	15
2.1 Text comments.	15
2.2 Elementary mathematical expressions	16
2.3 Variables in Scilab.....	19
2.4 Scilab system variables	22
2.5 Dynamic typing of Scilab variables.	24
2.6 Entering a real number and displaying the results of calculations.....	24
2.7 Boolean type of variables.	26
2.8 Functions of Scilab	27
2.8.1 Elementary mathematical functions	27
2.8.2 User-defined functions.	29
3 ARRAYS AND MATRICES IN Scilab	32
3.1 Input and formation of arrays and matrices.	33
3.2 Actions with vectors and matrices.	39
3.3 Special matrix functions in Scilab.....	44
3.3.1 Functions for definition of matrix.	44
3.3.1.1 The function of converting a matrix v to a matrix of a different size:....	44
3.3.1.2 The function creating a matrix which is composed of ones:	45
3.3.1.3 The function creating a matrix which is composed of zero:	47
3.3.1.4 The function creating the identity matrix with undefined dimensions. ..	48
3.3.1.5 The function to create a matrix of random numbers.	50
3.3.1.6 Array ordering function:.....	52
3.3.2 Functions for calculating some numeric characteristics of matrices.	53
3.3.2.1 Function for determining the size of an array:	53
3.3.2.2 Function for determining the length of the matrix:	54
3.3.2.3 Function for calculating the sum of array elements:	55
3.3.2.3 Function for calculating the product of array elements:	56
3.3.2.4 Function for calculating the determinant of a square matrix.	56
3.3.2.5 Function for calculating the largest element in an array.	57
3.3.2.6 Function for calculating the smallest element in an array.	58
3.3.2.7 Function for calculating the average value of array elements.	59
3.3.2.8 Function of inverse matrix calculation.....	60
3.3.2.9 The function of converting a matrix to a triangular form.	60
3.4 Symbolic matrices and operations on them.	61
3.5 Solving systems of linear algebraic equations.	62

4 PLOTTING A SET OF 2D CURVES.	65
4.1 plot function.	65
4.2 Modification of graphs.	69
4.3 Plotting several graphs in one graphic window.	72
4.5 Figuration of the graphs using the plot2 function.	76
4.6 Plotting dot graphs using the plot2d function.	81
4.7 Plotting graphs in the form of a stepped line.	83
4.8 Plotting graphs in polar coordinate system.	83
4.10 Graph formatting mode.	86
4.10.1 Formatting object «Figure(1)».	88
4.10.2 Formatting the Axes object.	91
4.10.3 Polyline object formatting.	99
5 PLOTTING THREE-DIMENSIONAL CHARTS IN Scilab.	102
5.1 Functions plot3d and plot3d1 .	103
5.2 meshgrid , surf and mesh functions.	107
5.3 plot3d2 и plot3d3 functions.	110
5.4 param3d and param3d1 commands.	113
5.5 contour function.	115
5.6 contourf function.	119
5.7 hist3d function.	122
6 NONLINEAR EQUATIONS AND SYSTEMS IN SCILAB.	123
6.1 Algebraic equations.	123
7 NUMERICAL INTEGRATION AND DIFFERENTIATION.	133
7.1 Integration by the method of trapezoids.	133
7.2 Integration by quadrature.	135
7.3 Integration of an external function.	136
7.4 Calculation of the derivative.	137
7.5 Calculation of the partial derivative.	138
8 PROGRAMMING IN Scilab.	138
8.1 Basic operators of SciLab language.	139
8.1.1 Input and Output functions in Scilab.	139
8.1.3 Loop conditional operator if .	142
8.1.4 Alternative select operator.	145
8.1.5 The while loop operator.	147
8.1.6 The non-conditional loop operator for .	148
8.2 An example of working with arrays.	149
8.3 Scilab functions for working with files.	150
8.3.1 File opening function mopen	151
8.3.2 The mfprintf function of writing a text to file.	152
8.3.3 Function mfscanf for reading data from a text file.	153
8.4 Sample program in Scilab.	156
9 CREATING GRAPHIC APPLICATIONS IN THE Scilab.	159
9.1. Working with the graphic window.	159

9.2 Dynamic creation of interface elements and description of the main functions.	161
.....	161
9.2.1 Command button.	163
9.2.2 Label.	165
9.2.3 Checkbox and radiobutton components.	167
9.2.4 Editing window component.	171
9.2.5 Lists of strings.	173
10. PROCESSING OF EXPERIMENTAL DATA	176
10.1 Least square method	176
LITERATURE	185