Ministry of Education and Sciense of Russian Federation

National Research Nuclear University «MEPhI»
(Moscow Engineering Physics Institute)

A.V. Krasavin, Ya.V. Zhumagulov

Practical course on MatLab for foreign students

Moscow 2018

УДК [(530.145+538.9):004.41](076.5)
ББК 22.314я7
К78

Krasavin A.V., Zhumagulov Ya.V. **Practical course on MatLab for foreign students**. [Electronic sourse] M.: NRNU MEPhI, 2018. 268 p.

The book discusses the basic techniques of programming in MatLab environment as applied to solving problems of condensed matter physics. The manual can be used as an introductory practical course of numerical simulation of quantum physical systems. The book is supplied with a lot of examples, illustrations and exercises.

Designed for students specializing in condensed matter physics. The book can also be useful for students and postgraduates of other physical specialties, as well as for teachers and specialists dealing with condensed matter physics.

Reviewer: Dr. Prof. Vladimir A. Kashurnikov

ISBN 978-5-7262-2453-4

Contents

1

Introsuction to MatLab

1.1. Simple calculations

In this chapter and later in the book, the typewriter font of the text means MatLab commands that can be typed directly in the command window.

`demo` overview of MatLab capabilities

Help information about any function can be obtained with the command `help <function name>`

Operators

`+` – addition;
`–` – subtraction;
`*` – multiplication;
`/` – division;
`^` – exponentiation;
`.'` – matrix transpose;
`'` – matrix transpose with complex conjugation.

Functions

`abs` – absolute value;
`sqrt` – square root;
`log` – natural logarithm;
`sin` – sine;
`cos` – cosine;
`tan` – tangent;
`sinh` – hyperbolic sine;
`fix` – rounding to the nearest integer towards zero;
`mod(x,y)` – remainder of dividing x by y.

Help about elementary functions: `help elfun`
Help about special functions: `help specfun`
Help about matrix functions: `help elmat`

Useful constants

`pi` – number of pi;
`i` – imaginary unit;
`j` – imaginary unit;
`eps` – relative accuracy of floating-point calculations;
`realmin` – the smallest floating-point number, 2^{-1022};
`realmax` – the largest floating-point number, $(2 - \text{eps})2^{1023}$;
`Inf` – infinity (`1/0 = Inf`);
`NaN` – not a number (for example, `0/0 = NaN`).

Example 1.1. Compute:

$$e^{-2.5}(\ln 11.3)^{0.3} - \sqrt{\frac{\sin(2.45\pi) + \cos(3.78\pi)}{\operatorname{tg}(3.3)}}.$$

Solution.

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

Answer:

```
ans = -3.2105
```

The ellipsis "..." means the continuation line; by default, the result of the calculation is stored in a variable named `ans`.

Variables

Variables are formed using the assignment operator "=", for example:

```
w = 3.54;
```

Lowercase and uppercase letters are different, for example, the variables `t1` and `T1` are two different variables. **The semicolon at the end of the line** is used to suppress the output of the calculation result on

the screen. **Comments** in MatLab are entered with the "`%`" command, for example:

```
c = a + b % calculation of the sum of vectors a and b
d = sum(a.*b) % calculation of the scalar product of
                vectors a and b
```

1.2. Arrays

Column-vectors ans row-vectors

```
a = [1.3; 5.4; 6.9] % column-vector a
b = [7.1; 3.5; 8.2] % column-vector b
c = a + b % sum of a and b

a = [1.3 5.4 6.9] % row-vector a
b = [7.1 3.5 8.2] % row-vector b
c = a + b % sum of a and b

size (c) % dimensions of array c
```

`d = sin(a)` – calculation the values of function `sin` for **all elements of array `a`**.

Access to elements of a vector

```
v = [1 2 6 4 8 34];
v(4) % access to the fourth element of vector v
v(2) = 100 % assigning new value to the second
element
v1 = [5 4 3 2 1];
v2 = [v v1] % forming new vector from two row-vectors
v(2:4) % access to elements of the vector from the
second to the fourth
v1(3:end) % access to elements of the vector v1 from
the second to the last
v3 = [v(2:4) v1(1:4)] % forming new vector from parts
of v and v1
ind = [4 2 5];
```

```
v4 = v(ind) % formation of vector by indexing
```

Application of data processing functions to vectors

```
v = [5 2 4 3 1];
p = prod(v) % multiplication of vector elements
s = sum(v) % sum of vector elements
l = length(v) % length of vector
mean(v) % arithmetic average of vector elements
% mean(v) = sum(v)/length(v)
M = max(v) % the maximum of vector v
m = min(v) % the minimum of vector v
[m, k] = min(v) % m the minimum of vector v, k is its
place in v
R = sort(v) % sorting of vector elements
[R, ind] = sort(v) % sorting with the output of the
array of correspondence
```

Elementwise operations with vectors

```
v1 = [2 -3 4 1];
v2 = [7 5 -6 9];
u = v1.*v2 % elementwise multiplication of vectors of
the same length
p = v1.^2 % elementwise exponentiation of the vector
v1
P = v1.^v2
d = v1./v2
s = v1 + 4
q = v2/2
```

`Q = v1*v2` – multiplication of vector by vector; **in this case will lead to an error**, since row-vectors can be multiplied only by column-vectors of the corresponding length.

```
V2 = v2' % transpose of v2, we obtain the column-
          vector V2
Q = v1*V2 % now the matrix multiplication is possible
```

Table of values of function

Example 1.2. Obtain the table of values of the function $f(x) = \sin x \cos^2 3x$ on the segment from 0 to 2π n increments of 0.01.

`x = [0 : 0.01 : 2*pi]` – creation of a row-vector containing the coordinates of the given points; an operation of the form `a:b:c` specifies an array of elements from `a` to `c` with step `b`.

`y = sin(x).*cos(3*x).^2` – creation of a row-vector containing the values of the function $f(x)$ at the points `x`. **Pay attention to the elementwise operations!**

Plotting a function of one variable

Plotting a function of one variable consists of the following steps.

1. Specifying the vector of the values of the argument x:

```
x = [0 : 0.01 : 2*pi];
```

2. Calculation of the vector y of the values of the function $y(x)$:

```
y = sin(x).*cos(3*x).^2;
```

3. Using `plot` command to plot the graph:

```
plot(x,y)
```

1.3. Matrices

Special matrices

`A = zeros(3,4)` – creation of zero matrix;

`I = eye(4)` – creation of identity matrix;

`E = ones(2,6)` – creation of matrix of ones;

`R = rand(3)` – creation of a square matrix filled with random numbers uniformly distributed between 0 and 1;

`Q = randn(3,4)` – creating a 3×4 matrix filled with random numbers distributed according to the normal law.

The `diag` function serves to isolate the diagonal of the matrix into a vector:

`d = diag(R)` – the main diagonal;

`d1 = diag(R,-1)` – the diagonal located one position down from the main diagonal.

The `diag` function also generates a diagonal matrix from the vector:

```
D = diag(d)
D1 = diag(d,2)
```

You can create a matrix directly in the command line:

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

Matrix operations

```
B = [4 3 -1; 2 7 0; -5 1 2]
E = ones(3)
A = B + E % addition of matrices
A = B – E % subtraction of matrices
A = 3*B % multiplication of matrix by a number
A = B^2 % exponentiation of the matrix B
A = B.^2 % exponentiation of each element of the
          matrix B
```

Selecting part of matrix

```
B = rand(4)
P = B(2:3, 2:3)
P = B(2,:) % selection of the second row
P = B(:,3) % selection of the third column
P = B(2, 2:end) % elements of the second row from the
                second to the last
```

Deleting rows and columns

```
P = B
size(P) % size of array P
P(1,:) = [] % deleting the first row
```

```
P(:,3) = [] % deleting the third column
size(P) % the size of array P is changed
B(:, 2:3) % column deletion from the second to the
third
```

Matrix visualization

Analyze the commands used in constructing the matrix G, and look at the result:

```
G = 2*eye(7)+diag(ones(1,6),1)+diag(ones(1,6),-1)
```

Portrait of the matrix: non-zero elements are marked; the total number of non-zero elements is indicated below the figure:

```
spy(G)
```

Plots of functions of two variables

The plotting includes two preliminary stages.

1. Partition of the domain of definition by a rectangular grid.
2. Calculating the values of the function at the grid nodes and writing them to the matrix.

Example 1.3. Construct a plot of the function $f(x,y) = x^2 + y^2$ in the square $[0,1] \times [0,1]$.

Generation of the grid array on the square$[0,1] \times [0,1]$ in increments of 0.2:

```
[X,Y] = meshgrid(0:0.2:1, 0:0.2:1)
```

Calculation of function values at grid nodes:

```
Z = X.^2 + Y.^2
```

Construction of the graph in the form of a wireframe surface:

```
mesh(X, Y, Z)
surf(X, Y, Z) % filled wireframe surface
shading interp % smooth fill
```

For a more accurate plotting, you may select a smaller grid step.

1.4. M-files

The most convenient way to execute MatLab commands is to use M-files, in which you can type commands, execute them all at once or in parts, save them in a file and use them in the future. To work with M-files, the M-file editor is designed. It is launched from the menu: File → New → M-file.

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

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

Save the file as, for example, `mydemo.m` in the default directory. To execute all the commands in the file, select Run from the menu.
Replace the function `g` with `cos(x)`, save the file and run all commands again.

Types of M-files

There are two types of M-files: script M-files and function M-files.

The script M-file has just been created for plotting the two functions. All variables declared in the script file become available in the workspace after it is executed. It is convenient to run the script file from the command line; to do so, just type the name of the script file and

press Enter. For example, by typing `mydemo` in the command line, you can run the newly created M-file again. After entering `mydemo` command, MatLab performs the following actions.

1. 1. Checks whether the entered command coincisdes with any variable defined in the workspace. If so, the value of this variable is displayed.

2. If there is no such variable, the entered command is searched for among the built-in functions. If the entered command is a built-in function, it is executed.

3. If there is no such built-in function, then MatLab searches for the M-file with the name of the entered command and the extension m. The search starts from the current directory; if the M-file is not found, the search continues in the directories from the list of set paths.

The list of paths is available from the menu: File → Set Path.

Function files

Example 1.4. Write an M-file for calculating function values:

$$f(x) = e^{-x}\sqrt{\frac{x^2+1}{x^4+0.1}}.$$

Solution. In the M-file editor, you need to create the following file:

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

Here `myfun` is the name of the function; `x` is the input argument; `f` is the output argument (the function value); elementwise operations in the expression for the function are used so that you can access the function with an input argument of the vector type.

Next, save the file under **the same name** `myfun` **as the function name** (it will be offered by default).

Run the following commands from the command line:

```
x = [0:0.05:4]
```

```
y = myfun(x)
plot(x, y)
```

Example 1.5. Write an M-file for calculating the sum and product of two square matrices.

Solution. There are two input and two output arguments in this task. The M-file may be as follows:

```
function [S, P] = sum_prod(A,B)
% A, B: input matrices
% S: sum of matrices
% P: product of matrices
S = A + B;
P = A*B;
```

Save the file as `sum_prod.m` and execute the following commands from the command line:

```
I = eye(5)
E = ones(5)
[A1,A2] = sum_prod(I,E)
```

1.5. Loop statements

For loop

```
for count = start : increment : final
      MatLab commands
end
```

Here `count` is the loop variable; `start` is its initial value; `final` is the final value; `count` is the incremented by the value of `increment` at each loop execution. The loop terminates as soon as the value of `count` is greater than `final`. The loop variable can take real values of any sign.

While loop

```
while condition
      MatLab commands
end
```

The loop is executed as long as the loop `condition` is satisfied. Relationships and logical operators are used to specify conditions:

- == is equal;
- < less than;
- > greater than;
- <= less or equal;
- >= greater or equal;
- ~= not equal;
- & logical «AND»;
- | logical «OR»;
- ~ negation.

For example, the condition $x < 3$ and $k = 4$ will be written as `(x<3)&(k==4)`.

The loop is interrupted by the `break` statement, and then the command is executed following the `end` statement, which marks the end of the current cycle. In the case of nested loops, `break` interrupts the inner loop.

Selection statement if

```
if condition
      MatLab commands
end
```

If the condition is true, then the commands between `if` and `end` are executed; otherwise the command after `end` statement is executed.

A more general form of the `if` statement:

```
if condition
      MatLab commands
```

```
elseif condition
      MatLab commands
else
      MatLab commands
end
```

Practice.

1.1. Construct the matrix:

$$M = \begin{pmatrix} 8 & 0 & 0 & 0 & 0 & 0 & 0 & 4 & 0 & 0 & 0 & 0 \\ 0 & 8 & 0 & 0 & 0 & 0 & 5 & 0 & 4 & 0 & 0 & 0 \\ 0 & 0 & 8 & 0 & 0 & 0 & 0 & 5 & 0 & 4 & 0 & 0 \\ 0 & 0 & 0 & 8 & 0 & 0 & 0 & 0 & 5 & 0 & 4 & 0 \\ 0 & 0 & 0 & 0 & 8 & 0 & 0 & 0 & 0 & 5 & 0 & 4 \\ 0 & 0 & 0 & 0 & 0 & 8 & 0 & 0 & 0 & 0 & 5 & 0 \\ 3 & 3 & 3 & 3 & 3 & 3 & 0 & 7 & 0 & 7 & 0 & 7 \\ 3 & 3 & 3 & 3 & 3 & 3 & 7 & 0 & 7 & 0 & 7 & 0 \\ 3 & 3 & 3 & 3 & 3 & 3 & 0 & 7 & 0 & 7 & 0 & 7 \\ 3 & 3 & 3 & 3 & 3 & 3 & 7 & 0 & 7 & 0 & 7 & 0 \\ 3 & 3 & 3 & 3 & 3 & 3 & 0 & 7 & 0 & 7 & 0 & 7 \\ 3 & 3 & 3 & 3 & 3 & 3 & 7 & 0 & 7 & 0 & 7 & 0 \end{pmatrix}.$$

1.2. Write a function file that calculates the factorial of a positive integer number.

1.3. Make the same plot as in Fig. 1.1. The figure on the left is *Bernoulli lemniscate:*

$$x = a\frac{t + t^3}{t^4 + 1}; y = a\frac{t - t^3}{t^4 + 1};\ t \in [-\infty; \infty]; a = 1; 2; 3; \dots; 10.$$

The figure on the right is *Maclaurin trisector:*

$$r = \frac{a}{\cos\left(\frac{\varphi}{3}\right)};\ \ \varphi \in \left[-\frac{5\pi}{4}; \frac{5\pi}{4}\right];\ \ a = 0.25; 0.5; 0.75; \dots; 2.5.$$

The plot in polar coordinates is constructed by `polar` command.

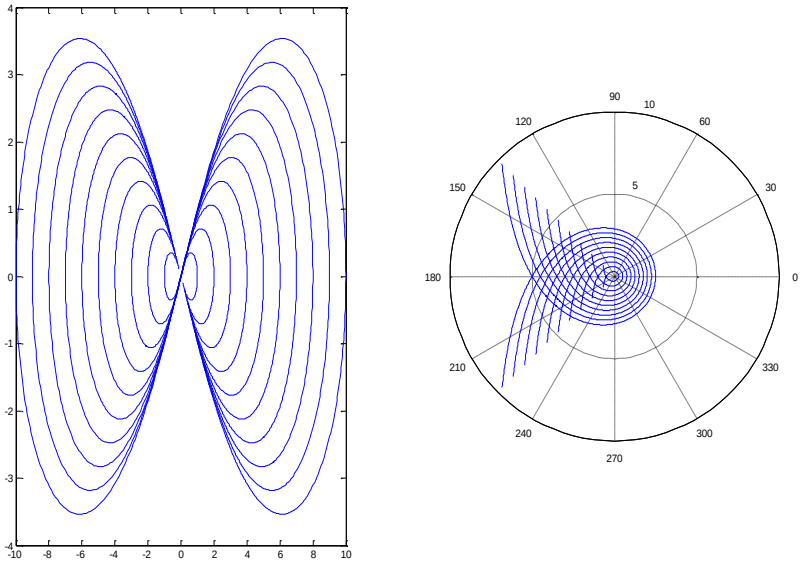


Fig. 1.1. Bernoulli lemniscate and Maclaurin trisector

1.4. Compute without using loops:

$$\sum_{k=1}^{20} \frac{\left(1+e^{1/k}(1+k^3)\right)\left(a_k^2+b_k^2\right)}{a_k b_k+\pi^{1/k}k^2};\ a_k = k\cos k;\ \ b_k = k^2\sin^2 k.$$

2

Vectors and matrices

2.1. Creating matrices

The simplest way to create matrices or vectors in MatLab is to define the components of a vector or a matrix:

```
x = [ 3 5 8 6 ]
```

The above command will result in the creation of a row-vector `x` containing four elements. To create column-vectors, separate the components with a semicolon:

```
x = [ 3; 5; 8; 6 ]
```

Accordingly, you can create a matrix in this way:

```
A = [ 1 5 7; 6 8 3; 2 2 0]
```

There are several ways to create matrices more efficiently.

To create vectors whose neighboring components differ by a given value, it is convenient to use the colon "`:`" statement. For example, the following commands

```
a = [1 : 3 : 16];
b = [100 : -10: 70];
c = [1:10];
```

will result in creation of vectors $a = (1, 4, 7, 10, 13, 16)$; $b = (100, 90, 80, 70)$ and $c = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)$. The colon statement has the following format:

```
initial value : increment : final value
```

The `increment` can be omitted; in this case it will be equal to one.

Another statement that allows to create vectors whose components differ by a sertain value, is `linspace`. It is used in the case when it is known not the difference between neighboring components, but the number of components on a given segment. For example, the command

```
x = linspace(0,2*pi,100)
```

will result in creation of a grid of 100 points equidistant from each other on the interval $[0, 2\pi]$.

To create special matrices, MatLab provides a large number of built-in functions. Among the most commonly used are `ones` for creating a matrix of ones, `zeros` for creating a matrix of zeros, and `eye` for creating an identity matrix, i.e. the matrix that have units on the main diagonal, and the remaining elements are zero. The format of these functions is as follows:

```
A = zeros(m,n)
```

Here `m` is the number of rows of the matrix; `n` is the number of columns. If the argument contains only one variable `m`, then a square matrix of size $m \times m$ is created. For example, the commands

```
W = ones(3);
E = eye(2,3);
```

will result in creation of matrices

$$W = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$$

and

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

Matrices can also be created in a block fashion with the use of existing matrices. For example, the command

```
A = [2*ones(2) eye(2); eye(2) [4 4; 3 3]];
```

creates the following matrix:

$$A = \begin{pmatrix} 2 & 0 & 1 & 0 \\ 0 & 2 & 0 & 1 \\ 1 & 0 & 4 & 4 \\ 0 & 1 & 3 & 3 \end{pmatrix}.$$

2.2. Dimensions of matrices

To determine the size of the matrix, the `size` statement is used. It returns a 1×2 vector with the number of rows and the number of columns of the original matrix. The following example illustrates the use of `size` statement.

Example 2.1.

```
A = ones(3,4); % creation of 3x4 matrix of ones
Z = size(A); % the dimensions of the matrix A are written in the
             vector Z: Z(1)=3; Z(2)=4
[r c] = size(A); % here the function size is called with two
             output arguments, so the number of rows will be
             written to the variable r, and the number of columns
             to the variable c: r=3; c=4
```

In MatLab, both vectors and even scalars are represented by matrices (the vector is a matrix of size $m \times 1$ or $1 \times m$, where m is the number of components, and the scalar is a 1×1 matrix). To determine the length of a vector, it is more convenient to use the `length` operator, which compute the number of rows and the number of columns of the input matrix and returns only one parameter, the largest of these values.

2.3. Access to matrix elements

To access to a single element of a matrix, the row and the column numbers of this element in the matrix should be specified: `z(2,3)`. The first argument is the row number, the second is the column number. The same statement can be used to assign a new value to a particular matrix

element (`z(2,3)=50`) or to display its value on the screen (`disp(z(2,3))`).

To access several neighboring matrix elements, use the "colon" statement. Its use is illustrated in Example 2.2.

Example 2.2.

```
Z = rand(6); % creation of 6x6 matrix filled with random numbers

Z =

    0.2438    0.4152    0.7236    0.6971    0.4762    0.9544
    0.0997    0.8191    0.4998    0.5970    0.1135    0.4516
    0.5606    0.6758    0.9471    0.9047    0.5157    0.5333
    0.0857    0.0464    0.9361    0.0330    0.8682    0.1265
    0.2993    0.6715    0.9491    0.8230    0.4606    0.9051
    0.2823    0.3165    0.7623    0.3662    0.3915    0.8013

 A = Z(2,2:4) % A is vector consisting of elements of the matrix Z
                in the second row and columns from the second to
                the fourth
 A =

     0.8191    0.4998    0.5970

 B = Z(1:2,3:5) % B is matrix consisting of elements of the matrix
                Z in the rows from the first to the second and in
                the columns from the third to the fifth
 B =

     0.7236    0.6971    0.4762
     0.4998    0.5970    0.1135

 C = Z(:,2) % C is vector consisting of the elements of the matrix
                Z in the second column
 C =

     0.4152
     0.8191
     0.6758
     0.0464
     0.6715
     0.3165

 D = Z(2:3,:) % D is matrix consisting of elements of the matrix Z
                in rows from the second to the third; the second
                colon operator in this case means "all columns"
```

```
D =

    0.0997    0.8191    0.4998    0.5970    0.1135    0.4516
    0.5606    0.6758    0.9471    0.9047    0.5157    0.5333

E = Z(4:end,:) % E is matrix consisting of elements of the matrix
                Z in rows from the fourth to the last; the end
                statement in this case indicates the last element
                of the range

E =

    0.0857    0.0464    0.9361    0.0330    0.8682    0.1265
    0.2993    0.6715    0.9491    0.8230    0.4606    0.9051
    0.2823    0.3165    0.7623    0.3662    0.3915    0.8013

Z(:,2:4) = [] % remove from the matrix Z columns from the second
                to the fourth

Z =

    0.2438    0.4762    0.9544
    0.0997    0.1135    0.4516
    0.5606    0.5157    0.5333
    0.0857    0.8682    0.1265
    0.2993    0.4606    0.9051
    0.2823    0.3915    0.8013
```

2.4. Elementwise operations with matrices

In addition to usual operations of matrix algebra (addition, subtraction and multiplication of matrices), *elementwise* operations are provided in MatLab. Elementwise operations on matrices assume that the matrices have the same dimensions. Example 2.3 illustrates the use of elementwise operations.

Example 2.3.

```
A = 12*ones(2) % input of matrix A

A =

    12    12
    12    12

B = [1 2; 3 4] % input of matrix B
```

```
B =

     1     2
     3     4

C = A./B % the matrix A is elementwise divided by the matrix B:
           each element of the matrix C is the corresponding
           element of the matrix A divided by the corresponding
           element of the matrix B

C =

    12     6
     4     3

D = A.*B % the matrix A is elementwise multiplied by the matrix B:
           each element of the matrix C is the corresponding
           element of the matrix A multiplied by the
           corresponding element of the matrix B

D =

    12    24
    36    48

E = A*B % usual multiplication of matrices according to the rules
          of linear algebra

E =

    48    72
    48    72

F = A^2 % the matrix A is squared, i.e. multiplication of the
          matrix A by itself

F =

   288   288
   288   288

G = A.^2 % elementwise squaring of the matrix A; elementwise
           operation thus replaces loops across all rows and columns

G =

   144   144
   144   144
```

2.5. Functions for working with matrices and vectors

The `sum` function sums all the components of a vector, and the `prod` function multiplies them:

```
sum([2 4 6])
      ans = 12
prod ([2 4 6])
      ans = 48
```

If we apply these functions to the matrix, and not to the vector, then all elements of the matrix *in each column* will be summed or multiplied. In order to find the sum or product of all elements of the matrix, you should apply the function twice:

```
M = magic(3)

M =
     8     1     6
     3     5     7
     4     9     2

sum(M)

ans =
    15    15    15

sum(sum(M))

ans =
    45
```

To find the vector minimum or maximum, the functions `min` and `max` are used. Likewise, if the input element for these functions is a matrix, rather than a vector, the functions produce a result for each column. These functions can also be used to determine the position of the maximum or minimum element in a vector or matrix. To do this, you need to address the function with two output arguments:

```
a = [3 9 5 7 4 1 2]

a =
     3     9     5     7     4     1     2

[m,b] = max(a)

m =
     9 % maxsimum element of vector a

b =
     2 % position of the maximum element in vector a
```

Example 2.4 contains one of the possible programming codes for determining the position of the maximum element in the matrix.

Example 2.4.

```
v = [2 6; 4 1; 0 3] % input of matrix
v =
     2     6
     4     1
     0     3
[BigCol, RowCol] = max(v)

BigCol =
     4     6 % maximum element in each column

RowCol =
     2      1 % the number of row which contains the corresponding
                maximum element
[Big,Col] = max(BigCol)

Big =
     6 % maximum element in the entire matrix

Col =
     2 % the number of column which contains the maximum element
        in the entire matrix
Row = RowCol(Col) % the number of row which contains the maximum
                    element
Row =
     1
```

Elements of a vector or matrix can be sorted using the `sort` function:

```
a =
     3     9     5     7     4     1     2

[b,c]=sort(a)

b =
     1     2     3     4     5     7     9 % sorted vector

c =
     6      7      1      5      3      4      2 % positions of the
elements of the sorted vector b in the original unsorted vector a
```

If the input element is a matrix, the `sort` function performs sorting in each column.

2.6. Filtering and logical indexing

Often there is a need of choosing from elements of a matrix or a vector such elements that satisfy a certain condition. The following examples illustrate possible solutions to this problem.

Example 2.5.

```
B = [1 5; 7 9] % input of matrix

B =
     1     5
     7     9
C  =  (B>5)  % creation of a logical matrix; unity means that the
             corresponding element of matrix B is greater than 5

C =
     0     0
     1     1

D  =  B(B>=5)  % selection from matrix B of all elements that are
              greater than or equal to 5, and writing them to vector
              D

D =
     7
```

```
     5
     9

B(B>=5)=13  %  assignment  to  all  elements  of  matrix  B  that  are
          greater than or equal to 5, the value of 13

B =
     1    13
    13    13
```

Example 2.6. Consider the `find` function:

```
A = [6 3 8 2 1]; % input of vector
loc = find(A>5) % finding the positions of all elements of vector
                 A that are greater than 5

loc =
     1       3 % the first and the third element in vector A are
               greater than 5
```

Practice.

2.1. Create the matrix:

$$M = \begin{pmatrix} 1 & 0 & 0 & 2 & 2 & 2 & 1 & 3 & 5 \\ 0 & 1 & 0 & 2 & 2 & 2 & 7 & 9 & 11 \\ 0 & 0 & 1 & 2 & 2 & 2 & 13 & 15 & 17 \\ 0 & 0 & 0 & 9 & 8 & 7 & 4 & 3 & 3 \\ 0 & 0 & 0 & 6 & 5 & 4 & 3 & 4 & 3 \\ 0 & 0 & 0 & 3 & 2 & 1 & 3 & 3 & 4 \\ 3 & 4 & 4 & 7 & 0 & 0 & 5 & 2 & 2 \\ 4 & 3 & 4 & 0 & 7 & 0 & 2 & 5 & 2 \\ 4 & 4 & 3 & 0 & 0 & 7 & 2 & 2 & 5 \end{pmatrix}.$$

2.2. Create a tridiagonal matrix using the function `diag`:

$$M = \begin{pmatrix} 2 & 3 & 0 & 0 & 0 & 0 \\ 3 & 2 & 3 & 0 & 0 & 0 \\ 0 & 3 & 2 & 3 & 0 & 0 \\ 0 & 0 & 3 & 2 & 3 & 0 \\ 0 & 0 & 0 & 3 & 2 & 3 \\ 0 & 0 & 0 & 0 & 3 & 2 \end{pmatrix}.$$

2.3. Using the `rand` function, create a 6×6 square matrix of random numbers and find the sum of all elements in rows 4 through 6 and in columns 4 through 6.

2.4. There are two vectors:

$$a = (2\ 4\ 6\ 8\ 10);$$

$$b = (1\ 4\ 7\ 10\ 13).$$

Compute

$$\sum_i a_i b_i\,;\quad \prod_i a_i b_i\,;\quad \sum_i a_i^2 b_i^3\,;\quad \sum_i \frac{e^{a_i^2}}{1+e^{b_i^2}}.$$

2.5. Using the `rand` function, create a square 10×10 matrix of random numbers. Find the sum of all matrix elements that are greater than 0.5. Determine the position of these elements in the matrix.

3

Selection statements

3.1. If

Selection statements are present in almost all programming languages and allow the creation of algorithms in which certain groups of commands are performed under certain conditions.

The `if` statement determines whether the commands within its structure are executed during the program. The structure of the `if` statement is as follows:

```
if condition
   statements
end
```

`condition` in this structure is a condition that determines the execution or non-execution of a group of commands `statements`; it can take one of two logical values: true or false. When the algorithm reaches the line containing the `if` statement, it checks the `condition`. If it is true, the algorithm proceeds to execute `statements` that are part of the structure of the selection statement; if it is false, the algorithm proceeds to the execution of statements following the `end` statement pointing to the end of the selection statement.

Example 3.1.

```
if number < 0
   number = 0
end
```

In the above code the `number` is compared to zero, and if it is less than zero, the variable `number` gets a new value of 0. Enter the following lines in the MatLab command window:

```
number = -1
if number < 0
   number = 0
```

```
end
```

After executing these operators, the result will appear in the command window:

```
number =
      0
```

Change the initial value of `number` and enter the following:

```
number = 1
if number < 0
   number = 0
end
```

After executing these operators, no messages will appear in the command window, since the `number < 0` condition in this case takes the value `false`, and the statement `number = 0` is not executed.

3.2. If-else, elseif

If an algorithm requires selection of more than two variants, then `if-else`, `elseif` statements are used.

The structure of `if-else` statement is as follows:

```
if condition
   statements1
else
   statements2
end
```

First, the `condition` is checked. If it is true, a group of `statements1` is executed, followed by `end` statement. If it is false, then the group of `statements2` is executed, followed by `end` statement.

Example 3.2.

```
if a == 0
   disp('a is equal zero') % displaying a message on the screen
else
   disp('a is not equal zero')
end
```

`if-else` structures can be nested within each other.

Example 3.3. Consider a piecewise continuous function defined by the following expression:

$$y(x) = \begin{cases} 1, if\ x < -1; \\ x^2, if\ -1 \le x \le 2; \\ 4, if\ x > 2. \end{cases}$$

Using `if` statement, this function can be specified as follows:

```
if x < -1
   y = 1;
end

if x >= -1 && x <= 2 % when checking conditions, the logical
                     operators AND (&) and OR (|) are written twice
   y = x^2;
end

if x > 2
   y = 4;
end
```

This code is obviously not effective, since it assumes that all three conditions for `if` statements are checked, regardless of the value of x. However, if it turns out that $x < -1$, i.e. the first condition is true, the need to check the remaining conditions will disappear, and processing of the two remaining `if` statements will turn into a loss of time. The code can be made faster by using `if-else` statements:

```
if x < -1
   y = 1;
else
   % into this branch the algorithm comes only if x > = -1
   if x <= 2
      y = x ^2;
   else
      % if the algorithm comes to this branch, then x > 2
      y = 4;
   end
end
```

In this example, two `if-else` structures were used, one external and one nested.

In Example 3.3, the algorithm was optimized by using two `if-else` constructions instead of three `if` statements. This algorithm can be

implemented in a third way, with the use of operator `elseif` MatLab statement:

```
if condition1
   operators1
elseif condition2
   operators2
elseif condition3
   operators3
   % it can be any number of branches
else
   operatorsn % the branch into which the algorithm
              comes if none of the previous
              conditions are satisfied
end
```

In this structure, the algorithm sequentially checks the conditions `condition1`, `condition2`, `condition2`, ... until it finds a condition that is true. In this case, the group of commands corresponding to this condition will be executed, after which the transition to the `end` statement will occur, which means the end of the `elseif` structure. If all conditions are false, a group of `operatorsn` commands will be executed, corresponding to the `else` statement, after which the algorithm goes to `end` statement. When using `elseif` structure, `end` operator is used only once. Example 3.4 shows how you can rewrite the code from Example 3.3 using the `elseif` structure. Example 3.5 demonstrates the use of this structure to check whether the input argument is a scalar, vector, or matrix.

Example 3.4.

```
if x < -1
   y = 1;
elseif x <= 2
   y = x^2;
else
   y = 4;
end
```

Example 3.5.

```
function typ = findtyp(inp)

[r, c] = size(inp); % determining the size of the variable inp;
         the number of rows of inp is written to variable r, and
         the number of columns of inp is written to variable c
if r == 1 && c == 1
   typ = 'scalar';
elseif r == 1 || c == 1
   typ = 'vector';
else
   typ = 'matrix';
end

end
```

Practice

3.1. Simplify the code:

```
if num > 100
   num = 100;
else
   num = num;
end
```

3.2. Simplify the code:

```
if val >= 10
   disp ('Too much!')
elseif val < 10
   disp ('Too low!')
end
```

3.3. Write a function which takes two input arguments m, n and forms a vector containing all integers lying between m and n (no matter which one is greater).

3.4. Write a function that has one input argument. If this argument is a row-vector or a column-vector, then the function must return also a row-vector or a column-vector, but with the reverse order of the elements (use the built-in functions `fliplr` and `flipud`). If the input argument is a scalar or a matrix, then the function should do nothing and return the original argument.

3.5. Write function `makematrix`, which receives two row-vectors as input arguments; the lengths of these vectors can be different and are not known in advance. The function must make a matrix with two rows containing input vectors. If the length of the vectors is different, it is necessary to add missing elements with zeros, for example:

```
vec1 = [2 3 4 5];
vec2 = [1 4 7 9 8 6];
makematrix(vec1,vec2)
ans =
   2 3 4 5 0 0
   1 4 7 9 8 6
```

4

Loops

4.1. For

The statement `for` is used if it is necessary to repeat the execution of some program elements, and it is known in advance how many times this repetition should be performed. The `for` loop has the following general form:

```
for loopvar = ini : step : fin
      statements
end
```

The variable `loopvar`, which is called loop variable, changes from the initial value `ini` to the final value `fin` with increment `step`. On the first execution of the loop, the variable `loopvar` takes the value `ini`, and `statements` are executed, which in general can depend on the value of `loopvar`. After that, the value of the loop variable is incremented by the value of `step` and becomes `ini + step`, and the whole sequence of actions is repeated. In the next step, the value of the loop variable becomes `ini + 2*step`, and so on, until the value of the loop variable exceeds `fin`. Then the program proceeds to statements following the `end` statement.

Example 4.1.

```
for i = 0:50:200
   fprintf('%3d\n',i)
end
```

In this case, there is only one `fprintf` statement in the body of the `for` loop, which provides the output of the loop variable `i` in accordance with the format specified by the string `%3d\n`. This format means that the number `i` will be displayed in integer format with three characters (`3d`), after which a new line (`\n`) begins. The result of the loop will be the following:

```
0
50
```

```
100
150
200
```

Example 4.2. `for` statement is convenient for calculating sums and products. For example, the sum of all prime numbers from 1 to 1000 can be found using the following loop:

```
total = 0
for k = 1:999 % loop for all natural numbers from 1 to 999
   if(isprime(k)) % checking whether k is a prime number
      total = total + k; % if k is a prime number, then the
            argument of the if statement takes a logical value of
            true, and statements are executed inside the if
            condition: the variable total is incremented by k
   end
end
disp(total)           % display the sum on the screen
```

As a result of this loop, we get 76127.

The above loop can be written more efficiently if we use the possibility of MatLab to specify elements of a vector as the loop variable:

```
total = 0
for k = primes(999)
      total = total + k;
end
disp(total)
```

In this case, the statement `primes(999)` creates a vector consisting of prime numbers belonging to the range from 1 to 999, and at each step the loop variable takes the value of the next prime number; all that remains is to add this number to the sum.

`for` loop should be used to calculate sums in complex cases; in simple situations it is more efficient to use element-wise operations or `sum` statement. For example, the above code can be optimized:

```
k = primes(999);
k = sum(k); % summation of all elements of vector k and writing
              the result to a variable with the same name k
disp(k)
```

Similarly, `for` loop is used to calculate products:

```
k = 1;
for i = 1:10
   k = k * i;         % calculation factorial of 10
end
disp(k)
```

4.2. While

`While` statement is used if it is necessary to repeat the execution of some program statements and it is not known in advance how many times this repetition is necessary. A while-loop has the following general form:

```
while condition
      statements
end
```

The execution of `statements` inside this loop will be executed as long as `condition` is `true`. Therefore, `statements` must affect somehow the loop condition and at some point should make it equal to `false`, otherwise the loop will last endlessly.

Example 4.3.

```
function [fact, i] = factgthigh(high)
% function factgthigh finds the smallest number i which factorial
(fact) is greater than the input number (high)

i = 0;
fact = 1;
while fact < high % the condition of the loop: the factorial fact
                  of the number i is less than the input
                  parameter high
   i = i + 1;
   fact = fact * i;
end % end of loop
end % end of function
```

As a result, we get:

```
[fact, i] = factgthigh(4000)

fact =
        5040

i =
     7
```

Often when using the while loop, you need to know how many times the loop has been executed before interruption. For this purpose, a *counter* should be put in the loop. The counter is initialized by zero value before the start of the loop and is incremented by one at each iteration:

```
counter = 0; % initialization of the counter
while condition
   counter = counter + 1;
   statements
end
```

Complex logical constructions formed by logical operators OR (`||`) and AND (`&&`) can be used as the condition of the loop, for example:

```
while (x >= 0) && (x <= 100)
while (a == b) || (c ~= d) % when checking the
                           condition, the equal sign
                           should be written twice
```

4.3. Converting a `for` loop into a `while` loop

Any for loop can be rewritten as a while loop. To do this, you need to complete three steps:

1) initialize the loop variable;

2) create a condition that is initially true, but becomes false when it is necessary to interrupt the execution of the loop; often this condition is specified as the range of the loop variable;

3) before each new iteration of the while loop, you must change the loop variable.

Example 4.4 shows a convertation of `for` loop into `while` loop.

Example 4.4.

```
% for loop
for x = 1:2:15
   disp(x)
end
```

```
% equivalent while loop
x = 1; % step 1
while (x < 15) % step 2
   disp(x)
   x = x + 2; % step 3
end
```

4.4. Nested loops

`For` and `while` loops can be placed inside each other, such loops are called *nested* loops. The necessary condition is that the first statement and `end` statement of the nested `for` or `while` loop must be placed between the beginning and the end of the loop in which they are nested. Fig. 4.1 shows one of the possible schemes of nested loops.

Nested loops are an effective tool for working with matrices. Example 4.5 shows how variables of outer and nested loops change when the program is running.

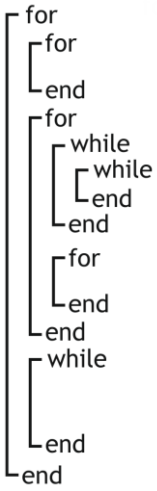


Fig. 4.1. Nested loops

Example 4.5.

```
for x = 1:3
   for y = 1:2
      fprintf('x= %.0f and y= %.0f\n',x,y)
   end
end
```

Running this code leads to the following result:

```
x= 1 and y= 1
x= 1 and y= 2
x= 2 and y= 1
x= 2 and y= 2
x= 3 and y= 1
x= 3 and y= 2
```

Thus, the variables of the inner loop change faster than the variables of the outer loop. The `fprintf` command in this case acts as follows: prints the string `x=`, then the first variable after the comma (that is, the variable `x`) in the format `%.0f` (floating-point number and zero decimal places), then the string `and y=` and the second variable after the comma (that is, the variable `y`) in the same format, and finally goes to a new line.

In Example 4.6, the transformation of a matrix into a vector consisting of the elements of the matrix taken line by line is considered.

Example 4.6.

```
% M is the given matrix
[r, c] = size (M) % size statement evaluates the dimensions of the
                  argument; in this case, the number of rows of the
                  array M is written into the variable r, and the
                  number of columns in the variable c
V = []; % initialization of vector V as an empty set
for col = 1:c % loop through columns
   for row = 1:r % loop through rows
      V(end + 1) = M(row,col); % the value of the element of the
                  matrix M in the row (row) and column (col) is
                  written in the element of vector V following the
                  last one, i.e. at each step of the inner cycle,
                  the length of the vector V increases by one, and
                  a number from the matrix M is written to the last
                  element of V
   end
end
disp(V)
```

If we take as the matrix M a magic matrix of size 4 as matrix M,

```
M = magic(4)

M =

    16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1
```

then the vector V will look like this:

```
16   5   9   4   2  11   7  14   3  10   6  15  13   8  12   1
```

4.5. Efficiency

`For` and `while` loops are quite simple in structure and easy to program. However, often they are not the most effective in terms of time spent by the computer for their implementation. To determine the time taken to perform an operation or a group of operations, the commands `tic` and `toc` are provided in MatLab. The first of them starts a timer clock, and the second stops it. Example 4.7 demonstrates how the preliminary initialization of arrays, as well as the use of built-in functions, speeds up the program.

Example 4.7. In this example, three programs are considered, each of which creates a vector of length 1 000 000, filled with ones.

In the first, most naive case, one element is added to the vector at each step of the cycle, and the value equal to one is written in it:

```
clear; % removing all variables from RAM
tic; % start a timer clock
for j = 1 : 1000000
   x(j) = 1;
end
time = toc % the elapsed time in seconds is written in variable
           time
```

In this case we get the following result:

```
time =

   0.464753821381304
```

Now pre-initialize the vector x, creating an array of the required size, filled with zeros. In this case, MatLab will allocate the necessary memory for all 1000000 elements of the vector x, and will not allocate additional space for each element at each step of the loop, as in the first case:

```
clear; % removing all variables from RAM
tic; % start a timer clock
x = zeros(1,1000000);
for j = 1 : 1000000
   x(j) = 1;
end
time = toc
```

On the same computer, the second program will give the result, several times faster than the first:

```
time =

   0.081696345144139
```

Now we get the vector x using the built-in MatLab function:

```
clear; % removing all variables from RAM
tic; % start a timer clock
x = ones(1,1000000);
time = toc
```

In this case, the calculation is even an order of magnitude faster:

```
time =

   0.004765046808881
```

4.6. Debugging

An important skill in programming is the ability to detect an error if the program does not work or produces wrong result. In general case, the process of localization of the error requires tracking the values of the variables. In the case of loops, this is quite simple. Example 4.8 shows a code that calculates Fibonacci numbers, and the values of all current variables are displayed on the screen at each iteration of the loop, as well as the iteration number. This technique of debugging the program allows you to quickly localize and fix the error.

Example 4.8.

```
A = 1;
B = 1;
C = A + B;
count = 3; % the first three Fibonacci numbers have already been
           counted, these are A, B, and C
while (C < 100000)
   fprintf('A:  %.0f,  B:  %.0f,  C:  %.0f,  count:  %.0f  \n',
           A,B,C,count);
   A = B;
   B = C;
   C = A + B;
   count = count + 1;
end
```

If somewhere in the program an error is made, and the program stops for any reason, then the values of all variables that they took immediately before the error will be displayed on the screen.

Practice

4.1. Write `for` loop that prints numbers in the column from 0.3 to 2.7 with 0.2 increments. Rewrite this loop in `while` loop.

4.2. Using `for` loop, write a function `steps2` that calculates the sum of numbers from 1 to n with increment of 2, where n is the argument of the function. Rewrite this loop in a `while` loop.

4.3. Write a code that calculates and displays the following table of values:

```
1
2 4
3 6 9
4 8 12 16
5 10 15 20 25
```

4.4. Write a program that calculates the approximate value of e^{-1} by the following expression:

$$\frac{1}{e} \approx \left(1 - \frac{1}{n}\right)^n.$$

The loop should be executed until the difference between the tabulated and approximate value becomes less than 0.0001. On the screen, it is necessary to give the exact value e^{-1}, the approximate value, and the iteration number n, at which the required accuracy was achieved.

4.5. Write a program that displays the elements of the input vector `vec` in the form of sentences. For example, for the vector `vec = [3.31 2.3 4.87]` the result should be

```
Element 1: 3.31
Element 2: 2.30
Element 3: 4.87
```

The length of the input vector `vec` can be arbitrary.

4.6. With the use of loops calculate the sum from the Practice 1.4. Use the statements `tic` and `toc`, and compare the calculation time using loops and using elementwise operations.

5

2D-plots

5.1. Simple plot

MatLab has powerful capabilities for creating graphical objects. Often, when writing a scientific article in a certain journal, there is a need to format all the drawings and plots in accordance with a given template. The most effective way in this case is to write a script that sets the parameters of all graphic objects in the drawing.

A simple plot can be constructed using `plot` function. The script presented in Example 5.1 creates a plot of the function $y(x) = \cos(6x)\, e^{x^2}$, specifies the color and style of the line, determines the parameters of the axes and their notation, and adds the legend to the plot. The result is shown in Fig. 5.1.

Example 5.1.

```
function simple_plot

x = linspace(-3,3,300); setting the grid of points in which the
function will be calculated
y = cos(6*x).*exp(-x.^2); calculation of function values

figure(1) % creating graphics window #1
plot(x,y,'k','LineWidth',2) % plotting; the width of the black
line is 2
axis([-2.75 2.753 -0.85 1.05]) % setting the limits for the axes
set(gca, 'LineWidth',1) % thickness of axis lines
set(gca, 'FontName', 'Trebuchet MS') % setting the font for the
notation on the axes
set(gca, 'FontSize', 8) % setting the size of the notation for the
axes
set(gca, 'FontWeight', 'bold') % setting the bold font of the
notation for the axes
xlabel('x', 'FontName', 'Trebuchet MS', 'FontAngle', 'italic',
'FontSize', 10, 'FontWeight', 'bold') % specifying the x-axis name
and style
ylabel('y=f(x)', 'FontName', 'Trebuchet MS', 'FontAngle',
'italic', 'FontSize', 10, 'FontWeight', 'bold')
```

```
leg = legend('y(x)=cos(6x)e^{-x^{2}}  ', 'Location', 'NorthWest');
specifying the legend in the upper right corner of the plot
set(leg, 'Box', 'off', 'FontName', 'Trebuchet MS', 'FontAngle',
'italic', 'FontSize', 10, 'FontWeight', 'bold') % specifying the
parameters and the style of the legend
```

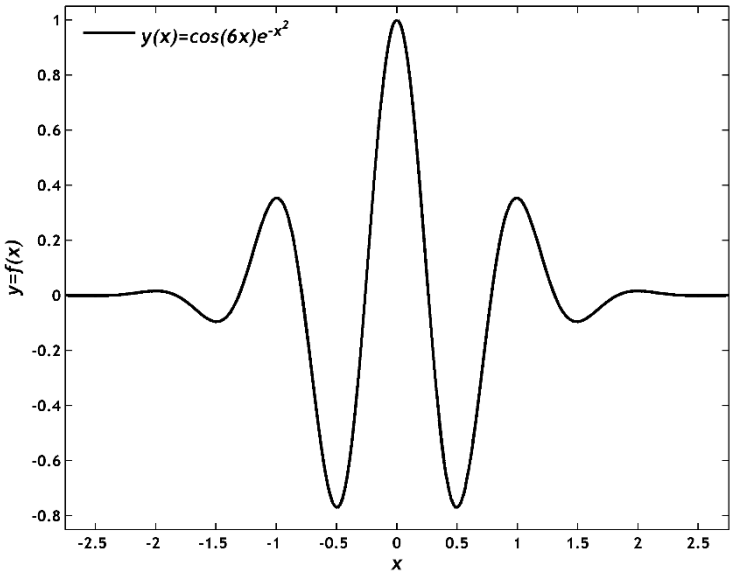


Fig. 5.1. Plot of the function $y(x) = \cos(6x)\, e^{x^2}$

The color and style of the graph line are specified as a parameter of the `plot` function by a string variable (in the Example 5.1 `'k'` means black color). Tables 5.1 – 5.3 provide several attributes of graph lines, such as the color of the line, the shape of markers and the line style.

Table 5.1

Colors of plot lines

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

Table 5.2

Shapes of markers

Symbol	Shape	Symbol	Shape
o	●	s	■
d	♦	*	★
h	⬢	v	▼
p	⬟	<	◀
+	+	>	▶
.	•	^	▲

Table 5.3

Line styles

Symbol	Line	Symbol	Line
--	Dashed	:	Dotted
-.	Dash dot	-	Solid

5.2. Plot with errors

To create a plot with errors, `errorbar` function can be used. It acts the same as `plot` function, but has an additional vector argument which contains error values for each point. Example 5.2 shows the use of this function. The function `set` in the Example is a universal MatLab function, and is used to set the properties of any object being its argument. In this case, `gca` function specifies the object which is the current axis. In addition to `gca`, functions `gcf` (specifying the current graphic window) `gco` (specifying the current graphic object) can be used. After the object, its attributes and the values of these attributes are listed, for example: `set (gco, 'Property1', 'Value1', 'Property2', 'Value2')`. The plot is shown in Fig. 5.2.

Example 5.2.

```
x = linspace(0,1,10); % setting the grid of points on the x-axis
from 1 to 10 in increments of 1
```

```
y = sin(4*x).*tan(x); % calculation of the function in these
points
e = 0.08*ones(1,10); % specifying the error values; in this case
0.08 for each point
errorbar(x,y,e,'r','LineWidth',2.5,'Marker','d','MarkerSize',8,'Ma
rkerEdgeColor','k','MarkerFaceColor','k') % drawing the plot with
errors and setting line properties
axis([-0.1 1.1 -1.3 0.6]) % setting the limits for the axes
set(gca, 'LineWidth',1) % thickness of axis lines
set(gca, 'FontName', 'Trebuchet MS') % setting font for the
notation on the axes
set(gca, 'FontSize', 8)
set(gca, 'FontWeight', 'bold')
xlabel('x', 'FontName', 'Trebuchet MS', 'FontAngle', 'italic',
'FontSize', 10, 'FontWeight', 'bold')
ylabel('y', 'FontName', 'Trebuchet MS', 'FontAngle', 'italic',
'FontSize', 10, 'FontWeight', 'bold')
leg = legend('Plot with errorbars', 'Location', 'Best');
set(leg, 'Box', 'on', 'FontName', 'Trebuchet MS', 'FontAngle',
'italic', 'FontSize', 10, 'FontWeight', 'bold')
```

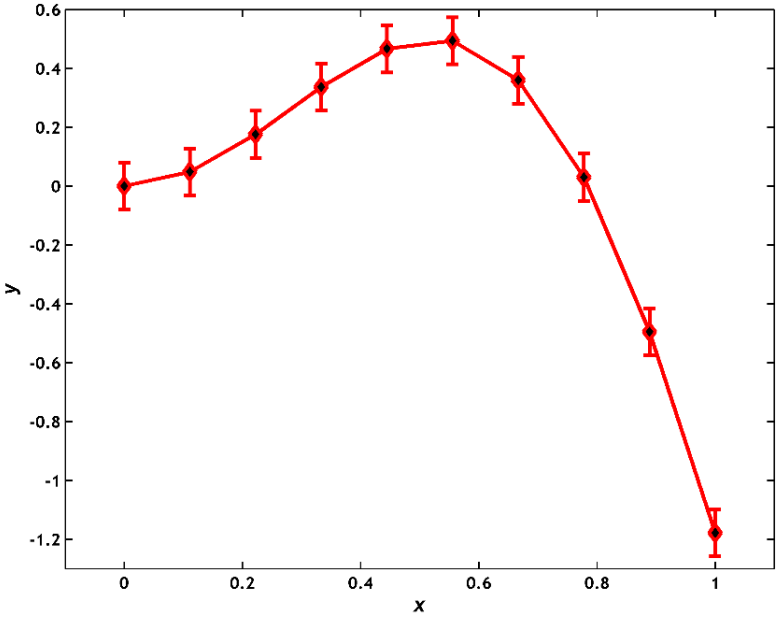


Fig. 5.2. Plot with errors

5.3. Multiple plots in one figure

The function `subplot(i,j,k)` is used to construct several plots in one graphical window. In this case, the graphic window is divided into a matrix of `i` graphical subwindows horizontally and `j` graphical subwindows vertically. The parameter `k` denotes the number of the current subwindow and is counted from the upper left corner line by line.

Example 5.3 shows the construction of four plots in one graphical window.

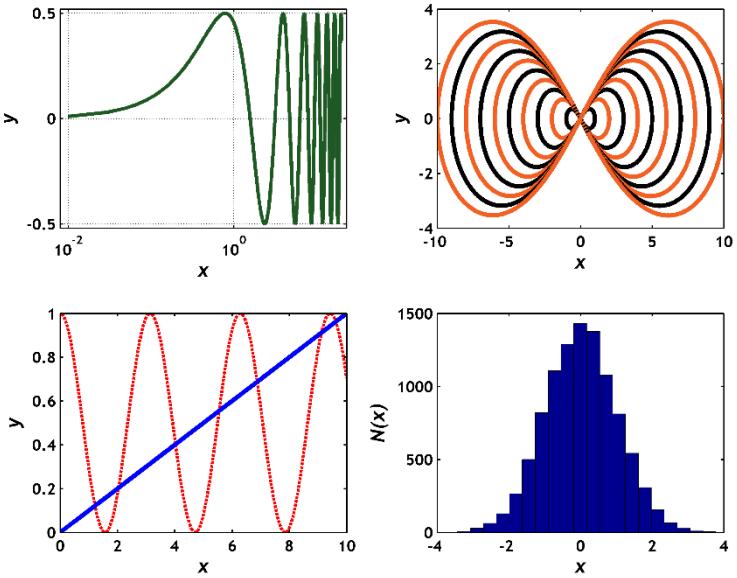


Fig. 5.3. Multiple plots in one graphical window

The first plot is drawn in a logarithmic scale along the *x* axis using `semilogx` function. The line color is specified in this case in RGB coordinates (for example, black color corresponds to the coordinates [0,0,0], white to [1,1,1], red to [1,0,0], etc.).

The second plot shows a family of curves defined parametrically by the parameter t: $x = a\frac{t+t^3}{t^4+1}; y = a\frac{t-t^3}{t^4+1}$ (*Bernoulli lemniscate*). The

statement `hold on` retains plots in the current graphical window so that new plots added do not delete existing plots. The color of the curve varies depending on the parity of the parameter a.

The third graph is a graphical solution of the equation $0.1x = \cos x^2$.

The fourth graph shows an example of constructing a histogram of the normal distribution from $N = 10000$ random points using the function `hist`.

The command `print -dtiff -r400 figure.tif` at the end of the example saves the contents of the graphical window to a tiff-file named figure.tif with resolution of 400 dots per inch.

Example 5.3.

```
figure(1)
subplot(2,2,1)

x = linspace(10^(-2),20,1000);
y = sin(x).*cos(x);

semilogx(x,y,'Color',[0.12,0.35,0.14],'LineWidth',2.5) % The
plotting the graph in the logarithmic scale along the x axis
axis([0.008 23 -0.52 0.52])
set(gca, 'LineWidth',1)
set(gca, 'FontName', 'Trebuchet MS')
set(gca, 'FontSize', 8)
set(gca, 'FontWeight', 'bold')
xlabel('x', 'FontName', 'Trebuchet MS', 'FontAngle', 'italic',
'FontSize', 10, 'FontWeight', 'bold')
ylabel('y', 'FontName', 'Trebuchet MS', 'FontAngle', 'italic',
'FontSize', 10, 'FontWeight', 'bold')

subplot(2,2,2)

t = [-20:0.001:20]; % setting the grid for the parameter t

for a = 1 : 10
    x = a*(t+t.^3)./(t.^4+1); % computing x on the grid of t
    y = a*(t-t.^3)./(t.^4+1); % computing y on the grid of t
    if mod(a,2) == 0 % checking the parity of the parameter a
        plot(x,y,'Color',[0.95,0.38,0.14],'LineWidth',3)
    else
        plot(x,y,'k','LineWidth',3)
    end

    hold on
end

set(gca, 'LineWidth',1)
```

```
set(gca, 'FontName', 'Trebuchet MS')
set(gca, 'FontSize', 8)
set(gca, 'FontWeight', 'bold')
xlabel('x', 'FontName', 'Trebuchet MS', 'FontAngle', 'italic',
'FontSize', 10, 'FontWeight', 'bold')
ylabel('y', 'FontName', 'Trebuchet MS', 'FontAngle', 'italic',
'FontSize', 10, 'FontWeight', 'bold')

subplot(2,2,3)

x1 = linspace(0,10,1000);
y1 = cos(x1).^2;
y2 = 0.1*x1;

plot(x1,y1,'--r','LineWidth',2)
hold on
plot(x1,y2,'b','LineWidth',3)
axis([0 10 0 1])
set(gca, 'LineWidth',1)
set(gca, 'FontName', 'Trebuchet MS')
set(gca, 'FontSize', 8)
set(gca, 'FontWeight', 'bold')
xlabel('x', 'FontName', 'Trebuchet MS', 'FontAngle', 'italic',
'FontSize', 10, 'FontWeight', 'bold')
ylabel('y', 'FontName', 'Trebuchet MS', 'FontAngle', 'italic',
'FontSize', 10, 'FontWeight', 'bold')

subplot(2,2,4)

r = randn(1,10000); % vector of 10000 random numbers normally
distributed on the interval [0,1]
hist(r,20) % drawing a histogram with 20 columns
set(gca, 'LineWidth',1)
set(gca, 'FontName', 'Trebuchet MS')
set(gca, 'FontSize', 8)
set(gca, 'FontWeight', 'bold')
xlabel('x', 'FontName', 'Trebuchet MS', 'FontAngle', 'italic',
'FontSize', 10, 'FontWeight', 'bold')
ylabel('N(x)', 'FontName', 'Trebuchet MS', 'FontAngle', 'italic',
'FontSize', 10, 'FontWeight', 'bold')

print -dtiff -r400 figure.tif
```

5.4. Plot with two different vertical axes

Quite often there is need to draw a graph of two functions defined on one segment but having a very different scale of values. In this case it is convenient to plot a graph with two different vertical axes. This can be done using the `plotyy` function. Example 5.4 gives one possible variant

of such plotting, where graphs of functions $y = 100e^{-2x} + +1000e^{-10x}$ and $y = e^{-(x-2.5)^2}$ are drawn in one graphical window. In addition to the specifying the vectors of data and function values, the list of arguments of `plotyy` function contains also string arguments defining the scale of the corresponding axis. In this case the logarithmic scale of y axis for the first plot and the usual axes for the second plot; the parameters of the axes and lines for the first and second plots are written in the variables `zaxes`, `zline1`, `zline2`, respectively. To create complex axes notations, the syntax of the LaTex language is used. The plot is shown in Fig. 5.4.

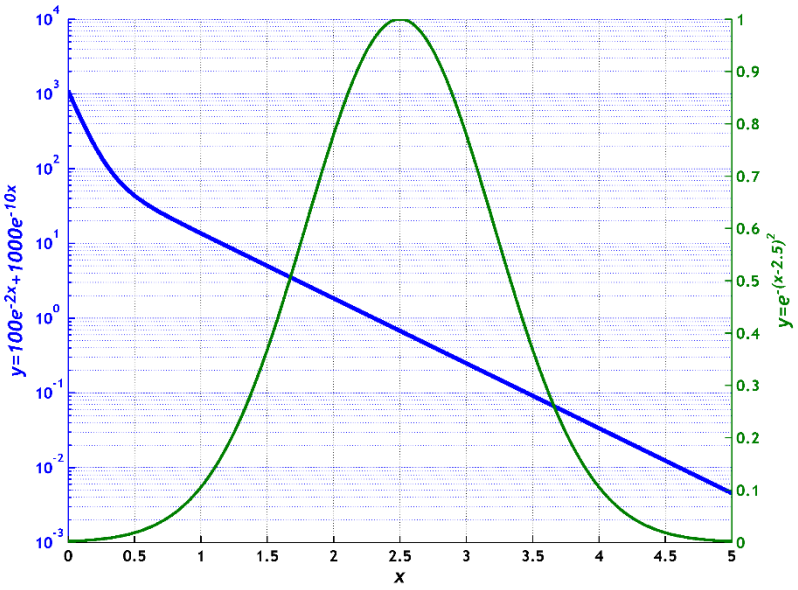


Fig. 5.4. Plot with two different vertical axes

Example 5.4.

```
x = linspace(0,5,1000);
y1 = 100*exp(-2*x) + 1000*exp(-10*x);
y2 = exp(-(x-2.5).^2);

figure(1)
[zaxes,zline1,zline2] = plotyy(x,y1,x,y2,'semilogy','plot');
```

```
set(zaxes(1),      'LineWidth',1,      'FontName',      'Trebuchet
MS','FontSize', 8, 'FontWeight', 'bold')
set(zaxes(2),      'LineWidth',1,      'FontName',      'Trebuchet
MS','FontSize', 8, 'FontWeight', 'bold')
set(zaxes(2), 'XTickLabel','') % disabling of notation on the
second x axis
xlabel(zaxes(1), 'x','FontName', 'Trebuchet MS', 'FontAngle',
'italic', 'FontSize', 10, 'FontWeight', 'bold')
ylabel(zaxes(1),      'y=100e^{-2x}+1000e^{-10x}',      'FontName',
'Trebuchet   MS',   'FontAngle',   'italic',   'FontSize',   10,
'FontWeight', 'bold')
ylabel(zaxes(2), 'y=e^{-(x-2.5)^2}', 'FontName', 'Trebuchet MS',
'FontAngle', 'italic', 'FontSize', 10, 'FontWeight', 'bold')
set(zline1,'LineWidth',3)
set(zline2,'LineWidth',2)
grid(zaxes(1),'on') % turn on the grid for the first plot
```

5.5. Plot with frames

To create a frame on a graph that demonstrates, for example, the behavior of a function on a certain segment on an enlarged scale, the function `axes` can be used, which creates new coordinate axes in the current graphics window. In Example 5.5, `axes` function has `Position` argument specifying the position of the new axes; the first pair of values of this argument defines the position of the origin of the new axes relative to the graphics window ([0,0] corresponds to the left bottom point, [1,1] to right upper one), and the second pair of values determines the length of the axes (the length equal to one corresponds to the entire graphics window). Fig. 5.5 shows the result of the script of Example 5.5.

Example 5.5.

```
x = linspace(0,2*pi,1000);
y1 = sin(x).*(cos(15*x)).^2;
y2 = sin(x);
figure(1)
plot(x,y1,'b','LineWidth',2)
hold on
plot(x,y2,'r','LineWidth',3)

axis([0 2*pi -1 1])
set(gca, 'LineWidth',1)
set(gca, 'FontName', 'Trebuchet MS')
```

```
set(gca, 'FontSize', 8)
set(gca, 'FontWeight', 'bold')
xlabel('x', 'FontName', 'Trebuchet MS', 'FontAngle', 'italic',
'FontSize', 10, 'FontWeight', 'bold')
ylabel('y=sin(x)cos^2(15x)', 'FontName', 'Trebuchet MS',
'FontAngle', 'italic', 'FontSize', 10, 'FontWeight', 'bold')

axes('Position', [0.25 0.2 0.2 0.2]); % the second pair of axes
x = linspace(2.64,3.64,100);
y1 = sin(x).*(cos(15*x)).^2;
y2 = sin(x);
plot(x,y1,'b','LineWidth',1.5)
hold on
plot(x,y2,'r','LineWidth',2.5)
axis([2.64 3.64 -0.5 0.5])
set(gca, 'LineWidth',1)
set(gca, 'FontName', 'Trebuchet MS')
set(gca, 'FontSize', 8)
set(gca, 'FontWeight', 'bold')
```

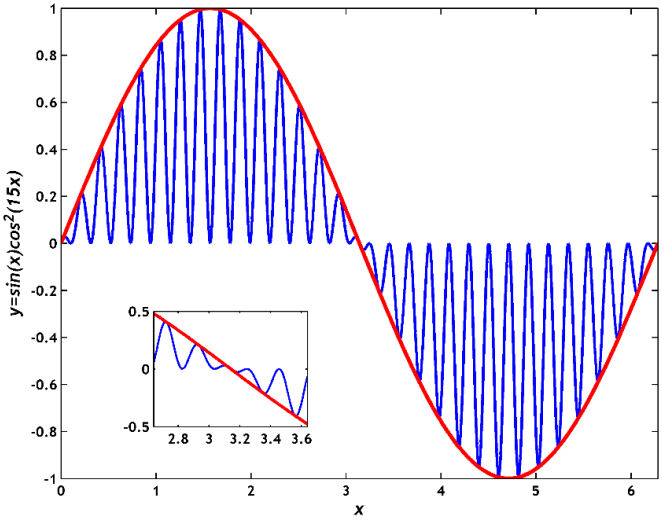


Fig. 5.5. Plot with a frame

6

3D-plots

6.1. Histograms

Many of MatLab functions related to the construction of 2D-plots have 3D analogues, differing from 2D ones by the presence of "3" in the name. For example, the function `bar3` allows drawing 3D histograms (Example 6.1, Fig. 6.1).

Example 6.1.

```
mat = spiral(5) % function spiral returns a square matrix whose
                  elements are integers increasing from the center
                  of the matrix to its edges
mat =
    21    22    23    24    25
    20     7     8     9    10
    19     6     1     2    11
    18     5     4     3    12
    17    16    15    14    13

figure
subplot(1,2,1)

bar3(mat) % 3D histogram of spiral matrix
set(gca, 'LineWidth',1)
set(gca, 'FontName', 'Trebuchet MS')
set(gca, 'FontSize', 8)
set(gca, 'FontWeight', 'bold')
zlabel('z', 'FontName', 'Trebuchet MS', 'FontAngle', 'italic',
'FontSize', 10, 'FontWeight', 'bold')

subplot(1,2,2)
bar3(1./mat) % 3D histogram of a matrix in which each element is
               the inverse of the corresponding element of the
               spiral matrix

set(gca, 'LineWidth',1)
set(gca, 'FontName', 'Trebuchet MS')
set(gca, 'FontSize', 8)
set(gca, 'FontWeight', 'bold')
zlabel('z', 'FontName', 'Trebuchet MS', 'FontAngle', 'italic',
'FontSize', 10, 'FontWeight', 'bold')
```

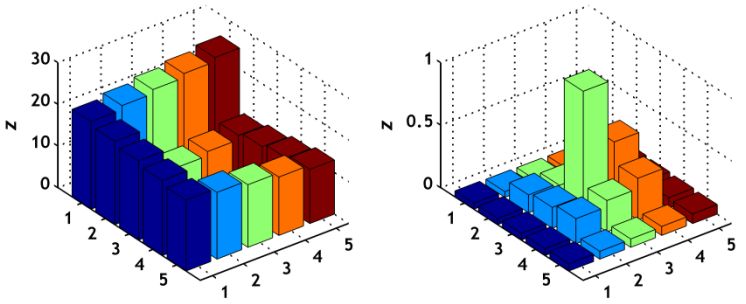


Fig. 6.1. 3D histograms

6.2. Linear plots in 3D

The function `plot3` (Example 6.2) is a three-dimensional analogue of the `plot` function. The function `plot(x,y,z)`, where x, y, z are vectors of the same length, draws a line passing through the points in space with coordinates (x, y, z). If x, y, z are matrices of the same size, then `plot(x,y,z)` draws a family of lines, each corresponding to one column of the matrices (x, y, z).

Example 6.2. Consider a plot of a line in 3D (Fig. 6.2).

```
t = linspace(0, 10*pi, 500); % a vector of 500 points uniformly
                             distributed on a segment from 0 to 10π
figure

plot3(sin(t),cos(t),t,'r','LineWidth',3) % plot of line in 3D

set(gca, 'LineWidth',1)
set(gca, 'FontName', 'Trebuchet MS')
set(gca, 'FontSize', 8)
set(gca, 'FontWeight', 'bold')
xlabel('sin(t)','FontName', 'Trebuchet MS', 'FontAngle', 'italic',
'FontSize', 10, 'FontWeight', 'bold')
ylabel('cos(t)','FontName', 'Trebuchet MS', 'FontAngle', 'italic',
'FontSize', 10, 'FontWeight', 'bold')
zlabel('t','FontName', 'Trebuchet MS', 'FontAngle', 'italic',
'FontSize', 10, 'FontWeight', 'bold')
grid on
```

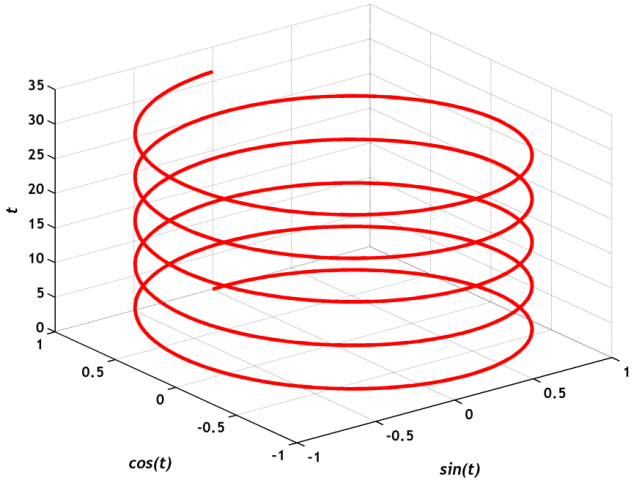


Fig. 6.2. Plot of a line in 3D

The function `view(a,b)` allows setting the viewing point angle of a 3D plot (Example 6.3, Fig. 6.3). The parameter `a` determines the rotation angle (in degrees) in the xy plane, and the parameter `b` is the ascent angle relative to the xy plane. By default, these values are $a = -37.5$; $b = 30$.

Example 6.3.

```
t = linspace(0,6*pi,500);

figure

subplot(2,2,1)
plot3(sin(t),cos(t),t,'r','LineWidth',3) % default 3D view
set(gca, 'LineWidth',1)
set(gca, 'FontName', 'Trebuchet MS')
set(gca, 'FontSize', 8)
set(gca, 'FontWeight', 'bold')
xlabel('sin(t)','FontName', 'Trebuchet MS', 'FontAngle', 'italic',
'FontSize', 10, 'FontWeight', 'bold')
ylabel('cos(t)','FontName', 'Trebuchet MS', 'FontAngle', 'italic',
'FontSize', 10, 'FontWeight', 'bold')
zlabel('t','FontName', 'Trebuchet MS', 'FontAngle', 'italic',
'FontSize', 10, 'FontWeight', 'bold')
grid on
title('3D')
```

```
subplot(2,2,2)
plot3(sin(t),cos(t),t,'g','LineWidth',3)
view(0,90)
set(gca, 'LineWidth',1)
set(gca, 'FontName', 'Trebuchet MS')
set(gca, 'FontSize', 8)
set(gca, 'FontWeight', 'bold')
xlabel('sin(t)','FontName', 'Trebuchet MS', 'FontAngle', 'italic',
'FontSize', 10, 'FontWeight', 'bold')
ylabel('cos(t)','FontName', 'Trebuchet MS', 'FontAngle', 'italic',
'FontSize', 10, 'FontWeight', 'bold')
zlabel('t','FontName', 'Trebuchet MS', 'FontAngle', 'italic',
'FontSize', 10, 'FontWeight', 'bold')
grid on
title('Look at the plane xy') % axis notations are automatically
adjusted depending on the viewing angle; in this case, the
notation for the z-axis is not shown

subplot(2,2,3)
plot3(sin(t),cos(t),t,'b','LineWidth',3)
view(0,0)
set(gca, 'LineWidth',1)
set(gca, 'FontName', 'Trebuchet MS')
set(gca, 'FontSize', 8)
set(gca, 'FontWeight', 'bold')
xlabel('sin(t)','FontName', 'Trebuchet MS', 'FontAngle', 'italic',
'FontSize', 10, 'FontWeight', 'bold')
ylabel('cos(t)','FontName', 'Trebuchet MS', 'FontAngle', 'italic',
'FontSize', 10, 'FontWeight', 'bold')
zlabel('t','FontName', 'Trebuchet MS', 'FontAngle', 'italic',
'FontSize', 10, 'FontWeight', 'bold')
grid on
title('Look at the plane xz')

subplot(2,2,4)
plot3(sin(t),cos(t),t,'m','LineWidth',3)
view(90,0)
set(gca, 'LineWidth',1)
set(gca, 'FontName', 'Trebuchet MS')
set(gca, 'FontSize', 8)
set(gca, 'FontWeight', 'bold')
xlabel('sin(t)','FontName', 'Trebuchet MS', 'FontAngle', 'italic',
'FontSize', 10, 'FontWeight', 'bold')
ylabel('cos(t)','FontName', 'Trebuchet MS', 'FontAngle', 'italic',
'FontSize', 10, 'FontWeight', 'bold')
zlabel('t','FontName', 'Trebuchet MS', 'FontAngle', 'italic',
'FontSize', 10, 'FontWeight', 'bold')
grid on
title('Look at the plane yz')
```

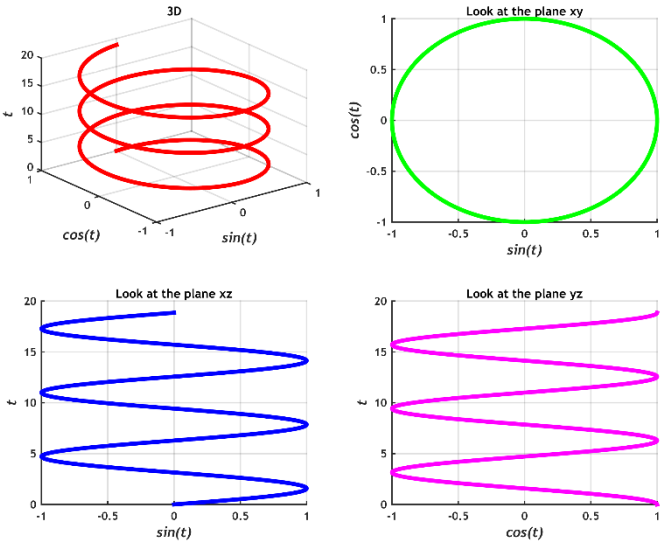


Fig. 6.3. Use of `view` function

6.3. Surfaces

The main functions used to draw 3D surfaces $z = f(x, y)$ in MatLab are `mesh`, `surf` and `meshgrid` (Example 6.4). The function `mesh` is used to draw wireframe surfaces (Fig. 6.4), and the `surf` function plots a colored parametric surface; the color is determined by the value of z for each calculated point (Fig. 6.5).

Example 6.4.

```
[x,y,z] = sphere(15); % the built-in function sphere(n) returns
                      vectors x, y, z of length n+1 containing the
                      coordinates of the unit sphere
figure

mesh(x,y,z) % creation of a wireframe surface of a sphere
set(gca, 'LineWidth',1)
set(gca, 'FontName', 'Trebuchet MS')
set(gca, 'FontSize', 8)
set(gca, 'FontWeight', 'bold')
title('Wireframe surface of a sphere')
```

```
figure

surf(x,y,z) % creation of a surface of a sphere
set(gca, 'LineWidth',2)
set(gca, 'FontName', 'Trebuchet MS')
set(gca, 'FontSize', 8)
set(gca, 'FontWeight', 'bold')
title('Surface of a sphere')

colorbar % Adding to the picture a legend containing information
         on the dependence of the color on the value of z
```

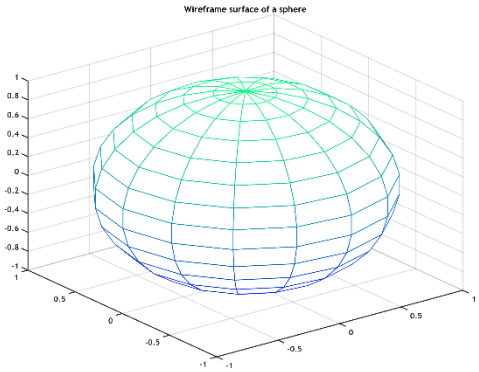


Fig. 6.4. Wireframe of a sphere drawn by use of `mesh` function

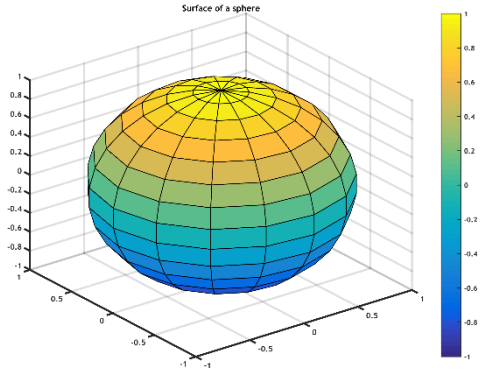


Fig. 6.5. Surface of a sphere drawn by use of `surf` function

With the use of the function `meshgrid` (Example 6.5, Fig. 6.6), a grid of points in the xy plane is defined, on which the function $z = f(x, y)$ is calculated; then the matrices x, y, z are used when referring to `mesh` or `surf`.

Example 6.5.

```
u=linspace(-5,5,81); % u is a vector of length 81 with coordinates
                     uniformly distributed on the segment from -5 to 5
v=linspace(0,10,81);
[x,y]=meshgrid(u,v); % in this case, function meshgrid specifies
                      the square 81x81 matrices x and y
z1=3*sin(x).*cos(y); % calculation of function values at points
                      (x,y)

figure
surf(x,y,z1) % 3D-surface z1
set(gca, 'LineWidth',1)
set(gca, 'FontName', 'Trebuchet MS')
set(gca, 'FontSize', 8)
set(gca, 'FontWeight', 'bold')
xlabel('x'),ylabel('y'),zlabel('z')
title('3D-surface and plane')
hold on

z2=0*x + 1;     % definition of a plane intersecting the z-axis at
               point 1; the statement 0*x creates a matrix of
               zeros of the size of the matrix x, then 1 is added
               to the whole matrix; similarly, z2 could be
               specified by the statement z2 = ones(size(x))
mesh(x,y,z2) % plane z2
hold off
```

The functions `meshc` and `surfc` (Example 6.6) supplement, respectively, the wireframe and the surface with contour graphs (Fig. 6.7), showing levels of the same height.

Example 6.6.

```
[X,Y,Z] = peaks; % peaks is built-in MatLab function specifying
          the coordinates of points of the surface with two maxima
figure
surfc(X,Y,Z)
axis([-3 3 -3 3 -10 5]) % range of axes
set(gca, 'LineWidth',1)
set(gca, 'FontName', 'Trebuchet MS')
set(gca, 'FontSize', 8)
set(gca, 'FontWeight', 'bold')
```

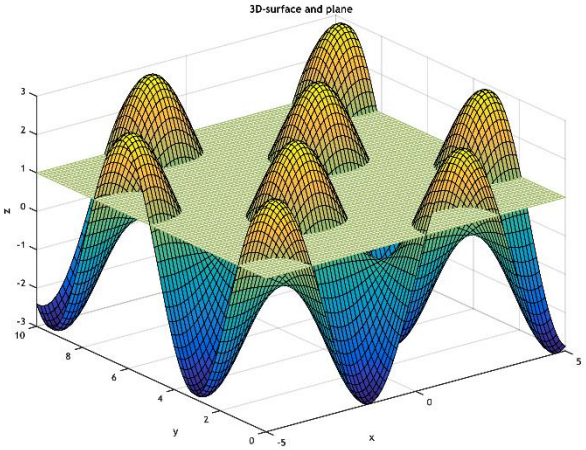


Fig. 6.6. Function `meshgrid`. Used to specify a grid in the xy plane for calculating values of $z = f(x, y)$ function

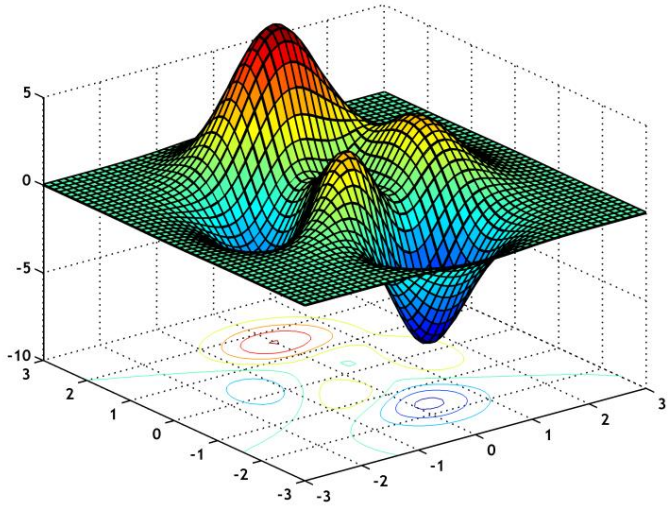


Fig. 6.7. Wireframe surface with countour plot

3D surface can be filled in various ways (Fig. 6.8). Example 6.7 illustrates several options.

Example 6.7.

```
figure
subplot(1,3,1)
sphere(16)
axis square % axes of the same scale
shading flat % flat shading, grid is not displayed
set(gca, 'LineWidth',1)
set(gca, 'FontName', 'Trebuchet MS')
set(gca, 'FontSize', 8)
set(gca, 'FontWeight', 'bold')
title('shading flat')

subplot(1,3,2)
sphere(16)
axis square
shading faceted % the default shading, grid is displayed
set(gca, 'LineWidth',1)
set(gca, 'FontName', 'Trebuchet MS')
set(gca, 'FontSize', 8)
set(gca, 'FontWeight', 'bold')
title('shading faceted')

subplot(1,3,3)
sphere(16)
axis square
shading interp % the matrix of colors is calculated by
                interpolation; grid is not displayed; color
                change becomes smooth
set(gca, 'LineWidth',1)
set(gca, 'FontName', 'Trebuchet MS')
set(gca, 'FontSize', 8)
set(gca, 'FontWeight', 'bold')
title('shading interp')
```

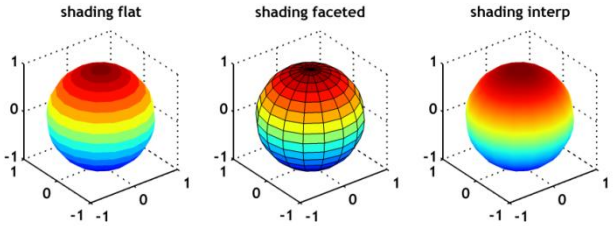


Fig. 6.8. Different surface filling options

6.4. Examples

Example 6.8 shows the use of various palettes (Fig. 6.9).

Example 6.8.

```
[x,y] = meshgrid(-3:0.01:3); grid of points in xy plain
z = peaks(x,y);

figure

surfl(x,y,z); % function surfl builds a surface and adds a light
source to the image
shading interp % gradient fill
colormap gray % gray palette
axis([-3 3 -3 3 -8 8]) % setting scales of the axes
set(gca, 'LineWidth',1)
set(gca, 'FontName', 'Trebuchet MS')
set(gca, 'FontSize', 8)
set(gca, 'FontWeight', 'bold')

figure

surfl(x,y,z);
view([25 25]) % change the viewing angle
shading interp
colormap copper % palette of copper colors
axis([-3 3 -3 3 -8 8])
set(gca, 'LineWidth',1)
set(gca, 'FontName', 'Trebuchet MS')
set(gca, 'FontSize', 8)
set(gca, 'FontWeight', 'bold')
```

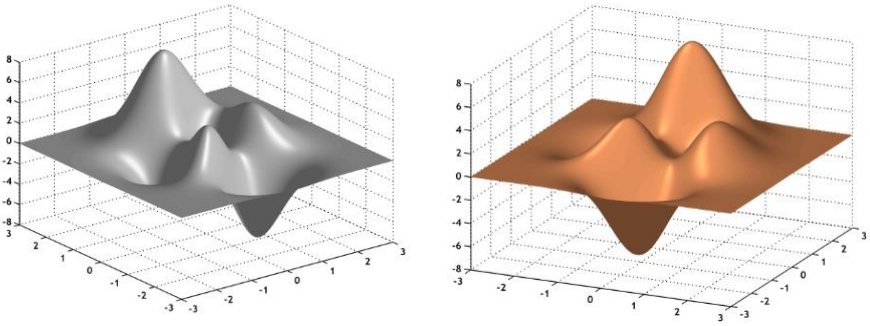


Fig. 6.9. Use of various palettes

In Example 6.9, various ways of constructing contour graphs are shown (Fig. 6.10).

Example 6.9.

```
figure
colormap jet % one of build-in palettes

subplot(2,2,1)
[x,y,z] = peaks;
contour(x,y,z,30) % contour plot of function peaks with 30 level
lines
set(gca, 'LineWidth',1)
set(gca, 'FontName', 'Trebuchet MS')
set(gca, 'FontSize', 8)
set(gca, 'FontWeight', 'bold')


subplot(2,2,2)
[c,h] = contour(x,y,z,8); % contour plot with 8 level lines;
matrix c contains data defining the contour lines
set(gca, 'LineWidth',1)
set(gca, 'FontName', 'Trebuchet MS')
set(gca, 'FontSize', 8)
set(gca, 'FontWeight', 'bold')
clabel(c,h,'FontName', 'Trebuchet MS','FontSize', 5,'FontWeight',
'bold') % function clabel(c,h) inserts numbers along the level
lines

subplot(2,2,3)
pcolor(x,y,z) % "pseudocolors"; the color of each point is
determined by the values of the nearest point with coordinates
(x,y,z)
shading interp
set(gca, 'LineWidth',1)
set(gca, 'FontName', 'Trebuchet MS')
set(gca, 'FontSize', 8)
set(gca, 'FontWeight', 'bold')

subplot(2,2,4)
contourf(x,y,z,15) % space between the level lines is filled with
one color
set(gca, 'LineWidth',1)
set(gca, 'FontName', 'Trebuchet MS')
set(gca, 'FontSize', 8)
set(gca, 'FontWeight', 'bold')
```

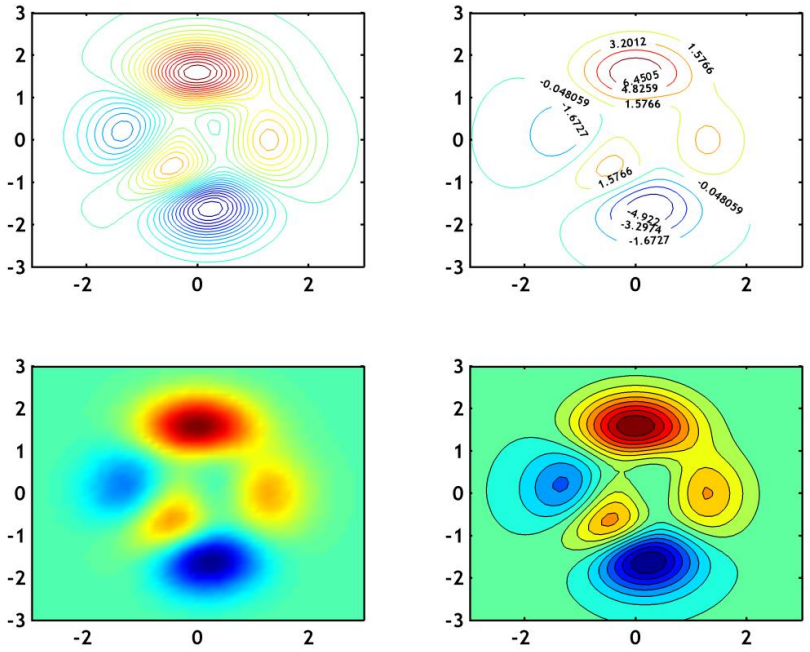


Fig. 6.10. Various contour plots

Example 6.10 shows various ways of visualizing vector fields (Fig. 6.11, 6.12).

Example 6.10.

```
figure

subplot(1,2,1)
[x,y,z] = peaks;
[dx,dy] = gradient(z,.5,.5); % calculation of the gradient of
function z with increment 0.5 along the x axis and along the y
axis
contour(x,y,z,10) % contour plot with 10 level lines
hold on
quiver(x,y,dx,dy) % plotting arrows denoting a vector field; the
length and direction of the arrows are determined by arrays dx and
dy; in this case these are the gradients of the function z
hold off
set(gca, 'LineWidth',1)
set(gca, 'FontName', 'Trebuchet MS')
```

```
set(gca, 'FontSize', 8)
set(gca, 'FontWeight', 'bold')
title('Vector field on the contour plot')

subplot(1,2,2)
[nx,ny,nz] = surfnorm(x,y,z); % calculation of normals to the
surface
surf(x,y,z)
hold on

quiver3(x,y,z,nx,ny,nz) % drawing arrows denoting a vector field,
the length and direction of the arrows are determined by arrays
nx, ny and nz; in this case these are the normals to the surface z
hold off

set(gca, 'LineWidth',1)
set(gca, 'FontName', 'Trebuchet MS')
set(gca, 'FontSize', 8)
set(gca, 'FontWeight', 'bold')

title('Normals to surface')
```

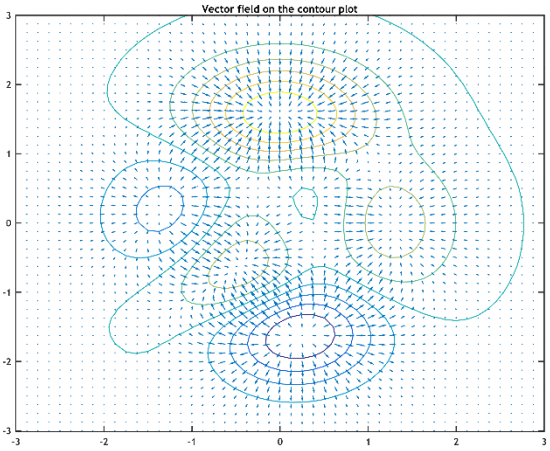


Fig. 6.11. Field of gradients

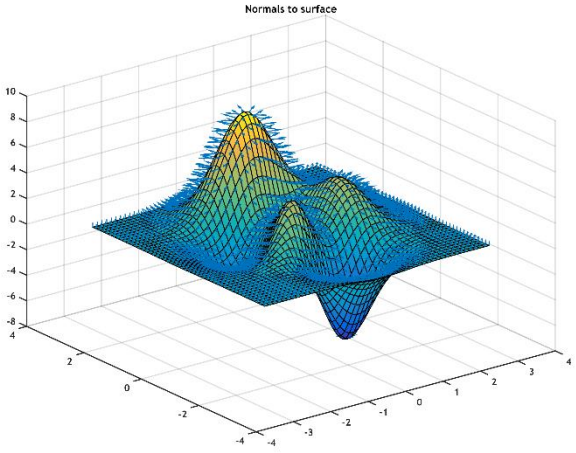


Fig. 6.12. Mapping of normals to a surface

In Example 6.11 the mapping of Hadamard matrix onto a spherical geometry is considered (Fig. 6.13).

Example 6.11.

```
k=5;
n = 2^k - 1;
theta = pi*(-n:2:n)/n; inclination angle of a spherical coordinate
system
phi = (pi/2)*(-n:2:n)'/n; % azimuth angle; the transposition of
the phi vector (') is used to enable matrix multiplication and to
compute x,y,z; if transposition is not used, then element-wise
operation should be used

x = cos(phi)*cos(theta); % alculation of coordinates of a sphere
y = cos(phi)*sin(theta);
z = sin(phi)*ones(size(theta));

colormap ([0 0 0; 1 1 1]) % a color palette of only two colors,
white and black
c = hadamard(2^k); % the statement hadamard(n) computes the
Hadamard matrix of order n; the Hadamard matrix consists of
numbers 1 and -1, its columns are orthogonal; it is used in such
areas as combinatorics, numerical analysis, signal processing, and
also in quantum computation
```

```
surf(x,y,z,c) % drawing a sphere of unit radius, each segment of
which is colored corresponding to the element of the Hadamard
matrix

axis square % axes of the same length
axis off % do not show axes
```

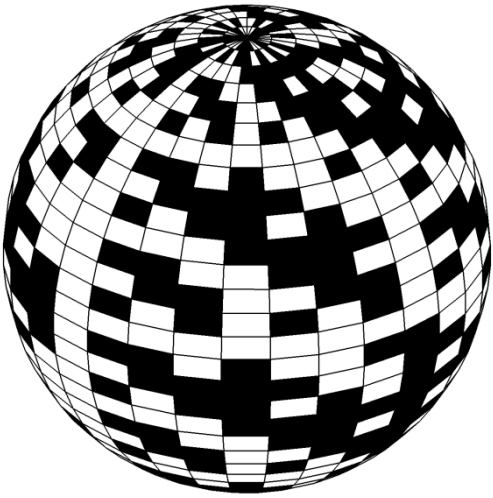


Fig. 6.13. Hadamard matrix on a sphere

7

Sorting

7.1. Basis states and matrices

The purpose of this section is to use selection statements, loops, and plot construction statements to write the simplest algorithms for sorting arrays and searching for an element in an ordered array.

A task of search and sorting arises in numerical modeling of quantum systems in the context of formation of basis functions of the system, as well as in constructing matrices of quantum operators in the chosen basis. It is easier to understand this problem with the help of a simple mathematical example.

Consider a system of three boxes and two identical balls placed in them. Obviously, there are six possible states in the system (see Fig. 7.1). We call these states *the basis* of the system.

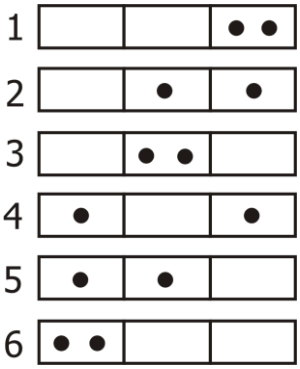


Fig. 7.1. The basis for the system of three boxes and two balls

If the balls were different, for example, black and white, then to the situation with one ball in the second box and one in the third box would correspond two possible configurations (Fig. 7.2, left). If the balls are

the same, i.e. *indistinguishable*, then there is only one configuration (Fig. 7.2, right).

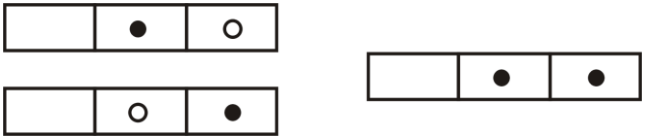


Fig. 7.2. Distinguishable (left) and indistinguishable (right) balls

Now let's have some device A, which works as follows: it shifts one ball from the third box to the second one. Let's see how the matrix reflecting the operation of this device will look in the basis shown in Fig. 7.1.

The matrix A will consist of the matrix elements A_{ij}, where the index of the column j corresponds to the number of the initial state, and the index of the row i to the state that is obtained from the state j after it is acted on by A. For example, if we take the state 1 (i.e., the index j is 1), in which both balls are in the third box, then after shifting one of the balls into the second box, the state will be as shown in Fig. 7.3.

Fig. 7.3. Under the action of device A, the left state transfroms to the right state

It can be seen that some other state was obtained that is different from state 1. To determine the value of the index i, it is necessary to find this state in the basis. *At this point, the problem arises of finding the obtained state in the initial basis.* Looking at Fig. 7.1 we see that the state that is obtained from state 1 is state 2. Hence,

$$A_{i1} = \begin{cases} 1, & if \quad i = 2; \\ 0, & if \quad i \neq 2, \end{cases}$$

since under the action of A state 1 goes to state 2 and to no other; therefore, the first column of matrix A is as follows:

$$A = \begin{pmatrix} 0 & \dots & \dots & \dots & \dots & \dots \\ 1 & \dots & \dots & \dots & \dots & \dots \\ 0 & \dots & \dots & \dots & \dots & \dots \\ 0 & \dots & \dots & \dots & \dots & \dots \\ 0 & \dots & \dots & \dots & \dots & \dots \\ 0 & \dots & \dots & \dots & \dots & \dots \end{pmatrix}.$$

Columns 3, 5 and 6 in matrix A will be zero, since in these states the third box is empty, and there is nothing to shift. Similarly, state 2 under the action of A goes to state 3, and state 4 goes to state 5; the final form of matrix A is as follows:

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

After the matrix reflecting the action of the device A has been obtained, it can be treated as a conventional matrix. For example, in order to find out what happens to the states of a basis, if they are acted on by A twice, it is sufficient to raise the matrix A to the square:

$$A^2 = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix},$$

this means that the double action of A leads to the transition of state 1 to state 3, and all other matrix elements are zero, since only in state 1 there are two balls in the third box.

Thus, some *operator* is introduced in the basis shown in Fig. 7.1, and a matrix corresponding to this operator is constructed in this basis by means of the action of this operator on the basis functions.

When modeling quantum systems, it is often necessary to form matrices of linear operators in bases consisting of a very large number of states, so if the procedure for finding the desired state in the basis is not organized in an efficient manner, the process of forming matrices can take a long time. The procedure for finding the desired state will be effective and fast only if the states of the basis are numbered not randomly, but in accordance with a specific scheme. For example, with each of the states shown in Fig. 7.1 a number can be associated, each digit of which will reflect the number of balls in the corresponding box (Fig. 7.4).

It is seen that the numbers corresponding to the states of the basis are ordered in ascending order, therefore, it is not difficult to organize an effective procedure for finding the desired state in such a basis. For this purpose, for example, a fast and easy-to-implement *bisection method* is suitable.

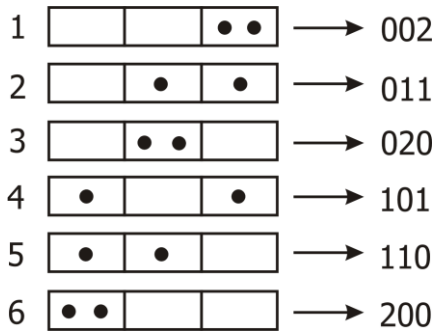


Fig. 7.4. The correspondence of the basis states and numbers sorted in ascending order

Depending on the specific task, sorting of states can be performed according to different criteria. As a rule, effective built-in search procedures are available for numerical simulation, but sometimes there are situations when you have to sort manually, without using built-in functions. Next, we will consider some of simple types of sorting.

7.2. Sorting by insertion

In this sorting method, the elements of an unordered array are scanned one at a time, and each next element is inserted into a suitable place among the previously ordered ones (Fig. 7.5).

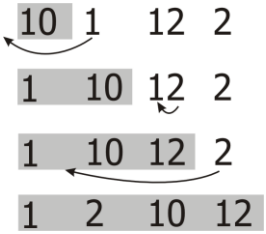


Fig. 7.5. Sorting by insertion.
Gray background shows ordered elements

The time expenditures for sorting by insertion are of the order of N^2 operations, where N is the number of elements in the unordered array. This sorting method is inefficient. Example 7.1 shows the MatLab code, which sorts an array of random numbers by insertion.

Example 7.1.

```
function [key, t] = vstavka(i)

% i – number of elements to be sorted
key = rand(i,1); % creation of a vector containing i random
numbers
tic % starting the time counter
N = length(key); % the length of the vector key
for k = 2 : N % loop over all elements of the array
   for l = 1 : k-1 % at each step of the k loop, key(k) is
                     compared to already ordered elements
      if key(l) > key(k)
         a = key(k); % auxiliary variable
         for m = k : -1 : l+1
            key(m) = key(m-1); % all previously ordered elements
                                 that are larger than key(k) are
                                 shifted one position to the right
         end
         key(l) = a; % key(k) is inserted in a place with the
                       number l
      end
   end
```

```
end
t = toc; % the time counter stops, and the elapsed time is written
           in variable t
```

7.3. Sorting by selection

In this sorting method, the smallest (or largest) element is selected from the unordered array and somehow separated from the others, then the smallest (largest) element of the remaining elements is selected, and so on (Fig. 7.6). This sorting method, like sorting by insertion, requires about N^2 operations.

7.4. Sorting by exchange

In this sorting method, two elements change places if they are not ordered (Fig. 7.7), this process is repeated until all possible pairs of elements are selected.

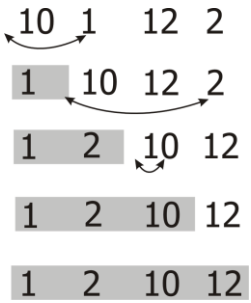


Fig. 7.6. Sorting by selection.
Gray background shows ordered elements

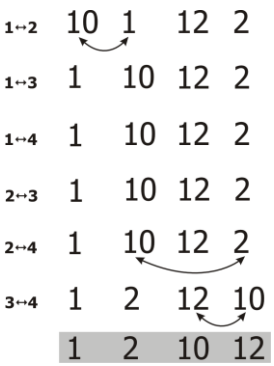


Fig. 7.7. Sorting by exchange. Gray background shows ordered elements. The left column shows the numbers of elements that are compared at the current step

The time expenditures for this sorting method are of the order of $N^2/2$ operations, this method is more economical compared to the previous ones and is often used in standard mathematical packages. A block diagram of the algorithm is shown in Fig. 7.8; in this algorithm, in addition to sorting the elements of the original key array, an array of indices ind is also created which contains pointers on the ordered elements of the key array.

7.5. Optimized method

The sorting methods presented above will perform sorting in a reasonable time when the number of elements is of the order of several thousand. For greater number of elements, sorting will slow down, and more effective methods are needed. One way to speed up sorting algorithm is to move not between individual elements, but between blocks of several tens or hundreds of elements.

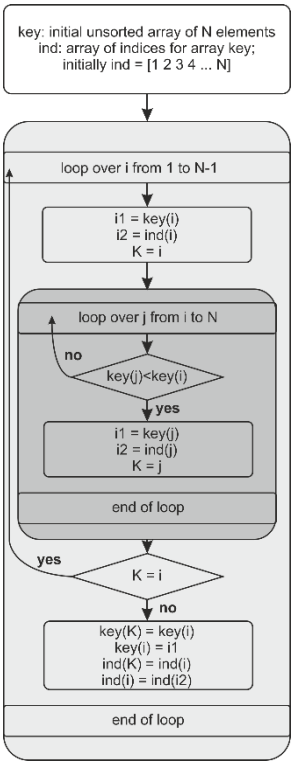


Fig. 7.8. Control-flow chart of sorting by exchange

Consider one of such optimal sorting. We divide the entire unordered array of N elements into clusters with $k \approx 3\sqrt{N} < N$ elements in each (with this cluster size, the speed of the sorting method described below will be maximal). The number of clusters in the full array will also be of the order of k, and in the last cluster there will be $k' \leq k$ elements, where k' is the remainder of the integer division of N by k (Fig. 7.9). Within each of the clusters, ordinary sorting (for example, by the exchange method) is carried out, which will take about $k\left(\frac{k^2}{2}\right)$ operations.

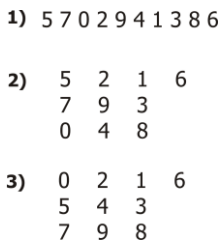


Fig. 7.9. Preparing an unordered array for optimal sorting:
1) the original array of ten elements;
2) the array is divided into blocks of three elements,
while in the last block there is one element;
3) sorting within each block

Next, we look at the minimal elements in each cluster (since all clusters have already been ordered, the minimal element in each cluster will be the first element) and choose the smallest of them. This process will take an order of k operations. The selected element is placed in a new array, in which the ordered elements will be accumulated, and is deleted from the cluster where it was located (Fig. 7.10). In this case, it is necessary to monitor the number of elements in each cluster and not consider clusters from which all elements have been moved to an ordered array. Filling the entire ordered array will take about k^2 operations, i.e. will be even faster than the original sorting; therefore, the algorithm of optimized sorting will require an order of $k^3 \sim N^{3/2}$ operations.

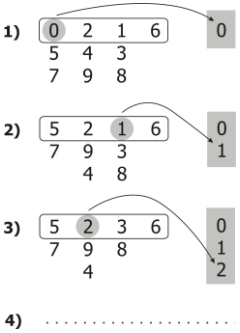


Fig. 7.10. Optimized sorting

The method presented is not the most optimal one. There are techniques that provide the sorting time of $\sim N log_2 N$ operations.

Practice

7.1. Implement algorithms for sorting one-dimensional arrays of arbitrary numbers:

1) by choice;
2) by insertion;
3) by exchange;
4) by the optimized method.

Sort 500, 1000, 2000, 4000, 8000, ... numbers by each of the methods and plot the dependence of the sorting time on the number of elements in the array for each method. At what number of elements does the optimized method become more efficient?

7.2. There is an array of N different numbers arranged in ascending order. Implement the algorithm for finding the position of a given element in an array:

1) by direct search;
2) by bisection method.

Plot the dependence of the search time on the length of the array N for each of the methods. Compare the efficiency of the two algorithms.

7.3. There is an array of N different numbers ordered in ascending order. Construct and implement the algorithm for finding the position of a given element in an array by bisection method for each digit.

8

Working with files

8.1. Writing to a file

When using data from external sources, as well as saving information for its subsequent processing, it becomes necessary to use functions for working with files.

There are three main methods for working with files: reading information from a file, writing information to a file (creating a file), and adding information to a file. Files in general can have arbitrary types and extensions. Here we consider files with the extensions `.dat`, `.txt`, `.mat`, which are most often used when working with MatLab.

The function `save` is used to write information from a matrix variable to a file or to create a file with a given matrix variable:

```
save filename matrixvariablename -ascii
```

The argument `-ascii` is used to create files in text format. For example,

```
mymat = rand(2,3)

mymat =
       0.4565    0.8214    0.6154
       0.0185    0.4447    0.7919

save testfile.dat mymat -ascii
```

The given set of commands will create a file `testfile.dat`, in which the matrix

$$\begin{pmatrix} 0.4565 & 0.8214 & 0.6154 \\ 0.0185 & 0.4447 & 0.7919 \end{pmatrix}$$

will be written in text format. If the file `testfile.dat` already exists, then this set of commands will rewrite it.

The function `type` is used to display the contents of a file:

```
type mymat

4.5646767e-001  8.2140716e-001  6.1543235e-001
1.8503643e-002  4.4470336e-001  7.9193704e-001
```

To add information to the file, the optional argument `-append` is used:

```
mat2 = rand(3,3)

mat2 =
       0.9218    0.4057    0.4103
       0.7382    0.9355    0.8936
       0.1763    0.9169    0.0579

save testfile.dat mat2 -ascii -append
```

As a result of executing these commands, the file `testfile.dat` is changed, and now contains the following data:

```
       0.4565    0.8214    0.6154
       0.0185    0.4447    0.7919
       0.9218    0.4057    0.4103
       0.7382    0.9355    0.8936
       0.1763    0.9169    0.0579
```

8.2. Reading from a file

To read data from a file, the function `load` can be used. If the file has `.txt` or `.dat` extension, then after reading the file, a matrix variable with the same name as the file name will be created. For example,

```
load testfile.dat

who % query for all variables in the workspace

Your variables are: testfile

testfile

testfile =
       0.4565    0.8214    0.6154
       0.0185    0.4447    0.7919
       0.9218    0.4057    0.4103
       0.7382    0.9355    0.8936
       0.1763    0.9169    0.0579
```

The function `load` works only when each row of the file has the same number of values, so the information can be stored as a matrix variable, since the function `save` can only store matrix variables. In other cases, statements of low-level communication with the file system should be used.

8.3. Functions of low-level communication with the file system

Working with files with the use of functions of low-level communication with the file system consists of three phases: opening a file; reading from a file or writing to a file; closing a file.

To open a file, the function `fopen` is used. By default, `fopen` opens the file for reading. To open a file in a different mode, for example, to write in it or to modify it, an additional string argument is used. If the function `fopen` returns `-1`, it means that the file was opened unsuccessfully. If the file was opened successfully, then a positive integer value is returned, which corresponds to the file identifier. This identifier is used later to call the low-level functions. For example,

```
fid = fopen('filename', 'permission string'); % here
fid is the file identifier, permission string is a
string argument
```

The string argument can take the following values:

`r` – to read data from a file;

`w` – to write data to a file (or to create a file);

`a` – to add data to a file.

After using `fopen`, the value of the returned file identifier should be checked to determine if the file was opened successfully. For example,

```
fid = fopen('samp.dat');
if fid == -1
      disp('File can not be opened')
   else
     ...
      % Commands for working with a file
     ...
end
```

After the work with a file is completed, the file should be closed. This is done by `fclose` function, which returns `0` if the file is closed successfully, or `-1` if the file could not be closed. Files can be closed individually by use of their identifiers, or you can close all the files together using `all` argument. For example,

```
closeresult = fclose(fid);

closeresult = fclose('all');
```

The value returned `closeresult` should also be checked:

```
fid = fopen('filename', 'permission string');
if fid == -1
     disp('File can not be opened')
else
     ...
```

```
    % Commands for working with a file
    ...
    closeresult = fclose(fid);
    if closeresult == 0
        disp('File is closed successfully')
    else
        disp('File can not be closed')
    end
end
```

There are several functions of low-level communication with the file system, intended for reading data from files. The function `fscanf` reads formatted data into a matrix variable and uses special arguments for type conversion, such as `%d` for integers, `%s` for string data, and `%f` for floating point numbers. The function `textscan` reads text information from a file and saves it as a cell array, also using special arguments for type conversion. The functions `fgetl` and `fgets` read from a file line by line, the difference between them is that `fgets` preserves the end-of-line character, and `fgetl` removes the end-of-line character. Before using all the above functions to read data from a file, the file should be opened, and closed after reading the data.

Typically, the functions `fgetl` and `fgets` are used inside `for` or `while` loop. The functions `fscanf` and `textscan` can read all the information from a file into one variable. It can be assumed that these two functions occupy an intermediate place between `load` function and such low-level functions as `fgetl`. When working with them, no `for` or `while` loop is required, the whole file will automatically be read into one variable.

The function `fgetl` provides more control over the process of reading data from a file, compared to other reading functions. The function `fgetl` reads one line from the file in the form of a string variable, then this variable can be converted with the help of string functions. Since the function `fgetl` reads one line only, it is usually placed in a loop that reads the file to the end. The function `feof` returns

logical one if the end of the file is reached, and logical zero otherwise. Thus, it is possible to write an algorithm for reading information from a file:

```
an attempt is made to open the file;
   checking whether the file was opened successfully or
   not;
if the file is opened successfully, then a loop runs until
the end of the file is reached;
   for each line of the file:
       reading line into a string variable (string);
       converting of the data read;
an attempt is made to close the file;
       checking whether the file was closed successfully or
not;.
```

In MatLab, the above algorithm corresponds to the following code:

```
fid = fopen('filename'); % if the function fopen is
       used without additional arguments, then the
       argument 'r', which is set by default, is used

if fid == -1
     disp('File can not be opened')
else
     while feof(fid) == 0 % Reading a single line in
                          a string variable
      aline = fgetl(fid);
        ...
        % Here you should use string functions to
        extract numbers, strings, etc. from the string
        variable aline.
        ...
     end

     closeresult = fclose(fid);
     if closeresult == 0
           disp('File is closed successfully')
     else
           disp('File can not be closed')
```

```
    end
end
```

The condition in the `while` loop can be interpreted as "doing a loop until the end of the file is reached." This condition can be written in an alternative way:

```
while ~feof(fid)
```

Suppose there is a file called `subjexp.dat`, which contains in each line a number, followed by a space and a certain character. Using the function `type`, you can display the data stored in this file:

```
type subjexp.dat
```

```
   5.3 a
   2.2 b
   3.3 a
   4.4 a
   1.1 b
```

Such a file can not be read into a matrix variable using the function `load`, since it contains both numbers and text. However, using the function `fgetl`, you can read this file line-by-line into string variables, and use string functions to format the data.

Example 8.1.

```
fid = fopen('subjexp.dat');
if fid == -1
    disp('File can not be opened')
else
    while feof(fid) == 0
       aline = fgetl(fid);
       [num, charcode] = strtok(aline); % splitting a string into
                                        a number and a character
       fprintf('%.2f %s\n', str2double(num), charcode)
end
   closeresult = fclose(fid);
   if closeresult == 0
        disp('File is closed successfully')
```

```
   else
end
```

As a result of the above commands, we get:

```
5.30 a
2.20 b
3.30 a
4.40 a
1.10 b
File is closed successfully
```

In Example 8.1, at each iteration of the loop, the function `fgetl` reads one line into the string variable `aline`. The function `strtok` splits the line into two lines with the names `num` and `charcode`. Next, the type of variable `num` is converted to a floating-point number using the function `str2double`.

The function `fscanf` can be used to read data from a file into a matrix variable:

```
mat = fscanf(fid, 'format', [dimensions])
```

Here the function `fscanf` reads data from the file with the identifier `fid` into the matrix variable `mat`. The variable `format` must contain special arguments for type conversion. The dimensions of the matrix variable `mat` are determined by the variable `dimensions`; if the number of values in the file is not known, then the value of `inf` should be used. Example 8.1 can be rewritten with the use of the function `fscanf` as follows:

```
fid = fopen('subjexp.dat');
mat = fscanf(fid,'%.2f %s',[2, inf])

mat =
      5.30 a
      2.20 b
      3.30 a
      4.40 a
      1.10 b
```

```
fclose(fid);
```

Example 8.2 shows the use of the function `fscanf` to handle data with complicated format.

Example 8.2. Suppose that there is a file `mymeas.dat`, which contains the following data: the first line contains the file header that should be ignored; the second line contains the number of measurement blocks; the third line is empty; then the measurement blocks follow. The measurement blocks are also a complex construction: each block consists of three parts, the first part containing information about the measurement time, the second about the date, then four lines of four measurement values in each:

```
Measurement Data
N=3

12:00:00
01-Jan-1977
4.21  6.55  6.78  6.55
9.15  0.35  7.57  NaN
7.92  8.49  7.43  7.06
9.59  9.33  3.92  0.31
09:10:02
23-Aug-1990
2.76  6.94  4.38  1.86
0.46  3.17  NaN 4.89
0.97  9.50  7.65  4.45
8.23  0.34  7.95  6.46
15:03:40
15-Apr-2003
7.09  6.55  9.59  7.51
7.54  1.62  3.40  2.55
NaN 1.19  5.85  5.05
6.79  4.98  2.23  6.99
```

Such a file can be read in the format of structure:

```
filename = 'mymeas.dat'; % name of the file
measrows = 4; % number of lines
meascols = 4; % number of columns

fid = fopen(filename); % opening the file
finfo = dir(filename); % reading file information
fsize = finfo.bytes; % determining the size of the file
```

```
if fsize > 0 % if the file is not empty, then

    % reading the top of the file and writing the value to the
variable N
    N = fscanf(fid, '%*s %*s\nN=%d\n\n', 1);

    % reading the file in blocks
    block = 1;
    while ~feof(fid) % until the end of the file is reached

        % reading and writing into the structure mystruct of
        variables mtime, mdate, means

        % read mtime as string data, one line
         mystruct(block).mtime = fscanf(fid, '%s', 1);

         % read mdate as string data, one line
         mystruct(block).mdate = fscanf(fid, '%s', 1);

        % read meas as floating point variables, four lines, four
        values in each line

         mystruct(block).meas   =   fscanf(fid,   '%f',   [measrows,
        meascols])'; % (2D array)

         block = block + 1; % block iterations
    end
end

fclose(fid); % close the file
```

The function `dir` is used to read the file information into the object of type 'structure' `finfo`, which is then used to determine the file size. Reading the file in the structure `mystruct` is done by blocks, the number of which is determined by the second line of the file. Each block of the structure is divided into three parts, which are written to the corresponding cells of the structure `mtime`, `mdate` or `meas` by the function `fscanf`.

Another function of low-level communication with the file system is `textscan`. The function `textscan` reads information from a file and stores it as a cell array:

```
cellarray = textscan(fid, 'format');
```

As in the case of the function `fscanf`, the format variable must contain special characters for type conversion, and `fid` is the file identifier. The variable `format` describes the type of columns in the cell array `cellarray`. For example:

```
fid = fopen('subjexp.dat');
subjdata = textscan(fid,'%f %c');
fclose(fid)
```

The argument `%f %c` means that each line of the file contains a floating-point number, one character, and a space between them. The function `textscan` creates an array `subjdata` consisting of two cells. The first cell is a column containing numbers; the second column contains the symbols:

```
subjdata
subjdata =
    [5x1 double]
subjdata{1}
ans = 5.3000
      2.2000
      3.3000
      4.4000
      1.1000
subjdata{2}
ans =
    a
    b
    a
    a
    b
```

There are several functions of low-level communication with the file system in MatLab for writing data to a file. As in the case of other low-level functions, the file must first be opened, and closed after all the information is written.

The function `fprintf` writes data to a file line-by-line:

```
fprintf(fid, 'format', variable);
```

The function `fprintf` returns the number of bytes that were written to the file. The writing is performed to the file with the identifier `fid`. By default, if there is no file identifier, `fprintf` prints the information to the screen, since the screen is the main output device. The variable `format` must contain special arguments for type conversion. The variable `variable` contains data for writing. For example:

```
fid = fopen('tryit.txt', 'w'); % argument 'w' means
           that the file is being created (or
           rewritten) to write information in it
for i = 1:3
     fprintf(fid, 'Цикл %d\n', i);
end
fclose(fid);
```

In order to check what was written into the file `tryit.txt`, let read it line by line using the function `fgetl`:

```
fid = fopen('tryit.txt');
while ~feof(fid)
aline = fgetl(fid);
      disp(aline)
end

Loop 1
Loop 2
Loop 3
fclose(fid);
```

The function `fprintf` can also be applied to matrix variables:

```
mat = [20 14 19 12; 8 12 17 5]

mat =
            20 14 19 12
            8 12 17 5
```

```
fid = fopen('randmat.dat','w');
fprintf(fid,'%d %d\n',mat);
fclose(fid);

type randmat.dat

20 8
14 12
19 17
12 5
```

In order to correctly write the matrix variable `mat`, the second argument `%d %d\n` is written in `fprintf`, which means that the matrix to be written consists of two columns. Further, since `mat` is a matrix variable, the function `load` can be used to read information from the file `randmat.dat`:

```
load randmat.dat
>> randmat

randmat =
    20   8
    14   12
    19   17
    12   5

randmat'  % transposing randmat, the form of the
            variable mat can be restored
ans =       20 14 19 12
            8 12 17 5
```

The function `fprintf` can also be used to add data to a file. In this case, the file should be opened by the function `fopen` as follows:

```
fid = fopen('filename', 'a')
```

The second argument means that the file is opened to add data to it.

8.4. Mat-files

In addition to the above functions of low-level communication with the file system intended for working with files of type `.dat`, `.txt`, etc., MatLab has functions that read variables from or write variables to special files with the extension `.mat`. These files are called mat-files. These files are different from files with the extensions `.dat`, `.txt`, etc.; among other things, they contain the names of variables and their types. Mat files are written in binary format.

Variables can be read from, written or added to the mat-file:

```
save filename variablename
```

Here `filename` is the name of the file (without the extension); `variablename` is the stored variable. With the use of the function `save` it is possible to save variables of any type, including matrix ones:

```
mymat = rand(3,5);
save mat mymat
who -file mat
Your variables are:
mymat
```

To add a variable to the mat-file, an additional argument `-append` is needed:

```
x = 1:6;
save -append mat x
who -file mat
Your variables are:
mymat  x  % there are now two variables in the file
        mat.mat: mymat and x
```

The function `load` is used to read variables from a mat-file:

```
load filename variable list
```

Here `filename` is the file name (without extension); `variable list` is the list of loaded variables; by default, the function `load` reads all the variables at once:

```
who
load mat
who
Your variables are:
mymat x
```

9

Working with strings

9.1. Simple ways to create strings

Strings in MatLab can consist of any number of characters enclosed in single quotes. In fact, strings are also vectors where each element represents a single character; this means that strings support some vector operations and functions. MatLab has a lot of built-in functions written specifically for working with strings.

There are many uses of strings, even in areas that are predominantly considered numeric. For example, when a file consists of a combination of numbers and symbols, it is often necessary to read each line of the file as a string, split the lines into parts, and convert parts containing numbers to numeric variables that can be used in further calculations.

Below are examples of strings:

```
''
'x'
'Cat'
'Cat house'
'123'
```

Substrings are parts of strings. For example, `'house'` is the substring of the string `'Cat house'`. *Symbols* are numbers, alphabet, punctuation marks, spaces, and special characters. *Special characters* are symbols that are not shown when displayed on a screen.

There are several ways to specify a string. One of them is the use of the assignment statement:

```
word = 'cat';
```

Another way is to type a string using the function `input`. The use of the `'s'` key as the second argument of the function is mandatory, since this key describes the variable as a string:

```
strvar = input('Enter a string: ', 's')
   Enter a string: xyzabc

strvar =
         xyzabc
```

9.2. Representing strings as vectors

Strings are vectors of symbols or, in other words, vectors, each element of which is a single symbol. Hence, some vector operations can be applied to strings. For example, to find the number of characters in a string, the function `length` can be used:

```
length('cat')
   ans =
      3
length(' ')
   ans =
      1
length('')
   ans =
      0
```

It is possible also to refer to a single character of a string (a character in a string) or to a subset of characters:

```
mystr = 'Hi';
mystr(1)
ans =
      H
sent = 'Hello there';
length(sent)
ans =
      11
```

```
sent(4:8)
ans =
      lo th
```

The result of creating a column-vector of strings is a matrix of characters:

```
wordmat = ['Hello';'Howdy']
   wordmat =
   Hello
   Howdy

size(wordmat)
ans =
   2    5 % The variable wordmat is a 2x5 character
             matrix
```

It is possible to refer to a single character or to a single line:

```
wordmat(2,4)
ans =
      d
wordmat(1,:)
ans =
      Hello
```

The strings within the matrix must always be of the same length. Short lines should be completed with spaces so that all lines have the same length; otherwise an error occurs:

```
greetmat = ['Hello'; 'Goodbye']
   Error using vertcat
   Dimensions of matrices being concatenated are not
consistent.

greetmat = ['Hello  '; 'Goodbye']
greetmat =
Hello
```

```
Goodbye

size(greetmat)
ans =
      2   7
```

9.3. String functions

The concatenation of strings means their joining. Since strings are also vectors, it is obvious that vector concatenation methods work with strings. For example, one string can be created from two others:

```
first = 'Cat';
last = 'house';
[first last]
ans =
      Cathouse
```

It should be noted that the names of variables (or strings) must be separated by a space in parentheses, but there are no spaces between the strings after they are concatenated.

The function `strcat` concatenates strings horizontally:

```
first = 'Cat';
last = 'house';

strcat(first,last)
ans =
      Cathouse
```

If there are spaces in strings, the above concatenation methods are significantly different. The method using square brackets combines all characters, including all spaces:

```
str1 = 'xxx   ';
str2 = '   yyy';

[str1 str2]
```

```
ans =
      xxx      yyy

length(ans)
ans =
      12
```

The function `strcat` removes last spaces in a string before concatenating, but at the same time does not remove opening spaces:

```
strcat(str1,str2)
ans =
      xxx   yyy

length(ans)
ans =
      9

strcat(str2,str1)
ans =
         yyyxxx
length(ans)
ans =
      9
```

The function `char` is used to vertically concatenate strings by creating row column-vectors (character matrices). The function `char` automatically completes short strings with spaces, so that all rows in the matrix have the same length:

```
greetmat = char('Hello','Goodbye')
greetmat =
Hello
Goodbye

size(greetmat)
ans =
      2   7
```

9.4. Built-in functions for creating strings

The function `blanks` creates a string of n spaces:

```
blanks(4)
ans =

length(ans)
ans =
      4
```

Typically, this function is most often used when concatenating strings and specifying the number of spaces between them. For example, the following code inserts five spaces between the words:

```
first = 'Cat';
last = 'house';
[first blanks(5) last]
ans =
      Cat     house
```

The function `sprintf` is also to create the strings in accordance with a specific pattern. This pattern is specified in the first argument, which contains special characters beginning with `%` and defines the type of variables that are inserted into the string as subsequent arguments. For example, the special character `%.2f` denotes a floating-point variable with two decimal digits, while `%5d` corresponds to an integer variable of five digits, and the special character `%s` is used for a string variable:

```
sent1 = sprintf('The value of pi is %.2f', pi)
sent1 =
   The value of pi is 3.14

sent2 = sprintf('Some numbers: %5d, %2d', 33, 6)
sent2 =
   Some numbers:    33,  6
```

```
length(sent2)
ans =
      23
```

In Example 9.1, the program is presented that reads data from the file `expnoanddata.dat` which contains the number of the experiment followed by experimental data:

```
123   4.4   5.6   2.5   7.2   4.6   5.3
```

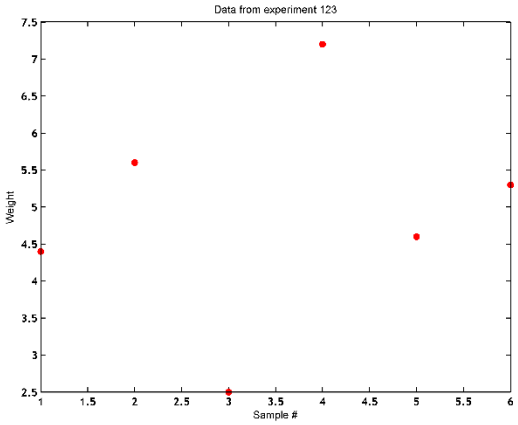


Fig. 9.1. The number of experiment is displayed in the title

Example 9.1.

```
% This program reads information from a file containing
% number of experiment and experimental data. Then it makes
% a plot and displays the number of experiment
% in the title (Fig. 9.1)

load expnoanddata.dat

experNo = expnoanddata(1);
data = expnoanddata(2:end);
figure
plot(data,'ro')
xlabel('Sample #')
```

```
ylabel('Weight')
title(sprintf('Data from experiment %d', experNo))
set(gca, 'LineWidth',1)
set(gca, 'FontName', 'Trebuchet MS')
set(gca, 'FontSize', 8)
set(gca, 'FontWeight', 'bold')
```

9.5. Removing spaces

There are functions in MatLab designed to remove spaces at the beginning and at the end of the string.

The function `deblank` removes spaces at the end of the string. For example, strings in the matrix of characters are completed with spaces, so that their lengths would be the same, but you may need to remove extra spaces in order to use the string in the original form:

```
names = char('Sue', 'Cathy','Xavier')
names =
Sue
Cathy
Xavier

name1 = names(1,:)
name1 =
Sue

length(name1)
ans =
      6

name1 = deblank(name1);
length(name1)
ans =
      3
```

The function `strtrim` removes both spaces at the beginning and at the end of the string but does not touch any spaces inside it. For

example, suppose that there are 3 spaces at the beginning, 4 spaces at the end, and 2 spaces inside the string:

```
strvar = [blanks(3) 'xx' blanks(2) 'yy' blanks(4)]
strvar =
   xx  yy

length(strvar)
ans =
      13

strtrim(strvar)
ans =
      xx  yy

length(ans)
ans =
      6
```

9.6. Changing register

MatLab has two functions `upper` and `lower` for changing register:

```
mystring = 'AbCDEfgh';

lower(mystring)
ans =
      abcdefgh

upper(ans)
ans =
      ABCDEFGH
```

9.7. String comparison

There are several functions for comparing strings that return a logical one if the strings are equivalent, and a logical zero otherwise. The function `stcmp` compares strings character by character. It returns a

logical one when the strings are absolutely identical (which assumes that they must be of the same length), or a logical zero if the strings are different. It should be noted that in the case of string comparisons, the comparison operator == is not valid. For example,

```
word1 = 'cat';
word2 = 'car';
word3 = ' cathedral';
word4 = 'CAR';

strcmp(word1,word3)
ans =
      0

strcmp(word1,word1)
ans =
      1
strcmp(word2,word4)
ans =
      0
```

The function `strncmp` compares only the first n characters in each string and ignores the rest. The first two arguments of this function are strings to be compared, the third argument is the number of characters:

```
strncmp(word1,word3,3)
ans =
      1
```

There are also functions `strcmpi` and `strncmpi`, corresponding to the functions `strcmp` and `strncmp`, but ignoring the differences of the character registers:

```
strcmpi(word2,word4)
ans =
      1
strncmpi(word3,word4,2)
ans =
      1
```

9.8. Searching, replacing, and splitting

The function `strfind` takes two strings as arguments: a string and a substring. The function finds all occurrences of a substring in a string and returns the positions in the string from which the substring begins. A substring can consist of either one character or several. If there are several occurrences of a substring in a string, then the vector of positions is returned. Note that only the positions of the beginning of a substring are returned:

```
strfind('abcde', 'd')
ans =
      4
strfind('abcde', 'bc')
ans =
      2
strfind('abcdeabcdedd', 'd')
ans =
      4    9    11    12
```

If the substring does not appear in the string, then an empty vector is returned:

```
strfind('abcdeabcde','ef')
ans =
      []
```

In Example 9.2, a program is presented that counts the number of spaces inside phrases. First, a vector of strings is created, consisting of phrases, then using `for` loop, the number of spaces in each string is counted, excluding the beginning and ending spaces.

Example 9.2.

```
phraseblanks.m

phrasemat = char('Hello and how are you?','Hi there everyone!',...
'How is it going?', 'Whazzup?')
```

```
[r c] = size(phrasemat);

for i = 1:r

howmany = countblanks(phrasemat(i,:));
fprintf('Phrase %d has %d blanks\n',i,howmany)

end
```

countblanks.m

```
function num = countblanks(phrase)

num = length(strfind(strtrim(phrase), ' '));

end
```

As a result of the program, the following will be displayed:

```
phraseblanks
   phrasemat =
   Hello and how are you?
   Hi there everyone!
   How is it going?
   Whazzup?
   Phrase 1 has 4 blanks
   Phrase 2 has 2 blanks
   Phrase 3 has 3 blanks
   Phrase 4 has 0 blanks
```

The function `strrep` finds all occurrences of a substring in a string and replaces them with a new substring. Function format: `strrep(string, old substring, new substring)`. For example,

```
strrep('abcdeabcde','e','x')
ans =
    abcdxabcdx
```

All strings can be of different length, so a new substring need not be the same length as the original.

In addition to the functions that find and replace substrings in a string, there are functions that separate the string into two substrings. The function `strtok` divides the string into two parts. There are several ways to call this function. The function takes a string as the first argument, then the first delimiter is searched, which can be a single character or a set of characters. By default, the delimiter is a space. Function format is as follows:

```
[token, rest] = strtok(string)
```

The function returns the beginning of the string to the first delimiter as variable `token`, and variable `rest` contains the rest of the string including the first delimiter. For example,

```
sentence1 = 'Hello there';
[word, rest] = strtok(sentence1)
word =
Hello
rest =
 there

length(word)
ans =
      5
length(rest)
ans =
      6 % the space at the beginning is included
```

Any character can be specified as a delimiter. Function format in this case is the following:

```
[token, rest] = strtok(string, delimeter)
```

For example, if you set the character `'l'` as the delimiter in the example above, the result will be

```
[word, rest] = strtok(sentence1,'l')
word =
He
```

```
rest =
llo there
```

The characters corresponding to the delimiter at the beginning of the string are ignored:

```
[firstpart, lastpart] = strtok(' materials science’)
firstpart =
materials % the leading space is ignored
lastpart =
science
```

9.9. String interpretation. Function `eval`

The function `eval` is used to interpret a string. If the string contains a function call, the function will be executed. For example, the string `'plot(x)'` contains a call of the function `plot`:

```
x = [2 6 8 3];
eval('plot(x)')
```

The function `eval` is often used when typing commands from the keyboard. For example, the user selects which type of plot to use for displaying data. The string entered by the user (in this case `'bar'`) is concatenated with the string `'(x)'`, thus creating the string `'bar(x)'`, which is interpreted by the function `eval` to draw a histogram:

```
x = [9 7 10 9];
whatplot = input('What type of plot?: ', 's');
What type of plot?: bar % keyboard input
eval([whatplot '(x)'])
```

9.10. Functions of string definition

The function `isletter` returns logical unit for each character in the string, if it is a symbol from the alphabet, and logical zero, if not. The

function `isspace` returns logical unit for each character in a string if it is a space:

```
isletter('EK127')
ans =
      1    1    0    0    0
isspace('a b')
ans =
      0    1    0
```

The function `ischar` returns logical unit if its argument is a vector of characters, and logical zero, if not:

```
vec = 'EK127';
ischar(vec)
ans =
      1

vec = 3:5;
ischar(vec)
ans =
      0
```

9.11. Conversion between strings and types of numbers

To convert numbers to strings, the following functions are used: `int2str` for integer values and `num2str` for floating-point numbers (this function also works with integer values). For example,

```
num = 38;
s1 = int2str(num)
s1 =
38
length(num) % num is a number
ans =
      1
>> length(s1) % s1 is a string
ans =
      2 %
```

The function `num2str` for converting floating-point numbers to strings can be called in several ways. If the argument is a number, then a string will be created that contains four decimal places of the initial number, which is the standard form of displaying floating-point numbers in MatLab. The number of characters can be specified as an integer argument, and the number of decimal places is set in the format of the string:

```
str2 = num2str(3.456789)
str2 = 3.4568
   >> length(str2)
ans =
      6
   >> str3 = num2str(3.456789,3)
str3 =
      3.46
   >> str = num2str(3.456789,'%6.2f')
str =
      3.46
```

The function `str2num` converts a string to a number that is stored in floating-point number (double) format:

```
num = str2num('123.456')
   num =
     123.4560
```

If the string contains several numbers separated by spaces, the function `str2num` returns a vector (in the double format by default). For example,

```
mystr = '66 2 111';
>> numvec = str2num(mystr)
numvec =
      66    2    111
>> sum(numvec)
ans =
      179
```

The function `str2double` is similar to `str2num` and is preferable to use. However, it can only be used to convert a scalar, it will not work with the `mystr` variable.

Practice

9.1. Write a function that takes two input string variables: the name `filename` and extension `ext`, and forms the filename `filename.ext` from them.

9.2. Write a function that generates two random numbers in the range from 10 to 99 inclusive. Then the function should concatenate these two numbers into a string variable, for example, if numbers 17 and 43 are generated, the result of the function will be the string `1743`.

9.3. Write the function `nchars`, which generates a string of n characters without using loops and selection statements:

```
nchars('&',7)
ans =
      &&&&&&&
```

Write a function that receives a list of words separated by `'/'` character. The function must return the same words separated by a space. The task should be solved without using loops.

9.4. Write a function `wordscr`, which takes some word and then rearranges the letters of this word in a random order, for example:

```
wordscr('argument')
ans =
      argentum
wordscr('argument')
ans =
      guterman
wordscr('argument')
ans =
      greatumn
```

10

Functions

10.1. Functions that return multiple variables

Functions in MatLab can be divided into three categories:
1) functions that evaluate and return one variable;
2) functions that evaluate and return multiple variables;
3) functions that do not return variables.

Although the above classification of functions is relative, the differences between these three types determine the format of the function header, as well as the way in which functions should be defined.

Each function in MatLab consists of the following parts: 1) the function header; 2) a comment describing the purpose of the function; 3) the body of the function, which includes the function code.

The function header includes the following elements:

- reserved word function;
- names (one or more) of the variables returned by the function followed by operator = if the function returns any variable; in the case that several variables are returned, the variable names must be enclosed in square brackets;
- the name of the function (the name of the function must match the name of the m-file containing this function);
- names (one or more) of the input variables enclosed in parentheses названия.

For example, the function `sorting`:

```
function [t, Asort, A] = sorting(R, n)
```

has two input variables (`R` and `n`) and three output variables (`t, Asort, A`). This function must be located on the hard disk in the file `sorting.m`.

A function that returns multiple variables has the following general form:

```
function [output variables] = name (input variables)
% comment describing the purpose of the function
Body of the function
```

In Example 10.1, a function is considered that calculates two values: the area and perimeter of a circle.

Example 10.1.

```
function [area, circum] = areacirc(rad)

% areacirc returns area and perimeter of a circle
% Format: areacirc(radius)

area = pi * rad .* rad;
circum = 2 * pi * rad;
```

Example of use:

```
[a, c] = areacirc(4)

a =
   50.2655
c =
   25.1327
```

It should be noted that the order of the returned variables is important. In this example, the function first returns the area, and then the perimeter of the circle.

The function `help` returns a comment that describes the purpose of the function below the function header:

```
help areacirc

areacirc returns area and perimeter of a circle
Format: areacirc(radius)
```

The function `areacirc` from Example 10.1 can be called from the command line, as is shown in the example, or from the script. In Example 10.2, the script `areacirc_call`, which prompts the user for the radius of the circle, calls the function `areacirc` to calculate the area and perimeter of the circle, and outputs the result.

Example 10.2.

```
radius = input('Input the radius of the circle: ');

[area, circ] = areacirc(radius);

fprintf('For the circle of radius %.1f\n', radius)
fprintf('the area is %.1f , the perometer is %.1f\n', area, circ)
```

Example of use:

```
reacirc_call

Input the radius of the circle: 5

For the circle of radius 5.0

the area is 78.5, the perometer is 31.4
```

Example 10.3 shows a function that returns three variables: the function takes the number of seconds and returns the number of hours, minutes, and seconds.

Example 10.3.

```
function [hours, minutes, secs] = breaktime(totseconds)

% function breaktime separates the total number of seconds into
% hours, minutes, and seconds
% Format: breaktime(total number of seconds)

hours = floor(totseconds/3600);
remsecs = rem(totseconds, 3600);
minutes = floor(remsecs/60);
secs = rem(remsecs,60);
```

Example of use:

```
[h, m, s] = breaktime(7515)

h=
   2
m=
   5
s=
   15
```

10.2. Functions that do not return variables

Many functions do not calculate variables, but only perform such specific operations as outputting formatted information. The general form of such functions is as follows:

```
function name(input arguments)
% comment describing the purpose of the function
Body of the function
```

In this case, there is no output variables in the function header, as well as the "=" operator. In Example 10.4, a function is considered that displays the numbers in a formatted form.

Example 10.4.

```
unction printem (a,b)

% printem prints two numbers in the form of a sentence
% Format: printem (num1, num2)

fprintf('The first number is %.1f and the second number is
%.1f\n',a,b)
```

Example of use:

```
printem(3.3, 2)

The first number is 3.3 and the second number is 2.0
```

In all the examples presented so far, each function had at least one input argument, which was displayed in the function header. This method of calling functions has the name "call by value". In some cases, functions do not require input variables. Consider a function that displays a random real number with two decimal places:

```
function printrand()

% printrand displays a random real number
% Format: printrand or printrand()
fprintf('Random number: %.2f\n',rand)
```

Example of use:

```
printrand()

Random number: 0.94
```

10.3. Anonimous functions

The advantage of anonymous functions is that they do not need to be stored in a separate m-file. This can greatly simplify programs, since if the repetitive calculations are simple, you can use anonymous functions, thereby reducing the total number of m-files. Anonymous functions can be created both in the command line, and in a script or function. The syntax of an anonymous function is as follows:

```
fnhandlevar = @ (arguments) functionbody;
```

where `fnhandlevar` is a function handle that can be used to call the anonymous function. The handle is returned using the @ operator and is then assigned to the variable `fnhandlevar`. The arguments `arguments` enclosed in parentheses correspond to the input arguments of the function, as in ordinary functions. `functionbody` is a function body that can contain any MatLab expression. For example, an anonymous function that calculates and returns the area of a circle can be written in the following form:

```
cirarea = @ (radius) pi * radius .^2;
```

The handle name for this function is `cirarea`. There is only one input argument (circle radius). The body of the function is the expression `pi*radius.^2`. The operator `.^` The operator provides the possibility of using the vector as an input argument. The function is called using a handle. Calling an anonymous function is similar to calling an ordinary function:

```
cirarea(4)

ans = 50.2655

areas = cirarea(1:4)

areas =

    3.1416 12.5664 28.2743 50.2655
```

Unlike the functions stored in m-files, in the case of an anonymous function, parentheses for input arguments must be present, even if the input arguments are missing, for example:

```
prtran = @ () fprintf('%.2f\n',rand);

prtran()

0.95
```

If only the handle name is specified in the command line, the contents of the anonymous function is displayed:

```
prtran

   prtran =

       @ () fprintf('%.2f\n',rand)
```

Anonymous functions can be stored in mat-file and loaded later if necessary:

```
cirarea = @ (radius) pi * radius .^2;

save anonfns cirarea
```

A single mat-file can store several anonymous functions. This is their advantage: anonymous functions do not need to be stored in separate m-files, but can be grouped and stored in a single mat-file.

10.4. Using the function handle

A function handle can be created not just for anonymous functions, but for an ordinary function. For example, when executing a command

```
facth = @factorial;
```

the operator @ calls the handle of the function `factorial` and stores it in the variable `facth`. This handle can be used as a function to which it is attached:

```
facth(5)

ans =

   120
```

One of the reasons for using function handles is the ability to use functions as input arguments for other functions. For example, suppose there is a function that creates a vector `x`. The vector `y` is created as some function calculated at each point of the vector `x`, and after that a plot of the dependence $y(x)$ is drawn:

```
function fnfnexamp(funh)

% fnfnexamp gets the function handle
% and plots this function on the segment x
% format: fnfnexamp (function handle)

x = 1:.25:6;
y = funh(x);
plot(x,y,'ko')
xlabel('x')
ylabel('fn(x)')
title(func2str(funh))
```

If we now use a standard function as the input argument, for example sin, cos, or tan, without using operator @, then an error will occur:

```
fnfnexamp(sin)

   Error using sin
   Not enough input arguments.
```

Instead, the function handle should be used:

```
fnfnexamp(@sin)
```

The function `func2str` in the body of the function `fnfnexamp` converts the name of the function handle into a string for further use in the header of the plot.

Any handle of any function can be used as an input argument of the `fnfnexamp` function. It should be noted that if a variable already contains a function handle, its use as an input argument does not require @ operator, for example:

```
fnfnexamp(cirarea)
```

There are a number of built-in functions in MatLab, the arguments of which can be other functions. One of such function is `fplot`; this function draws a plot of a function in a given range. The format of `fplot` is as follows:

```
fplot(fnhandle, [xmin, xmax])
```

For example, to construct the function $y = \sin(x)$ on the interval $[-\pi, \pi]$, we need to use the handle of the function $\sin(x)$:

```
fplot(@sin, [-pi, pi])
```

Another example of such a built-in function is `feval`, which calls a function by the handle with a specific input argument:

```
feval(@sin, 3.2)

ans =
```

```
    -0.0584
```

The function `fzero` finds the zero value of the function near the specified value:

```
fzero(@cos,4)

ans =

    4.7124
```

10.5. Variable number of arguments

Often there are situations where it is convenient to create a function that allows the use of a variable number of both input arguments and output variables. The built-in array of cells `varargin` can be used to store a variable number of input arguments, and the array `varargout` is used for the output variables. These variables `varargin` and `varargout` have the type of cell array, because both input arguments and output variables may have different types. The function `nargin` returns the number of input arguments, and `nargout` determines the number of variables returned by the function. In Example 10.5, a function is considered that provides a variable number of both input and output parameters [1].

Example 10.5. The task is to write a function whose input parameters are the coordinates of a point (px, py), as well as the parameters of circles $(x1, y1, r1)$, $(x2, y2, r2)$, ..., where xi, yi are the coordinates of centers of circles; ri are the radii of the circles; the number of circles can be arbitrary. The function must determine whether the given point lies within any circle, find the number of circles inside which the point lies, and also give the numbers of these circles in the list of output arguments.

```
function [w, varargout] = point2(varargin)

% the function determines if the given point with coordinates
% (px, py) lies in circles with centers in (x1,y1), (x2, y2), etc.
% and radii r1, r2 etc.
%
% Usage:
% w = point(px,py,[x1,y1,R1],[x2,y2,R2],...)
% w is 1 if the point is in any circle, 0 otherwise
```

```
%
% [w, NC] = point(px,py,[x1,y1,R1],[x2,y2,R2],...)
% NC is equal to the number of circles containing the point
%
% [w, NC, Nums] = point(px,py,[x1,y1,R1],[x2,y2,R2],...)
% Vector Nums contains the numbers of circles in which the point
% is located
% defining the coordinates of the point from the first two cells
Xpoint = varargin{1};
Ypoint = varargin{2};

% Finding the number of given circles
% (the number of varargin cells without the first two)
Ncircle = length(varargin) - 2;

% Extraction of the coordinates of the centers and radii
for i = 1 : Ncircle
   Xcircle(i) = varargin{i+2}(1);
   Ycircle(i) = varargin{i+2}(2);
   Rcircle(i) = varargin{i+2}(3);
end

w = 0;

NC = 0;

% Loop through circles
for i = 1:Ncircle

   % Calculation of the distance from the point to the center of
   % the current circle
   dist = sqrt((Xpoint-Xcircle(i))^2+(Ypoint-Ycircle(i))^2);

   % Comparison of the distance with the radius of the circle
   if  dist <= Rcircle(i)
      w = 1;  % Required circle is found
      % Increase the number of circles found
      NC = NC + 1;
      % Save circle number
      Nums(NC) = i;
   end
end

% Record results in the output array of cells depending on the
% number of output parameters

if nargout == 2
    varargout{1} = NC;
elseif nargout == 3
```

```
    varargout{1} = NC;
    varargout{2} = Nums;
end

% Displaying data in a graphics window
figure;

% Drawing circles
t = 0:pi/20:2*pi;
for i = 1:Ncircle
   x = Rcircle(i)*cos(t) + Xcircle(i);
   y = Rcircle(i)*sin(t) + Ycircle(i);
   plot(x,y)
   hold on
end

% Drawing the point in red
plot(Xpoint,Ypoint, 'or')
hold off
axis square % equal scale of axes
```

Practice

10.1. Write an anonymous function that calculates the value of the hyperbolic sine of the argument.

10.2. Write an anonymous function that receives two vectors x and y of the same length, and a handle for drawing a plot. The function must draw the desired type of the plot of the dependence of y on x, for example:

```
fun(x,y,@bar)
```

10.3. Write a function that receives a various number of input parameters: length, width and depth. If only the first two parameters are entered, the function must return the area of the rectangle; if all three parameters are entered the function must return the volume of the parallelepiped and the area of the rectangle of its base.

11

Cell arrays

11.1. Creating a cell array

There is a data type in MatLab that is missing in most programming languages, a *cell array*. The cell arra is an array, but unlike conventional vectors and matrices, the elements of this array can contain variables of different types.

There are several ways to specify a cell array. Suppose you want to create an array of four elements, with the first element is an integer number, the second is a character, the third is a vector, and the fourth is a string. Cell arrays are specified using curly brackets. For example:

```
cellrowvec = {23, 'a', 1:2:9, 'hello'} % specifying a
                cell array in the form of a row-vector
cellrowvec =
           [23]    'a'    [1x5 double]    'hello'
```

The cell array can also be specified as a column or matrix:

```
cellcolvec = {23; 'a'; 1:2:9; 'hello'} % specifying a
                  cell array as a column-vector
cellcolvec =
                  [23]
                  'a'
                  [1x5 double]
                  'hello'


cellmat = {23 'a'; 1:2:9 'hello'} % specifying a cell
                         array as a matrix

cellmat =
    [        23]    'a'
```

```
    [1x5 double]    'hello'
```

The data type of a cell array in MatLab is named `cell`:

```
class(cellmat)  % determining the type of variable
                  cellmat
ans =
                  cell
```

Another way to create a cell array is elementwise assignment of values to cells. In this case, if the size of the cell array is known in advance, the cell array should be initialized first:

```
mycellmat = cell(2,2) % creation of cell array of
                required dimension; cells are
                created empty

mycellmat =
    []    []
    []    []
```

11.2. Displaying and reffering to elements of a cell array

There are two types of referring to cell array elements: referring to the cell and referring to the cell contents.

When curly brackets are used, the contents of the cell is referred to; this method of addressing is called *indexing the contents of a cell*. For example:

```
cellrowvec{2}
ans =
                  a
class(ans)  % determining the type of the variable
                  ans, to which the result of the
                  previous command was written
ans =
```

```
            char % type of variable ans is character
```

Similarly, to refer to the contents of the cells being the part of a matrix structure:

```
cellmat{1,1}
ans =
                  23
```

Assignment of values to elements of a cell array is also analogous to the operation of assignment of values to matrix elements and vectors; for example, for the array `mycellmat` initialized in 11.1, assignment the value of 23 to the element in the first line and in the first column is done as follows:

```
mycellmat{1,1} = 23
mycellmat =
                  [23]    []
                    []    []
```

When parentheses are used, the *cell* is referred to; such a method of addressing is called *cell indexing*. For example,

```
cellcolvec(2)
ans =
                  'a'
class(ans)  % determining the type of the variable
                  ans, to which the result of the
                  previous command was written
ans =
                  cell % type of variable ans is cell
```

If the referred cell is itself a structure, only the type of this structure will be displayed when the cell is indexed; the object itself can be displayed when indexing the contents of the cell:

```
cellmat(2,1)
ans =
                [1x5 double] % this cell contains a
                vector of length 5
cellmat{2,1} % this command displays the contents of
                the vector
ans =
                1  3  5  7  9
cellmat{2,1}(4)  % a certain element of this vector
                can    be    referred    to    using
                parentheses
ans =
                7
```

There are several ways to display a cell array. The function `celldisp` displays the contents of all elements:

```
celldisp(cellrowvec)

cellrowvec{1} =
     23
cellrowvec{2} =
a
cellrowvec{3} =
     1     3     5     7     9
cellrowvec{4} =
hello
```

The function `cellplot` displays the cell array in the graphics window; this method does not display the contents of the array, but only its structure (Fig. 11.1).

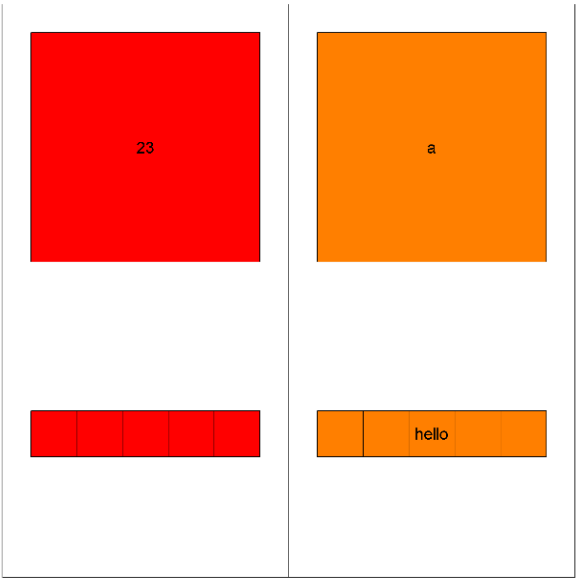


Fig. 11.1. Displaying a cell array using the function `cellplot`

Many functions and operators that can be applied to vectors and matrices are also applicable to cell arrays:

```
length(cellrowvec)   % the length of the array
                     cellrowvec
ans =
             4
size(cellcolvec) % dimensions of the array cellcolvec
ans =
             4     1
cellrowvec{end}   % the last element of the array
                  cellrowvec
ans =
            hello
```

To delete an element from a cell array, cell indexing is used:

```
cellrowvec
            [23]    'a'    [1x5 double]    'hello'
cellrowvec(2) = [] % deleting the second element
cellrowvec =
            [23]    [1x5 double]    'hello'
```

11.3. Storing strings in a cell array

One of the most useful features of cell arrays is the ability to store strings of various lengths:

```
names = {'Andrew', 'Anton', 'Alexander'}

names =
    ' Andrew'    ' Anton'    ' Alexander'
```

Each string has its own length and does not contain unnecessary spaces. The length of each string can be displayed using the `for` loop:

```
for i = 1 : length(names)
      disp(length(names{i}))
end

     6

     5

     9
```

Using the function `cellstr`, an array of characters can be converted into a cell array, with extra spaces being ignored:

```
greetman  =  char('Hello','Goodbye')  %  creating  a
                                  character array

greetman =
```

```
Hello
Goodbye

size(greetman)

ans =
     2     7 % The first string is completed with two
                 spaces to ensure that all the
                 strings in the character array are
                 of the same length

cellgreets = cellstr(greetman) % converting a
                 character array to a cell array

cellgreets =

    'Hello'
    'Goodbye'

size(cellgreets{1})

ans =
     1     5 % the first element has a length of 5
```

Practice

11.1. Write a program that initializes three cell arrays, the first of which contains several names (Andrey, Egor, Alexey, etc.), the second contains several verbs (eats, likes, sees, etc.), and the third contains several nouns (apples, stones, books, etc.). The program should randomly select one item from each cell array and display the resulting sentence on the screen.

11.2. Write the function `buildstr`, which has two input arguments: a character and a positive integer n. The function has to create and display an array of n cells, each of which contains a string of length from 1 to n, with the first cell containing a string that matches the input character, and each next line is obtained from the previous by attaching

a character whose ASCII code is greater than the ASCII code of the previous character by one. For example:

```
buildstr('щ',3)

ans =
      'щ'
      'щъ'
      'щъы'
```

The ASCII code of a character can be specified using the function `double`:

```
double('щ')

ans =
        1097
```

12

Structures

12.1. Creating structures

The *structures* in MatLab are the way to represent data that has a logical relation. This relationship is implemented in the form of *fields.* Each field has its own name, which helps to make it clear what kind of variable is stored in the structure. Structures are not arrays, so they do not have indexing of elements; it is not possible also to implement vector operations and loops on them.

A structure can be created simply by storing variables in fields with the use of assignment operators or structure functions. Consider creating a structure containing information about software that is being sold in a certain store. The structure should consist of the following fields: product numbers (article); purchase prices; sale prices; identifier of the type of software. The structure for one of the products might look like this:

Package			
item_no	cost	price	code
123	1999	3999	g

This structure has the name "Package" and contains four fields: "item_no", "cost", "price", "code".

One way to create a structure is to use the function `struct`. The field names of a structure are specified using string variables, followed by the value of this field:

```
package = struct('item_no', 123, 'cost', 1999,
'price', 3995, 'code', 'g')
```

```
package =
    item_no: 123
       cost: 1999
      price: 3995
       code: 'g'

class(package) % defining the type of the variable
package

ans =
      struct % type of variable is structure
```

An alternative way to create structures that is less efficient than the previous one is to use the dot operator "." to indicate fields within the structure, for example:

```
package.item_no = 123;
package.cost = 1999;
package.price = 3995;
package.code = 'g';
```

The structure can be completely assigned to another structure with the help of the operator "="; both structures will have similar fields and their contents. For example, below the variables of one structure are copied to another structure, and then two fields are changed:

```
newpack = package;
newpack.item_no = 111;
newpack.price = 3495
newpack =
     item_no: 111
     cost: 1999
     price: 3495
     code: 'g'
```

To display the structure, the universal function `disp` is used, which shows all the fields and their contents:

```
disp(package)
      item_no: 123
      cost: 1999
      price: 3995
      code: 'g'

disp(package.cost)
      1999
```

The function `rmfield` removes a field from the structure. This function returns a structure with a deleted field, but does not change the initial structure:

```
b = rmfield(package, 'code')

b =
     item_no: 123
     cost: 1999
     price: 3995
```

12.2. Using structures as function arguments

Both the entire structure and individual fields can be used as arguments to a function. In Example 12.1, two versions of the program that calculates the profit from the sale of software are considered. In the first variant, the entire structure variable is used as the input argument, and only some structure fields in the second.

Example 12.1.

```
function profit = calcprof(packstruct) % the input argument of the
                                       function is the structure

profit = packstruct.price - packstruct.cost;
end

calcprof(package)
ans = 1996
```

```
function profit = calcprof2(oneprice, onecost) % the input
                            arguments of the function are the
                            values of fields of the structure

profit = oneprice - onecost;
end

calcprof2(package.price, package.cost) % in this case it is
                         necessary to specify the required fields
                         of the structure
ans = 19.9600
```

12.3. Vectors of structures

Many programs, such as programs for working with databases, use vectors of structures to store information. There are several ways of forming a vector of structures. One of them is the creation of structure variables with the subsequent forming of a vector:

```
packages(1) = struct('item_no', 123, 'cost', 1999,
'price', 3995, 'code', 'g');
packages(2) = struct('item_no', 456, 'cost', 599,
'price', 4999, 'code', 'l');
packages(3) = struct('item_no', 587, 'cost', 1111,
'price', 3333, 'code', 'w');
```

An alternative way is to use the function `repmat`, which specifies a vector of structures with predefined dimensions. `Repmat` creates a matrix that contains copies of the first argument, for example:

```
example_struct = struct('item_no', 123, 'cost', 1999,
'price', 3995, 'code', 'g')
packages = repmat(example_struct,1,3);
packages(2) = struct('item_no', 456, 'cost', 599,
'price', 4999, 'code', 'l');
packages(3) = struct('item_no', 587, 'cost', 1111,
'price', 3333, 'code', 'w');
```

Thus, in this variant, a matrix of dimensions 1×3 is created containing copies of the structure `example_struct`, and then the matrix elements (vectors) are replaced element by element.

If you enter a variable name in the command line, only the size of the vector of structures and the field names will be displayed:

```
packages

packages =
1x3 struct array with fields:
     item_no
     cost
     price
     code
```

The variable `packages` is now a vector of structures, so each element of the vector is a structure. To display the element of the vector (i.e. one structure), indexing is needed:

```
packages(2)

ans =
      item_no: 456
      cost: 599
      price: 4999
      code: 'l'
```

To refer to the contents of a specific field, first the structure should be referred to, and then the field. This means that first the indexing of the structure vector should be done, and then the dot operator "." to refer to the field:

```
packages(1).code

ans = g
```

Thus, there are three levels of referring to this structure. The variable `packages`, the highest level, is the vector of structures. Each element of

the vector is a structure. The fields of each structure are the lowest level. Let use the `for` loop to display the contents of `packages`:

```
for i = 1:length(packages)
    disp(packages(i))
end
```

```
     item_no: 123
     cost: 1999
     price: 3995
     code: 'g'

     item_no: 456
     cost: 599
     price: 4999
     code: 'l'

     item_no: 587
     cost: 1111
     price: 3333
     code: 'w'
```

Using the dot operator ".", all values of the vector of structures corresponding to a specific field name can be displayed:

```
packages.cost
```

```
      ans = 1999
      ans = 599
      ans = 1111
```

These values can be stored as a vector:

```
pc = [packages.cost]
```

```
pc =
      1999   599  1111
```

Example 12.2 shows a function that receives a structure and displays it on the screen in tabular form.

Example 12.2.

```
function printpackages(packstruct)

fprintf('\nItem #  Cost  Price  Code\n\n')
no_packs = length(packstruct);

for i = 1 : no_packs
    fprintf('%6d %6d %6d %6c\n',...
    packstruct(i).item_no, ...
    packstruct(i).cost, ...
    packstruct(i).price, ...
    packstruct(i).code)
end
```

The function implements a loop of vector structure elements, and uses the dot operator to display the contents of fields. Example of using `printpackages`:

```
printpackages(packages)

Item #  Cost  Price  Code
   123  1999  3995     g
   456   599  4999     l
   587  1111  3333     w
```

Practice

12.1. Implement a structure that contains information about the elements of the Periodic table. The number of structure vectors, as well as the field names, can be selected at your choice. Display the contents of the structure on the screen.

13

Fitting and interpolation

13.1. Polynomials

Often it is necessary to approximate the available tabular data of any variable f at points $\{x_i\}$ on a segment $[x_{\min}, x_{\max}]$ by some function in order to perform *interpolation* (i.e., to approximate the values of f at the points $x_j \in [x_{\min}, x_{\max}]$; $x_j \neq \{x_i\}$) or *extrapolation* (that is, to approximate the values of f at the points $x_j \in [x_{\min}, x_{\max}]$; $x_j \neq \{x_i\}$). One of the simplest methods of fitting is the use of polynomial functions of various orders. In MatLab, polynomials are represented by vectors containing polynomial coefficients, for example, the polynomial $2x^3 - 4x^2 + x - 3$ is represented by the vector `[2 -4 1 -3]`, and the polynomial $x^5 + 4x^2 + x$ by the vector `[1 0 0 4 1 0]`. The function `roots` is used to find the zeros of a polynomial.

Example 13.1. Find solutions of the equation $4x^3 - 2x^2 - 8x + 3 = 0$.

```
roots([4 -2 -8 3])

ans =
       -1.3660
        1.5000
        0.3660
```

The function `polyval` evaluates the value of a polynomial at a given point.

Example 13.2. Calculate the value of the function $f(x) = -2x^2 + x + 4$ at the point $x = 3$.

```
p = [-2 1 4];
polyval(p,3)
ans =
       -11
```

The function `polyfit` uses the least squares method to find the coefficients of the polynomial that best approximates the tabulated values of the function. The function has three input arguments: the vector of coordinates, the vector of the values of the function in these coordinates, and the degree of the approximating polynomial. The coefficients calculated by `polyfit` can be used to construct an approximating polynomial using the function `polyval`.

Example 13.3. Consider fitting data (Fig. 13.1):

```
x = 2:6; % points at which measurements were performed
y = [65 67 72 71 63]; % the values

coeff1 = polyfit(x,y,1); % calculation of the coefficients of
                         polynomials of various orders
coeff2 = polyfit(x,y,2);
coeff3 = polyfit(x,y,3);
coeff4 = polyfit(x,y,4);

data_x = linspace(min(x),max(x),100); % grid of points for the
                                      construction of smooth curves

curve1 = polyval(coeff1,data_x); % calculation of approximating
                                 polynomials
curve2 = polyval(coeff2,data_x);
curve3 = polyval(coeff3,data_x);
curve4 = polyval(coeff4,data_x);

figure

plot(x,y,'r.','MarkerSize',25)
hold on
plot(data_x,curve1,'k','LineWidth',2)
plot(data_x,curve2,'b','LineWidth',2)
plot(data_x,curve3,'g','LineWidth',2)
plot(data_x,curve4,'m','LineWidth',2)
legend('data','n=1','n=2','n=3','n=4')
axis([1.5 6.5 62 73])

set(gca, 'LineWidth',1)
set(gca, 'FontName', 'Trebuchet MS')
set(gca, 'FontSize', 8)
set(gca, 'FontWeight', 'bold')
```

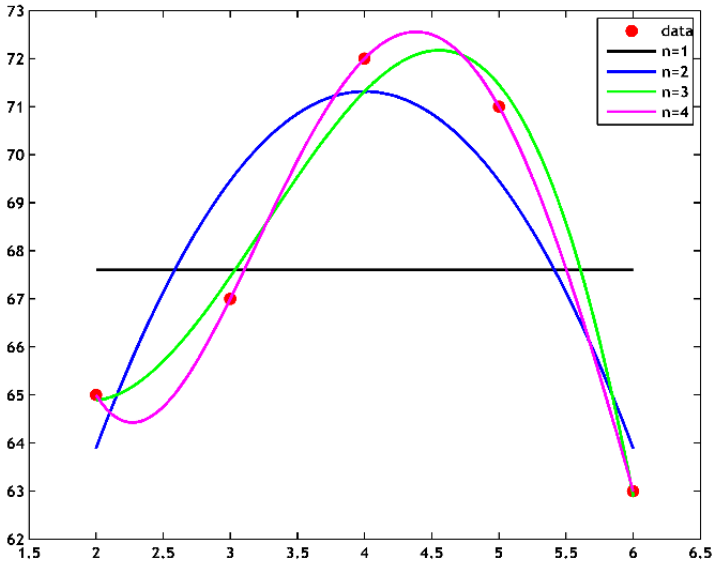


Fig. 13.1. Fitting data with polynomials of various orders

Example 13.4 shows the fitting data on the size of the US population in various years (Fig. 13.2).

Example 13.4.

```
load census % loading data on the US population in various years;
            this file comes with MatLab

% pop is the number of population
% cdate is a grid by years
[p,ErrorEst] = polyfit(cdate,pop,2); % approximation of data by a
                                     polynomial of the second order
% p are polynomial coefficients
% ErrorEst is the approximation discrepancy (error)

% calculation of approximated data from the obtained dependence
[pop_fit,delta] = polyval(p,cdate,ErrorEst);
% pop_fit is the approximated data
% delta is the error

% drawing a plot
figure

plot(cdate,pop,'d','MarkerSize',10,'MarkerFaceColor','b')
```

```
hold on
plot(cdate,pop_fit,'g-','LineWidth',2)
plot(cdate,pop_fit+3*delta,'r:','LineWidth',2)
plot(cdate,pop_fit-3*delta,'r:','LineWidth',2)

xlabel('Years', 'FontName', 'Trebuchet MS', 'FontSize', 10,
'FontWeight', 'bold');
ylabel('Population (millions)', 'FontName', 'Trebuchet MS',
'FontSize', 10,'FontWeight', 'bold');

grid on

legend({'Data','Fit','Error'}, 'Location', 'NorthWest')
set(gca, 'LineWidth',1)
set(gca, 'FontName', 'Trebuchet MS')
set(gca, 'FontSize', 10)
set(gca, 'FontWeight', 'bold')
```

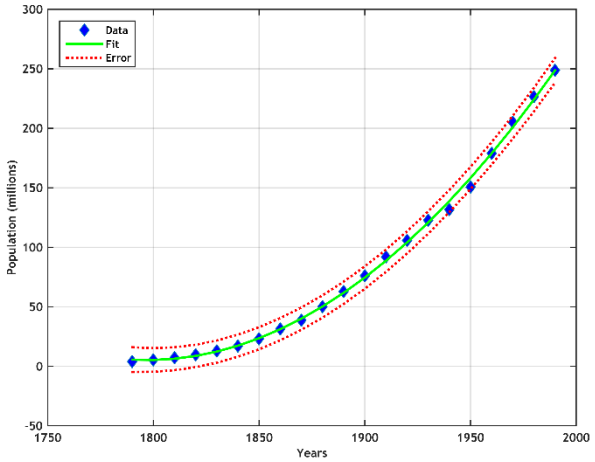


Fig. 13.2. Fitting the US population data

13.2. Least Squares Approximation

If the grid of coordinates $\{x_i\}$ and the vector of values $\{y_i\}$ on this grid are given, then the function `lsqcurvefit` is used to fit the values $\{y_i\}$ by the specified functional dependence $f = f(\alpha, \{x_i\})$, where α are the parameters of the function f determined by least squares method.

The function `lsqcurvefit` finds the values α_i minimizing the expression

$$\min_x \sum_i (f(\alpha, x_i) - y_i)^2.$$

Example 13.5 illustrates the use of `lsqcurvefit` to fit an existing data set to the sum of two exponents (see Fig. 13.3).

Example 13.5.

```
% initial data
Data = ...
  [0.0000    5.8955
   0.1000    3.5639
   0.2000    2.5173
   0.3000    1.9790
   0.4000    1.8990
   0.5000    1.3938
   0.6000    1.1359
   0.7000    1.0096
   0.8000    1.0343
   0.9000    0.8435
   1.0000    0.6856
   1.1000    0.6100
   1.2000    0.5392
   1.3000    0.3946
   1.4000    0.3903
   1.5000    0.5474
   1.6000    0.3459
   1.7000    0.1370
   1.8000    0.2211
   1.9000    0.1704
   2.0000    0.2636];

t = Data(:,1); % first column
y = Data(:,2); % second column

% plot the dependence of y on t
figure
axis([0 2 -0.5 6])
hold on
plot(t,y,'r.','MarkerSize',20)

% Finding a solution with the least squares method in the form
% y = c(1)*exp(-lambda(1)*t) + c(2)*exp(-lambda(2)*t), where
% x(1) = c(1)
```

```
% x(2) = lambda(1)
% x(3) = c(2)
% x(4) = lambda(2)

% Defining the required dependence in the form of anonymous
function
F = @(x,xdata)x(1)*exp(-x(2)*xdata) + x(3)*exp(-x(4)*xdata);

x0 = [1 1 1 1]; % initial values of c(1), lambda(1), c(2) and
                  lambda(2)
x = lsqcurvefit(F,x0,t,y) % least squares method

% drawing a plot
plot(t,F(x,t),'LineWidth',2)
hold off
set(gca, 'LineWidth',1)
set(gca, 'FontName', 'Trebuchet MS')
set(gca, 'FontSize', 8)
set(gca, 'FontWeight', 'bold')
```

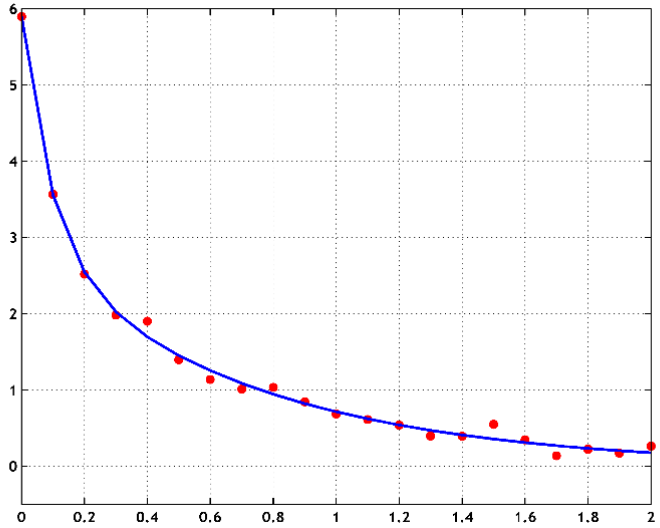


Fig. 13.3. Fiting data to the sum of two exponents

```
x =

    3.0069   10.5864    2.8890    1.4003
```

13.3. Splines

A *spline* is a piecewise polynomial function defined on the interval $[a, b]$ and having on it several number of continuous derivatives. Splines are often used in the problem of interpolating table values of a function. On each of the segments between the given points $\{x_i\}$, the values of the function $\{y_i\}$ are interpolated by polynomials of a certain degree that are joined at the boundaries of the segments. In MatLab, spline interpolation (Fig. 13.4) is performed using the function `interp1`. Example 13.6 illustrates the use of this function to interpolate the values of a simple function $y = \sin x$.

Example 13.6.

```
x = 0:pi/2:2*pi; % grid of points x
v = sin(x); % calculation of sine values on the grid

xq = 0 : pi/16 : 2*pi; % interpolation grid

vq1 = interp1(x,v,xq,'nearest'); % approximation of neighboring
elements
vq2 = interp1(x,v,xq); % linear interpolation
vq3 = interp1(x,v,xq,'spline'); % interpolation by cubic splines
vq4 = interp1(x,v,xq,'pchip'); % interpolation by cubic Hermitian
                               splines

% drawing a plot
figure

plot(xq,sin(xq),'Color',[0.1 0.7 1],'LineWidth',2.5)
hold on
plot(xq,vq1,'r--','LineWidth',2.5);
plot(xq,vq2,'Color',[0.5 0.1 1],'LineWidth',2.5);
plot(xq,vq3,'Color','k','LineWidth',2.5);
plot(xq,vq4,'Color',[1 0.5 0.1],'LineWidth',2.5);
plot(x,v,'o')
axis([0 2*pi -1.1 1.1])
box on
grid on
legend({'Original','Nearest','Linear','Spline','Pchip'})

set(gca, 'LineWidth',1)
set(gca, 'FontName', 'Trebuchet MS')
set(gca, 'FontSize', 8)
set(gca, 'FontWeight', 'bold')
```

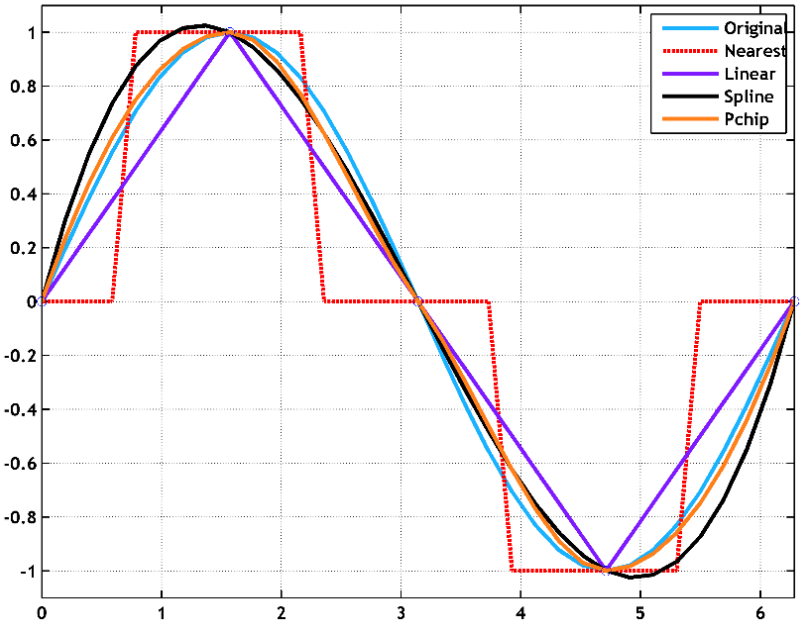


Fig. 13.4. Spline interpolation

14

Particle in potential well. Numerical solution of Schrödinger equation

14.1. Schrödinger equation

Consider the quantum-mechanical problem of particle motion in a potential field. The square of the modulus of the wave function $|\psi(\vec{r},t)|^2$ for such system determines the probability density to detect a particle at time t at a point of space with a radius vector $\vec{r}$. The time evolution of the function $\psi(\vec{r},t)$ is described by *the nonstationary Schrödinger equation*

$$i\hbar\frac{\partial\psi(\vec{r},t)}{\partial t} = -\frac{\hbar^2}{2m}\Delta\psi(\vec{r},t) + U(\vec{r},t)\psi(\vec{r},t),$$

where m is the mass of the particle; $U(\vec{r},t)$ is the external potential field.

For a time-independent potential, solutions of the nonstationary Schrödinger equation can be find in the form

$$\psi(\vec{r},t) = \phi(\vec{r})e^{-\frac{iEt}{\hbar}}.$$

The particle in the state described by the wave function $\psi(\vec{r},t)$ has a specific value of energy E. Substituting $\psi(\vec{r},t)$ in the first equation, we obtain *the stationary Schrödinger equation*:

$$-\frac{\hbar^2}{2m}\Delta\phi(\vec{r}) + U(\vec{r})\phi(\vec{r}) = E\phi(\vec{r})$$

or

$$H\phi(\vec{r}) = E\phi(\vec{r}),$$

where

$$H = -\frac{\hbar^2}{2m}\Delta + U(\vec{r})$$

is the energy operator (or Hamiltonian) of the system.

Suppose that the operator H has n eigenfunctions ϕ_n and n corresponding eigenvalues of energy E_n. The number n can be finite or infinite; the values of E_n can be discrete (*discrete spectrum*) or continuous (*continuous spectrum*), some values of E_n can coincide (*degenerate states*), Fig. 14.1. The state with the lowest energy is called *the ground state* of the system. The general solution of the Schrödinger equation can be represented as a superposition of the eigenfunctions of the Hamiltonian:

$$\psi(\vec{r}) = \sum_n C_n \phi_n(\vec{r}).$$

Here the symbol $\sum_n$ means summation over all discrete states and integration over the states of the continuous spectrum.

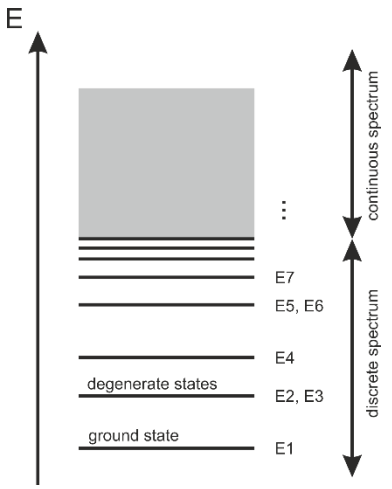


Fig. 14.1. Classification of eigenvalues of the energy operator

14.2. Infinite potential well

Consider a particle in a one-dimensional potential well of the width a with infinitely high walls (Fig. 14.2).

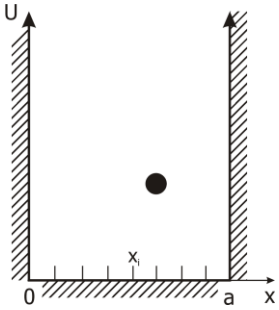


Fig. 14.2. Infinite one-dimensional potential well

The Hamiltonian of this system has the form

$$H = -\frac{\hbar^2}{2m}\frac{d}{dx^2} + U(\vec{r}),$$

where m is the mass of the particle; $U(x)$ is the potential of the well:

$$U(x) = \begin{cases} 0, & if \;\; 0 < x < a; \\ \infty, & if \;\; x < 0, x > a. \end{cases}$$

For convenience, let's change to a dimensionless system of units, setting $m \to 1;\ \hbar \to 1$, then the Hamiltonian can be written as follows:

$$H = -\frac{1}{2}\frac{d}{dx^2} + U(x).$$

Since the potential of the well is infinite for $x < 0$ and $x > a$, the solution of the Schrödinger equation exists only in the region $0 < x < a$. Substituting the expression for the Hamiltonian into the Schrödinger equation, we obtain:

$$-\frac{1}{2}\Psi''(x) - E\Psi(x) = 0; \quad 0 < x < a.$$

We divide now the domain $0 < x < a$ into n segments $[x_i, x_{i+1}], i = 1, \dots, n$ of length $h = a/n$, with $x_1 = 0;\ x_2 = h; \dots; x_{n+1} = nh = a$, and approximate $\Psi''(x_i)$ by the three-point difference formula:

$$\Psi''(x_i) = \frac{1}{h^2}\big(\Psi(x_{i-1}) - 2\Psi(x_i) + \Psi(x_{i+1})\big),$$

then we obtain:

$$-\frac{1}{h^2}(\Psi_{i-1} - 2\Psi_i + \Psi_{i+1}) - E\Psi_i = 0; \quad i = 1, \dots, n,$$

where $\Psi_i \equiv \Psi(x_i)$.

As an orthonormal basis consider a system of functions

$$\Phi_1 = |100 \dots 0\rangle; \quad \Phi_2 = |010 \dots 0\rangle; \quad \dots \quad ; \Phi_n = |000 \dots 1\rangle,$$

where unity means that the particle is on the corresponding segment, for example, the basis function Φ_2 corresponds to the situation when the particle is on the interval $[x_2, x_3]$. The dimension of this basis will be equal to the number of segments n of the partition. Any wave function $\Psi(x)$ can be expanded in terms of the basis functions Φ:

$$\Psi(x) = \sum_{i=1}^{n} C_i \Phi_i,$$

where the expansion coefficients C_i will be obtained further in the calculation, and the energy operator acts on the basis functions as follows:

$$H\Phi_i = -\frac{1}{2h^2}(\Phi_{i-1} - 2\Phi_i + \Phi_{i+1}).$$

Thus, the problem reduces to a system of linear equations

$$H\Psi = E\Psi,$$

where the matrix H has dimensions $n \times n$ and is tridiagonal:

$$H = \begin{pmatrix} 1/h^2 & -1/2h^2 & 0 & 0 & \dots & 0 \\ -1/2h^2 & 1/h^2 & -1/2h^2 & 0 & \dots & 0 \\ 0 & -1/2h^2 & 1/h^2 & -1/2h^2 & \dots & 0 \\ 0 & 0 & -1/2h^2 & 1/h^2 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & 0 & \dots & 1/h^2 \end{pmatrix}.$$

To solve the system of linear equations and find the eigenfunctions and eigenvalues of the matrix H, one must find a transformation F that tranfroms the basis Φ into an eigenbasis Φ' in which the matrix H is diagonal:

$$\Phi \overset{F}{\rightarrow} \Phi',$$

$$H\Phi_i' = E_i\Phi_i'.$$

The process of transition to an eigenbasis is called *the diagonalization* of the matrix H. In many modern mathematical packages there are sufficiently powerful built-in diagonalization procedures for matrices that allow finding eigenvectors and eigenvalues of arbitrary matrices of rather large sizes. The result of the diagonalization procedure is the column vector of the eigenvalues of the Hamiltonian, or *the spectrum* of the system,

$$E = \begin{pmatrix} E_1 \\ E_2 \\ \dots \\ E_n \end{pmatrix},$$

and the matrix C consisting of the column vectors corresponding to the expansion of the eigenfunctions Φ' in the initial basis Φ:

$$C = \begin{pmatrix} \begin{pmatrix} C_{11} \\ C_{21} \\ C_{31} \\ \dots \\ C_{n1} \end{pmatrix} & \begin{pmatrix} C_{12} \\ C_{22} \\ C_{32} \\ \dots \\ C_{n2} \end{pmatrix} & \dots & \begin{pmatrix} C_{1n} \\ C_{2n} \\ C_{3n} \\ \dots \\ C_{nn} \end{pmatrix} \end{pmatrix};$$

$$\Phi_i' = \sum_j C_{ji}\Phi_j.$$

Example 14.1 shows the code realizing the solution of the Schrödinger equation for a particle in an infinite well.

Example 14.1.

```
function [H,E] = InfiniteWell(a,n,k)

% Input arguments:
% a is the width of the well
% n is the number of segments
% k is the number of solutions

% Output arguments:
% H is Hamiltonian
% E is the spectrum

h = a/n; % width of the segment
x = [h/2 : h : a - h/2]; % coordinates of midpoints of segments

% creation of matrix H

H = zeros(n); % initialization; square matrix of zeros
H = H + (1/h^2)*diag(ones(1,n)); % adding the main diagonal
H = H - (1/2/h^2)*diag(ones(1,n-1),1); % adding the upper diagonal
H = H - (1/2/h^2)*diag(ones(1,n-1),-1); % adding the bottom
                                          diagonal

% diagonalization of the matrix H

[V,E] = eigs(H,k,0); % eigs is matrix diagonalization function;
          the first argument is the original matrix; the second
          argument is the number of solutions to be found; the
          third argument is the search area, the solutions will be
          found near this point
% E is a matrix of size k x k with eigenvalues along the main
          diagonal
% V is a matrix of the size n x k of the expansion of
          eigenfunctions in the basis functions
```

```
[E,DE] = sort(diag(E)); % transformation of the diagonal matrix E
into a vector and sorting its elements

% normalization of matrix V
A = sum(V(:,1).^2)*h;
V = V / sqrt(A);

% plotting the eigenfunctions
col = 'bgrcmk'; % color designation

figure

for i = 1 : k
    subplot(3,2,i)
    plot(x,V(:,DE(i)).^2,col(i),'LineWidth',3) % numerical
                                                solution
    hold on
    plot(x,2*(sin(pi*i*x)).^2,'ok') % exact solution
    set(gca, 'LineWidth',1)
    set(gca, 'FontName', 'Trebuchet MS')
    set(gca, 'FontSize', 8)
    set(gca, 'FontWeight', 'bold')
    xlabel('x', 'FontName', 'Trebuchet MS', 'FontAngle', 'italic',
     'FontSize', 10, 'FontWeight', 'bold')
    ylabel('y', 'FontName', 'Trebuchet MS', 'FontAngle', 'italic',
     'FontSize', 10, 'FontWeight', 'bold')
    title(['E=',num2str(E(i))],'FontSize', 12)
end
```

As a result of the program, the particle spectrum in an infinite potential well (the eigenvalues of the Hamiltonian) and the squares of the moduli of the wave functions (the eigenfunctions of the Hamiltonian, Fig. 14.3) are calculated, which are also compared with the exact solution

$$\Phi_i(x) = \sqrt{2}\sin(\pi x i);$$

$$E_i = \frac{\pi^2 i^2}{2}; \quad i = 1,2,\dots,\infty$$

for the well of the width $a = 1$. Numerical results for $n = 100$ segments and $k = 6$ solutions are shown by solid lines in Fig. 14.3, exact solutions are shown with black circles. Although the accuracy of the numerical solution is rather high, especially for the ground state with energy E_1, it is clear that with the increase in the energy level number, the calculation error increases. It should also be noted that in solving single-particle problems, the oscillation theorem always holds, that states that the number of zeros of the eigenfunction is one less than the number of the energy level to which it responds (degenerate states are not taken into account here).

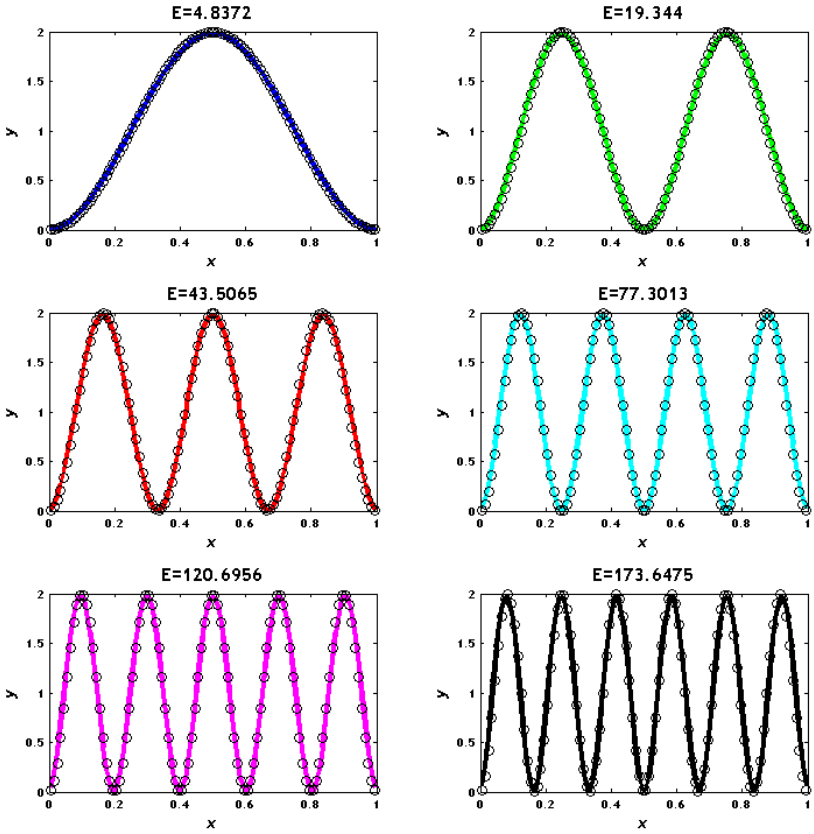


Fig. 14.3. Squares of moduli of the wave functions of a particle
in an infinite potential well

14.3. Finite potential well

Consider now a particle in a finite potential well (Fig. 14.4), where

$$U(x) = \begin{cases} \infty, & \text{if } x \le 0; \\ -U_0, & \text{if } 0 < x < a; \\ 0, & \text{if } x \ge a. \end{cases}$$

In a well of finite depth the states of the particles are divided into *bound states* E_d and *states of the continuous spectrum* E_c, whose energy satisfies the relation

$$E_c > \min\big(U(\pm\infty)\big).$$

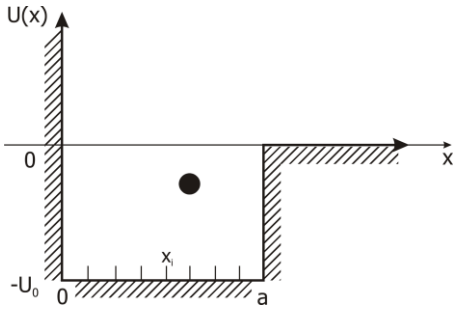


Fig. 14.4. Finite potential well

The eigenfunctions corresponding to the values of the energy E_c from the continuous spectrum outside the well behave like plane waves:

$$\Psi_k \underset{x\to\infty}{\longrightarrow} \text{const} \times e^{ikx},$$

while the eigenfunctions corresponding to bound states fade out exponentially outside the well (Fig. 14.5):

$$\Psi_n \sim e^{-\kappa x}, \quad \kappa > 0.$$

Consequently, for numerical calculation of the wave functions of bound states it is not enough to take into account the dimensions of the well, since the wave functions exist at $x > a$. The boundary a' of the region on which the solution is calculated must be determined from the condition of fading out of the solution obtained at distances $x \sim a'$ (see Fig. 14.5). If, after solving the problem and finding the eigenfunctions, it turns out that they have a finite value at the point $x = a'$, then it is necessary to increase the size of the region on which the solution is calculated and solve the problem once more.

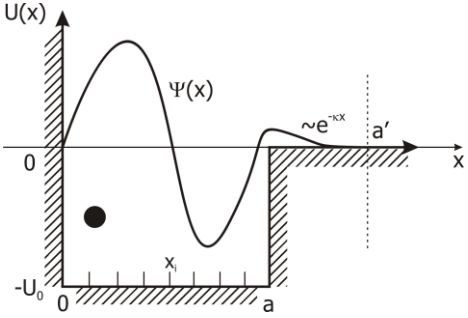


Fig. 14.5. Outside the well, the wave functions corresponding to the bound fade out exponentially

In comparison with the problem of a well with infinite walls, the spectral problem is now formulated as follows:

$$-\frac{1}{2}\Psi''(x) + (U(x) - E)\Psi(x) = 0; \quad 0 < x < a'.$$

Splitting the region $0 < x < a'$ into n segments of length h and approximating the functions similarly to the case of an infinite well, we obtain:

$$-\frac{1}{h^2}(\Psi_{i-1} - 2\Psi_i + \Psi_{i+1}) + (U_i - E)\Psi_i = 0; \quad i = 1, \dots, n,$$

where

$$U_i \equiv U(x_i) = \begin{cases} -U_0, & \text{if } 0 \le x_i \le a; \\ 0, & \text{if } a \le x_i \le a'. \end{cases}$$

The matrix H is now

$$H = \begin{pmatrix} 1/h^2 + U_1 & -1/2h^2 & 0 & 0 & \dots & 0 \\ -1/2h^2 & 1/h^2 + U_2 & -1/2h^2 & 0 & \dots & 0 \\ 0 & -1/2h^2 & 1/h^2 + U_3 & -1/2h^2 & \dots & 0 \\ 0 & 0 & -1/2h^2 & 1/h^2 + U_4 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & 0 & \dots & 1/h^2 + U_n \end{pmatrix}.$$

Because there is only a finite number of bound states in a well of finite depth, among all n solutions there are $n-k$ solutions corresponding to the states of the continuous spectrum. At numerical diagonalization of the matrix, this can lead to a problem that the matrix H can be *ill-conditioned*, i.e. some of its columns will be almost linearly dependent. The degree of conditionality of the matrix is characterized by the *conditioning number*, which for a symmetric real matrix is equal to the ratio of the moduli of the maximum and minimum eigenvalues. Large conditioning numbers (of the order of 1000 or greater) correspond to ill-conditioned matrices, leading to incorrect results at numerical diagonalization. Because the values of the energy of the particle corresponding to the states of the continuous spectrum can take any positive values $E_c > 0$, the conditioning number of the matrix H can take any arbitrarily large values, determined only by the number of the partition n. In modern mathematical packages, there are built-in procedures for estimating the conditioning number of matrices prior to their diagonalization.

To reduce the conditioning number, the following trick can be used. Let us shift the entire potential picture in Fig. 14.5 by a sufficiently large value of $U' > 0$, so that

$$|U'| \gg |U_0|,$$

and, accordingly, a term equal to U' will be added to each element of the main diagonal of the matrix H,

$$H' = H + \begin{pmatrix} U' & 0 & \dots & 0 \\ 0 & U' & \dots & 0 \\ \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & U' \end{pmatrix}.$$

After this transformation all quantities in the problem will be of the same order, and the conditioning number of the matrix H becomes smaller.

Accordingly, all eigenvalues of the matrix H' will differ from the eigenvalues of the matrix H by the value of U':

$$E' = E + U',$$

while on the eigenfunctions the shift of the potential picture will have no effect:

$$\Psi' = \Psi.$$

Example 14.2 shows the code realizing the solution of the Schrödinger equation for a particle in a finite well.

Example 14.2.

```
function [H,E] = FiniteWell(a,a1,n,k,U0)

% Input arguments:
% a is the width of the well
% a1 is the region where the solution is calculated
% n is the number of segments
% k is the number of solutions
% U0 is the depth of the well

% Output arguments:
% H is Hamiltonian
% E is the spectrum

h = a1/n; % % width of the segment
x = [h/2 : h : a1 - h/2]; % coordinates of midpoints of segments

% creation of matrix H
H = zeros(n); % initialization
H = H + (1/h^2)*diag(ones(1,n)); % adding the main diagonal
H = H - (1/2/h^2)*diag(ones(1,n-1),1); % adding the upper diagonal
H = H - (1/2/h^2)*diag(ones(1,n-1),-1); % adding the bottom
                                        diagonal

b = find(x < a); % finding the numbers of segments located in the
                   well
U = zeros(size(x)); % initialization
U(b) = -U0; % for the segments located in the well, the value of
                the potential is -U0

H = H + diag(U); % adding the potential of the well to H

% diagonalization of the matrix H
[V,E] = eigs(H,k,-U0); % the first argument is the matrix H; the
                         second argument is the number of solutions;
                         the third argument is the search area for
                         these solutions, here it is the bottom of
                         the well
```

```
% E is a matrix of size k x k with eigenvalues along the main
          diagonal
% V is a matrix of the size n x k of the expansion of
          eigenfunctions in the basis functions

[E,DE] = sort(diag(E)); % transformation of the diagonal matrix E
into a vector and sorting its elements

% normalization of matrix V
A = sum(V(:,1).^2)*h;
V = V / sqrt(A);

% plotting the eigenfunctions
col = 'bgrcmk';

figure

for i = 1 : k

    subplot(3,2,i)
    plot(x,V(:,DE(i)).^2,col(i),'LineWidth',2) % numerical
                                                        solution

    set(gca, 'LineWidth',1)
    set(gca, 'FontName', 'Trebuchet MS')
    set(gca, 'FontSize', 8)
    set(gca, 'FontWeight', 'bold')
    xlabel('x', 'FontName', 'Trebuchet MS', 'FontAngle', 'italic',
'FontSize', 10, 'FontWeight', 'bold')
    ylabel('y', 'FontName', 'Trebuchet MS', 'FontAngle', 'italic',
'FontSize', 10, 'FontWeight', 'bold')
    title(['E=',num2str(E(i))],'FontSize', 12)

    hold on

    plot(x,U/U0,'k','LineWidth',3) % profile of the well (not to
                                        scale)
    plot(x(b),E(i)/U0*ones(size(x(b))),col(i),'LineWidth',2)
end
```

As a result of the program, the particle spectrum in the finite potential well (the eigenvalues of the Hamiltonian) and the squares of the moduli of the wave functions of the particle are calculated. Plots of these functions for the parameters $a = 1$; $a1 = 2$; $n = 200$; $U0 = 100$; $k = 6$ are shown in Fig. 14.6. There are five bound states in this well, and the sixth state belongs to the continuous spectrum.

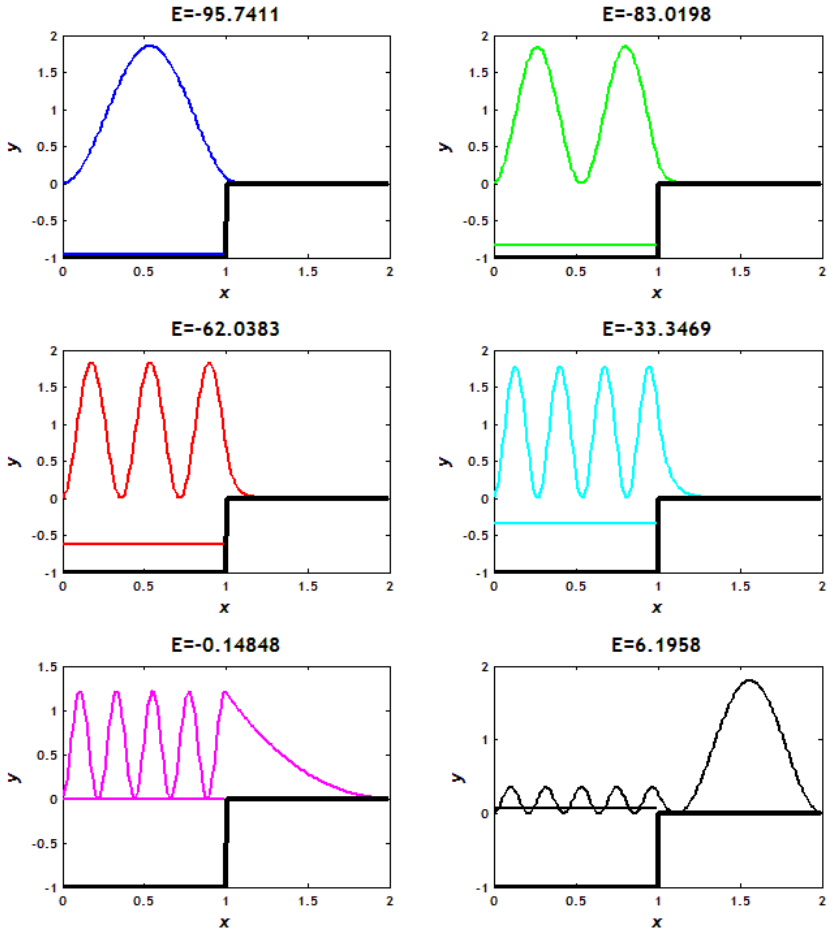


Fig. 14.6. Particle in a finite potential well. The squares of the moduli of the wave functions, the energy of the particle, and also the profile of the well (not to scale) are shown

The matrix H calculated in the example is a general matrix for the particle problem in a one-dimensional finite potential well. Setting different values of U_i, it is possible to solve the Schrödinger equation for a potential well of arbitrary shape.

Practice

14.1. Obtain bound energy levels and the corresponding wave functions of the particle in the potential $U(x) = -U_0$ for $-a < x < a$; $U(x) = 0$ otherwise. Choose such a value of U_0, at which there are five bound states in the well. $a = 1$.

14.2. Obtain bound energy levels and the corresponding wave functions of the particle in the potential $U(x) = -U_0$ for $1 - a < x < 1 + a$; $U(x) = \infty, x < 0$; $U(x) = 0$ otherwise. Choose such a value of U_0 at which there are five bound states in the well. At this value of U_0 study the behavior of the bound states depending on parameter a; obtain resonance values of a corresponding to a change in the number of bound states in the well.

14.3. Obtain bound energy levels and the corresponding wave functions of the particle in the potential $U(x) = -U_0 e^{-x^2/2a^2}$. Study the behavior of the bound states depending on the depth of the well; obtain resonance values of U_0 corresponding to the appearance of new bound states in the well.

14.4. Obtain bound energy levels and the corresponding wave functions of the particle in the potential of two two identical rectangular wells of depth U_0, the centers of which are located at a distance a from the origin, and the width of each well is $h < 2a$. Compare the degeneracy of the levels to the statement of the oscillator theorem.

15

Hydrogen atom

Consider a hydrogen atom placed in the coordinate origin. Since the Coulomb field of the nucleus has spherical symmetry, it is convenient to study the motion of an electron in a spherical coordinate system r, θ, φ. The potential energy of the Coulomb interaction of the electron and the nucleus has the form

$$U(r) = -\frac{e^2}{4\pi\varepsilon_0 r}.$$

We write the Schrödinger equation in the form

$$\Delta\psi + \frac{2m_e}{\hbar^2}\left(E + \frac{e^2}{4\pi\varepsilon_0 r}\right)\psi = 0,$$

where E is the total energy of the electron in the atom, which is to be found.

In spherical coordinates, the Schrödinger equation has the form

$$\frac{1}{r^2}\frac{\partial}{\partial r}\left(r^2\frac{\partial\psi}{\partial r}\right) + \frac{1}{r^2\sin\theta}\frac{\partial}{\partial\theta}\left(\sin\theta\frac{\partial\psi}{\partial\theta}\right) + \frac{1}{r^2\sin^2\theta}\frac{\partial^2\psi}{\partial\varphi^2} +$$

$$+\frac{2m_e}{\hbar^2}\left(E + \frac{e^2}{4\pi\varepsilon_0 r}\right)\psi = 0.$$

We represent the function $\psi(r, \theta, \varphi)$ as the product of two independent functions, depending on the radial and angular coordinates, respectively:

$$\psi(r, \theta, \varphi) = R(r)Y(\theta, \varphi).$$

Multiply all the terms of the equation by r^2 and divide by RY, then we get:

$$\frac{1}{R}\frac{\partial}{\partial r}\left(r^2\frac{\partial R}{\partial r}\right)+\frac{1}{Y\sin\theta}\frac{\partial}{\partial\theta}\left(\sin\theta\frac{\partial Y}{\partial\theta}\right)+\frac{1}{Y\sin^2\theta}\frac{\partial^2 Y}{\partial\varphi^2}+$$

$$+\frac{2m_e r^2}{\hbar^2}\left(E+\frac{e^2}{4\pi\varepsilon_0 r}\right)\psi=0.$$

In this equation, the first and the last terms depend only on r, and the second and third terms depend only on θ and φ, so the sum of all the terms of the equation can be zero only if the sum of terms that depend only on r and the sum of terms that depend only on θ and φ are equal in absolute value to the same constant λ and have opposite signs:

$$\frac{1}{Y\sin\theta}\frac{\partial}{\partial\theta}\left(\sin\theta\frac{\partial Y}{\partial\theta}\right)+\frac{1}{Y\sin^2\theta}\frac{\partial^2 Y}{\partial\varphi^2}=-\lambda;$$

$$\frac{1}{R}\frac{\partial}{\partial r}\left(r^2\frac{\partial R}{\partial r}\right)+\frac{2m_e r^2}{\hbar^2}\left(E+\frac{e^2}{4\pi\varepsilon_0 r}\right)\psi=\lambda.$$

The first of these equations is called *Legendre equation*; the eigenvalues $\lambda=l(l+1)$, and the quantity λ characterizes the square of the angular momentum of an electron in an atom:

$$\boldsymbol{l}^2=\frac{M_e^2}{\hbar^2}=l(l+1),$$

while the projection of this moment on an arbitrary direction is $l_z=m_l$.

Substituting the value of λ in the second equation, we obtain

$$\frac{\partial^2 R}{\partial r^2}+\frac{2}{R}\frac{\partial R}{\partial r}+\frac{2m_e}{\hbar^2}\left[E+\frac{e^2}{4\pi\varepsilon_0 r}-\frac{l(l+1)\hbar^2}{2m_e r^2}\right]R=0.$$

In this equation, the last two terms in square brackets are

$$U=\frac{e^2}{4\pi\varepsilon_0 r}-\frac{l(l+1)\hbar^2}{2m_e r^2},$$

which can be regarded as an effective potential energy consisting of Coulomb and centrifugal energy. The plot of the function U is shown in Fig. 15.1.

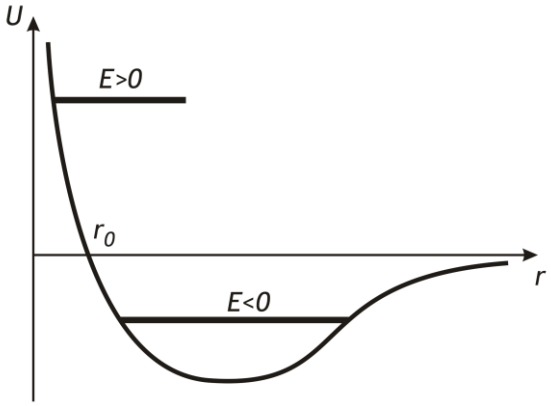


Fig. 15.1. Effective potential

For $r < r_0$ the potential energy is negative, and if the total energy of the electron is $E < 0$, then the spectrum of the system is discrete:

$$E = -\frac{m_e \alpha^2}{2\hbar^2 n^2};$$

$$\alpha = \frac{e^2}{4\pi\varepsilon_0}.$$

Finally, the wave functions of the hydrogen atom can be written in the form

$$\psi(r, \theta, \varphi) = R_{nl}(r) Y_{lm}(\theta, \varphi),$$

where $R_{nl}(r)$ is the radial part; $Y_{lm}(\theta, \varphi)$ are spherical harmonics.

The radial wave functions $R_{nl}(r)$ can be expressed in terms of *Laguerre polynomials* L_{n+l}^{2l+1}:

$$R_{nl}(r) = -\frac{2}{a^{\frac{3}{2}} n^2} \sqrt{\frac{(n-l-1)!}{((n+l)!)^3}}\, e^{-\frac{r}{na}} \left(\frac{2r}{na}\right)^l L_{n+l}^{2l+1}\left(\frac{2r}{na}\right),$$

which, in turn, are expressed in terms of a hypergeometric function:

$$L_n^k(x) = (-1)^k \frac{(n!)^2}{k!\,(n-k)!} F(k-n, k+1, x).$$

Below are the radial wave functions for several lower states:

$$R_{10}(r) = \left(\frac{1}{a}\right)^{\frac{3}{2}} e^{-\frac{\rho_n}{2}} 2;$$

$$R_{20}(r) = \left(\frac{1}{a}\right)^{\frac{3}{2}} e^{-\frac{\rho_n}{2}} \frac{1}{2\sqrt{2}} (2-\rho_n);$$

$$R_{21}(r) = \left(\frac{1}{a}\right)^{\frac{3}{2}} e^{-\frac{\rho_n}{2}} \frac{1}{2\sqrt{6}} \rho_n;$$

$$R_{30}(r) = \left(\frac{1}{a}\right)^{\frac{3}{2}} e^{-\frac{\rho_n}{2}} \frac{1}{9\sqrt{3}} (6-6\rho_n+\rho_n^2);$$

$$R_{31}(r) = \left(\frac{1}{a}\right)^{\frac{3}{2}} e^{-\frac{\rho_n}{2}} \frac{1}{9\sqrt{6}} (4-\rho_n)\rho_n;$$

$$R_{32}(r) = \left(\frac{1}{a}\right)^{\frac{3}{2}} e^{-\frac{\rho_n}{2}} \frac{1}{9\sqrt{30}} \rho_n^2;$$

$$R_{40}(r) = \left(\frac{1}{a}\right)^{\frac{3}{2}} e^{-\frac{\rho_n}{2}} \frac{1}{96} (24-36\rho_n+12\rho_n^2-\rho_n^3);$$

$$R_{41}(r) = \left(\frac{1}{a}\right)^{\frac{3}{2}} e^{-\frac{\rho_n}{2}} \frac{1}{32\sqrt{15}} (20-10\rho_n+\rho_n^2);$$

$$R_{42}(r) = \left(\frac{1}{a}\right)^{\frac{3}{2}} e^{-\frac{\rho_n}{2}} \frac{1}{96\sqrt{5}} (6 - \rho_n)\rho_n^2;$$

$$R_{43}(r) = \left(\frac{1}{a}\right)^{\frac{3}{2}} e^{-\frac{\rho_n}{2}} \frac{1}{96\sqrt{35}} \rho_n^3;$$

$$\rho_n = \frac{2r}{an}.$$

Spherical harmonics are expressed in terms of *Legendre polynomials*:

$$Y_{lm}(\theta, \varphi) = (-1)^{\frac{m+|m|}{2}} i^l \sqrt{\frac{(2l+1)}{4\pi} \frac{(l-|m|)!}{(l+|m|)!}} P_l^{|m|}(\cos\theta) e^{im\varphi};$$

$$P_l^{|m|}(\cos\theta) = \sin^{|m|}\theta \frac{d^{|m|} P_l(\cos\theta)}{d(\cos\theta)^{|m|}};$$

$$P_l(x) = \frac{1}{2^l l!} \frac{d^l}{dx^l} (x^2 - 1)^l.$$

Examples 15.1 and 15.2 show the codes that implement the construction of radial functions and spherical harmonics (Fig. 15.2 – 15.4).

Example 15.1. Consider radial functions [2].

```
% Constants
mec2 = 511000.0;  % m_e*c^2 = 511 KeV
hbarc = 2000.0;   % h_Planck*c = 2000 eV * Angstrom

labe = hbarc/mec2;
alf = 1.0 ./137; % fine structure constant
ao = labe ./alf; % the Bohr radius
%
Eo = -(mec2 .*alf .*alf) ./2.0;   % ground state energy
E1 = Eo ./4; % first excited state
E2 = Eo ./9; % second excited state

for i = 1:2
    EEo(i) = Eo;
```

```
    EE1(i) = E1;
    EE2(i) = E2;
end

ro = (3.0.*ao)/2.0; % average distance of the electron from the
nucleus
r1 = ro.*4;
r2 = ro.*9;
rro = linspace(0,ro,2);
rr1 = linspace(0,r1,2);
rr2 = linspace(0,r2,2);
%
figure % рис. 15.2
plot(rro,EEo,'b',rr1,EE1,'r',rr2,EE2,'k','LineWidth',3)
xlabel('Average    distance,    Angstrom','FontName',    'Trebuchet
MS','FontSize', 8)
ylabel('Energy, eV','FontName', 'Trebuchet MS','FontSize', 8)
legend('n=1','n=2','n=3')
set(gca, 'LineWidth',1) % thickness of axis lines
set(gca, 'FontName', 'Trebuchet MS')
set(gca, 'FontSize', 8)
set(gca, 'FontWeight', 'bold')

r = linspace(0,10);   % distance in Angstroms
q = r./ao;
psi0 = (exp(-q)).^2;

psi1 = ((exp(-q./2.0).*(1-q./2.0))).^2;

psi2 = (exp(-q./3.0).*(1-(2.0.*q)./3.0 + (2.0.*q.*q)./27.0)).^2;

figure % рис. 15.3
semilogy(q, psi0, 'b', q, psi1, 'r', q, psi2, 'k', 'LineWidth', 3)
xlabel('r/a')
ylabel('|\Psi(r)|^2')
legend('n=1','n=2','n=3')

axis([0 20 0.00001 2]);
set(gca, 'LineWidth',1)
set(gca, 'FontName', 'Trebuchet MS')
set(gca, 'FontSize', 8)
set(gca, 'FontWeight', 'bold')

figure % рис. 15.4
plot(q, psi0.*5.*q.*q, 'b', q, psi1.*2.*q.*q, 'r', q, psi2.*q.*q,
'k', 'LineWidth', 3)
xlabel('r/a')
ylabel('|\Psi(r)|^2*r^2')
legend('n=1','n=2','n=3')
```

```
set(gca, 'LineWidth',1)
set(gca, 'FontName', 'Trebuchet MS')
set(gca, 'FontSize', 8)
set(gca, 'FontWeight', 'bold')
```

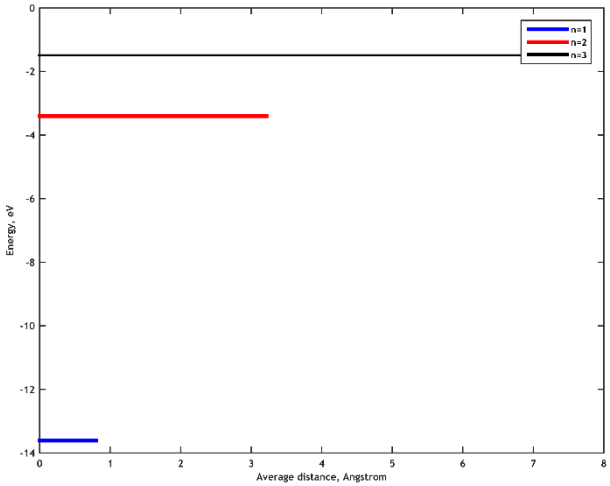


Fig. 15.2. Energy levels in hydrogen atom

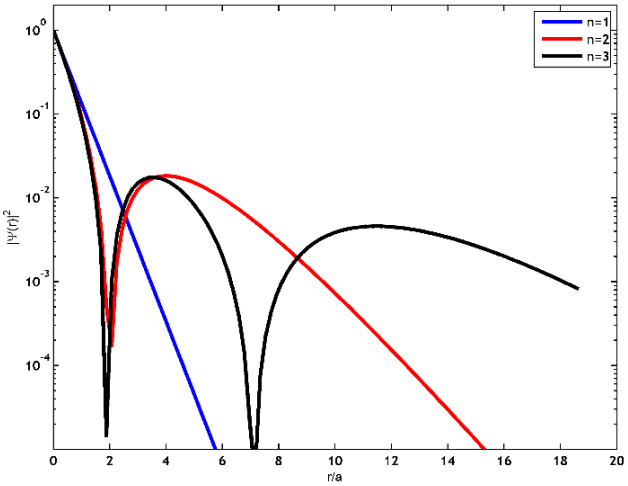


Fig. 15.3. Squares of the wave functions of the hydrogen atom

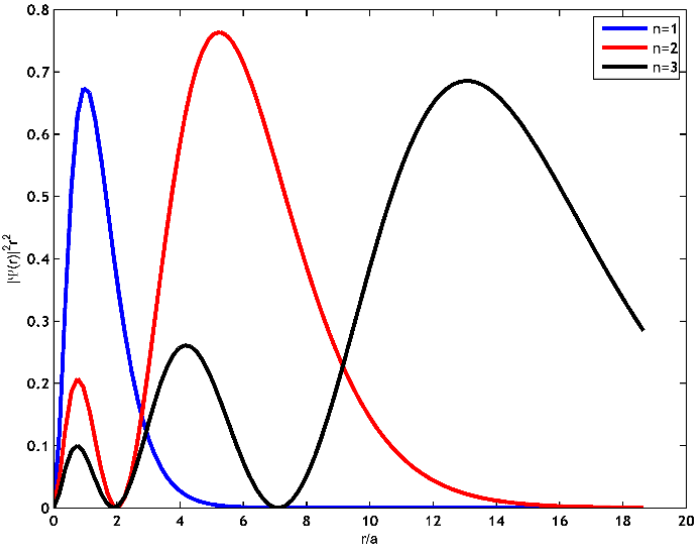


Fig. 15.4. Probability density for the three lower states

Example 15.2. Consider spherical harmonics [3] (Fig. 15.5 – 15.9).

```
function spharmPlot(L, resolution)

% grid of coordinates
delta = pi/resolution;
theta = 0:delta:pi;
phi = 0:2*delta:2*pi;
[phi,theta] = meshgrid(phi,theta);

% white background
figure('Color',[1 1 1])

for M = -L : L % loop over all angular momentum projections
    % Legendre polynomials
    P_LM = legendre(L,cos(theta(:,1)));
    P_LM = P_LM(abs(M)+1,:)';
    P_LM = repmat(P_LM, [1, size(theta, 1)]);

    % normalizing coefficients
    N_LM = sqrt((2*L+1)/4/pi*factorial(L-...
           abs(M))/factorial(L+abs(M)));
```

```
    % spherical harmonics
    if M >= 0
        Y_LM = sqrt(2) * N_LM * P_LM .* cos(M*phi);
    else
        Y_LM = sqrt(2) * N_LM * P_LM .* sin(abs(M)*phi);
    end

    % transition to spherical coordinates
    r = Y_LM;
    r = Y_LM;
    x = abs(r).*sin(theta).*cos(phi);
    y = abs(r).*sin(theta).*sin(phi);
    z = abs(r).*cos(theta);

    % plotting surfaces
    subplot(ceil(sqrt(2*L+1)),ceil(sqrt(2*L+1)),M+1+L)
    h = surf(x,y,z,double(r>=0));

    view(40,30)
    camlight left
    camlight right
    lighting phong

    axis([-1 1 -1 1 -1 1])
    set(gca, 'LineWidth',1)
    set(gca, 'FontName', 'Trebuchet MS')
    set(gca, 'FontSize', 8)
    set(gca, 'FontWeight', 'bold')

    % filling positive areas with red, negative with green
    colormap(redgreencmap([2]))

    set(h, 'LineStyle','none')

    grid off
    title(['L = ', num2str(L), ', M = ', num2str(M)],
'FontName',...
    'Trebuchet MS', 'FontSize', 8, 'FontWeight', 'bold')
end
```

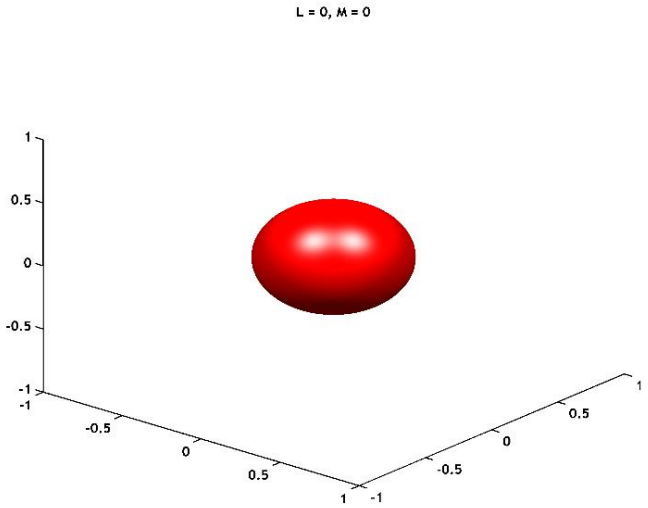


Fig. 15.5. Spherical harmonic for $n = 0$

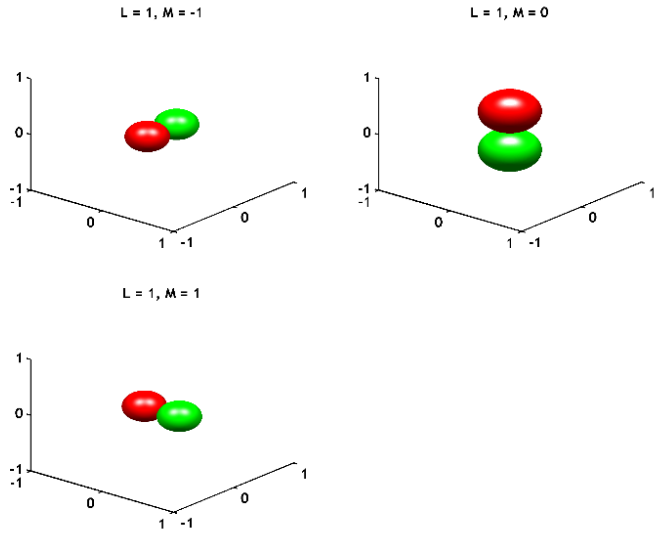


Fig. 15.6. Spherical harmonics for $n = 1$

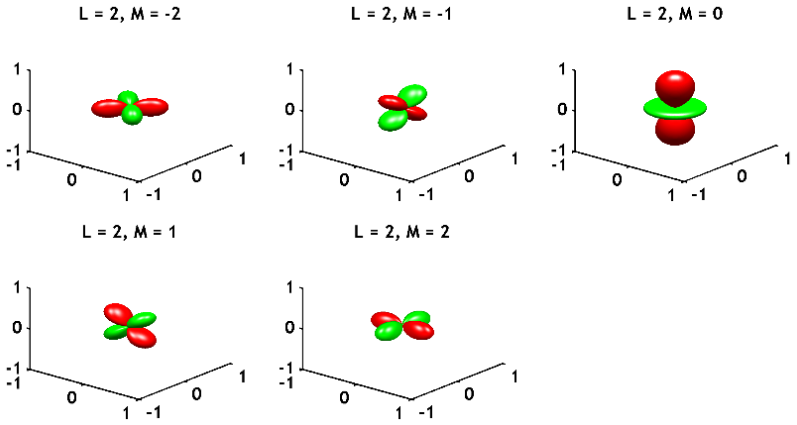


Fig. 15.7. Spherical harmonics for $n = 2$

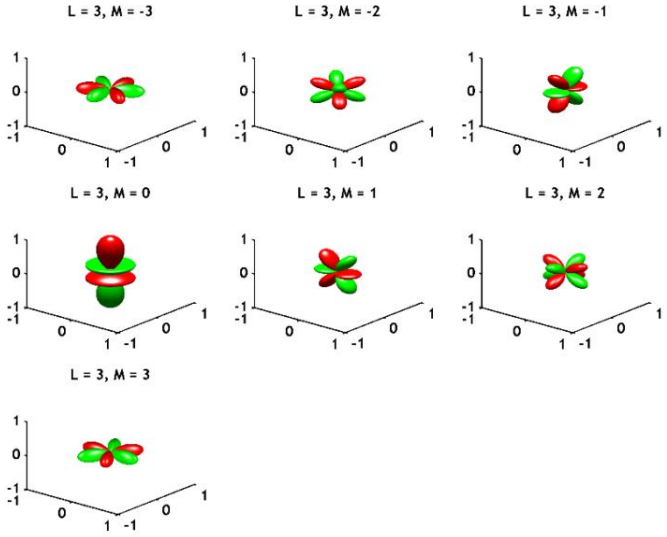


Fig. 15.8. Spherical harmonics for $n = 3$

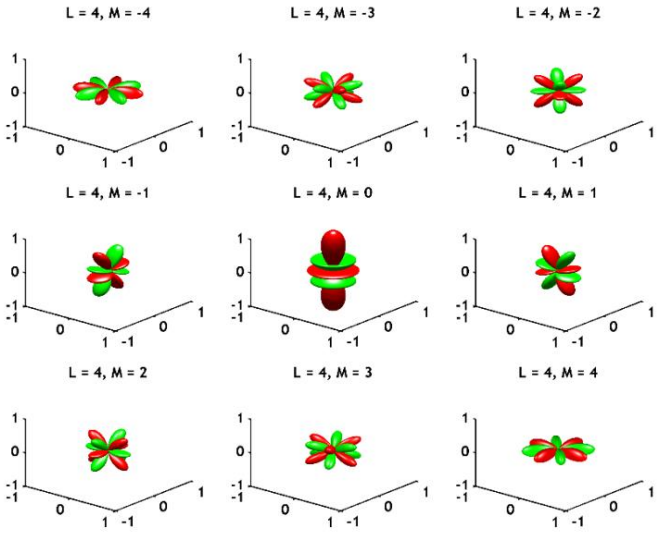


Fig. 15.9. Spherical harmonics for $n = 4$

16

Random distributions

16.1. The inverse function method

In various problems there is a necessity to obtain random values distributed by some law, by means of the generator the random numbers uniformly distributed on an interval $(0, 1)$. There are two simple and at the same time general methods for obtaining the necessary distributions: the inverse function method and von Neumann method.

Let the random variable ξ is defined on the interval (a, b), and has a value less than or equal to x with probability P (x). Then the function

$$F(x) = P\{\xi \le x\}$$

is called *the cumulative distribution function* of the random variable ξ and always monotonically increases from 0 to 1:

$$F(x_1) \le F(x_2), \;\; if \;\; x_1 \le x_2;$$

$$F(a) = 0; \;\; F(b) = 1.$$

The density function $f(x)$ of the random variable ξ is defined as follows:

$$F(x) = \int_a^x f(t)dt,$$

where the upper limit varies within the domain of definition (a, b).

Suppose that there exists an inverse function $F^{-1}(y)$ such that if $0 < y < 1$, then $y = F(x)$ if and only if $x = F^{-1}(y)$. Therefore, to find the desired distribution, we must put

$$\xi = F^{-1}(R),$$

where R is a random variable uniformly distributed in $(0, 1)$. Indeed,

$$P\{\xi \le x\} = P\{F^{-1}(R) \le x\} = P\{R \le F(x)\} = F(x).$$

In practice, the inverse function method is used as follows. Let it be necessary to obtain the values of the random variable ξ distributed with the density function $p(x)$ on the interval (a, b). Now we prove that the values of ξ can be found from the equation

$$\int_a^{\xi} p(x)dx = R,$$

where R is a random variable uniformly distributed in $(0, 1)$.

Consider the function

$$y(x) = \int_a^x p(t)dt,$$

while from the properties of the density function we have

$$y(a) = 0; \quad y(b) = 1; \quad y'(x) = p(x) > 0.$$

Therefore, the function $y(x)$ increases monotonically from 0 to 1, and any line $y = R$, where $0 < R < 1$, intersects the plot of the function $y = y(x)$ at one single point with its x-coordinate, as will be clear further, is the desired number ξ (Fig. 16.1). Thus, the uniqueness of the solution is proved.

Next, we choose an arbitrary interval (a', b') inside the interval (a, b). The points of this interval $a' < x < b'$ correspond to the y-coordinates of the curve $y(a') < y < y(b')$. If $\xi \in (a', b')$, then $R \in (y(a'), y(b'))$, and vice versa (Fig. 16.2), which means that the probabilities of these events are equal:

$$P\{a' < \xi < b'\} = P\{y(a') < R < y(b')\}.$$

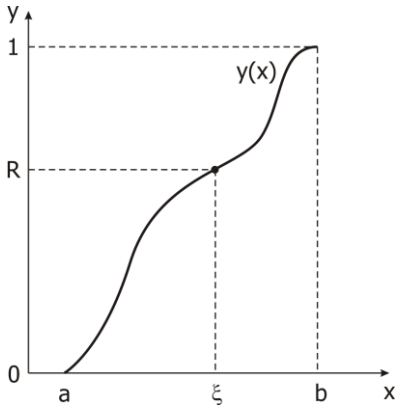
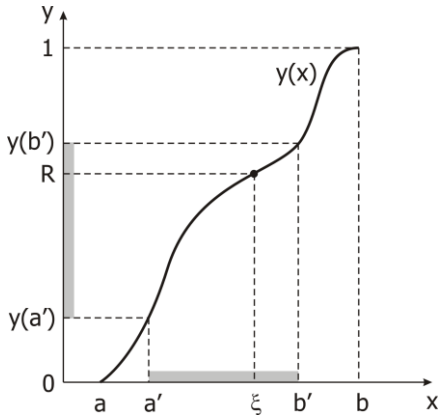


Fig. 16.1. Line $y = R$ intersects the plot of the function $y = y(x)$ in one single point



Fog. 16.2. If $\xi \in (a', b')$, then $R \in (y(a'), y(b'))$, and vice versa, therefore, the probabilities of these events are equal

Since R is uniformly distributed in $(0, 1)$, then

$$P\{y(a') < R < y(b')\} = y(b') - y(a') = \int_{a'}^{b'} p(x)dx.$$

Therefore,

$$P\{a' < \xi < b'\} = \int_{a'}^{b'} p(x)dx,$$

which means that ξ has the density function $p(x)$.

Thus, to find the required distribution, the inverse function $F^{-1}(x)$ must be obtained.

Example 16.1. Let it be necessary to obtain values ξ uniformly distributed in the interval (a, b). The normalized density function has the form

$$p(x) = \frac{1}{b-a}.$$

Then, according to the inverse function method,

$$\int_{a}^{\xi} \frac{dx}{b-a} = R.$$

Hence we find:

$$\xi = a + R(b-a).$$

In this case it is necessary only to shift the origin and change the scale of the initial random variable R.

Consider now *the exponential distribution*. Random variables with exponential distribution are used, for example, in problems of radioactive decay.

Let it be necessary to obtain a random variable distributed in $(0, \infty)$ with the cumulative distribution function

$$F(x) = 1 - e^{-x}$$

and, accordingly, with the density function

$$f(x) = e^{-x}.$$

Applying the inverse function method, we find

$$y = F(x) = 1 - e^{-x}; \quad x = F^{-1}(y) = -\ln(1-y).$$

Thus, the random variable

$$\xi = -\ln(1-R)$$

will have the exponential distribution. Example 16.2 shows the code, which realizes the generation of random numbers with the exponential distribution.

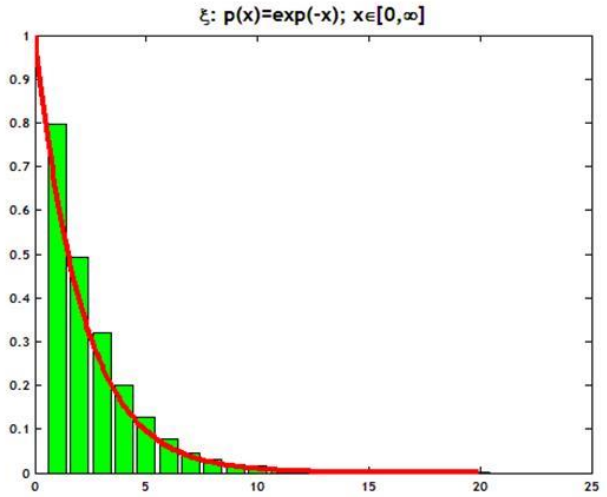


Fig. 16.3. Histogram of the exponential distribution

Example 16.2.

```
function exp_raspr(R)

y = zeros(R,1);
x = zeros(R,1);

for i = 1 : R
    y(i) = rand; % random numbers uniformly distributed in (0,1)
    x(i)   =   -log(1-y(i));   %   random   numbers   exponentially
                               distributed in (0, ∞)
end

D = max(x);
delta = D/20;
```

```
NN = R * delta;

figure % Fig. 16.3

H = hist(x,20)/NN;
bar(H,'g') % plotting a distribution histogram

hold on
x = [0 : 0.05 : 20*delta];
f = exp(-x);
plot(x/delta,f,'r','LineWidth',3)
    set(gca, 'LineWidth',1)
    set(gca, 'FontName', 'Trebuchet MS')
    set(gca, 'FontSize', 8)
    set(gca, 'FontWeight', 'bold')
    title('\xi:   p(x)=exp(-x);   x\in[0,\infty]','FontSize',   12,
'FontName', 'Trebuchet MS')
```

16.2. Neumann method

The integral, which must be computed in the inverse function method, can not always be calculated analytically. J. von Neumann suggested the following way to avoid this problem.

Suppose that the random variable ξ is defined on the interval (a, b), and its density function is confined:

$$p(x) \leq M_0.$$

Then, two random numbers R_1, R_2 are generated, which are uniformly distributed in $(0, 1)$. These numbers correspond to a point on a plane with coordinates (x_1, x_2), where $x_1 = a + R_1(b - a)$, $x_2 = R_2 M_0$. If this point lies below the curve $y = p(x)$, i.e. $x_2 < p(x_1)$ (point A in Fig. 16.4), then the required number $\xi = x_1$ is found; if the point lies above the curve (point B in Fig. 16.4), then the pair R_1, R_2 is discarded, and a new pair is selected.

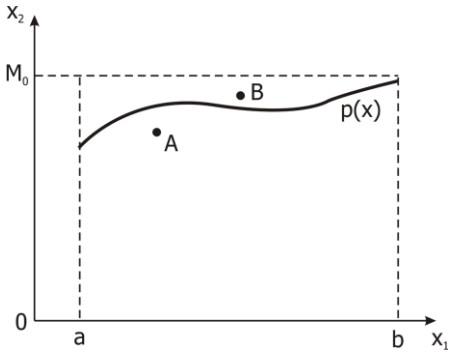


Fig. 16.4. Neumann's method of generating random numbers with density function $p(x)$

Example 16.3 shows the code, which implements the generation of random numbers with the density function $p(x) \propto e^{-x^2}$ in the interval $(0, \infty)$ by Neumann method (Fig. 16.5).

Example 16.3.

```
function k = Neumann(N)

M = 1;
a = 0;
b = 20; % right limit of distribution

x1 = zeros(N,1);
x2 = zeros(N,1);
k=0;
x = zeros(N,1);

for i = 1 : N
    R1 = rand;
    R2 = rand;

    x1(i) = a + R1 * (b - a);
    x2(i) = R2 * M;
    if x2(i) < exp(-x1(i)^2)
        k = k + 1; % counter of numbers having a given
                   distribution law
        x(k) = x1(i); % array of numbers having a given
                   distribution law
    end

end
```

```
x(k+1:end) = []; % unfilled elements of array x are deleted
D = max(x);
delta = D/20;

figure % рис. 16.5
H = hist(x,20);
bar(H) % plotting a distribution histogram

hold on
xr = [0 : 0.05 : 20*delta];
f = H(1) * exp(-xr.^2); % normalization of the function to the
                        first column of the histogram
plot(xr/delta,f,'r','LineWidth',3) % construction of the envelope
function
set(gca, 'LineWidth',1)
set(gca, 'FontName', 'Trebuchet MS')
set(gca, 'FontSize', 8)
set(gca, 'FontWeight', 'bold')
title('\xi:   p(x)=exp(-x^2);   x\in[0,\infty]','FontSize',   12,
'FontName', 'Trebuchet MS')
```

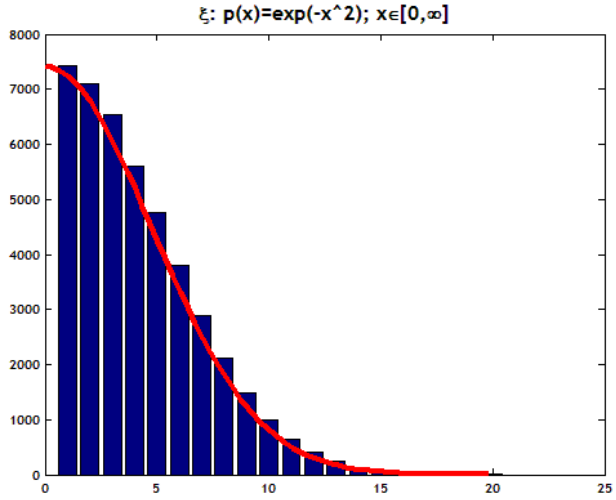


Fig. 16.5. Histogram of random numbers distributed with density function $p(x) \propto e^{-x^2}$ in the interval $(0, \infty)$

Practice

16.1. Using the inverse function method, generate random numbers distributed on a segment $[0, 3]$ with density: a) $p(x) = \alpha x$; b) $p(x) = \beta x^2$. Plot histograms of distributions.

16.2. Using Neumann's method, generate random numbers distributed with density function $p(x) \propto \sqrt{x}$ on the interval $[0, 3]$. Plot a histogram of the distribution.

17

Calculation of integrals by Monte Carlo method

Consider the function $g(x)$ defined on the interval (a, b), and the integral

$$I = \int_a^b g(x)dx.$$

To calculate the integral, we choose an arbitrary density function $p(x)$ defined on the same interval, which has the normalization

$$\int_a^b p(x)dx = 1.$$

Then, define a random variable

$$\eta = \frac{g(\xi)}{p(\xi)},$$

where ξ is a random variable distributed with density function $p(x)$ on (a, b). Then the mathematical expectation of η will be equal to the desired integral:

$$M\eta = \int_a^b \frac{g(x)}{p(x)} p(x)dx = I.$$

Consider N independent random variables $\eta_1, \eta_2, \dots, \eta_N$ and apply the central limit theorem to their sum, then

$$P\left\{\left|\frac{1}{N}\sum_{i=1}^{N}\eta_i - I\right| < 3\sqrt{\frac{D\eta}{N}}\right\} \approx 0.997.$$

Therefore, if N random values $\xi_1, \xi_2, \ldots, \xi_N$ are chosen, then for sufficiently large N

$$\frac{1}{N}\sum_{i=1}^{N}\frac{g(\xi_i)}{p(\xi_i)} \approx I,$$

and the calculation error does not exceed $3\sqrt{D\eta/N}$, where

$$D\eta = M\eta^2 - I^2 = \int_a^b \frac{g^2(x)}{p(x)}dx - I^2 \cong \frac{1}{N}\sum_{i=1}^{N}\frac{g^2(\xi_i)}{p(\xi_i)} - I^2.$$

It should be noted that the convergence of the result to the exact value also follows from the law of large numbers, according to which for any $\varepsilon > 0$

$$\lim_{N\to\infty} P\left\{\left|\frac{1}{N}\sum_{i=1}^{N}\eta_i - I\right| < \varepsilon\right\} = 1.$$

Moreover, the strong law of large numbers is valid:

$$P\left\{\lim_{N\to\infty}\frac{1}{N}\sum_{i=1}^{N}\eta_i = I\right\} = 1.$$

For an optimal calculation of the integral with a minimum error it is necessary to choose the density function $p(x)$ which is proportional to $|g(x)|$ or, if possible, is close to it. Let us prove this conclusion.

Let use the well-known Cauchy-Bunyakovsky inequality in an integral form:

$$\left(\int_a^b |u(x)v(x)|dx\right)^2 \le \int_a^b u^2(x)dx \int_a^b v^2(x)dx.$$

Let put

$$u(x) = \frac{g(x)}{\sqrt{p(x)}}; \quad v(x) = \sqrt{p(x)},$$

then

$$\left(\int_a^b |g(x)|dx\right)^2 \le \int_a^b \left(\frac{g^2(x)}{p(x)}\right)dx \int_a^b p(x)dx \equiv \int_a^b \left(\frac{g^2(x)}{p(x)}\right)dx.$$

Hence, we obtain a lower bound for the variance:

$$D\eta \ge \left(\int_a^b |g(x)|dx\right)^2 - I^2.$$

Let choose the distribution

$$p_0(x) = C|g(x)|; \quad C^{-1} = \int_a^b |g(x)|dx.$$

From the distribution it follows that

$$\int_a^b \frac{g^2(x)}{p_0(x)}dx = \frac{1}{C}\int_a^b |g(x)|dx = \left(\int_a^b |g(x)|dx\right)^2.$$

Substituting this expression in the expression for the variance, we have

$$D\eta = \left(\int_a^b |g(x)|dx\right)^2 - I^2.$$

Thus, the choice of the function $p_0(x) = C|g(x)|$ as the density function leads to the smallest error, i.e. the lower bound for the variance. Such calculation of the integral with the density function which is the closest to $p_0(x)$ is called *an essential sample*.

To illustrate the effectiveness of this choice, consider the integral

$$I = \int_0^{\pi/2} \sin x \, dx = 1.$$

We use various normalized density functions defined on the interval $(0, \pi/2)$ to calculate the integral (Fig. 17.1):

$$p_1(x) = \frac{2}{\pi}; \quad p_2(x) = \frac{8x}{\pi^2}.$$

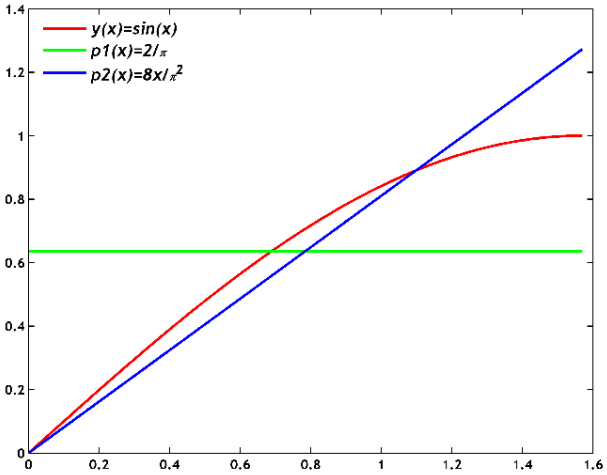


Fig. 17.1. The integrand $f(x)$ and the different density functions for calculating the integral by Monte Carlo method

Random numbers distributed on the interval $(0, \pi/2)$ with density functions $p_1(x)$ and $p_2(x)$ can be obtained by the inverse function method:

$$\xi_1 = \frac{\pi}{2} R;$$

$$\xi_2 = \frac{\pi}{2} \sqrt{R},$$

where R is a random number uniformly distributed in the interval $(0,1)$.

Accordingly, for the estimation of the integral I after N iterations, we obtain:

$$I_1(N) = \frac{\pi}{2N} \sum_{i=1}^{N} \sin \xi_{1i};$$

$$I_2(N) = \frac{\pi^2}{8N} \sum_{i=1}^{N} \frac{\sin \xi_{2i}}{\xi_{2i}}.$$

Fig. 17.2 shows the process of convergence of the calculated value of integral I to the exact value depending on the number N of generated random points.

It can be seen that the values of $I(N)$ converge faster to the exact value when choosing the density function $p_2(x)$. This is explained by the fact that $p_2(x)$ is close to the integrand function $y(x)$ (see Fig. 17.1), so the variance of the values of $I(N)$ is smaller in this case.

The efficiency of Monte Carlo method grows with the dimension of the calculated integral. The calculation of two-dimensional and three-dimensional integrals by Monte Carlo method is more efficient than the calculation by means of difference schemes. Monte Carlo method is successfully used for various physical and mathematical problems and processes: for modeling queuing systems, information flows, percolation processes, processes of neutron propagation in media, etc.

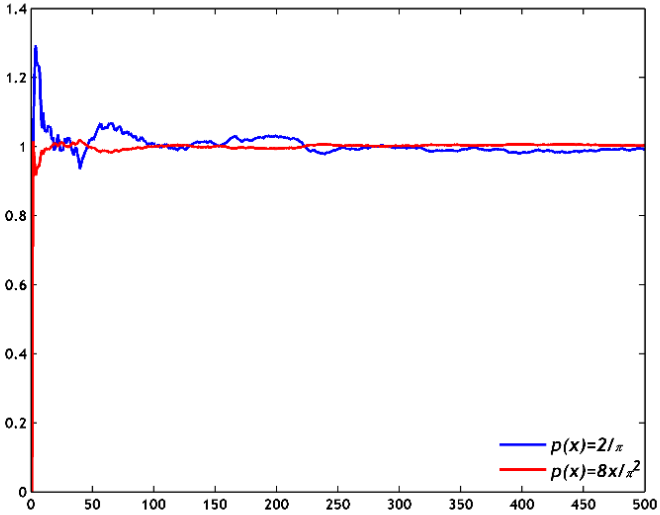


Fig. 17.2. The choice of the density function strongly affects the rate of convergence of the calculation of the integral by Monte Carlo method

Example 17.1 shows the code realizing the approximation of the integral $I(N)$ with the use of density functions distributions $p_1(x)$ and $p_2(x)$.

Example 17.1.

```
function IntMonte(N)

R = rand(1,N);
I1 = zeros(1,N);
I2 = zeros(1,N);

for i = 1 : N

I1(i) = pi/2/i*sum(sin(pi/2*R(1:i)));
I2(i) = pi/4/i*sum(sin(pi*sqrt(R(1:i))/2)./sqrt(R(1:i)));

end

figure
plot(I1,'b','LineWidth',2)
```

```
hold on
plot(I2,'r','LineWidth',2)
set(gca, 'LineWidth',1)
set(gca, 'FontName', 'Trebuchet MS')
set(gca, 'FontSize', 8)
set(gca, 'FontWeight', 'bold')
leg    =    legend('p(x)=2/\pi',    'p(x)=8x/\pi^2',    'Location',
'SouthEast');
set(leg, 'Box', 'off', 'FontName', 'Trebuchet MS', 'FontAngle',
'italic', 'FontSize', 10, 'FontWeight', 'bold')
```

Practice

17.1. Optimize the code from Example 17.1, using recurrence relations for $I(N)$.

17.2. Approximate the integral I by means of random numbers distributed with density function $p_3(x) = \beta\sqrt{x}$. Compare the rate of convergence with the first two cases.

17.3. Approximate the integral

$$\int_0^1 e^{-t^2}\, dt,$$

using the distributions

$$p_1(x) = \text{const};$$

$$p_2(x) = ae^{-x}.$$

Compare the rate of convergence of the integral to the exact value.

18

Metropolis algorithm

18.1. Markov chain. The concept of ergodicity

Let's assume that the Brownian motion is modeled, and at each computational (and temporal) step, one of the particles moves to some distance, which leads to a new arrangement of the particles. After moving, the particle "does not remember" its initial position, i.e. information about the previous state is "erased". A random walk is an example of a Markov chain. At each step a new state of the system appears, and the process is a chain of consequent states. The transition from the previous state to the new one depends only on the previous state, or, more accurately, the probability of finding the system in a given state depends only on the previous state. Denote as

$$x_0, x_1, \dots, x_n, \dots$$

a sequence of states where x_i means all degrees of freedom of the system under consideration (for example, a set of coordinates and momenta) describing its state (the system can be multiparticule). For example, x_i can be a set of coordinates and momenta of the particles or x_i can denote any of the basic functions of the system.

Denote as

$$P_{x_0, x_1, \dots, x_{n-1} \to x_n}$$

the probability of the appearance of a new state x_n under condition of realization of the previous states $x_0, x_1, \dots, x_{n-1}$ (conditional transition probability). Then *Markov chain* can be defined as the sequence x_0, $x_0, x_1, \dots, x_n, \dots$ of system states if for any n the following condition is fulfilled:

$$P_{x_0,x_1,\dots,x_{n-1}\to x_n} = P_{x_{n-1}\to x_n}.$$

The absolute probability of realization of the sequence $x_0, x_1, \dots, x_n$

$$P(x_0, x_1, \dots, x_n) = P(x_0)P_{x_0\to x_1} \dots P_{x_{n-2}\to x_{n-1}}P_{x_{n-1}\to x_n},$$

where $P(x_0)$ is the absolute probability of realization of the state x_0.

Any realization of the sequence of states $x_0, x_1, \dots, x_n$ can be obtained from the initial state x_0; the probability of realization of such a sequence is

$$P = P_{x_0\to x_1} \dots P_{x_{n-2}\to x_{n-1}}P_{x_{n-1}\to x_n}.$$

There exists an invariant distribution of states of the system $P(x_i)$, which does not depend on the initial conditions and which Markov chain allows to reach.

For example, for a canonical ensemble such an invariant distribution is Gibbs distribution:

$$P(x_i) \sim e^{-\frac{H(x_i)}{T}},$$

where H is the Hamiltonian of the system.

In order for the Markov chain be able to reach an invariant distribution, a number of conditions must be imposed on the transition probabilities.

First, we define the concept of the *invariant* or *stationary* probability distribution $P(x_i)$ by the following conditions:

$$P(x_i) > 0; \quad \sum_i P(x_i) = 1; \quad P(x_j) = \sum_i P(x_i)P_{x_i\to x_j}.$$

The last condition means that the absolute probability of each state consists of all possible transitions of the system to this state. The transition matrix $P_{x_i\to x_j}$ is called *stochastic matrix*.

A Markov chain is said to be *irreducible* if each of its states can be obtained from each other state (possibly through a series of other states and transitions). Thus, there can not be "traps" in an irreducible Markov chain, i.e. states or groups of states that the system can not exit from.

A state of Markov chain is called *periodic* if, upon reaching this state, the system returns to it in a certain number of steps (a period). If there are no such states, the Markov chain is called *aperiodic*, and then the states are also called *aperiodic states*.

Let $P_{x_i \to x_j}^{(n)}$ denote the probability that in the process starting from the state x_i, the first transition to x_j takes place at the n-th step. Then $P_{x_i \to x_j} = \sum_{n=1}^{\infty} P_{x_i \to x_j}^{(n)}$ is the probability that, starting from the state x_i, the system passes through the state x_j. If, in addition, $P_{x_i \to x_j} = 1$, then the state x_i is called *stable state*, and the quantity $\mu_i = \sum_{n=1}^{\infty} n P_{x_i \to x_j}^{(n)}$ is called *average return time*.

The Markov chain, consisting of aperiodic and stable states with a finite return time, is called *ergodic*, or *connected*, and the states that make it up are also called *ergodic states*.

An irreducible aperiodic Markov chain has an invariant distribution if and only if it is ergodic. In other words, to achieve an invariant distribution, the Markov process must be constructed so that after some finite number of steps any state x_j could be reached from any state x_i. The number of such steps should not be comparable with the length of the entire Markov chain. This is the practical guide for implementing the ergodic algorithm.

18.2. Principle of detailed balance

The main task of statistical mechanics is to calculate the observed thermodynamic quantities from statistical averaging:

$$\langle A\rangle = \frac{\int d\Omega\, A(\Omega)\rho(\Omega)}{Z};\quad Z = \int d\Omega\, \rho(\Omega).$$

Here, the integration is over the entire phase space Ω, ρ is the distribution function (in the particular case, the Gibbs distribution).

The mean value in general case is a multidimensional integral over the phase space. The general rules for calculating this integral by the essential sampling within the framework of the Monte Carlo algorithm are also valid here. Suppose that a chain of random states Ω_i is created with a certain predetermined distribution $P(\Omega)$ to estimate the integral for $\langle A\rangle$. Then the following estimate is true:

$$\langle A\rangle \approx \frac{\sum_{i=1}^{N} A(\Omega_i)\rho(\Omega_i)P^{-1}(\Omega_i)}{\sum_{i=1}^{N} \rho(\Omega_i)P^{-1}(\Omega_i)}.$$

If we choose the density function ρ for the probability $P(\Omega)$,

$$P(\Omega) = \frac{\rho(\Omega)}{Z},$$

then the calculation of $\langle A\rangle$ is reduced to a simple arithmetic mean:

$$\langle A\rangle \approx \frac{1}{N}\sum_{i=1}^{N} A(\Omega_i).$$

Since the distribution $P(\Omega)$ is invariant for the system under consideration, a Markov chain based on such a distribution must be ergodic. Let us formulate the principle according to which it is possible to construct an algorithm for realizing the distribution $P(\Omega)$.

First of all, the conditions for the ergodicity must be satisfied, and for the practical implementation of the algorithm, it is necessary to impose additional limiting conditions on the transition probabilities:

$$P_{\Omega'\to\Omega} \geq 0;\quad \sum_{\Omega'} P_{\Omega\to\Omega'} = 1;\quad P(\Omega) = \sum_{\Omega'} P_{\Omega'\to\Omega}P(\Omega').$$

The first condition is that the transition probabilities must be chosen non-negative. The second condition means that the total probability that the system goes from any state Ω to some other state is equal to one, i.e. from the state Ω there is necessarily an exit, it is not a "trap". The third condition is equivalent to the last of the relations of the ergodicity conditions and means that the required distribution $P(\Omega)$ is invariant, that is, the sum of the transition probabilities from all states to a given state Ω realizes the probability of this event $P(\Omega)$.

Each step $\Omega_i \to \Omega_j$ of the Markov process can be associated with the time interval dt of the calculation time of this step; this time reflects the scale of the real relaxation time of the physical system. The limit of the ratio of the probability of transition to this time interval

$$W_{\Omega' \to \Omega} = \lim_{dt \to 0} \frac{P_{\Omega' \to \Omega}}{dt}$$

is called the *transition intensity*, or the *transition probability density*. The limit is understood in the sense that the total calculation time is much greater than dt, so that it is possible to approximate the discrete steps by a continuous process.

Taking into account the evolution of the system in time, we can write:

$$P(\Omega, t + dt) = \sum_{\Omega'} P_{\Omega' \to \Omega, dt} P(\Omega', t),$$

where $P_{\Omega' \to \Omega, dt}$ is the conditional probability of transition of the system from the state Ω' to the state Ω during the time dt. Next, select the term with $\Omega' = \Omega$:

$$P(\Omega, t + dt) = P_{\Omega \to \Omega, dt} P(\Omega, t) + \sum_{\Omega' \neq \Omega} P_{\Omega' \to \Omega, dt} P(\Omega', t).$$

The condition of the conservation of probabilities can be represented as follows:

$$P_{\Omega\to\Omega,dt} = 1 - \sum_{\Omega'\neq\Omega} P_{\Omega'\to\Omega,dt}.$$

Hence,

$$P(\Omega, t+dt) = P(\Omega,t)\left(1 - \sum_{\Omega'\neq\Omega} P_{\Omega'\to\Omega,dt}\right) + \sum_{\Omega'\neq\Omega} P_{\Omega'\to\Omega,dt} P(\Omega', t)$$

$$\Rightarrow$$

$$\Rightarrow\ P(\Omega, t+dt) - P(\Omega,t) =$$

$$= -\sum_{\Omega'\neq\Omega} P_{\Omega\to\Omega',dt} P(\Omega,t) + \sum_{\Omega'\neq\Omega} P_{\Omega'\to\Omega,dt} P(\Omega', t).$$

The evolution of the probability $P(\Omega)$ can be described as a kind of *balance equation*, or a *velocity equation*:

$$\frac{dP(\Omega)}{dt} = -\sum_{\Omega'\neq\Omega} W_{\Omega\to\Omega'} P(\Omega) + \sum_{\Omega'\neq\Omega} W_{\Omega'\to\Omega} P(\Omega').$$

The first term on the right describes the rate of all transitions from the state Ω to all other states, and the second is the rate of transitions from all states other than Ω to the state Ω. The terms with $\Omega' = \Omega$ in both sums cancel each other. This expression is also called *Kolmogorov equation* (or *Kolmogorov-Chapman equation*).

In the equilibrium state, the derivative $\frac{dP}{dt}$ is equal to zero, and

$$\sum_{\Omega'} W_{\Omega\to\Omega'} P(\Omega) = \sum_{\Omega'} W_{\Omega'\to\Omega} P(\Omega').$$

To facilitate further practical application of the detailed balance equation, stronger constraints can be imposed to the last equation. Let require that this equation be valid for *each* state Ω' under the summation sign:

$$W_{\Omega\to\Omega'} P(\Omega) = W_{\Omega'\to\Omega} P(\Omega').$$

This relationship is called the condition of the *detailed balance*. This ratio gives a considerable freedom in choosing the intensity of the transitions.

One of the most common options for choosing the intensity of transitions that satisfies the detailed balance is the *Metropolis algorithm*, which uses the following expression for $W_{\Omega' \to \Omega}$:

$$W_{\Omega' \to \Omega} = \begin{cases} \dfrac{P(\Omega')}{P(\Omega)}, & \text{если } \dfrac{P(\Omega')}{P(\Omega)} < 1; \\ 1, & \text{если } \dfrac{P(\Omega')}{P(\Omega)} \geq 1. \end{cases}$$

We can formulate a more general version of the Metropolis algorithm by choosing $W_{\Omega' \to \Omega}$ as follows:

$$W_{\Omega' \to \Omega} = \min\left(\frac{1}{\tau}; \frac{1}{\tau}\frac{P(\Omega')}{P(\Omega)}\right),$$

where $\tau \sim 1$ is an arbitrary constant. By varying the parameter τ, we can change the rate of convergence of the Monte Carlo algorithm.

18.3. 1D-version of the Metropolis algorithm

Let it be necessary to generate random numbers $x_0, x_1, \ldots, x_n, \ldots$ distributed with a density function $p(x)$. Then the Metropolis algorithm for $\tau = 1$ will be as follows.

0. Suppose that the system is at the point x_n. To form a new point x_{n+1}, the following actions are performed.

1. Select a random point $x_t = x_n + \delta_n$, where δ_n is a random number uniformly distributed on the interval $[-\delta, \delta]$.

2. Compute $w = p(x_t)/p(x_n)$.

3. If $w \geq 1$, the new point x_t is accepted: $x_{n+1} = x_t$.

4. If $w < 1$, then generate a random number r uniformly distributed on the interval $[0, 1]$.

5. If $r \leq w$, the new point x_t is accepted: $x_{n+1} = x_t$.

6. If $r > w$, then x_n is accepted as a new point: $x_{n+1} = x_n$.

The main problem in the effective implementation of the above algorithm is the correct choice of the number δ. If this number is chosen too large, then the probability of step 6 in the algorithm is increased, and the system will stay at the same point for a long time. If the number δ is chosen too small, then the probability that the new point x_{n+1} will be different from the previous point x_n increases, but because of the smallness of δ, the system will still move near the previous state, and the generated random numbers will again give a bad sampling. In practice, the optimal choice of δ is usually the number at which approximately half of the system's transitions to a new point are accepted.

Another problem of the Metropolis algorithm is the fact that the random variables x_i generated by the algorithm are not independent, since there is an obvious correlation between the new and the current random points. Thus, the Metropolis algorithm generates random points $x_0, x_1, \dots, x_n, \dots$, which are distributed with the required density function $p(x)$ but are not statistically independent. Therefore, for example, the estimate of the variance DI for calculating the integral of the function $f(x)$ by Monte Carlo method using $p(x)$,

$$I = \frac{\int w(x) f(x) dx}{\int w(x) dx},$$

will be incorrect, since to calculate the variance a sample of independent random variables is necessary. To isolate from the sequence $x_0, x_1, \dots, x_n, \dots$ a subsequence containing independent random variables, an *autocorrelation function* is used:

$$C(k) = \frac{\langle f_i f_{i+k}\rangle - \langle f_i\rangle^2}{\langle f_i^2\rangle - \langle f_i\rangle^2},$$

$$f_i \equiv f(x_i),$$

$$\langle f_i f_{i+k}\rangle = \frac{1}{N-k}\sum_{i=1}^{N-k} f(x_i) f(x_{i+k}).$$

It is assumed that the random points x_i and x_{i+k} separated by an interval k so that $C(k) < 0.1$, can be considered statistically independent, i.e. a subsequence $\{x_0, x_k, x_{2k}, x_{3k}, \ldots\}$ can be used to calculate the variance and other statistical processing of the results.

Example 18.1 demonstrates an algorithm (Fig. 18.1) that generates random numbers whaving Gaussian distribution [4]

$$p(x) = \frac{1}{\sqrt{10\pi}} e^{-x^2/10}.$$

Example 18.1.

```
% specification of the density function and call a function that
implements the Metropolis algorithm

p=inline('1/((2*pi*5)^0.5)*exp(-x^2/10)','x');
N=2*10^5; % number of random points
x(1)=0; % the first term of a random sequence

for i=2:N
    x(i) = Metropolis(x(i-1),0.5,p); % delta = 0.5
end


% visualization
figure % рис. 18.1

i= 1 : N;
plot(i,x,'k');
set(gca, 'LineWidth',1)
set(gca, 'FontName', 'Trebuchet MS')
set(gca, 'FontSize', 8)
set(gca, 'FontWeight', 'bold')
```

```
% histogram of distribution
Ni = 50; % number of histogram bins
k = 1 : Ni;
dx = (max(x) - min(x))/(Ni-1);
Interval(k) = min(x)+dx*(k-1);

s = hist(x,Interval);

for j = 1 : Ni
    z(j) = min(x) + dx*(j-1);
    Z(j) = N*p(z(j))*dx;
end

figure % Fig. 18.2

bar(Interval,s);
colormap white
hold on

plot(z,Z,'r','LineWidth',2.5)
hold off

set(gca, 'LineWidth',1)
set(gca, 'FontName', 'Trebuchet MS')
set(gca, 'FontSize', 8)
set(gca, 'FontWeight', 'bold')


% Metropolis algorithm
function z = Metropolis(x,delta,p)
% x is coordinate of initial point

xt = x+delta*(2*rand - 1);
w= p(xt)/p(x);

if w >= 1
    xnew = xt;
end

if w < 1
    r = rand;
    if r<=w
    xnew = xt;
        else
        xnew = x;
    end
end

z = xnew;
```

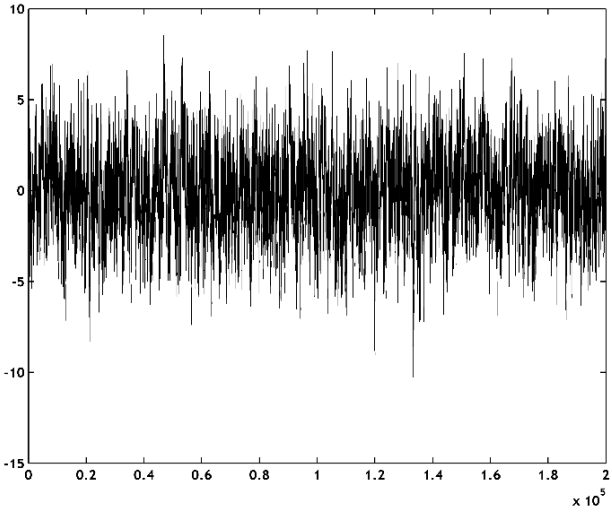


Fig. 18.1. Random numbers generated by the Metropolis algorithm and having Gaussian distribution

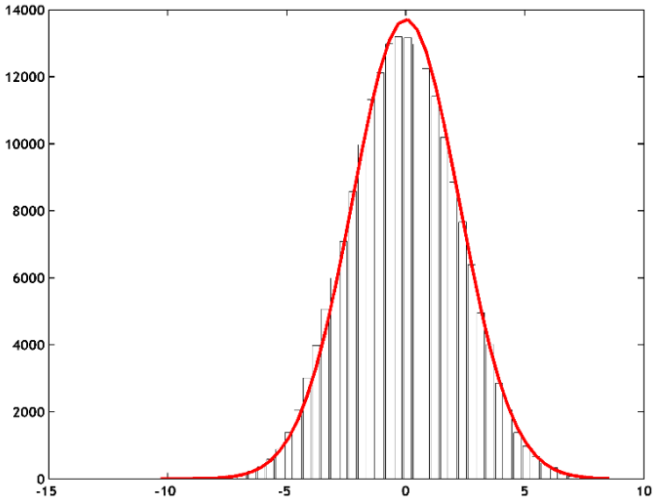


Fig. 18.2. The histogram of random numbers of Fig. 18.1 and the envelope function

$$p(x) = \frac{1}{\sqrt{10\pi}} e^{-x^2/10}$$

Practice

18.1. Compute the autocorrelation function for the random numbers of Example 18.1. Determine the value of k for which random numbers can be considered independent.

18.2. Investigate the dependence of the number of successful configuration changes (a successful configuration change means a situation when $x_{n+1} \neq x_n$) on δ.

18.3. Using the Metropolis algorithm generate a sequence of random variables having a normal distribution, extract from this sequence a subsequence of independent random variables, and use this subsequence to calculate the integrals:

$$\int_{-\infty}^{\infty} x^2 e^{-\frac{x^2}{2}} dx;$$

$$\int_{0}^{4} x^2 e^{-x} dx.$$

19

Fourier transform

Consider a periodic function $f(x)$ with period 1 defined on the whole real axis, and expand it into Fourier series:

$$f(x) = \sum_{q=-\infty}^{\infty} a_q e^{2\pi i q x},$$

where q is integer, while for convergence we require

$$\sum_{q=-\infty}^{\infty} |a_q| < \infty.$$

We choose on the x-axis a discrete spatial grid $x_i = i/N$, where N is fixed, i is an integer. The number N is called the *sampling frequency*. Let $f_i \equiv f(x_i)$ be the values of the function at the grid nodes. Then there will be many summands with the same exponential factors in the Fourier expansion of $f(x)$. Indeed, suppose that there are two such momenta q_1 and q_2 that $q_2 - q_1 = kN$, k is an integer. Then $q_2 x_i - q_1 x_i = \frac{i}{N} kN = ki$ is also an integer. Thus, $e^{2\pi i q_2 x_i} = e^{2\pi i q_1 x_i}$. The Fourier series can be rewritten as follows:

$$f(x) = \sum_{q=0}^{N-1} A_q e^{2\pi i q x}, \quad A_q = \sum_{s=-\infty}^{\infty} a_{q+sN},$$

where s is an integer. This transformation is valid only for the variable x, which belongs to the discrete grid x_i.

Thus, now the problem is formulated on the domain with the length of only one period, which is divided into segments ("nodes") of length $2\pi/N$. To solve the problem, it is necessary to determine the Fourier coefficients A_q.

Let introduce the scalar product of two functions on a discrete grid:

$$(f,g) = \frac{1}{N}\sum_{i=0}^{N-1} f_i g_i^*.$$

The factor 1 / N is needed for the correct transition to a continuous limit:

$$(f,g) \xrightarrow[|x_{i+1}-x_i|\to 0]{} \int_0^1 f(x)g^*(x)dx.$$

The functions $g_q(x) = e^{2\pi i q x_i}$ for $0 \le q < N$ form an orthonormal basis system. Indeed, let calculate the scalar product

$$\left(g_q, g_j\right) = \frac{1}{N}\sum_{k=0}^{N-1} e^{2\pi i (q-j)\frac{k}{N}}.$$

If $q \neq j$, then $\left(g_q, g_j\right) = \frac{1}{N}\left[\frac{1-e^{2\pi i(q-j)}}{1-e^{2\pi i(q-j)/N}}\right] = 0$; if $q = j$, then the scalar product can be calculated in two ways:

а) $\left(g_q, g_j\right) = \frac{1}{N}\left[\frac{1-e^{2\pi i\alpha}}{1-e^{2\pi i\alpha/N}}\right]_{|\alpha\to 0} = \frac{1}{N}\left[\frac{-2\pi i\alpha}{-2\pi i\alpha/N}\right]_{|\alpha\to 0} \to 1;$

б) $\left(g_q, g_j\right) = \frac{1}{N}\sum_{j=0}^{N-1} 1 = 1.$

Thus, it is proved that $\left(g_q, g_j\right) = \delta_{qj}, 0 \le q, j < N$, i.e. the functions $g_q(x) = e^{2\pi i q x_i}$ are orthonormal.

Next we obtain the desired Fourier coefficients A_q. To do this, we find the scalar product of the functions f and g_j and obtain the coefficients with the use of the inverse Fourier transform:

$$\left(f, g_j\right) = \frac{1}{N}\sum_{k=0}^{N-1}\sum_{q=0}^{N-1} A_q g_j^* e^{2\pi i q x_k} = \sum_{k=0}^{N-1} A_q \frac{1}{N}\sum_{q=0}^{N-1} e^{2\pi i q x_k - 2\pi i j x_k} =$$

$$= \sum_{q=0}^{N-1} A_q \left(g_q, g_j\right) = \sum_{q=0}^{N-1} A_q \delta_{qj} = A_j.$$

Thus,

$$A_j = \left(f, g_j\right) = \frac{1}{N} \sum_{k=0}^{N-1} f_k \, e^{-2\pi i j x_k}.$$

If the coefficients a_q decrease rapidly with increasing q, then

$$A_q = \sum_{s=-\infty}^{\infty} a_{q+sN} \to a_q.$$

For the numerical calculation of all Fourier coefficients in the general case, it is necessary to perform $\sim N^2$ operations, and for a sufficiently large N the procedure for calculating the coefficients can take a considerable time. There is an algorithm that allows the Fourier series to be expanded much faster, in $\sim N \log_2 N$ operations: the so-called *fast Fourier transform*.

Suppose that there are Fourier coefficients of some periodic function f:

$$A_q = \left(f, g_q\right) = \frac{1}{N} \sum_{j=0}^{N-1} f_j e^{-2\pi i q x_j}.$$

We divide the number N into two multipliers: $N = p_1 p_2$, $p_1 p_2 \neq 1$ (for example, $= 100,\ p_1 = 20,\ p_2 = 5$). Then the numbers q and j can be represented in the form $q = q_1 + p_1 q_2$, $j = j_2 + p_2 j_1$, and $0 \leq q_1, j_1 < p_1;\ \ 0 \leq q_2, j_2 < p_2$. Formally substitute these expansions in the expression for the Fourier coefficients:

$$A_q = A(q_1, q_2) = \frac{1}{p_1 p_2} \sum_{j_1=0}^{p_1-1} \sum_{j_2=0}^{p_2-1} f_{j_2+p_2 j_1} e^{-\frac{2\pi i}{p_1 p_2}(j_2+p_2 j_1)(q_1+p_1 q_2)}.$$

Convert the expression to the exponential:

$$\frac{1}{p_1 p_2}(j_2 + p_2 j_1)(q_1 + p_1 q_2) = \frac{j_2 q_1}{p_1 p_2} + \frac{j_2 p_1 q_2}{p_1 p_2} + \frac{p_2 j_1 q_1}{p_1 p_2} + \frac{p_2 j_1 p_1 q_2}{p_1 p_2} =$$

$$= \frac{q}{N} j_2 + \frac{j_1 q_1}{p_1} + q_2 j_1.$$

Note that the product $q_2 j_1$ is an integer, i.e. $e^{-2\pi i q_2 j_1} = 1$. Then we have:

$$A_q = \frac{1}{p_1 p_2} \sum_{j_1=0}^{p_1-1} \sum_{j_2=0}^{p_2-1} f_{j_2+p_2 j_1} e^{-2\pi i \left(\frac{q j_2}{N} + \frac{j_1 q_1}{p_1}\right)} =$$

$$= \frac{1}{p_2} \sum_{j_2=0}^{p_2-1} e^{-2\pi i \frac{q j_2}{N}} \frac{1}{p_1} \sum_{j_1=0}^{p_1-1} f_{j_2+p_2 j_1} e^{-2\pi i \frac{j_1 q_1}{p_1}}.$$

We introduce

$$A^{(1)}(j_2, q_1) = \frac{1}{p_1} \sum_{j_1=0}^{p_1-1} f_{j_2+p_2 j_1} e^{-2\pi i \frac{j_1 q_1}{p_1}}.$$

Then the Fourier component can be rewritten as follows:

$$A_q = \frac{1}{p_2} \sum_{j_2=0}^{p_2-1} A^{(1)}(j_2, q_1) e^{-2\pi i \frac{q j_2}{N}}.$$

The problem was divided into two parts: first to determine the coefficients $A^{(1)}$, and then to substitute them in the expression for the Fourier coefficients A_q.

Determine the number of operations necessary to calculate the Fourier components. It is easy to estimate that to calculate all the coefficients $A^{(1)}$, one should do p_1 operations for the internal sum, p_2 operations for the external variable j_2, and p_1 operations for the external

variable q_1. Thus, to calculate all the coefficients $A^{(1)}$, it is required $\sim p_1 p_2 p_1 = N p_1$ operations. Similarly, the number of operations necessary to calculate the coefficients A_q is $\sim p_2 p_1 p_2 = N p_2$ operations. In the general case, when the number N is factorized into arbitrary multipliers, there are $\sim N \sum_{i=1}^{r} p_i$ Fourier operations, where r is the number of multipliers. The most efficient factorizing is achieved with $p_1 = p_2 = \cdots = p_r = 2$, i.e. on the basis of 2, as in binary code. In this case we have $2^r = N$, i.e. $r = \log_2 N$. Hence, the total number of operations is $\sim N \log_2 N \ll N^2$.

Write down the algorithm specifically for the binary factrozation, which is often used in practice. First, represent the numbers q and j in the form

$$q = \sum_{k=1}^{r} q_k 2^{k-1}, \quad j = \sum_{m=1}^{r} j_{r+1-m} 2^{m-1}; \quad q_k, j_m = 0,1.$$

Then, for the calculation of the coefficients $A^{(i)}$ there exists a sequence of recurrence relations:

$$A^{(m)}(q_1, \dots, q_m, j_{m+1}, \dots, j_r) =$$

$$= \frac{1}{2} \sum_{j_m=0}^{1} e^{-2\pi i j_m 2^{-m} \sum_{k=0}^{m} q_k 2^{k-1}} A^{(m-1)}(q_1, \dots, q_{m-1}, j_m, \dots, j_r);$$

$$A^{(0)}(j_1, \dots, j_r) = f_{j_r + j_{r-1} 2 + \cdots + j_1 2^{r-1}}; \quad m = 1, \dots, r.$$

Note that we must take the number N фы multiple of powers of 2: $N = 2^n$.

In MatLab, a fast Fourier transform is realized using the function `fft`. Example 19.1 shows the use of this function to restore a noisy signal (Fig. 19.1).

Example 19.1.

```
Fs = 100000; % Sampling frequency
T = 1/Fs; % Time Sampling
L = 10000; % Signal length
t = (0:L-1)*T; % Vector of time

% The sum of two sinusoids of 500 and 1200 Hz
x = 0.7*sin(2*pi*500*t) + sin(2*pi*1200*t);

figure
% plotting of the amplitude of the pure signal versus time

subplot(2,2,1)

plot(Fs*t(1:1000),x(1:1000))
title('Source signal (2 sinusoids 500 and 1200 Hz)')
xlabel('Time, ms')

y = x + 2*randn(size(t));     % adding of noise to the signal

% plotting the amplitude of the noisy signal versus time
subplot(2,2,2)

plot(Fs*t(1:1000),y(1:1000))
title('Signal with noise')
xlabel('Time, ms')

Y = fft(y); % fast Fourier transform of a noisy signal
f = Fs*linspace(0,1,L); % frequency grid

% plotting of the spectrum of the noisy signal in the frequency
range [0 Hz, 2000 Hz]
subplot(2,2,3)

plot(f(f<2000),abs(Y(f<2000)))
title('Signal spectrum with noise')
xlabel('Frequency (Hz)')
ylabel('|Y(f)|')

% finding of two main peaks in the spectrum of the noisy signal in
the frequency range [0 Hz, 2000 Hz]
[psor,lsor]  =  findpeaks(abs(Y(f<2000)),  'SortStr',  'descend',
'NPeaks', 2);
fr=f(lsor); x=0; % finding the frequency of peaks
for i=1:1:numel(fr) % frequency loop for signal recovery
    x=x+psor(i)/max(psor)*sin(2*pi*fr(i)*t);
end
```

```
% plotting of the amplitude of the reconstructed signal versus
time

subplot(2,2,4)

plot(Fs*t(1:1000),x(1:1000),'r')
title('Recovered signal')
xlabel('Time, ms')
```

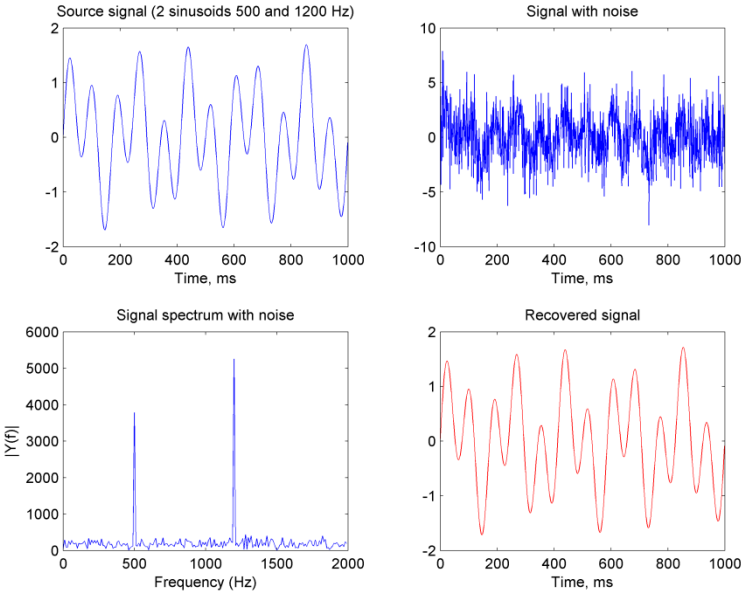


Fig. 19.1. Recovering a noisy signal with the help of Fourier transform

In Example 19.2, spectral analysis of an audio file is considered using the example of a blue whale call (Fig. 19.2).

Example 19.2.

```
% Spectral analysis of blue whale call
whaleFile = fullfile(matlabroot, 'examples', 'matlab',
'bluewhale.au'); % link to the audio file with the voice of the
blue whale; this file comes with MatLab
[x,fs] = audioread(whaleFile); % reading audio file
sound(x,fs); % playback of the audio file
```

```
figure % Fig. 19.2

subplot(3,1,1)

t=10*(0:1/fs:(length(x)-1)/fs); % time discretization
plot(t,x) % plotting the amplitude versus time
xlabel('Time, s')
ylabel('Amplitude')
title('{\bf Blue Whale Call}')


subplot(3,1,2)

bCall = x(2.45e4:3.10e4); % bCall is the "second word" of the blue
whale
tb = 10*(0:1/fs:(length(bCall)-1)/fs); % transition of the
discretization of the "second word" into the time grid
plot(tb,bCall,'m') % plotting the amplitude versus time for the
"second word"
xlim([0 tb(end)]) % setting the limits along the X axis
xlabel('Time, s')
ylabel('Amplitude')
title('{\bf Second word }')

m = length(bCall);      % length of the "second word"
n = pow2(nextpow2(m)); % finding a number of the form 2^k, which
is either equal to or greater than the length of the "second word"
y = fft(bCall,n);       % fast Fourier transform
f = (0:n-1)*(fs/n)/10; % frequency range
p = y.*conj(y)/n;       % power of the set of y values

subplot(3,1,3)
plot(f(1:floor(n/2)),p(1:floor(n/2)),'r') % plotting the signal
power versus frequency
xlabel('Frequency, Hz')
ylabel('Power')
```

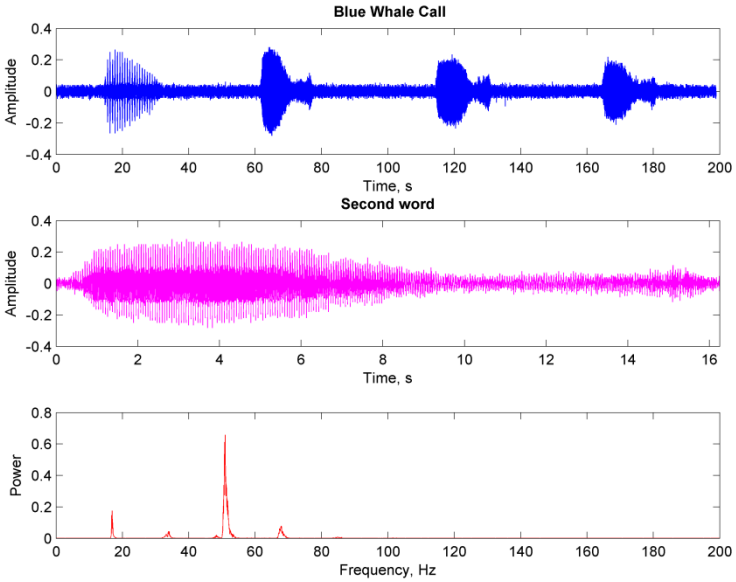


Fig. 19.2. Spectral analysis of the blue whale call

Example 19.3 shows the code analyzing the frequency spectrum of a one-dimensional atomic chain (Fig. 19.3, 19.4).

Example 19.3. In the example, it is possible to observe oscillations of atoms in real time; to do this, uncomment the commented lines.

```
Fs = 10; %Fs is the sample rate
dt = 1/Fs; % dt is time incrfement
a = 1; % a is the interatomic distance
N = 50; %N is the number of atoms
L = 10000; %L is the number of time steps
alpha = 1; %alpha is the coefficient of atomic interaction
lattice = [a:a:(a*N)]; % the initial position of the atoms
M = 1; %M is the atomic mass, by default all atoms are the same
     and have a mass of 1

coord = lattice + a*(2*rand(1,N)-1)/100; %coord is the vector of
       coordinates of atoms; we set the random initial deviation

% uncomment for visualization of oscillations
%coord=lattice;
```

```
%coord(1)=coord(1)+rand/10;
%

coordPast = coord; % coordPast is the vector of coordinates of
                   atoms on the previous iteration
coordFuture = coord; % coordFuture is the vector of coordinates of
                     atoms on the next iteration; it is intended
                     to implement the Euler difference scheme

position = zeros(L,N); % position is the matrix of positions of
                       atoms in time
h = waitbar(0,'Period calculating'); % waitbar initialization to
                           display the process of calculating
                           coordinates;     %     comment     when
                           visualizing the oscillations

%figure
for j = 1:1:L % loop over time
    for i = 1:1:N % loop over coordinates
        if i == 1 % accounting for the Born-Karman boundary
                    conditions
            coordFuture(1) = dt^2*(-alpha/M)*(2*coord(1)-
            (coord(N)-(lattice(end)))-coord(2))+2*coord(1)-
            coordPast(1);

        elseif i == N % implementation of the Euler difference
                      scheme
           coordFuture(N)   =   dt^2*(-alpha/M)*(2*coord(N)-coord(N-
          1)-(coord(1)+lattice(end)))+2*coord(N)-coordPast(N);

        else
           coordFuture(i)   =   dt^2*(-alpha/M)*(2*coord(i)-coord(i-
          1)-coord(i+1))+2*coord(i)-coordPast(i);
        end
    end

    coordPast = coord; % go to the next iteration
    coord = coordFuture;
    position(j,:) = coordFuture; % current position of atoms

% uncomment for visualization of oscillations
%plot(lattice,lattice-coordFuture,'or')
%axis([min(lattice)-1 max(lattice)+1 -1/10 1/10])
%drawnow
%pause(dt)

waitbar(j/L) % comment when visualizing the oscillations
end
```

```
close(h) % comment when visualizing the oscillations

f = [0:1:(L-1)]*Fs/L*2*pi; % f is the frequency grid for a
                           discrete Fourier transform
Y = zeros(N,L); % Y is the future Fourier image of the dependence
                of the positions of atoms on time
h = waitbar(0,'FFT'); % waitbar initialization to display the
discrete Fourier transform process; comment when visualizing the
oscillations

for i = 1:1:N
    position(:,i) = position(:,i) - sum(position(:,i))
     /numel(position(:,i)); % normalization
    Y(i,:) = fft(position(:,i)); % discrete Fourier transform
    Y(i,:) = abs(Y(i,:)); % amplitude value of the Fourier
                          transform
    waitbar(i/N) % comment when visualizing the oscillations
end

close(h) % comment when visualizing the oscillations

sumY = sum(Y(i,:),1); % sum of the Fourier transforms into a
                      single Fourier transform of the whole system
k = (2*pi/a)*(0:1:(N-1))/N; % k is the array of wave vectors of
                            the system
omega = 2*sqrt(alpha/M)*sin(k*a/2); % omega omega is the vector
                    of the acoustic oscillation frequencies of the
                    system
analit_omega = omega(1:end/2); % analit_omega: consider only one
                               branch of frequencies
[pks,locs] = findpeaks(sumY(1:end/20), f(1:end/20), 'Npeaks',
numel(analit_omega)-1); % seek the values of the frequencies of
the acoustic oscillations in the resulting total Fourier transform

locs = sort(locs); % sort the obtained values of frequencies

figure % Fig. 19.3

plot(locs,pks,'sr')
hold on
plot(f(1:end/20),sumY(1:end/20)) % build the frequency response of
the system; consider only 1/20, since the rest does not carry
useful information
xlabel('Frequency (rad/s)')
ylabel('Amplitude')
title('{\bf Frequency components of oscillations of a one-
dimensional atomic chain}')

figure % Fig. 19.4
```

```
plot([-k(end/2:-1:1)  k(1:end/2)],[locs(end:-1:1)  0  0  locs],'*b')
% plot the dispersion
hold on
plot([-k(end/2:-1:1)      k(1:end/2)],      [analit_omega(end:-1:1)
analit_omega],'r')

xlabel('Wave vector')
ylabel('Frequency (rad/s)')

title('{\bf The dispersion law of the acoustic branch of a one-
dimensional atomic chain}')
legend({'Obtained values', 'Analytical expression'}, 'Location',
'SouthWest')
```

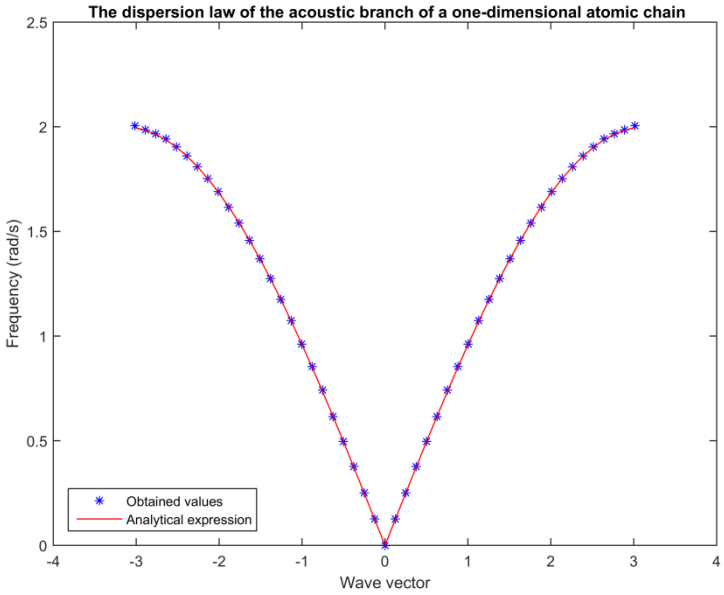


Fig. 19.3. The acoustic branch of oscillations of a chain of atoms

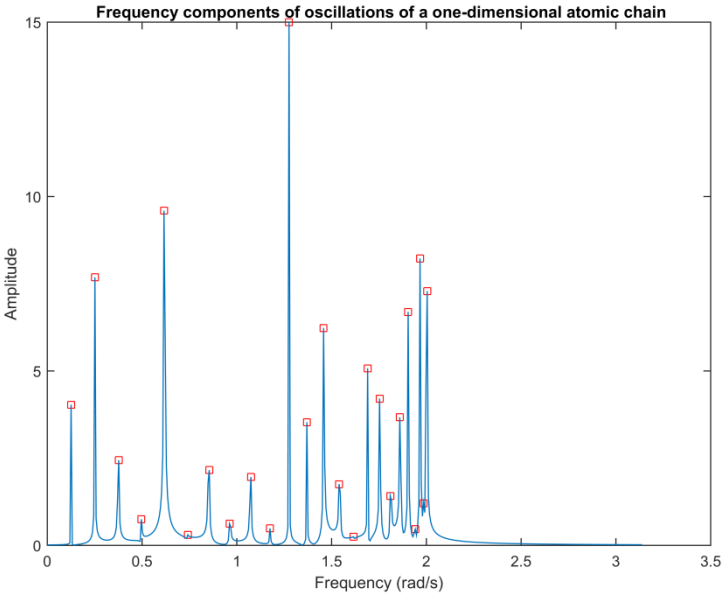


Fig. 19.4. Frequency spectrum of oscillations of a chain of atoms

20

Sparse matrices

20.1. Storage scheme

Sparse matrices are matrices containing zero elements, the number of which is much greater than the number of non-zero elements. Sparse matrices occur in various scientific fields; to work with them, special algorithms are created. Strictly speaking, a matrix can be considered sparse if a compact storage scheme for its elements and algorithms for sparse matrices provide a significant gain in processing speed compared to conventional or dense matrices.

Elements of sparse matrices are stored in special arrays of type `double array (sparse)`. Only non-zero elements are stored in these arrays, as well as information about the place in the matrix where these elements are located. The function `sparse` converts the matrix storage scheme from ordinary to sparse:

```
A = [ 4 0 0 -1 0
      0 0 2 0 1
      1 1 0 0 0
      3 0 0 0 -7
      0 0 5 0 5]

A =
     4     0     0    -1     0
     0     0     2     0     1
     1     1     0     0     0
     3     0     0     0    -7
     0     0     8     0     5

AS = sparse(A)

AS =
```

```
(1,1)         4
(3,1)         1
(4,1)         3
(3,2)         1
(2,3)         2
(5,3)         8
(1,4)        -1
(2,5)         1
(4,5)        -7
(5,5)         5
```

Fig. 20.1 shows the storage scheme of the sparse *AS* matrix in MatLab.

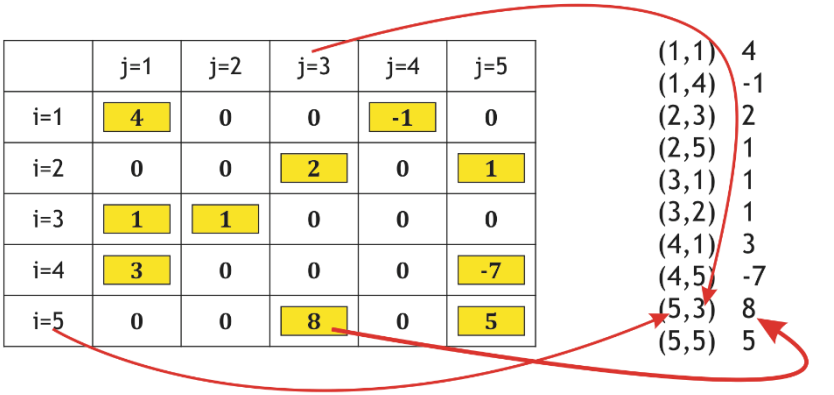


Fig. 20.1. The sparse matrix storage scheme in MatLab

20.2. Creating sparse matrices

The function `sparse`, in addition to converting the full matrix into a sparse one, provides the possibility of creating a sparse matrix using vectors containing values of non-zero elements and their positions in the matrix:

```
AS = sparse(i,j,nzer,m,n)
```

Here `i` is the vector of the row indices of non-zero elements; `j` is a vector of the column indices of non-zero elements; `nzer` is the vector of non-zero elements; `m,n` are the dimensions of the full matrix. The matrix *AS* in Fig. 20.1 can be specified using the following commands:

```
i = [1 3 4 3 2 5 1 2 4 5]; % elements of a sparse
                           matrix are listed by columns
j = [1 1 1 2 3 3 4 5 5 5];
nzer = [4 1 3 1 2 8 -1 1 -7 5];
AS = sparse(i,j,nzer,5,5);
```

In the following examples, various ways of specifying sparse matrices with the use of the function `sparse`, efficient and ineffective, are shown.

Example 20.1. Using the following commands, a tridiagonal sparse matrix of the following form can be created:

$$\begin{pmatrix} 2 & 1 & & & \\ 1 & 2 & 1 & & \\ & \ddots & \ddots & \ddots & \\ & & 1 & 2 & 1 \\ & & & 1 & 2 \end{pmatrix}.$$

```
i = [1:10 2:10 1:9]; % row indiced
j = [1:10 1:9 2:10]; % column indices
s = [2*ones(1,10) ones(1,9) ones(1,9)]; % matrix elements
B = sparse(i,j,s); % in the absence of the fourth and the fifth
                   arguments, the dimensions of the matrix are
                   computed automatically from the maximum values of
                   the elements of the vectors i and j
```

Example 20.2. Let's analyze effective and inefficient ways of creating sparse matrices.

```
% 1
% The most inefficient way
% n is the size of the square matrix
% nz is the number of non-zero elements

A = sparse(n,n); %  initialization of sparse matrix of size n x n

for k = 1:nz
    i = 1 + fix(n*rand(1)); % rows and columns are selected
randomly
```

```
    j = 1 + fix(n*rand(1));
    x = rand(1);
    A(i,j) = A(i,j) + x; % a random number is written in the
                chosen place of the matrix; such an operation
                works very slowly with sparse matrices
end


% 2
% It is more efficient to pre-initialize the arrays of row and
column indices, as well as element values
i = zeros(nz,1);
j = zeros(nz,1);
x = zeros(nz,1);

for k = 1:nz
    i(k) = 1 + fix(n*rand(1));
    j(k) = 1 + fix(n*rand(1));
    x(k) = rand(1);
end

A = sparse(i,j,x);


% 3
% The best way is to completely avoid loops

e = rand(3,nz);
e(1,:) = 1 + fix(n*e(1,:));
e(2,:) = 1 + fix(n*e(1,:));
A = sparse(e(1,:),e(2,:),e(3,:));
```

Example 20.3. Consider the function `spy`.

```
p32 = pascal(32); % Pascal matrix is a matrix whose elements are
                  binomial coefficients
s32 = rem(p32,2); % function rem calculates the remainder of
                  dividing the elements of the matrix by 2; we
                  obtain a matrix of zeros and ones
sps32 = sparse(s32); % creation of sparse matrix

figure % рис. 20.2
spy(sps32) % function spy draws a portrait of the matrix; nonzero
           elements are marked with dots; under the plot, the
           number of non-zero elements in the matrix are shown;
           the portrait of this matrix coincides with the
           Sierpinski carpet
```

```
set(gca, 'LineWidth',1)
set(gca, 'FontName', 'Trebuchet MS')
set(gca, 'FontSize', 8)
set(gca, 'FontWeight', 'bold')
```

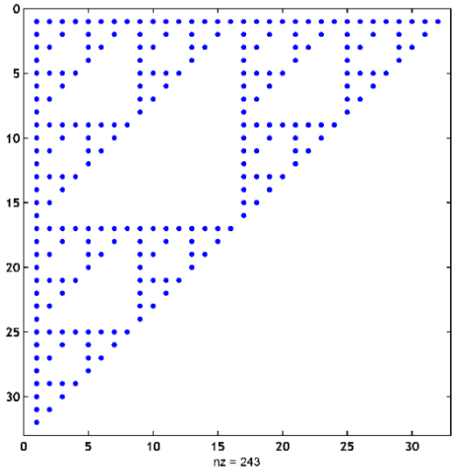


Fig. 20.2. Sparse matrix portrait

20.3. Operations with sparse matrices

All operations of linear algebra such as addition, subtraction, multiplication of matrices, etc., are realized in MatLab for sparse matrices. That is, for example, operation `A+B`, where `A` and `B` are sparse matrices, will result also in a sparse matrix. If one of the matrices is full, then the result can be a sparse or full matrix, depending on the type of operation being performed. Table 20.1 shows the types of the resulting matrix for various operations with sparse matrices; `R` means a sparse matrix, and `F` means a full matrix.

Table 20.1

The type of result when performing operations with sparse matrices

R+R ⟹ R	R+F ⟹ F
R-R ⟹ R	R-F ⟹ F
R*R ⟹ R	R*F ⟹ F
R.*R ⟹ R	R.*F ⟹ R
R./R ⟹ R	R./F ⟹ R

The function `spdiags` is an analog of the `diag` function for sparse matrices and allows forming sparse matrices with given diagonals or to select diagonals from sparse matrices. For example, for the matrix `A`

```
A = [0 1 0 2 0 0; 0 0 3 0 4 0; 5 0 0 6 0 7;...
    8 9 0 0 10 0; 0 11 12 0 0 13];

A =
     0     1     0     2     0     0
     0     0     3     0     4     0
     5     0     0     6     0     7
     8     9     0     0    10     0
     0    11    12     0     0    13
```

the statement

```
[B, d] = spdiags(A)
```

forms the matrix `B` consisting of columns containing diagonals of the matrix `A`; the vector `d` contains the positions of these diagonals in the matrix:

```
B =
     0     0     1     2
     0     0     3     4
     0     5     6     7
     8     9    10     0
    11    12    13     0

d =
    -3
    -2
     1
     3
```

The positions of the diagonals in the matrix are shown in Fig. 20.3. For diagonals lying under the main diagonal, the values of the elements are added to the end of the corresponding column of the matrix `B`, and for diagonals lying above the main diagonal to the beginning of the corresponding column.

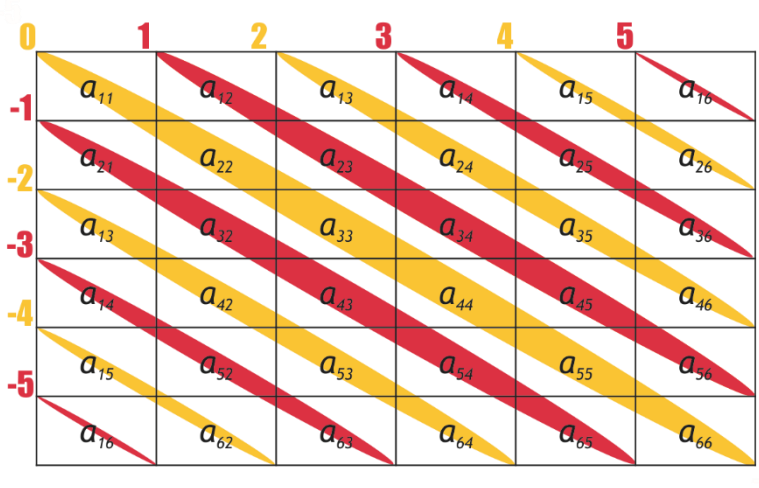


Fig. 20.3. The positions of the diagonals in the matrix

The function `speye(m,n)` creates an identity sparse matrix of size $m \times n$.

The function `full` performs the transformation of a sparse matrix into a full matrix.

The function `find` searches for non-zero elements in the matrix:

```
[i,j,x] = find(A); % such a call to the function find
          with three output parameters leads to the
          creation of two vectors of indices and a
          vector containing values of nonzero elements
i =
     3   4   1   4   5   2   5   1   3   2   4   3   5
j =
     1   1   2   2   2   3   3   4   4   5   5   6   6
x =
     5   8   1   9   11   3   12   2   6   4   10   7   13
```

The function `spconvert` creates a sparse matrix with the use of a three-columns matrix containing two columns of indices and a column of values. The following two commands are equivalent:

```
A = spconvert(x);
A = sparse(x(:,1),x(:,2),x(:,3));
```

If the matrix `x` has four columns, then the last two columns are used as an array of values, and the matrix `A` will be complex; the values of the real part of the elements are taken from the third column of `x`, and the imaginary part is taken from the fourth column.

The function `nnz` returns the number of non-zero elements of the matrix:

```
nnz(A)

ans =
    13
```

The function `spfun` is used to calculate functions of non-zero matrix elements. The function has the following format: `C = spfun(@f,A)`, where `f` is a function. The values of the function will be computed only for non-zero elements of the matrix `A`: `C(i,j)=f(A(i,j))`. For example,

```
CN = spfun(@cos,A)
CN =
   (3,1)        0.2837
   (4,1)       -0.1455
   (1,2)        0.5403
   (4,2)       -0.9111
   (5,2)        0.0044
   (2,3)       -0.9900
   (5,3)        0.8439
   (1,4)       -0.4161
   (3,4)        0.9602
   (2,5)       -0.6536
   (4,5)       -0.8391
```

```
(3,6)        0.7539
(5,6)        0.9074
```

The function `eigs` calculates the eigenvalues of the matrix and is optimized to work with sparse matrices. It has already been used to calculate the energy of a particle in a potential well (see Chapter 14).

Practice

20.1. Rewrite the algorithms from the Chapter 14 with the use of tridiagonal sparse matrices.

21

Fractals. Recursion

21.1. Recursive functions

The term "recursion" means "something defined in terms of itself". In programming function is called *recursive* if it calls itself.

A classic example of recursion is the recursive definition of the factorial of an integer:

$$n! = n \cdot (n-1)! \quad \text{common case;}$$

$$1! = 1 \quad \text{particular case.}$$

This definition of the factorial is recursive, because the factorial is defined in terms of another factorial. In the recursive definition of any object, there is always a general and particular case. In general case, the factorial of the number n is equal to the number n multiplied by the factorial of the number $(n-1)$. In the particular case, the factorial of the number 1 is 1. The achieving of the particular case stops the recursion. For example,

$$3! = 3 \cdot 2!$$

$$2! = 2 \cdot 1!$$

$$1! = 1$$

Calculation of 3! is impossible without calculation of 2!; calculation of 2! is impossible without calculation of 1!; calculation of 1! is the particular case, the result is 1. After reaching the particular case, the calculations are reversed: knowing 1!, it is possible to calculate 2!, and knowing 2!, it is possible to calculate 3!.

When programming a recursion, it is necessary to ensure that a particular case is realized sooner or later; otherwise the recursion will be infinite. Example 21.1 shows the code that calculates the factorial by the recursion method.

Example 21.1.

```
function facn = fact(n)

if n == 1
    facn = 1;
else
    facn = n * facn(n-1);
end
end
```

The program in Example 21.1 works as follows. Suppose that we again need to compute 3!. At the first call to the function, the program tries to assign the result to the variable `3 * facn(2)`. However, the value of `facn(2)` is not defined at this time, and the calculation is interrupted by a recursive call to the `facn` function to evaluate `facn(2)`. In this call, the attempt is made to calculate the value `2 * facn(1)`. But the value of `facn(1)` at this point is also not defined, and the calculation is interrupted by a recursive call to the function `facn` to evaluate `facn(1)`. In this call, the `if`-branch of the `if-else` construction is implemented, and the recursion reaches the particular case. The value of `facn(1)` is calculated, then it is passed to the process that expects it to evaluate `facn(2)`, which in turn is passed to a process that expects it to evaluate `facn(3)`.

In Example 21.2 the application of a recursive function to rearranging words of a sentence in reverse order is considered.

Example 21.2.

```
function reverse_words(sent)
% sent is initial sentence
[word, rest] = strtok(sent); % the function strtok splits the
                             input argument into two parts: the
                             first word is written to the variable
```

```
                            word, and the remaining text is
                            written to the variable rest
if ~isempty(rest) % if the variable rest is not an empty array, a
                            recursive call to the function occurs;
                            when there are no words in the
                            variable rest, a particular case
                            occurs, and the recursion stops
    reverse_words(rest);
end

disp(word) % displaying the word stored in the variable word; the
                            last word in the sentence will be
                            displayed first, since it is on this
                            word the recursion stops
end


% Example of function working

reverse_words('Masha loves porridge')

porridge
loves
Masha
```

Recursive functions are often used in the construction of fractal objects.

21.2. Sierpinsky carpet

The Sierpinsky carpet is a fractal object with a dimension of 1.585 [5]. The method of construction of the carpet is quite simple. As an initial set, we choose an equilateral triangle S_0. Then we remove the insideness of the central triangular part and denote the remaining set S_1. This process is repeated for each of the three remaining triangles, and the next approximation S_2 is obtained. Continuing in the same way, we obtain a sequence of sets S_n, which forms the Sierpinski carpet (Fig. 21.1).

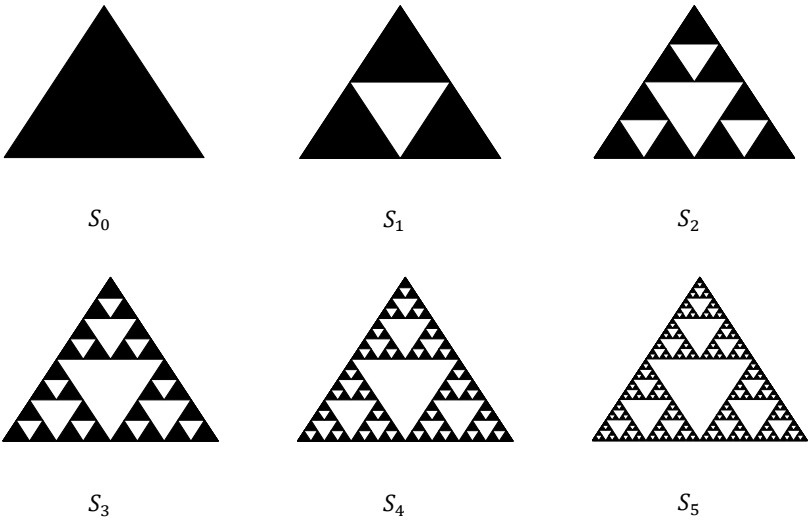


Fig. 21.1. Construction of the Sierpinsky carpet

To construct the Sierpinsky carpet, the following algorithm is implemented.

1. Set the order of carpet N.
2. Set the coordinates of the vertices of the original triangle ABC: $(X_A, Y_A), (X_B, Y_B), (X_C, Y_C)$.
3. Plot an equilateral triangle ABC and fill it with black.
4. Calculate the coordinates of the means of the sides of the triangle ABC:

$$dx = \frac{X_B - X_A}{2};\ dy = \frac{Y_B - Y_A}{2};$$

$$X_{A'} = X_A + dx;\ Y_{A'} = Y_A;$$

$$X_{B'} = X_A + dx + \frac{dx}{2};\ Y_{B'} = Y_A + dy;$$

$$X_{C'} = X_A + \frac{dx}{2};\ Y_{C'} = Y_A + dy.$$

5. Plot the triangle $A'B'C'$ and fill it with white.

6. Repeat the actions of items 4 and 5 N times for the triangles $AA'C'$, $A'BB'$, $C'B'C$ respectively.

The algorithm can be implemented using a recursive procedure that performs the sequence of actions described in items 4 and 5. The full code [4] is given in Example 21.2.

Example 21.2.

```
function z = Serpinsky(Lmax)

% Lmax is the order of the carpet


% specification of the coordinates of the vertices of an
equilateral triangle
x1=0;
y1=0;

x2=1;
y2=0;

x3=0.5;
y3=sin(pi/3);

h=figure(1); % initializing the graphics window
hold on

fill([x1 x2 x3],[y1 y2 y3],'k'); % draw an equilateral triangle
                        and fill it with black; the function fill
                        draws the polygon defined by the x and y
                        vectors

set(gca,'xtick',[],'ytick',[]); % switching off axis ticks
set(gca,'XColor','w','YColor','w'); % setting color for drawing
                                      axes

% call of the function that draws an equilateral triangle of white
color

Simplex(x1,y1,x2,y2,x3,y3,0,Lmax);
hold off %
```

```
function z=Simplex(x1,y1,x2,y2,x3,y3,n,Lmax)

if n < Lmax % the condition for achieving the particular case

% setting the coordinates of the vertices of the current
equilateral triangle
     dx=(x2-x1)/2;
     dy=(y3-y1)/2;
     x1n=x1+dx;
     y1n=y1;
     x2n=x1+dx+dx/2;
     y2n=y1+dy;
     x3n=x1+dx/2;
     y3n=y1+dy;

% draw the current equilateral triangle
     n=n+1;
     fill([x1n x2n x3n],[y1n y2n y3n],'w');

     % application of Simplex function to a triangle with vertices
     % (Xa',Ya); (Xa',Ya); (Xc',Yc')
     Simplex(x1,y1,x1n,y1n,x3n,y3n,n,Lmax);

     % application of Simplex function to a triangle with vertices
     % (Xa',Ya); (Xb',Yb'); (Xb',Yb')
     Simplex(x1n,y1n,x2,y2,x2n,y2n,n,Lmax);

     % application of Simplex function to a triangle with vertices
     % (Xc',Yc'); (Xb',Yb'); (Xc,Yc)
     Simplex(x3n,y3n,x2n,y2n,x3,y3,n,Lmax);
end
```

21.3. Menger sponge

The Menger sponge is a three-dimensional analogue of the Sierpinsky carpet. In principle, the construction of the sponge does not differ from the carpet construction scheme, the difference consists only in the number of iterations at each level. Example 21.3 shows the code that implements the construction of a sponge and its filling with various colors (Fig. 21.2).

Example 21.3.

```
function menger(n)

switch nargin  % if there are no input arguments
```

```
    case 0
        n = 2; % sponge order will be selected by default 2
end

figure % Fig. 21.2

% calculation of the matrix consisting of zeros and ones that
denote the presence or absence of cubes of the size corresponding
to the current iteration

% M is final matrix
M = 0; % unit cube

for k = 1 : n

% create cube A, 3 times bigger than the cube M in each dimension
    A = zeros([3^k, 3^k, 3^k]);

% a cube of size M is cut out on the lower face of the cube A
    A(:,:,1:3^(k-1)) = [M,M,M; M,ones(size(M)),M; M,M,M];

% a cross of five cubes M is cut out in the cube A
    A(:,:,3^(k-1)+1:2*3^(k-1)) = ...
                [M,ones(size(M)),M; ...
      ones(size(M)),ones(size(M)),ones(size(M));...
                M,ones(size(M)), M];

% a cube of size M is cut out on the upper face of the cube A
    A(:,:,2*3^(k-1)+1:3^k) = [M,M,M; M,ones(size(M)),M; M,M,M];

    M=A; % the resulting shape is assigned the name M for the next
          iteration
end

% transformation of the matrix (cube) M into 3D image

hold on
d = 1;
```

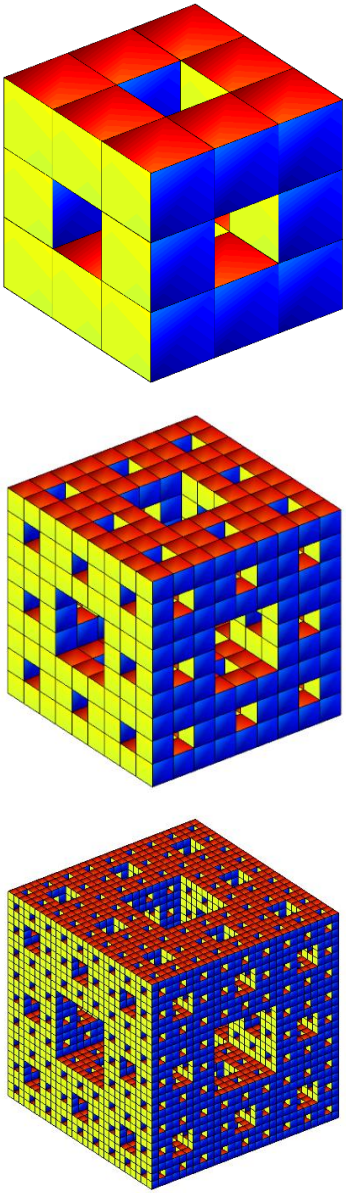


Fig. 21.2. Menger sponges of the first, second and third orders

```
% loop over all cells of the matrix (cube) M
for i = 1:3^n
    for j = 1:3^n
        for k = 1:3^n
            % checking for a cube
            if M(i,j,k) == 0
                cube(i,j,k,d,n); % call the cube drawing function
            end
        end
    end
end

axis equal;
axis off;
hold off;

view(3);
hold off;


function cube(i,j,k,d,n)
% The function of drawing a small cube
% d is the ratio of the size of a small cube to the size of the
sponge;
% value of d = 1 is equivalent to the entire Menger sponge
% i,j,k are the coordinates of the cube
% n is the iteration number

% coordinates of faces of the cube (polygons):
X = [0 0 0 0 0 1; ... % each array column specifies the
     1 0 1 1 1 1; ... % coordinates in 3D space corresponding to
     1 0 1 1 1 1; ...   % the coordinates of the vertices of the
faces of the cube
     0 0 0 0 0 1];      % 6 faces of the cube == 6 columns in the
array

Y = [0 0 0 0 1 0; ...
     0 1 0 0 1 1; ...
     0 1 1 1 1 1; ...
     0 0 1 1 1 0];

Z = [0 0 1 0 0 0; ...
     0 0 1 0 0 0; ...
     1 1 1 0 1 1; ...
     1 1 1 0 1 1];
```

```
% setting gradient for cube faces
% each column of the array sets the color values at the vertex of
the cube face; between the vertices a smooth color transition
occurs, thereby the gradient is modeled; in this implementation,
the back edge of the sponge will not be visible, but it becomes
visible when you change the viewing angle using the view function
C = [0.1 0.84 1.1 1.1 0.1 0.84 ; ...
     0.2 0.86 1.2 1.2 0.2 0.86 ; ...
     0.3 0.88 1.3 1.3 0.3 0.88 ; ...
     0.4 0.90 1.4 1.4 0.4 0.90 ];

% create a cube of size d with a center point (i,j,k)
% and reduction of its size to 3^(-n)
X = (d*(X-0.5) + i)/3^n;
Y = (d*(Y-0.5) + j)/3^n;
Z = (d*(Z-0.5) + k)/3^n;

% drawing a cube
fill3(X,Y,Z,C);
```

Practice

21.1. Modify the code from Example 21.2 and build a square Sierpinsky carpet.

21.2. Modify the code from Example 21.3 and build a pyramidal Menger sponge.

22

L-systems

The concept of L-systems was introduced in 1968 by A. Lindenmayer; originally, they were used in the study of formal languages, as well as in biological models [5].

L-system is a set consisting of an alphabet, an axiom and several rules.

An alphabet is a finite set, and its elements are *symbols*. The nature of the symbols is not important; their only function is to differ from each other. *An alphabetic string* is a finite sequence of alphabet characters.

Axiom is some string of the alphabet.

Each *rule* is a pair consisting of a predecessor and a successor. *The predecessor* is the symbol of the alphabet, and *the successor* is the alphabet string, for example A → FA + A (the arrow separates the predecessor from the successor). In the rule list, the predecessor symbols must be unique.

Once the L-system is defined, it begins to develop in accordance with its rules. The initial state of L-system is its axiom. At further development, this string describing the state will change. The development of the L-system occurs cyclically. In each development cycle, the string is scanned from the beginning to the end, symbol by symbol. For each symbol, the rule is searched for, for which this symbol serves as a predecessor. If there is no such rule, the symbol is left unchanged. If the corresponding rule is found, the predecessor is replaced by the successor string from this rule.

L-systems are used in modeling the growth processes of various objects (for example, crystals, mollusk shells or honeycombs). For illustration, consider the following L-system (it is called Algæ, as its development simulates the growth of one of the alga species):

Axiom	Rules
A	$A \rightarrow B$ $B \rightarrow AB$

Below are the states of this system after six cycles of development.

Cycle	A
0	B
1	AB
2	BAB
3	$ABBAB$
4	$BABABBAB$
5	$ABBABBABABBAB$

Note that each next string is obtained by concatenating the two previous strings.

To model L-system, it is necessary to give meaning to the symbols of its alphabet. For graphic representation of L-systems, the alphabet symbols are interpreted on a "turtle" language, which consists of the following commands:

F: move one step forward, drawing a trace;

b: move one step forward without drawing a trace;

[: open the branch;

]: close the branch;

+: increase the angle by θ;

–: decrease the angle by θ.

At the beginning of the simulation, the turtle is at some point in the (x, y) plane and looks in the direction of the initial angle α. On the first cycle, the turtle draws the axiom on the plane. At each next step of the cycle, the characters in the current string are replaced in accordance with the rules.

Sometimes auxiliary variables X, Y, Z are used, which do not involve changes in the state of the turtle but allow the introduction of several generating rules into the L-system.

The pseudocode for the generating rules $F \rightarrow \text{new } F; X \rightarrow \text{new } X$ is as follows (here $W(j)$ is the j-th character of the string W, $\{T+\}$ is the string T to which the + sign is attached):

```
Entrance:
    axiom (initialization word)
    newF (the generating rule)
    newX (the generating rule)
    level (number of iterations)
```

```
Exit:
    word (resulting string)

Initialization:
    W = axiom
    n = length(W)
    T = {} (empty set)

Steps:
    while level > 0
        for j = 1 to n
           if W(j) = +, T = {T +}, end if
           if W(j) = -, T = {T -}, end if
           if W(j) = [, T = {T [}, end if
           if W(j) = ], T = {T ]}, end if
           if W(j) = F, T = {T newF}, end if
           if W(j) = X, T = {T newX}, end if
        end for
        W = T
        level = level - 1
    end while
    word = W
```

Example 22.1 illustrates the implementation of the code in MatLab for the construction of Harter–Heighway dragon [4]. The dragon is specified by the following rules:

Axiom: FX

Generating rules:

New $f = F$

New $x = X + YF +$

New y $= -$ FX$-$ Y

Accordingly, for the first few steps, the L-system will have the following form:

First step: $FX + YF +$

Second step: $FX + YF + +- FX - YF +$

Third step: $FX + YF + +- FX - YF + +- FX + YF +- - FX - YF +$

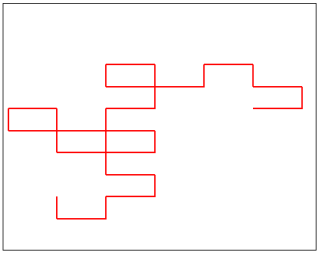
Example 22.1.

```
function [X,Y]=dragon(Lmax)
% function that returns the dragon image
% Lmax is the order of the maximum iteration
```

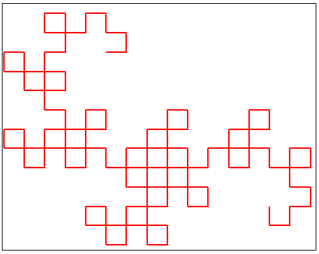
```
% generating rules
Axiom='FX';
Newf='F';
Newx='X+YF+';
Newy='-FX-Y';
teta=pi/2;
alpha=0;

p=[0;0]; % coordinate of the starting point

% call the function that returns the coordinates of the dragon
p=Coord(p,Lmax,Axiom,Newf,Newx,Newy,alpha,teta);

% visualization
M=size(p,2); % number of points

X=p(1:1,1:M); % vector containing x-coordinates of the dragon
Y=p(2:2,1:M); % vector containing y-coordinates of the dragon

figure;
plot(X,Y,'Color','r','LineWidth',2);
set(gca,'XTick',[]) % disables x-axis ticks
set(gca,'XTickLabel',[]) % disables x labels
set(gca,'YTick',[])
set(gca,'YTickLabel',[])

% function that returns the coordinates of the dragon
function z=Coord(p,Lmax,Axiom,Newf,Newx,Newy,alpha,teta)

Rule=DraconString(Lmax,Axiom,Newf,Newx,Newy,1,''); % defining the
L-system

M=length(Rule);
for i=1:M
 tmp=p(1:2,size(p,2):size(p,2));

 if Rule(i)=='F' % step forward
  R=[cos(alpha);sin(alpha)];
  R=R/(2^Lmax); % scaling of the picture depending on the number
   of iterations
  tmp=tmp+R;
  p=cat(2,p,tmp);
 end

 if Rule(i)=='+' % increase the angle
  alpha=alpha+teta;
 end

 if Rule(i)=='-' % decrease the angle
  alpha=alpha-teta;
 end
 end
z=p;
```

```
% function that returns L-system
function z=DraconString(Lmax,Axiom,Newf,Newx,Newy,n,tmp_str)

if n <= Lmax
 if n==1
  tmp_str=Axiom;
 end

 M=length(tmp_str);
 tmp=''; % create an empty string

 for i=1:M

  if tmp_str(i)=='F'
   tmp=strcat(tmp,Newf);
  end

  if tmp_str(i)=='X'
   tmp=strcat(tmp,Newx);
  end

  if tmp_str(i)=='Y'
   tmp=strcat(tmp,Newy);
  end

  if not(tmp_str(i)=='F') && not(tmp_str(i)=='X') &&
  not(tmp_str(i)=='Y')
   tmp=strcat(tmp,tmp_str(i));
  end
end

 tmp_str=tmp;
 n=n+1;
 tmp_str=DraconString(Lmax,Axiom,Newf,Newx,Newy,n,tmp_str); %
рекурсия
end
z=tmp_str;
```

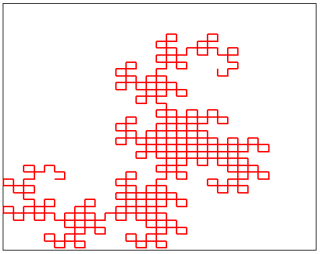
Fig. 22.1 shows the Harter–Heighway dragon for various number of iterations. The calculation time of the L-system increases very rapidly with the number of iterations.



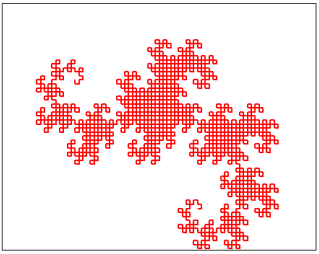
$L_{\max} = 5$



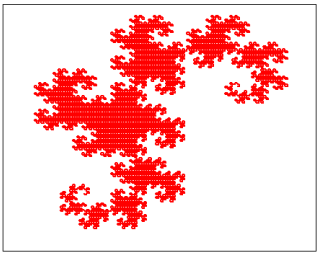
$L_{\max} = 7$



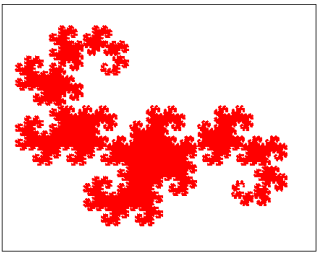
$L_{\max} = 9$



$L_{\max} = 11$



$L_{\max} = 13$



$L_{\max} = 15$

Fig. 22.1. Harter–Heighway dragon for various number of iterations

Practice

22.1. Using the above code for the Harter–Heighway dragon, implement the construction of the following figures (Fig. 22.2):

a) the Hilbert curve:

Axiom: X

Generating rules:

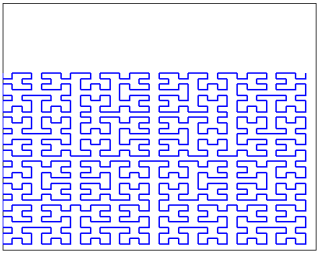
New $f = F$
New $x = -YF + XFX + FY-$
New $y = +XF-YFY-FX+$
$\alpha = 0;\ \theta = \pi/2$

b) Gosper curve:
Axiom: XF
Generating rules:
New $f = F$
New $x = X + YF + +YF-FX- -FXFX-YF+$
New $y = -FX + YFYF + +YF + FX--FX-Y$
$\alpha = 0;\ \theta = \pi/3$

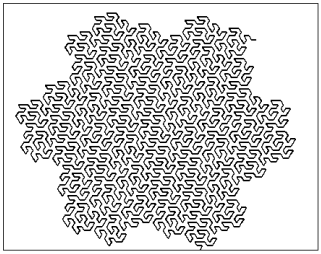
c) Sierpinsky curve:
Axiom: $F + XF + F + XF$
Generating rules:
New $f = F$
New $x = XF-F + F-XF + F + XF-F + F-X$
New $y = ''$
$\alpha = \frac{\pi}{4};\ \theta = \pi/2$

d) island:
Axiom: $F + F + F + F$
Generating rules:
New $f = F + F-F-FFF + F + F-F$
New $x = ''$
New $y = ''$
$\alpha = 0;\ \theta = \pi/2$

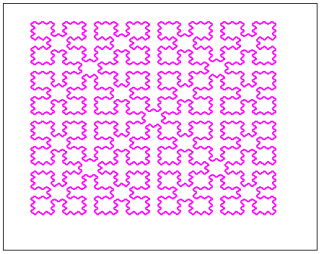
e) ornament:
Axiom: $F + F + F + F$
Generating rules:
New $f = FF + F + F + F + FF$
New $x = ''$
New $y = ''$
$\alpha = 0;\ \theta = \pi/2$



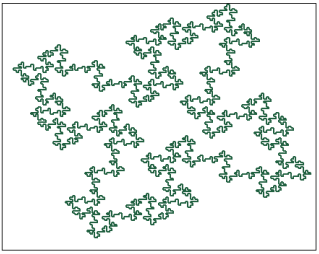
a



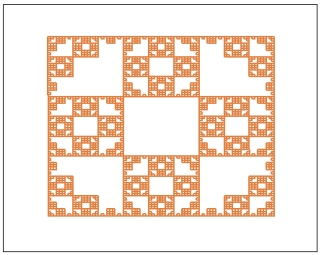
b



c



d



e

Fig. 22.2. Images of curves from Practice 22.1

The images of these curves are shown in Fig. 22.2, where *a* is Hilbert curve of the 5-th order; *b* is Gosper curve of the 4th order; *c* is Sierpinsky curve of the 4th order; *d* is curve "island" of the third order; *e* is curve "ornament" of the 4th order.

22.2. Modify the above code for the Harter–Heighway dragon so that it would be possible to account for the b command, i.e. 'forward movement without drawing a trace'. Using the new algorithm, build the following curves:

a) mosaic:

Axiom: $F-F-F-F$
Generating rules:
New $f = F-b+FF-F-FF-Fb-FF+b-FF+F+FF+$
$+Fb+FFF$
New $b = bbbbbb$
$\alpha = 0$
$\theta = \pi/2$

б) carpet:

Axiom: F
Generating rules:
New $f = F+F-F-F-b+F+F+F-F$
New $b = bbb$
$\alpha = 0$
$\theta = \pi/2$

22.3. Modify the above code for the Harter–Heighway dragon so that it would be possible to take into account branching commands [and]. The symbol [opens a branch in the L-system; the current coordinates of the turtle and the angle at which it looks (x, y, α) must be stored in a separate variable at this point saving. These coordinates must be returned if the symbol] (close the branch) is encountered in the L-system. Using the new algorithm, build the following curves:

a) flower:

Axiom: $F\,[\,F\,+\,F\,]\,[\,-\,F\,-\,F\,]\,[\,+\,+\,F\,]\,[\,-\,-\,F\,]\,F$
Generating rules:
New $f = F\,F\,[+\,+\,F\,]\,[\,+F\,]\,[\,F\,]\,[\,-\,F\,]\,[\,-\,-\,F\,]$

$\alpha = \pi/2$
$\theta = \pi/2$

б) bush:

Axiom: F
Generating rules:
New f $= - F + F + [+ F - F -] - [- F + F + F]$
$\alpha = 0$
$\theta = \pi/8$

23

Parallel Computing

23.1. Mathematical basis of parallel computations

Many tasks require calculations with a large number of operations, which demand considerable resources, moreover, it can be confidently assumed that no matter what speeds the computing equipment has achieved, there will always be problems that would require considerable time to solve. Many of these complex tasks require that the result be obtained in as little time as possible. Such modern problems of science and technology as climate modeling, genetic engineering, design of integrated circuits, analysis of environmental pollution, creation of medicines, etc., require for their analysis a computer with a capacity of more than 1,000 billion floating point operations per second (1 teraflop). On the other hand, it is a great technical problem to reduce the execution time of each operation in the microprocessor. An obvious way to increase the speed of computing would be to use not one computing device, but several, working together to solve one task. This approach is called *parallel computing*. Despite the seeming simplicity of the solution, it is often a very nontrivial task to design computer technology and develop algorithms. The first problem lies in the fact that in order for the problem to be solved with the help of parallel computations, the algorithm for solving it must allow parallelization, moreover, not every problem can be solved by a parallel algorithm. Another problem, which is no less important, is the construction of a system on which parallel computations could be implemented.

Parallel calculations can speed up the optimization process, provided that the calculations require much more time than sending, receiving and processing data. However, effective planning of operations and control of the system is a complex and hard-to-solve task.

Traditionally, a program was written as a sequence of calculations. It was run on a single computer with one central processor. Program instructions were executed one after another. Some of today's programs try to use multiprocessors for computations. The program is divided into parts that can be solved simultaneously. These parts are computed on different processors. Computing resources can contain either a computer with a multiprocessor, or several computers connected over a network, or a combination of both.

Within the conditions of the transition of processor manufacturers to multi-core architectures, parallel programming becomes a vital tool for every advanced programmer and researcher. The *de facto* parallel programming standard is the Message Passing Interface (MPI), but not everyone can master it. Therefore, the developers of software packages, including MatLab, are concerned about the introduction of parallel and distributed computing in their packages.

Parallel calculations are possible when there is no need to complete the previous operation to start the next one. As an example, consider the following expression:

$$5 \cdot 7 + 3 \cdot 13.$$

In order to compute the second multiplication, it is not necessary to know the result of the first one, therefore, both multiplications can be made in parallel, and only after this addition is performed. Obviously, not every calculation can be parallelized. The expression

$$5 \cdot 7 \cdot 3 + 13$$

can only be calculated sequentially, at first the first multiplication, then the second, and only after that the addition.

23.2. Parallel Computing Toolbox

Parallel Computing Toolbox allows solving tasks which require to process a significant amount of data and calculations, using MatLab and

Simulink on multi-core and multiprocessor computers. Such methods of parallel processing as parallel `for` loops, distributed arrays, parallel numerical algorithms, and MPI functions allow parallel algorithms to be implemented in MatLab at a high level, without programming for special hardware and network architectures. As a result, converting MatLab applications to parallel versions requires minor code changes and does not require programming in the lower level language at all. Applications run online or offline, in batch environments.

23.3. Parallel loop `for`

Many applications contain repeated code segments. Often `for` loop is used in such cases. The ability to execute code in parallel on a separate computer or on a cluster can significantly improve performance in the following cases.

1. Programs with independent segments that perform a number of unrelated tasks can be performed simultaneously on separate resources. The `for` loop is not normally used for such distinctly different tasks, but a parallel `for` loop can be an appropriate solution.

2. Applications that involve moving of parameters, that is, the application loads any initial data that does not change in the process, and the resulting data is collected at the output. With a large number of iterations, moving parameters can take a long time: each iteration can be performed quite quickly, but it will take a lot of time to execute thousands or millions of such iterations. On the other hand, the process may not require a large number of iterations, but each of them can be performed for a long time. Typically, the only difference between iterations is determined by different input data. In these cases, the ability to perform separate iterations at the same time can significantly improve performance. Performing these iterations in parallel is an ideal way of processing large or multiple sets of data. The only limitation is that the iterations can not depend on each other.

Parallel Computing Toolbox improves the performance of such loops, allowing several system processes (in the MatLab environment they are called *workers*) to perform separate iterations of the loop simultaneously. For example, a loop of 100 iterations can be run on a cluster of 20 workers so that simultaneously each system process performs five iterations. Of course, the speed of work will not improve by 20 times due to network traffic, but the acceleration will be significant. Even running local system processes on one PC shows a significant improvement in performance on a multi-core computer. So, regardless of whether the loop takes a lot of time because of the number of iterations, or because of their size, the work speed can be increased by distributing iterations between MatLab workers.

Example 23.1 shows how to change a simple `for` loop to run it in parallel. This example demonstrates technical differences, and there are no significant improvements in terms of performance.

Example 23.1. Let there be a loop for calculating the values of the sine and its graphical display:

```
for i=1:1024
    A(i) = sin(i*2*pi/1024);
end
plot(A)
```

To interactively execute the code that contains a parallel loop, first the MatLab *pool* must be opened. This will reserve several work sessions for loop iterations (see Fig. 23.1).

```
parpool
```

After that, we change the source code for its parallel execution:

```
parfor i=1:1024
    A(i) = sin(i*2*pi/1024);
end
plot(A)
```

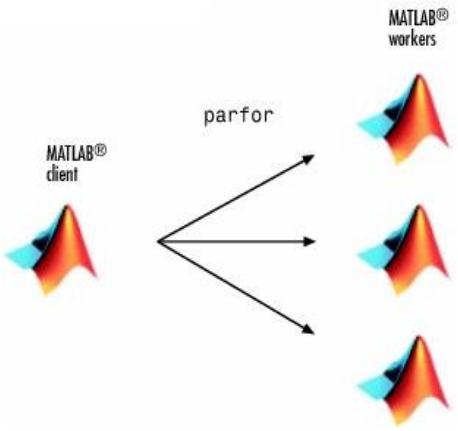


Fig. 23.1. Parallelization of the `for` loop

The only difference in this loop is the keyword `parfor` instead of `for`. After the loop is completed, the results look the same as the ones generated in the previous ordinary loop.

Since the iterations are performed in parallel in all MatLab sessions, each iteration must be completely independent of all others. For instance, the system process that computes the value for A(100) will not necessarily be the system process that computes A(500). Also, there is no guarantee that A(900) will be calculated later than A(400). The values of all the elements of array A will be available in the MatLab client session only after all the data is returned from the workers of MatLab, and the loop is completed.

After completing the calculations, the pool should be closed:

```
delete(gcp)
```

23.4. spmd

For multi-level monitoring of parallel schemes, the Parallel Computing Toolbox provides such constructs as `spmd` (the "one program – many data" paradigm), and a number of message transfer procedures based on the standard MPI library. The `spmd` construct allows selecting sections of code that will be run simultaneously on the workers involved in parallel computations. During execution of the

program, spmd automatically transfers the data and code inside this section to the system processes and, once execution is complete, takes the results back to the MatLab client session.

23.5. Parallel computations with pmode

Using the pmode mode, it is possible to access workers directly from the MatLab command window, view their local variables, and exchange data between them. In the pmode mode, the commands entered in the command window of MatLab will be executed by all working processes. This mode is most often used as a debugging tool for parallel programs.

The pmode command allows interactively performing a parallel task that is executed simultaneously by several workers. Commands typed in the Parallel Command Window are executed by all workers. Each worker executes commands in their own workspace and with their own variables.

Synchronization of workers means that each worker after the completion of the command is idle, waiting for all workers which are working on this task to complete the same command. Only after all workers complete the execution of one command, they all go to the next pmode command.

The command pmode provides a display for each worker performing the task, where commands can be entered, results can be viewed, access to the workspace of each worker can be performed, and so on. pmode does not allow alternating sequential and parallel work. After exiting the pmode session, all information and data on the workers are deleted. The next pmode session always starts with an empty state. The following example illustrates how to work in pmode mode.

Example 23.2. Run pmode with the command `pmode`:

```
pmode start local 2
```

This command starts two local workers, creates a parallel task to work with them and opens the Parallel Command Window (Fig. 23.2).

To illustrate that the commands in `pmode` are executed by all workers, type the following in the command window:

```
P>> help magic
```

Introduce the variable in pmode (Fig. 23.3):

```
P>> x = pi
```

A variable does not necessarily have the same value for each worker. The function `labindex` returns the ID of each worker working on the parallel task. For example, the variable `x` has different values in the workspace of each worker:

```
x = labindex
```

The function `numlabs` returns the number of workers working on the task:

```
all = numlabs
```

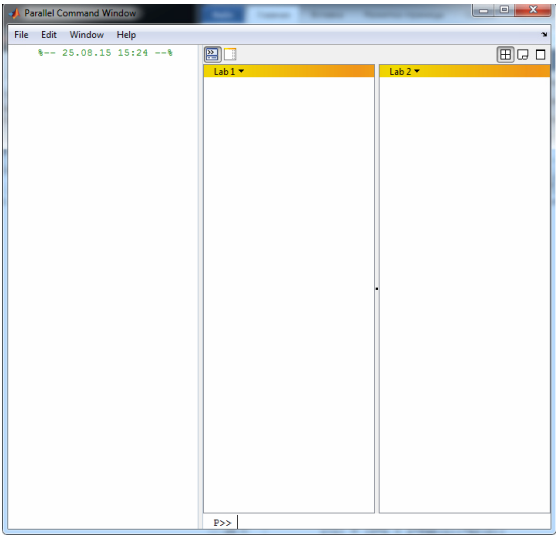


Fig. 23.2. Start of Parallel Command Window

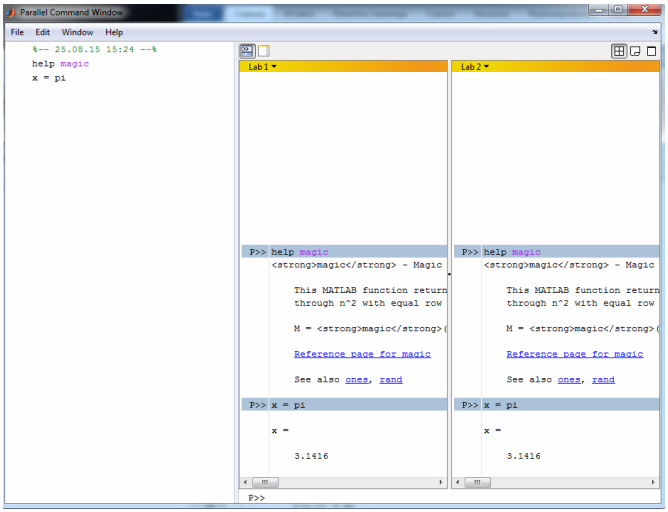


Fig. 23.3. Introducing a variable in `pmode`

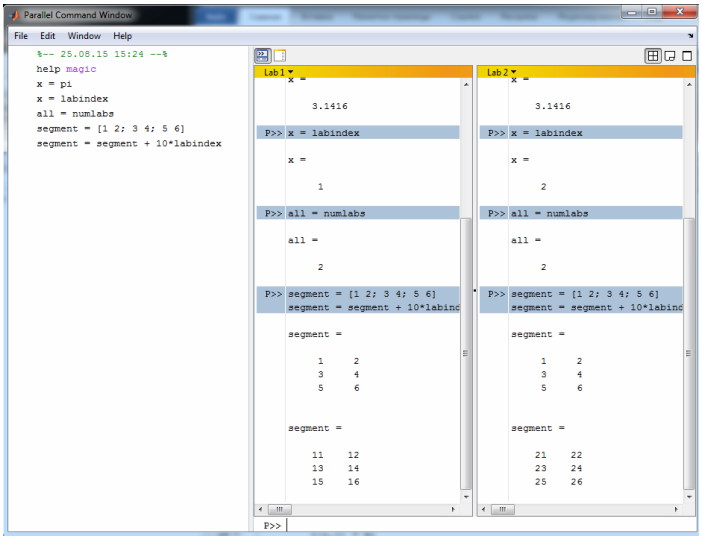


Fig. 23.4. Arrays in `pmode`

To create arrays (Fig. 23.4):

```
segment = [1 2; 3 4; 5 6]
segment = segment + 10*labindex
```

To end the work with `pmode`, use the `pmode exit` command.

Example 23.3 presents the construction of the Mandelbrot set using the parallel loop `parfor`.

Example 23.3. The Mandelbrot set $\mathcal{M}$ (Fig. 23.5) for the polynomial $(z) = z^2 + c$ is defined as the set of all $c \in \mathbb{C}$ for which the orbit of the origin (i.e., the point 0) is bounded:

$$\mathcal{M} = \{c \in \mathbb{C} : \{f^n(0)\}_{n=0}^{\infty} \text{ bounded}\}.$$

The equivalent definition can be written as follows:

$$\mathcal{M} = \{c \in \mathbb{C} : f^n(0) \nrightarrow \infty \text{ as } n \to \infty\}.$$

The function f here is defined on the complex plane $\mathbb{C}$. MatLab supports complex numbers; the imaginary unit is specified using the letters `i` or `j`, in general form the complex number $z = a + bi$ is specified by the command

```
z = a + bi
```

or

```
z = a + bj
```

To plot the Mandelbrot set, it is necessary to choose a window on the complex plane, and for each point c inside this window find out whether the sequence $\{f^n(0)\}_{n=0}^{\infty}$ is divergent or not. The number of iterations N should not be taken too small; a certain color is assigned to the point c after N iterations depending on the distance of the orbit z to the origin:

```
function par_mandel(N,max_iter, epsilon)

% N is the number of points in the complex plane in each direction
W = zeros(N,N); % create an image matrix

tic; % time counter starts
parfor i=1:N % parallel loop of calculating the image matrix by
             columns
    Z=zeros(1,N); % a column-vector of iteration values
    W_local=zeros(1,N);  % create a column-vector of the image
                         matrix
```

```
    C=(2*(1:N)/(N/2)-2.5)+1.5i*(i./(N/2)-1); % define a window on
              the complex plane; note that there is no conflict
              between the variable i denoting the iteration number
              and the letter i denoting the imaginary part of the
              complex number
    for j=1:max_iter % iteration loop
        Z=Z.*Z+C % iteration relation
        W_local=W_local+(abs(Z)>epsilon);  % if at the current
              iteration the point extends beyond the radius defined
              by epsilon, a unit is added to the W_local array
              specifying the color of the point on the plot; the
              earlier the point extends outside the circle with the
              radius of epsilon, the greater is the value for it in
              the array W_local
    end

    W(i,:)=W_local;
end

toc; % time counter stops

imagesc(W); % function imagesc normalizes the values of W to build
              the image, and displays the image on the screen
colormap(jet); % jet is one of the built-in palettes of MatLab

axis equal
axis off

end
```

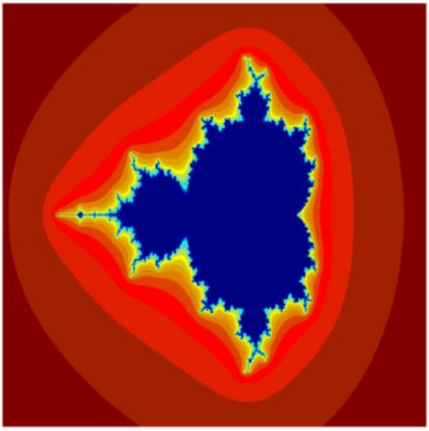


Fig. 23.5. Mandelbrot set

Practice

23.1. Implement an algorithm for constructing Mandelbrot set without a parallel loop. Construct Mandelbrot set using parameters $N = 1000;\ \max\_iter = 23;\ epsilon = 10$. Compare the calculation time with the time of a similar calculation using a parallel loop.

23.2. Consider different parts of the complex plane and plot images for them using the function `par_mandel`.

23.3. Write a parallel code that calculates the number π by Monte Carlo method.

23.4. Write the code "Euler's Knight", based on a stochastic method using a parallel `parfor` loop to solve the problem of the progress of the "knight". The task of the "knight" is to find a route for the chess knight that passes through all the fields of the board of a given size only once.

One example of the implementation of this algorithm can be an algorithm consisting of the following parts:

1) making a map of the moves of the "knight": to number the board fields, and for each field find the neighbors to which the "knight" can jump;

2) realization of the loop: the "knight" passes the field on the basis of the stochastic method with the aid of map of moves; when the "knight" moves to a certain field, this field is eliminated from the map;

3) realization of a random search of ways of passing the board.

24

Lattices

A numerical solution of a problem in condensed matter physics is often based on the use of lattice models. Such models are either formulated in a natural way, when the lattice sites are determined by the crystal structure of the material substance being investigated or are a discrete approximation of the phase space of the problem. The primary task in this case, as a rule, is the determination of the geometry of the lattice, i.e. calculating the coordinates of sites, determining the coordination spheres and the number of sites that they contain, determining the unique distances between nodes on the grid, and so on (Fig. 24.1).

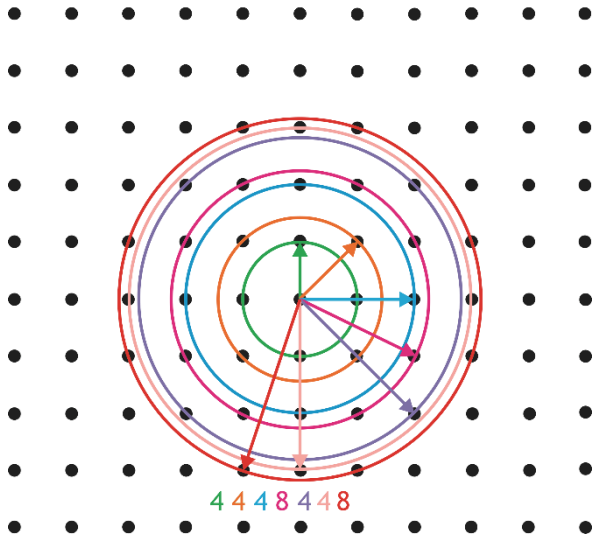


Fig. 24.1. A square lattice. The first seven coordination spheres are shown and the number of sites in them

In the study of 2D problems, periodic boundary conditions in both directions are often introduced to simulate the continuum, and the system becomes a torus (Fig. 24.2).

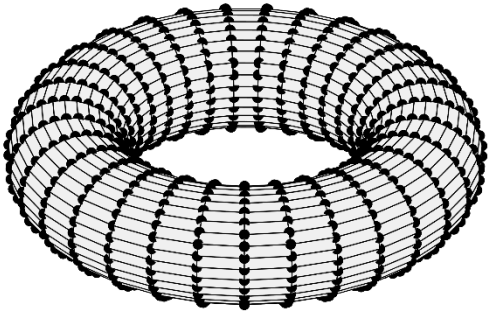


Fig. 24.2. A square lattice with periodic boundary conditions in both directions

Consider the code that solves these problems for a simple square lattice (Example 24.1) and for a graphene lattice (Example 24.2).

Example 24.1.

```
function [lattice]=SquareLattice(Lx,Ly)

% Lx is the number of atoms along the X axis, Ly is the number of
atoms along the X axis

eX = 1;
eY = 1; % eY and eX are the basis vectors of the lattice

N = Lx*Ly; % N is the total number of atoms

coord = zeros(3,N); % creation of the matrix coord for the
                      description of the lattice structure
coord(3,:) = (1:1:N); % The third row of the matrix coord is
responsible for numbering the atoms in the lattice
coord(1,:) = mod(coord(3,:)-1,Lx)*eX; % X-coordinates of atoms
coord(2,:) = floor((coord(3,:)-1)/Lx)*eY; % Y-coordinates of atoms
lattice.structure = coord; % creation of the lattice structure for
a full description of the lattice

xMax = max(coord(1,:))+eX; % xMax is the maximum value of the X
coordinate, yMax is the maximum value of the Y coordinate
```

```
yMax = max(coord(2,:))+eY; % values of the maximum coordinates are
                introduced to account for the Born-Karman boundary
                conditions

% implementation of the algorithm for enumeration of all unique
interatomic distances

coord(1,:) = coord(1,:)+xMax/2; % "scrolling" the atomic lattice
as a "canvas"
coord(2,:) = coord(2,:)+yMax/2; % the first lattice atom is put in
the center of the "canvas"
coord(1,coord(1,:)>=xMax)   =   coord(1,coord(1,:)>=xMax)-xMax;   %
"scrolling" takes into account the Born-Karman boundary conditions
coord(2,coord(2,:)>=yMax) = coord(2,coord(2,:)>=yMax)-yMax;

% create a field radiuses in the lattice structure
% which is the matrix of size N x N
% for storing the interatomic distances between atom i and atom j
in the form of lattice.radiuses(i,j) or lattice.radiuses(j,i)

lattice.radiuses=zeros(N,N);
for i=1:1:N % loop over atoms
    for j=1:1:N % second loop over atoms
        lattice.radiuses(i,j)=round(10^8*sqrt((coord(1,i)-
         coord(1,j))^2+(coord(2,i)-coord(2,j))^2))/(10^8);
         % rounding to eight decimal places
    end
    coord(1,:)=coord(1,:)-eX; % "canvas" shift
    coord(2,:)=coord(2,:)-eY*(mod(i,Lx)==0);  % putting the next
                        atom in the center of the "canvas"
          coord(1,(coord(1,:)<0))=coord(1,(coord(1,:)<0))+xMax; %
         taking into account the Born-Karman boundary conditions
    coord(2,(coord(2,:)<0))=coord(2,(coord(2,:)<0))+yMax;
end

uniqueRadiuses=unique(lattice.radiuses); % determination of unique
interatomic distances from the matrix radiuses and storing them in
the vector uniqueRadiuses

numberOfUniqueRadiuses=size(uniqueRadiuses,1);
% numberOfUniqueRadiuses is the number of unique interatomic
distances (0 is included)

% creating the field neighbors of the lattice structure as an
array of cells N x numberOfUniqueRadiuses-1 to describe the
neighbors of each atom in the lattice
lattice.neighbors=cell(N,numberOfUniqueRadiuses-1);
for i=1:1:N % loop over atoms
    for j=1:1:N % loop over atoms
```

```
        for   k=2:1:numberOfUniqueRadiuses   %   loop   over   unique
              interatomic distances
             if   uniqueRadiuses(k)==lattice.radiuses(i,j)   %   the
              search condition for neighbors of the k-th order
             lattice.neighbors{i,k-1}=[lattice.neighbors{i,k-
              1} j];     % storing the neighbor
             end
        end
    end
end
end
```

```
function  SquareLatticePlot(lattice)
% visualization of a square lattice

xMax=max(lattice.structure(1,:)); % xMax and yMax are the maximum
values of the x and y coordinates, respectively
yMax=max(lattice.structure(2,:));

figure

plot(lattice.structure(1,:),lattice.structure(2,:),'b.',
'MarkerSize',20)
hold on
for  i=1:1:size(lattice.structure,2)  % numbering  of  atoms  in  the
figure
    text(lattice.structure(1,i)+0.15,lattice.structure(2,i),
     num2str(lattice.structure(3,i)))

end
axis([-1  (xMax+1) -1 (yMax+1)]);
end
```

The 8×8 square lattice is shown in Fig. 24.3.

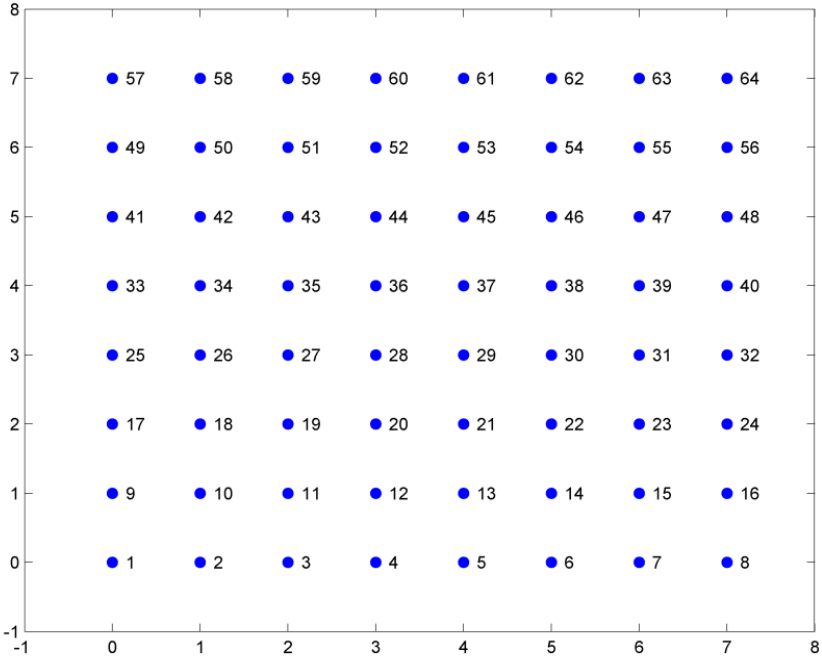


Fig. 24.3. The square lattice 8×8. Coordinates and numbering of sites

Fig. 24.4 shows the structure `lattice` of the square lattice. The field `structure` contains the x-coordinates of the atoms (the first line), y-coordinates of atoms (second line) and their numbers (third line); the field `radiuses` is the matrix that determines the distances between each two atoms on the lattice; the field `neighbors` for each atom contains the information about the number and labels of atoms of the corresponding coordination sphere; for 8×8 lattice with periodic boundary conditions, there are 14 coordination spheres.

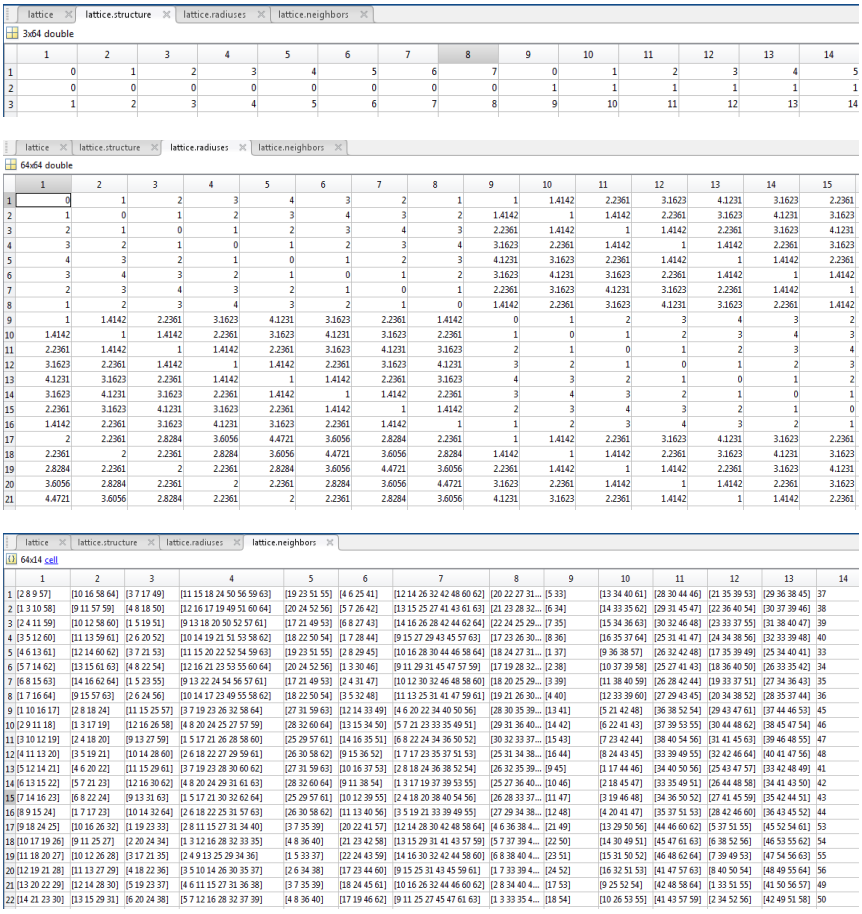


Fig. 24.4. Structure `lattice` of a square lattice

The graphene lattice (Example 24.2) is a hexagonal lattice formed by carbon atoms. From physical considerations, it is convenient to represent it as two triangular sublattices A and B nested inside each other (Fig. 24.5-24.7).

Atoms of even coordination spheres belong to the same sublattice as the atom under consideration; atoms of odd coordination spheres belong to another sublattice.

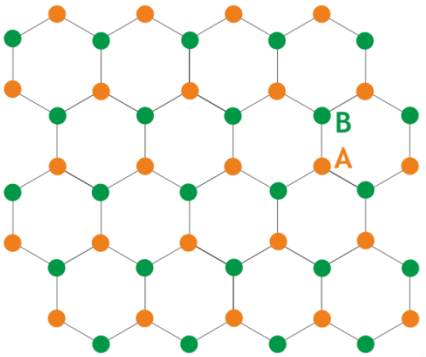


Fig. 24.5. Graphen lattice

Example 24.2. The program code is similar to the code of Example 24.1, with the difference that atoms are divided into two sublattices, and the basis vectors are not orthogonal to each other:

```
function [lattice]=GrapheneLattice(Lx,Ly)
% Lx is the number of atoms along the X axis, Ly * 2 is the number
of atoms along the Y axis

eY=0.5;
eX=sqrt(1-0.5^2); % eY and eX are the basis vectors of the lattice
xMax=Lx*eX*2;

N=Lx*Ly; fullN=Lx*Ly*2; % N is the number of atoms in one of the
two sublattices, fullN is the total number of atoms

ACoord=zeros(3,N);
BCoord=zeros(3,N);

ACoord(3,:)=1:1:N;
BCoord(3,:)=ACoord(3,:)+Lx*Ly;

ACoord(1,:)=(floor((ACoord(3,:)-1)/Lx)+2*mod(ACoord(3,:)-
1,Lx))*eX;
ACoord(2,:)=floor((ACoord(3,:)-1)/Lx)*3*eY;
ACoord(1,(ACoord(1,:)>=xMax))=ACoord(1,(ACoord(1,:)>=xMax))-xMax;
BCoord(1,:)=ACoord(1,:);
BCoord(2,:)=ACoord(2,:)+2*eY;

fullCoord=[ACoord,BCoord];
lattice.structure=fullCoord;
yMax=max(fullCoord(2,:))+eY;

fullCoord(1,:)=fullCoord(1,:)+xMax/2;
fullCoord(2,:)=fullCoord(2,:)+yMax/2;
fullCoord(1,(fullCoord(1,:)>=xMax))=fullCoord(1,(fullCoord(1,:)>=
```

```
xMax))-xMax;
fullCoord(2,(fullCoord(2,:)>=yMax))=fullCoord(2,(fullCoord(2,:)>=
yMax))-yMax;

lattice.radiuses=zeros(fullN,fullN);
for i=1:1:N
    for j=1:1:fullN
        lattice.radiuses(i,j)=round(10^8*sqrt((fullCoord(1,i)-
fullCoord(1,j))^2+(fullCoord(2,i)-fullCoord(2,j))^2))/(10^8);
    end
    fullCoord(1,:)=fullCoord(1,:)-eX*(mod(i,Lx)==0)-2*eX;
    fullCoord(2,:)=fullCoord(2,:)-eY*3*(mod(i,Lx)==0);

fullCoord(1,(fullCoord(1,:)<0))=fullCoord(1,(fullCoord(1,:)<0))+
xMax;

fullCoord(2,(fullCoord(2,:)<0))=fullCoord(2,(fullCoord(2,:)<0))+
yMax;
end

fullCoord=lattice.structure;
fullCoord(1,:)=fullCoord(1,:)+xMax/2;
fullCoord(2,:)=fullCoord(2,:)+yMax/2-2*eY;
fullCoord(1,(fullCoord(1,:)>=xMax))=fullCoord(1,(fullCoord(1,:)>=
xMax))-xMax;
fullCoord(2,(fullCoord(2,:)>=yMax))=fullCoord(2,(fullCoord(2,:)>=
yMax))-yMax;

for i=(N+1):1:fullN
    for j=1:1:fullN
        lattice.radiuses(i,j)=round(10^8*sqrt((fullCoord(1,i)-
fullCoord(1,j))^2+(fullCoord(2,i)-fullCoord(2,j))^2))/(10^8);
    end
    fullCoord(1,:)=fullCoord(1,:)-eX*(mod(i,Lx)==0)-2*eX;
    fullCoord(2,:)=fullCoord(2,:)-eY*3*(mod(i,Lx)==0);

fullCoord(1,(fullCoord(1,:)<0))=fullCoord(1,(fullCoord(1,:)<0))+
xMax;

fullCoord(2,(fullCoord(2,:)<0))=fullCoord(2,(fullCoord(2,:)<0))+
yMax;

end

uniqueRadiuses=unique(lattice.radiuses);
numberOfUniqueRadiuses=size(uniqueRadiuses,1);

lattice.neighbors=cell(fullN,numberOfUniqueRadiuses-1);

for i=1:1:fullN
    for j=1:1:fullN
        for k=2:1:numberOfUniqueRadiuses
            if uniqueRadiuses(k)==lattice.radiuses(i,j)
                lattice.neighbors{i,k-1}=[lattice.neighbors{i,k-
                 1} j];
```

```
            end
        end
    end
end

function  GrapheneLatticePlot(lattice)

N=size(lattice.radiuses,1);

xMax=max(lattice.structure(1,:));
yMax=max(lattice.structure(2,:));

lattice_A=lattice.structure(:,1:N/2);
lattice_B=lattice.structure(:,N/2+1:N);

figure % рис. 24.6, 24.7

plot(lattice_A(1,:),lattice_A(2,:),'b.','MarkerSize',20)
hold on
plot(lattice_B(1,:),lattice_B(2,:),'r.','MarkerSize',20)

for i=1:1:size(lattice_A,2)
text(lattice_A(1,i)+0.15,lattice_A(2,i),num2str(lattice_A(3,i)))
text(lattice_B(1,i)+0.15,lattice_B(2,i),num2str(lattice_B(3,i)))
end
axis([-1  (xMax+1) -1 (yMax+1)]);
end
```

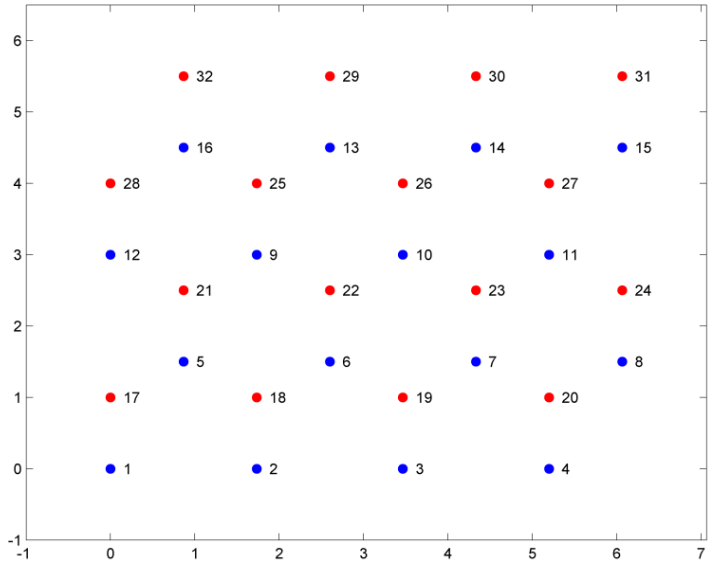


Fig. 24.6. Graphene lattice 4 × 4. Coordinates and numbering of sites

lattice | lattice.structure | lattice.radiuses | lattice.neighbors

3x32 double

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	0	1.7321	3.4641	5.1962	0.8660	2.5981	4.3301	6.0622	1.7321	3.4641	5.1962	0	2.5981	4.3301
2	0	0	0	0	1.5000	1.5000	1.5000	1.5000	3	3	3	3	4.5000	4.5000
3	1	2	3	4	5	6	7	8	9	10	11	12	13	14

lattice | lattice.structure | lattice.radiuses | lattice.neighbors

32x32 double

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	0	1.7321	3.4641	1.7321	1.7321	3	3	1.7321	3.4641	4.5826	3.4641	3	3	3
2	1.7321	0	1.7321	3.4641	1.7321	1.7321	3	3	3	3.4641	4.5826	3.4641	1.7321	3
3	3.4641	1.7321	0	1.7321	3	1.7321	1.7321	3	3.4641	3	3.4641	4.5826	1.7321	1.7321
4	1.7321	3.4641	1.7321	0	3	3	1.7321	1.7321	4.5826	3.4641	3	3.4641	3	1.7321
5	1.7321	1.7321	3	3	0	1.7321	3.4641	1.7321	1.7321	3	3	1.7321	3.4641	4.5826
6	3	1.7321	1.7321	3	1.7321	0	1.7321	3.4641	1.7321	1.7321	3	3	3	3.4641
7	3	3	1.7321	1.7321	3.4641	1.7321	0	1.7321	3	1.7321	1.7321	3	3.4641	3
8	1.7321	3	3	1.7321	1.7321	3.4641	1.7321	0	3	3	1.7321	1.7321	4.5826	3.4641
9	3.4641	3	3.4641	4.5826	1.7321	1.7321	3	3	0	1.7321	3.4641	1.7321	1.7321	3
10	4.5826	3.4641	3	3.4641	3	1.7321	1.7321	3	1.7321	0	1.7321	3.4641	1.7321	1.7321
11	3.4641	4.5826	3.4641	3	3	3	1.7321	1.7321	3.4641	1.7321	0	1.7321	3	1.7321
12	3	3.4641	4.5826	3.4641	1.7321	3	3	1.7321	1.7321	3.4641	1.7321	0	3	3
13	3	1.7321	1.7321	3	3.4641	3	3.4641	4.5826	1.7321	1.7321	3	3	0	1.7321
14	3	3	1.7321	1.7321	4.5826	3.4641	3	3.4641	3	1.7321	1.7321	3	1.7321	0
15	1.7321	3	3	1.7321	3.4641	4.5826	3.4641	3	3	3	1.7321	1.7321	3.4641	1.7321
16	1.7321	1.7321	3	3	3	3.4641	4.5826	3.4641	1.7321	3	3	1.7321	1.7321	3.4641
17	1	2	3.6056	2	1	2.6458	2.6458	1	2.6458	4	2.6458	2	3.6056	3.6056
18	2	1	2	3.6056	1	1	2.6458	2.6458	2	2.6458	4	2.6458	2.6458	3.6056
19	3.6056	2	1	2	2.6458	1	1	2.6458	2.6458	2	2.6458	4	2.6458	2.6458
20	2	3.6056	2	1	2.6458	2.6458	1	1	4	2.6458	2	2.6458	3.6056	2.6458
21	2.6458	2.6458	3.6056	3.6056	1	2	3.6056	2	1	2.6458	2.6458	1	2.6458	4

lattice | lattice.structure | lattice.radiuses | lattice.neighbors

32x9 cell

	1	2	3	4	5	6	7	8	9
1	[17 31 32]	[2 4 5 8 15 16]	[18 20 28]	[21 24 25 27 29 30]	[6 7 12 13 14]	[3 9 11]	[19 22 23]	26	10
2	[18 29 32]	[1 3 5 6 13 16]	[17 19 25]	[21 22 26 28 30 31]	[7 8 9 14 15]	[4 10 12]	[20 23 24]	27	11
3	[19 29 30]	[2 4 6 7 13 14]	[18 20 26]	[22 23 25 27 31 32]	[5 8 10 15 16]	[1 9 11]	[17 21 24]	28	12
4	[20 30 31]	[1 3 7 8 14 15]	[17 19 27]	[23 24 26 28 29 32]	[5 6 11 13 16]	[2 10 12]	[18 21 22]	25	9
5	[17 18 21]	[1 2 6 8 9 12]	[22 24 32]	[19 20 25 28 29 31]	[3 4 10 11 16]	[7 13 15]	[23 26 27]	30	14
6	[18 19 22]	[2 3 5 7 9 10]	[21 23 29]	[17 20 25 26 30 32]	[1 4 11 12 13]	[8 14 16]	[24 27 28]	31	15
7	[19 20 23]	[3 4 6 8 10 11]	[22 24 30]	[17 18 26 27 29 31]	[1 2 9 12 14]	[5 13 15]	[21 25 28]	32	16
8	[17 20 24]	[1 4 5 7 11 12]	[21 23 31]	[18 19 27 28 30 32]	[2 3 9 10 15]	[6 14 16]	[22 25 26]	29	13
9	[21 22 25]	[5 6 10 12 13 16]	[18 26 28]	[17 19 23 24 29 32]	[2 7 8 14 15]	[1 3 11]	[27 30 31]	20	4
10	[22 23 26]	[6 7 9 11 13 14]	[19 25 27]	[18 20 21 24 29 30]	[3 5 8 15 16]	[2 4 12]	[28 31 32]	17	1
11	[23 24 27]	[7 8 10 12 14 15]	[20 26 28]	[17 19 21 22 30 31]	[4 5 6 13 16]	[1 3 9]	[25 29 32]	18	2
12	[21 24 28]	[5 8 9 11 15 16]	[17 25 27]	[18 20 22 23 31 32]	[1 6 7 13 14]	[2 4 10]	[26 29 30]	19	3
13	[25 26 29]	[2 3 9 10 14 16]	[22 30 32]	[18 19 21 23 27 28]	[1 4 6 11 12]	[5 7 15]	[17 20 31]	24	8
14	[26 27 30]	[3 4 10 11 13 15]	[23 29 31]	[19 20 22 24 25 28]	[1 2 7 9 12]	[6 8 16]	[17 18 32]	21	5
15	[27 28 31]	[1 4 11 12 14 16]	[24 30 32]	[17 20 21 23 25 26]	[2 3 8 9 10]	[5 7 13]	[18 19 29]	22	6
16	[25 28 32]	[1 2 9 12 13 15]	[21 29 31]	[17 18 22 24 26 27]	[3 4 5 10 11]	[6 8 14]	[19 20 30]	23	7
17	[1 5 8]	[18 20 21 24 31 32]	[2 4 12]	[6 7 9 11 15 16]	[22 23 28 29...	[19 25 27]	[3 13 14]	10	26
18	[2 5 6]	[17 19 21 22 29 32]	[1 3 9]	[7 8 10 12 13 16]	[23 24 25 30...	[20 26 28]	[4 14 15]	11	27
19	[3 6 7]	[18 20 22 23 29 30]	[2 4 10]	[5 8 9 11 13 14]	[21 24 26 31...	[17 25 27]	[1 15 16]	12	28
20	[4 7 8]	[17 19 23 24 30 31]	[1 3 11]	[5 6 10 12 14 15]	[21 22 27 29...	[18 26 28]	[2 13 16]	9	25
21	[5 9 12]	[17 18 22 24 25 28]	[6 8 16]	[1 2 10 11 13 15]	[19 20 26 27...	[23 29 31]	[3 4 7]	14	30

Fig. 24.7. Structure `lattice` of graphene lattice

List of literature and Internet resources

1. S. Attaway. MATLAB. A Practical Introduction to Programming and Problem Solving. Butterworth-Heinemann, USA. 2013. 538 p.

2. Dan Green. One Hundred Physics Visualizations with Matlab. – World Scientific Publishing Company, 2013. – 200 p.

3. Mengliu Zhao, 2014.

http://www.mathworks.com/matlabcentral/fileexchange/41613-plot-spherical-harmonics/content/spharmPlot.m

4. S.V. Porshnev. Computer modeling of physical processes in MatLab (in Russian). Lan', Saint-Petersburg. 2011. 736 p.

5. R. Crownover. Introduction to Fractals and Chaos. Jones and Bartlett Publishers, London. 1999. 306 p.

A.V. Krasavin, Ya.V. Zhumagulov

Practical course on MatLab for foreign students

Корректор М.В. Макарова

Подписано в печать 01.02.2018. Формат 60x84 1/16
Уч.-изд.л. 16,75. П.л. 16,75.
Изд. № 003-1

Национальный исследовательский ядерный университет «МИФИ».
115409, г. Москва, Каширское ш., 31.