Основы программирования в MatLab

Наместников С.М. / Сборник лекций: УлГТУ, Ульяновск. - 2011

Оглавление

Введение
Глава 1. Структура программы. Основные математические операции и типы данных
1.1. Структура программы пакета MatLab
1.2. Простые переменные и основные типы данных в MatLab
1.3. Арифметические операции с простыми переменными
1.4. Основные математические функции MatLab
1.5. Векторы и матрицы в MatLab
1.6. Операции над матрицами и векторами
1.7. Структуры в MatLab
1.8. Ячейки в MatLab
Глава 2. Условные операторы и циклы в MatLab
2.1. Условный оператор if
2.2. Условный оператор switch
2.3. Оператор цикла while
2.4. Оператор цикла for
Глава 3. Работа с графиками в MatLab
3.1. Функция plot
3.2. Оформление графиков
3.3. Отображение трехмерных графиков
3.4. Отображение растровых изображений
Глава 4. Программирование функций в MatLab
4.1. Порядок определения и вызова функций
4.2. Область видимости переменных
Глава 5. Работа с файлами в MatLab
5.1. Функции save и load
5.2. Функции fwrite и fread
5.3. Функции fscanf и fprintf
5.4. Функции imread и imwrite

Введение

Среди множества существующих математических пакетов, таких как Mathematica, MathCad и др., система MatLab занимает лидирующее место благодаря удобному встроенному языку программирования для реализации самых разнообразных математических алгоритмов и задач математического моделирования. Кроме того, данный пакет имеет дополнительно инструмент визуального моделирования Simulink, позволяющий строить и исследовать математические модели, не прибегая к их программированию.

В данном учебном пособии рассматривается внутренний язык программирования MatLab, дающий наибольшую гибкость, богатство функционала и удобство при решении и исследовании математических задач. При изложении материала предпочтение отдавалось наиболее простым конструкциям языка, изучая которые можно создавать самые разнообразные и нетривиальные математические алгоритмы.

Глава 1. Структура программы. Основные математические операции и типы данных

Первым шагом на пути создания математических алгоритмов является изучение структуры программы и набора математических операций, доступных языку программирования. В частности, в данной главе будут рассмотрены математические операции и функции пакета MatLab, связанные с обработкой как скалярных, так и матричных переменных.

1.1. Структура программы пакета MatLab

Как правило, каждая программа в MatLab представляет собой функцию и начинается с ключевого слова function, за которым через пробел следует ее название. Например,

```
function Lab1
a = 5;
b = 2;
c = a*b;
```

Данная программа заключена в функции с именем Lab1 и вычисляет произведение двух переменных a и b. При сохранении программы в m-файл рекомендуется указывать имя файла, совпадающее с именем функции, т.е. в данном случае – Lab1.

Следует отметить, что в одном m-файле можно задавать множество дополнительных функций. Для этого достаточно написать в конце листинга основной программы еще одно ключевое слово function и задать ее имя, например,

```
function Lab1
a = 5;
b = 2;
c = a*b;
out_c(c);                  % вызов функции out_c()

function out_c(arg_c)    % определение функции out_c()
disp(arg_c);
```

Обратите внимание, что функцию out_c() можно вызывать в основной программе до ее определения. Это особенность языка MatLab, позволяющая не беспокоиться программисту о последовательности задания функций. В приведенном примере функция out_c() имеет один входной параметр с именем arg_c, который выводится на экран (в командное окно MatLab) с помощью встроенной функции disp(). В итоге, при выполнении приведенной программы в командном окне MatLab будет отображено значение переменной c.

Дополнительные функции можно оформлять и в отдельных m-файлах. Например, если есть необходимость какую-либо функцию описать в одном m-файле, а вызывать ее в другом, то это можно реализовать следующим образом.

1-й файл (Lab1.m)

```
function Lab1
a = 5;
b = 2;
c = square(a,b);         % вызов функции square()
out_c(c);                  % вызов функции out_c()

function out_c(arg_c)    % определение функции out_c()
disp(arg_c);
```

2-й файл (square.m)

```
function res=square(a, b)
res = a*b;
```

При выполнении функции Lab1 система MatLab вызовет функцию square из файла square.m. Это будет сделано автоматически, т.к. встроенные функции языка MatLab определены также и вызываются из файлов, имена которых, как правило, соответствуют именам вызываемых функций. Обратите также внимание на то, что функция square() не только принимает два аргумента a и b, но и возвращает их произведение с помощью переменной res. Представленный синтаксис следует использовать всякий раз, когда требуется возвратить результат вычислений основной программе. В четвертой главе данного пособия более подробно изложены конструкции вызова функций для реализации разнообразных алгоритмов.

1.2. Простые переменные и основные типы данных в MatLab

Создание программы, как правило, начинается с определения переменных и способа представления данных. Следовательно, чтобы правильно организовать описание данных программы, необходимо знать как задавать переменные в MatLab и какие виды переменных возможны.

Самый простой и наиболее распространенный тип данных – это число. В MatLab число хранится в переменной, которая имеет некоторое уникальное имя, например,

```
a = 5;
```

задает переменную с именем a и присваивает ей значение 5. По умолчанию переменная a является вещественной (тип double), т.е. может принимать дробные значения, например,

```
a = -7.8;
```

задает значение переменной a равное -7,8. Изменить тип переменной можно, указав тип присваиваемого числа с помощью соответствующего ключевого слова, например,

```
a = int16(5);
```

выполнит присваивание числа 5 как целочисленного 16-битового значения. В результате выполнения такой операции тип переменной a будет соответствовать int16.

Типы данных, доступные в MatLab, представлены в табл. 1.1.

Таблица 1.1. Основные типы данных в MatLab

double	вещественный, 64 бит
single	вещественный, 32 бит
int8	знаковый целочисленный, 8 бит
int16	знаковый целочисленный, 16 бит
int32	знаковый целочисленный, 32 бит
int64	знаковый целочисленный, 64 бит
uint8	беззнаковый целочисленный, 8 бит
uint16	беззнаковый целочисленный, 16 бит
uint32	беззнаковый целочисленный, 32 бит
uint64	беззнаковый целочисленный, 64 бит

По умолчанию используется тип double, который имеет наибольшую точность представления вещественного числа и является потому универсальным типом. Однако, если необходимо экономить память ЭВМ, то можно указывать самостоятельно желаемый тип.

Последнее, что следует знать при задании переменных – это правило определения их имен. В MatLab имена переменных могут задаваться только латинскими буквами, цифрами и символом '_'. Причем, первый символ в имени должен соответствовать букве латинского алфавита. Также следует отметить, что имена

```
arg = 1;
Arg = 2;
ARG = 3;
```

это три разных имени, т.е. три разные переменные со значениями 1, 2 и 3 соответственно. Данный пример показывает, что MatLab различает регистр в именах переменных.

При программировании лучше всего задавать осмысленные имена переменных, по которым можно было бы понять какие данные они представляют. Это позволяет избежать путаницы при построении больших программ.

1.3. Арифметические операции с простыми переменными

Рассмотрим базовые арифметические операции над простыми переменными в MatLab. Пусть заданы две переменные a и b. Тогда операции сложения, вычитания, умножения и деления запишутся так:

```
c = a+b;  % сложение
c = a-b;  % вычитание
c = a*b;  % умножение
c = a/b;  % деление
```

Дополнительно, в MatLab определена операция возведения в степень, которая записывается так:

```
c = a^2;  % возведение переменной a в квадрат
c = a^0.5; % извлечение квадратного корня из переменной a
```

Приоритет арифметических операций * и / выше операций + и - , т.е. сначала выполняется умножение и деление, а затем, сложение и вычитание. Операция возведения в степень имеет наивысший приоритет. Для изменения приоритетов арифметических операций используются круглые скобки, как и в обычной математике, например,

```
c = 7+2*2;      % значение c = 28
c = (7+2)*2;    % значение c = 18
```

В практике программирования есть несколько устоявшихся подходов использования арифметических операций. Например, если необходимо увеличить значение переменной arg на 1, то этого можно добиться следующим образом:

```
arg = arg + 1;
```

При этом совершенно не важно чему равна переменная arg, в любом случае ее значение будет увеличено на 1. Аналогичным образом можно выполнять увеличение или уменьшение значения переменной на любую величину.

Другим устоявшимся приемом программирования является обмен значений между двумя переменными. Например, заданы переменные arg_a и arg_b и необходимо произвести обмен данными между ними. Это достигается следующим образом:

```
temp = arg_a;
arg_a = arg_b;
arg_b = temp;
```

Здесь temp – это временная переменная, необходимая для сохранения значения переменной arg_a, т.к. оно затирается во второй строчке значением arg_b. Поэтому в третьей строке переменной arg_b присваивается сохраненное в temp значение переменной arg_a.

Если в результате выполнения вычислений появляется **комплексное число**, то MatLab автоматически будет оперировать с такими числами в соответствии с арифметикой комплексных чисел. Например, при извлечении квадратного корня из -1, получим следующий результат:

```
c = (-1)^0.5;        % c = 0.0000 + 1.0000i
```

Здесь i – зарезервированное имя мнимой единицы и представленная запись обозначает комплексное число с нулевой действительной частью и единичной мнимой. В MatLab в качестве зарезервированного имени мнимой единицы также используется буква j.

Для того, чтобы задать комплексную переменную достаточно указать значения их действительной и мнимой частей как показано ниже

```
c = 6 + 5i;          % комплексное число
```

и с заданными комплексными переменными также можно выполнять описанные выше арифметические операции.

При работе с комплексными числами существуют две специальные функции:

real(x) – взятие действительной части комплексного числа x;
imag(x) – взятие мнимой части комплексного числа x;
abs(x) – вычисление абсолютного значения комплексного числа x;
conj(x) – вычисление комплексно-сопряженного числа x;
angle(x) – вычисление аргумента комплексного числа x;

1.4. Основные математические функции MatLab

MatLab содержит в себе все распространенные математические функции, которые доступны по их имени при реализации алгоритмов. Например, функция sqrt() позволяет вычислять квадрат числа и может быть использована в программе следующим образом:

```
x = 2;
y = 4;
d = sqrt(x^2+y^2);  %вычисление евклидового расстояния
```

Аналогичным образом вызываются и все другие математические функции, представленные в табл. 1.2.

Таблица 1.2. Основные математические функции MatLab

sqrt(x)	вычисление квадратного корня
exp(x)	возведение в степень числа е
pow2(x)	возведение в степень числа 2
log(x)	вычисление натурального логарифма
log10(x)	вычисление десятичного логарифма
log2(x)	вычисление логарифма по основанию 2
sin(x)	синус угла x, заданного в радианах
cos(x)	косинус угла x, заданного в радианах
tan(x)	тангенс угла x, заданного в радианах
cot(x)	котангенс угла x, заданного в радианах
asin(x)	арксинус
acos(x)	арккосинус
atan(x)	арктангенс
pi	число пи
round(x)	округление до ближайшего целого

fix(x)	усечение дробной части числа
floor(x)	округление до меньшего целого
ceil(x)	округление до большего целого
mod(x)	остаток от деления с учётом знака
sign(x)	знак числа
factor(x)	разложение числа на простые множители
isprime(x)	истинно, если число простое
rand	генерация псевдослучайного числа с равномерным законом распределения
randn	генерация псевдослучайного числа с нормальным законом распределения
abs(x)	вычисление модуля числа

Почти все элементарные функции допускают вычисления и с комплексными аргументами. Например:

```
res = sin(2+3i)*atan(4i)/(1 — 6i);     % res = -1.8009 — 1.9190i
```

или

```
exp(i*x) = cos(x)+i*sin(x);
```

Ниже показан пример задания вектора с именем а, и содержащий значения 1, 2, 3, 4:

```
a = [1 2 3 4];      % вектор-строка
```

Для доступа к тому или иному элементу вектора используется следующая конструкция языка:

```
disp( a(1) );  % отображение значения 1-го элемента вектора
disp( a(2) );  % отображение значения 2-го элемента вектора
disp( a(3) );  % отображение значения 3-го элемента вектора
disp( a(4) );  % отображение значения 4-го элемента вектора
```

т.е. нужно указать имя вектора и в круглых скобках написать номер индекса элемента, с которым предполагается работать. Например, для изменения значения 2-го элемента массива на 10 достаточно записать

```
a(2) = 10; % изменение значения 2-го элемента на 10
```

Часто возникает необходимость определения общего числа элементов в векторе, т.е. определения его размера. Это можно сделать, воспользовавшись функцией length() следующим образом:

```
N = length(a); % (N=4) число элементов массива a
```

Если требуется задать вектор-столбец, то это можно сделать так

```
a = [1; 2; 3; 4];   % вектор-столбец
```

или так

```
b = [1 2 3 4]’;         % вектор-столбец
```

при этом доступ к элементам векторов осуществляется также как и для векторов-строк.

Следует отметить, что векторы можно составлять не только из отдельных чисел или переменных, но и из векторов. Например, следующий фрагмент программы показывает, как можно создавать один вектор на основе другого:

```
a = [1 2 3 4];      % начальный вектор a = [1 2 3 4]
b = [a 5 6];        % второй вектор b = [1 2 3 4 5 6]
```

Здесь вектор b состоит из шести элементов и создан на основе вектора а. Используя этот прием, можно осуществлять увеличение размера векторов в процессе работы программы:

```
a = [a 5];          % увеличение вектора a на один элемент
```

Недостатком описанного способа задания (инициализации) векторов является сложность определения векторов больших размеров, состоящих, например, из 100 или 1000 элементов. Чтобы решить данную задачу, в MatLab существуют функции инициализации векторов нулями, единицами или случайными значениями:

```
a1 = zeros(1, 100);     % вектор-строка, 100 элементов с
% нулевыми значениями
a2 = zeros(100, 1);     % вектор-столбец, 100 элементов с
% нулевыми значениями
a3 = ones(1, 1000);     % вектор-строка, 1000 элементов с
% единичными значениями
a4 = ones(1000, 1);     % вектор-столбец, 1000 элементов с
% единичными значениями
a5 = rand(1000, 1);     % вектор-столбец, 1000 элементов со
% случайными значениями
```

Матрицы в MatLab задаются аналогично векторам с той лишь разницей, что указываются обе размерности. Приведем пример инициализации единичной матрицы размером 3х3:

```
E = [1 0 0; 0 1 0; 0 01];         % единичная матрица 3х3
```

или

```
E = [1 0 0
    0 1 0
     0 0 1];               % единичная матрица 3х3
```

Аналогичным образом можно задавать любые другие матрицы, а также использовать приведенные выше функции zeros(), ones() и rand(), например:

```
A1 = zeros(10,10);      % нулевая матрица 10х10 элементов
```

или

```
A2 = zeros(10);              % нулевая матрица 10х10 элементов
A3 = ones(5);           % матрица 5х5, состоящая из единиц
A4 = rand(100);              % матрица 100х100, из случайных чисел
```

Для доступа к элементам матрицы применяется такой же синтаксис как и для векторов, но с указанием строки и столбца где находится требуемый элемент:

```
A = [1 2 3;4 5 6;7 8 9]; % матрица 3х3
disp( A(2,1) );         % вывод на экран элемента, стоящего во
                    % второй строке первого столбца, т.е. 4
disp( A(1,2) );         % вывод на экран элемента, стоящего в
                    % первой строке второго столбца, т.е. 2
```

Также возможны операции выделения указанной части матрицы, например:

```
B1 = A(:,1);   % B1 = [1; 4; 7] – выделение первого столбца
B2 = A(2,:);   % B2 = [1 2 3] – выделение первой строки
B3 = A(1:2,2:3);    % B3 = [2 3; 5 6] – выделение первых двух
% строк и 2-го и 3-го столбцов матрицы А.
```

Размерность любой матрицы или вектора в MatLab можно определить с помощью функции size(), которая возвращает число строк и столбцов переменной, указанной в качестве аргумента:

```
a = 5;              % переменная a
A = [1 2 3];        % вектор-строка
B = [1 2 3; 4 5 6]; % матрица 2х3
size(a)             % 1х1
size(A)             % 1х3
size(B)             % 2х3
```

1.6. Операции над матрицами и векторами

В системе MatLab достаточно просто выполняются математические операции над матрицами и векторами. Рассмотрим сначала простые операции сложения и умножения матриц и векторов. Пусть даны два вектора

```
a = [1 2 3 4 5];         % вектор-строка
b = [1; 1; 1; 1; 1];          % вектор-столбец
```

тогда умножение этих двух векторов можно записать так

```
c = a*b;                % c=1+2+3+4+5=16
d = b*a;                % d – матрица 5х5 элементов
```

В соответствии с операциями над векторами, умножение вектор-строки на вектор-столбец дает число, а умножение вектор-столбца на вектор-строку дает двумерную матрицу, что и является результатом вычислений в приведенном примере, т.е.

$$c = \sum_{i=1}^{5} a_i b_i = 1+2+3+4+5+6 = 16,$$

$$d = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 3 & 4 & 5 \end{bmatrix}.$$

Сложение и вычитание двух векторов записывается так

```
a1 = [1 2 3 4 5];
a2 = [5 4 3 2 1];
c = a1+a2;               % c = [1+5, 2+4, 3+3, 4+2, 5+1];
c = a2-a1;              % c = [5-1, 4-2, 3-3, 2-4, 1-5];
```

Следует обратить внимание, что операции сложения и вычитания можно выполнять между двумя векторами-столбцами или двумя векторами-строками. Иначе MatLab выдаст сообщение об ошибке, т.к. разнотипные векторы складывать нельзя. Так обстоит дело со всеми недопустимыми арифметическими операциями: в случае невозможности их вычисления система MatLab сообщит об ошибке и выполнение программы будет завершено на соответствующей строке.

Аналогичным образом выполняются операции умножения и сложения между матрицами:

```
A = [1 2 3; 4 5 6; 7 8 9];
B = ones(3);
C = A+B;            % сложение двух матриц одинакового размера
D = A+5;            % сложение матрицы и числа
E = A*B;            % умножение матрицы А на В
F = B*A;            % умножение матрицы В на А
G = 5*A;            % умножение матрицы на число
```

Операции вычисления обратной матрицы, а также транспонирования матриц и векторов, записываются следующим образом:

```
a = [1 1 1];        % вектор-строка
b = a’;             % вектор-столбец, образованный
% транспонированием вектора-строки a.
A = [1 2 3; 4 5 6; 7 8 9]; % матрица 3х3 элемента
B = a*A;            % B = [12 15 18] – вектор-строка
C = A*b;            % C = [6; 15; 24] – вектор-столбец
D = a*A*a’;         % D = 45 – число, сумма эл-ов матрицы А
E = A’;             % E – транспонированная матрица А
F = inv(A);         % F – обратная матрица А
G = A^-1;           % G – обратная матрица А
```

Из приведенного примера видно, что операция транспонирования матриц и векторов обозначается символом ‘ (апостроф), который ставится после имени вектора или матрицы. Вычисление обратной матрицы можно делать путем вызова функции inv() или возводя матрицу в степень -1. Результат в обоих случаях будет одинаковым, а два способа вычисления сделано для удобства использования при реализации различных алгоритмов.

Если в процессе вычислений требуется поэлементно умножить, разделить или возвести в степень элементы вектора или матрицы, то для этого используются операторы:

.* - поэлементное умножение;
./ и .\ - поэлементные деления;
.^ - поэлементное возведение в степень.

Рассмотрим работу данных операторов на следующем примере.

```
a = [1 2 3];         % вектор-строка
b = [3 2 1];         % вектор-строка
c = a.*b;            % c = [3 4 3]
A = ones(3);         % матрица 3x3, состоящая из единиц
B = [1 2 3;4 5 6; 7 8 9];         % матрица 3x3
C = A.*B;            % матрица 3x3, состоящая из $\|c_{ij} = a_{ij} b_{ij}\|_{i,j=\overline{1,3}}$
D = A./B;            % матрица 3x3, состоящая из $\|d_{ij} = a_{ij} / b_{ij}\|_{i,j=\overline{1,3}}$
E = A.\B;            % матрица 3x3, состоящая из $\|e_{ij} = b_{ij} / a_{ij}\|_{i,j=\overline{1,3}}$
F = A.^2;            % возведение элементов матрицы A в квадрат
```

В заключении данного параграфа рассмотрим несколько функций полезных при работе с векторами и матрицами.

Для поиска максимального значения элемента вектора используется стандартная функция max(), которая возвращает найденное максимальное значение элемента и его позицию (индекс):

```
a = [1 6 3 4];
[v, i] = max(a);                 % v = 6, i = 2;
```

или

```
v = max(a);                      % v = 6;
```

Приведенный пример показывает два разных способа вызова функции max(). В первом случае определяется и максимальное значение элемента и его индекс в векторе, а во втором – только максимальное значение элемента.

В случае с матрицами, данная функция определяет максимальные значения, стоящие в столбцах, как показано ниже в примере:

```
A = [4 3 5; 6 7 2; 3 1 8];
[V, I] = max(A);                 % V=[6 7 8], I = [2 2 3]
V = max(A);                      % V=[6 7 8]
```

Полный синтаксис функции max() можно узнать, набрав в командном окне MatLab команду

```
help <название функции>
```

Аналогичным образом работает функция min(), которая определяет минимальное значение элемента вектора или матрицы и его индекс.

Другой полезной функцией работы с матрицами и векторами является функция sum(), которая вычисляет сумму значений элементов вектора или столбцов матрицы:

```
a = [3 5 4 2 1];
s = sum(a);                       % s = 3+5+4+2+1=15
A = [4 3 5; 6 7 2; 3 1 8];
S1 = sum(A);                      % S1=[13 11 15]
S2 = sum(sum(A));                % S2=39
```

При вычислении суммы S2 сначала вычисляется сумма значений элементов матрицы A по столбцам, а затем, по строкам. В результате, переменная S2 содержит сумму значений всех элементов матрицы A.

Для сортировки значений элементов вектора или матрицы по возрастанию или убыванию используется функция sort() следующим образом:

```
a = [3 5 4 2 1];

b1 = sort(a);                 % b1=[1 2 3 4 5]
b2 = sort(a, 'descend');       % b2=[5 4 3 2 1]
b3 = sort(a, 'ascend');       % b3=[1 2 3 4 5]
```

для матриц

```
A = [4 3 5; 6 7 2; 3 1 8];
B1 = sort(A);                 % B1=[3 1 2
                             %     4 3 5
                              %     6 7 8]
B2 = sort(A, 'descend');      % B2=[6 7 8
                              %     4 3 5
                              %     3 1 2]
```

Во многих практических задачах часто требуется найти определенный элемент в векторе или матрице. Это можно выполнить с помощью стандартной функции find(), которая в качестве аргумента принимает условие, в соответствии с которым и находятся требуемые элементы, например:

```
a = [3 5 4 2 1];
b1 = find(a == 2);            % b1 = 4 – индекс элемента 2
b2 = find(a ~= 2);            % b2 = [1 2 3 5] – индексы без 2
b3 = find(a > 3);             % b3 = [2 3]
```

В приведенном примере символ '==' означает проверку на равенство, а символ '~=' выполняет проверку на неравенство значений элементов вектора a. Более подробно об этих операторах будет описано в разделе условные операторы.

Еще одной полезной функцией работы с векторами и матрицами является функция mean() для вычисления среднего арифметического значения, которая работает следующим образом:

```
a = [3 5 4 2 1];
m = mean(a);                  % m = 3
A = [4 3 5; 6 7 2; 3 1 8];
M1 = mean(A);                 % M1 = [4.333 3.667 5.000]
M2 = mean(mean(A));           % M2 = 4.333
```

1.7. Структуры в MatLab

При разработке программ важным является выбор эффективного способа представления данных. Во многих случаях недостаточно объявить простую переменную или массив, а нужна более гибкая форма представления данных. Таким элементом может быть структура, которая позволяет включать в себя разные типы данных и даже другие структуры. Структуры задаются следующим образом:

```
S = struct('field1',VALUES1,'field2',VALUES2,...);
```

где field1 – название первого поля структуры; VALUES1 – переменная первого поля структуры, и т.д.

Приведем пример, в котором использование структуры позволяет эффективно представить данные. Таким примером будет инвентарный перечень книг, в котором для каждой книги необходимо указывать ее наименование, автора и год издания. Причем количество книг может быть разным, но будем полгать, что не более 100. Для хранения информации об одной книге будем использовать структуру, которая задается в MatLab с помощью ключевого слова struct следующим образом:

```
S = struct('title','','author','','year',0);
```

В итоге задается структура с тремя полями: title, author и year. Каждое поле имеет свой тип данных и значение.

Для того, чтобы записать в эту структуру конкретные значения используется оператор '.' (точка) для доступа к тому или иному полю структуры:

```
S.title = 'Евгений Онегин';
S.author = 'Пушкин';
S.year = 2000;
```

и таким образом, переменная S хранит информацию о выбранной книге.

Однако по условиям задачи необходимо осуществлять запись не по одной, а по 100 книгам. В этом случае целесообразно использовать вектор структур lib, который можно задать следующим образом:

```
lib(100,1) = struct('title','','author','','year',0);
```

и записывать информацию о книгах так:

```
lib(1).title = 'Евгений Онегин';
lib(1).author = 'Пушкин';
lib(1).year = 2000;
```

Данный пример показывает удобство хранения информации по книгам. Графически массив структур можно представить в виде таблицы, в которой роль столбцов играют поля, а роль строк элементы массива структур (рис. 1.1).

При работе со структурами полезными являются следующие функции:

```
isstruct( S ) – возвращает истину, если аргумент структура
isfield( S, 'name') – возвращает истину, если имеется такое поле
fieldnames( S ) – возвращает массив строк с именами всех полей
```

которые позволяют программно определить всю необходимую информацию о той или иной структуре и корректно выполнять обработку ее полей.

	название	автор	год издания
lib[1]	lib[1].title	lib[1].author	lib[1].year
lib[2]	lib[2].title	lib[2].author	lib[2].year
lib[3]	lib[3].title	lib[3].author	lib[3].year
⋮			
lib[100]	lib[100].title	lib[100].author	lib[100].year

Рис. 1.1. Графическое представление массива структур хранения информации по 100 книгам

1.8. Ячейки в MatLab

Ячейки также как и структуры могут содержать разные типы данных, объединенные одной переменной, но в отличии от вектора структур, вектор ячеек может менять тип данных в каждом элементе. Таким образом, вектор ячеек является универсальным контейнером – его элементы могут содержать любые типы и структуры данных, с которыми работает MatLab – векторы чисел любой размерности, строки, векторы структур и другие (вложенные) векторы ячеек.

Методы создания вектора ячеек похожи на методы создания вектора структур. Как и в случае структур, векторы ячеек могут быть созданы либо путём последовательного присваивания значений отдельным элементам массива, либо созданы целиком при помощи специальной функции cell(). Однако в любом случае важно различать ячейку (элемент вектора ячеек) и её содержимое. Ячейка – это содержимое плюс некоторая оболочка (служебная структура данных) вокруг этого содержимого, позволяющая хранить в ячейке произвольные типы данных любого размера.

Приведем пример создания вектора ячеек хранения разных типов данных.

```
book = struct('title','Онегин','author','Пушкин','year',2000);
MyCell(1)={book};
MyCell(2)={'Пушкин'};
MyCell(3)={2000};
```

Здесь задан вектор ячеек MyCell с тремя элементами. Первый элемент соответствует структуре, второй – строке, а третий – числу. В этом и заключается особенность организации данных с помощью ячеек: у каждого элемента свой тип данных.

Для обращения к содержимому той или иной ячейки используются фигурные скобки, внутри которых ставится индекс элемента с которым предполагается работа:

```
MyCell{1}
```

выведет на экран

```
     title: 'Евгений Онегин'
    author: 'Пушкин'
      year: 2000
```

Если же используются круглые скобки, то будет возвращена структура данных вместо отдельных значений, например

```
MyCell(1)
```

выведет

```
[1x1 struct]
```

Для того чтобы задать вектор или матрицу ячеек с пустыми (неопределенными) значениями, используется функция cell() как показано ниже.

```
MyCellArray = cell(2, 2);
```

задается матрица размером 2x2. Данную инициализацию целесообразно выполнять когда нужно определить большой вектор или матрицу ячеек и в цикле задавать их значения. В этом случае MatLab сразу создает массивы нужных размеров, в результате чего повышается скорость выполнения программ.

В заключении рассмотрим возможность программирования функции с произвольным числом аргументов благодаря использованию ячеек. Для этого в качестве аргумента функции указывается ключевое слово varargin, которое интерпретируется внутри функции как вектор ячеек с переданными аргументами:

```
function len = SumSquare( varargin )
n= length( varargin );
len = 0;
for k = 1 : n
         len = len + varargin{ k }(1)^2 +varargin{ k }(2)^2;
end
```

Данная функция вычисляет сумму квадратов чисел, которые передаются ей следующим образом:

```
SumSquare( [ 1 2], [3 4] )             % ответ 30
SumSquare( [ 1 2], [3 4], [ 5 6] )         % ответ 91
```

Таким образом, благодаря использованию ячеек функции SumSquare() можно передавать произвольное число двумерных векторов.

Глава 2. Условные операторы и циклы в MatLab

Вторым шагом создания полноценных программ на языке MatLab является изучение операторов ветвления и циклов. С их помощью можно реализовывать логику выполнения математических алгоритмов и создавать повторяющиеся (итерационные, рекуррентные) вычисления.

2.1. Условный оператор if

Для того чтобы иметь возможность реализовать логику в программе используются условные операторы. Умозрительно эти операторы можно представить в виде узловых пунктов, достигая которых программа делает выбор по какому из возможных направлений двигаться дальше. Например, требуется определить, содержит ли некоторая переменная arg положительное или отрицательное число и вывести соответствующее сообщение на экран. Для этого можно воспользоваться оператором if (если), который и выполняет подобные проверки.

В самом простом случае синтаксис данного оператора if имеет вид:

```
if <выражение>
<операторы>
end
```

Если значение параметра «выражение» соответствует значению «истинно», то выполняется оператор, иначе он пропускается программой. Следует отметить, что «выражение» является условным выражением, в котором выполняется проверка некоторого условия. В табл. 2.1 представлены варианты простых логических выражений оператора if.

Таблица 2.1. Простые логические выражения

if a < b	Истинно, если переменная a меньше переменной b и ложно в противном случае.
if a > b	Истинно, если переменная a больше переменной b и ложно в противном случае.
if a == b	Истинно, если переменная a равна переменной b и ложно в противном случае.
if a <= b	Истинно, если переменная a меньше либо равна переменной b и ложно в противном случае.
if a >= b	Истинно, если переменная a больше либо равна переменной b и ложно в противном случае.
if a ~= b	Истинно, если переменная a не равна переменной b и ложно в противном случае.

Ниже представлен пример реализации функции sign(), которая возвращает +1, если число больше нуля, -1 – если число меньше нуля и 0, если число равно нулю:

```
function my_sign
x = 5;
if x > 0
disp(1);
end
if x < 0
disp(-1);
end
if x == 0
disp(0);
end
```

Анализ приведенного примера показывает, что все эти три условия являются взаимоисключающими, т.е. при срабатывании одного из них нет необходимости проверять другие. Реализация именно такой логики позволит увеличить скорость выполнения программы. Этого можно добиться путем использования конструкции

```
if <выражение>
<операторы1>        % выполняются, если истинно условие
else
<операторы2>        % выполняются, если условие ложно
end
```

Тогда приведенный выше пример можно записать следующим образом:

```
function my_sign
x = 5;
if x > 0
disp(1);
else
if x < 0
disp(-1);
else
disp(0);
end
end
```

В данной программе сначала выполняется проверка на положительность переменной x, и если это так, то на экран выводится значение 1, а все другие условия игнорируются. Если же первое условие оказалось ложным, то выполнение программы переходит по else (иначе) на второе условие, где выполняется проверка переменной x на отрицательность, и в случае истинности условия, на экран выводится значение -1. Если оба условия оказались ложными, то выводится значение 0.

Приведенный выше пример можно записать в более простой форме, используя еще одну конструкцию оператора if языка MatLab:

```
if <выражение1>
<операторы1>        % выполняются, если истинно выражение1
```

```
elseif <выражение2>
<операторы2>        % выполняются, если истинно выражение2
...
elseif <выражениеN>
<операторыN>        % выполняются, если истинно выражениеN
end
```

и записывается следующим образом:

```
function my_sign
x = 5;
if x > 0
     disp(1);        % выполняется, если x > 0
elseif x < 0
     disp(-1);       % выполняется, если x < 0
else
     disp(0);        % выполняется, если x = 0
end
```

С помощью условного оператора if можно выполнять проверку более сложных (составных) условий. Например, необходимо определить: попадает ли переменная x в диапазон значений от 0 до 2? Это можно реализовать одновременной проверкой сразу двух условий: x >= 0 и x <=2. Если эти оба условия истинны, то x попадает в диапазон от 0 до 2.

Для реализации составных условий в MatLab используются логические операторы:

& - логическое И
| - логическое ИЛИ
~ - логическое НЕ

Рассмотрим пример использования составных условий. Пусть требуется проверить попадание переменной x в диапазон от 0 до 2. Программа запишется следующим образом:

```
function my_if
x = 1;
if x >= 0 & x <= 2
   disp('x принадлежит диапазону от 0 до 2');
else
   disp('x  не принадлежит диапазону от 0 до 2');
end
```

Во втором примере выполним проверку на не принадлежность переменной x диапазону от 0 до 2. Это достигается срабатыванием одного из двух условий: x < 0 или x > 2:

```
function my_if
x = 1;
if x < 0 | x > 2
   disp('x не принадлежит диапазону от 0 до 2');
else
   disp('x  принадлежит диапазону от 0 до 2');
end
```

Используя логические операторы И, ИЛИ, НЕ, можно создавать разнообразные составные условия. Например, можно сделать проверку, что переменная x попадает в диапазон от -5 до 5, но не принадлежит диапазону от 0 до 1. Очевидно, это можно реализовать следующим образом:

```
function my_if
x = 1;
if (x >= -5 & x <= 5) & (x < 0 | x > 1)
   disp('x принадлежит [-5, 5], но не входит в [0, 1]');
else
   disp('x или не входит в [-5, 5] или в [0, 1]');
end
```

Обратите внимание, что при сложном составном условии были использованы круглые скобки. Дело в том, что приоритет операции И выше приоритета операции ИЛИ, и если бы не было круглых скобок, то условие выглядело бы так: (x >= -5 и x <= 5 и x < 0) или x > 1. Очевидно, что такая проверка давала бы другой результат от ожидаемого.

Круглые скобки в программировании используются для изменения приоритетов выполнения операторов. Подобно арифметическим операторам, логические также могут быть изменены по желанию программиста. Благодаря использованию круглых скобок, сначала выполняется проверка внутри них, а, затем, за их пределами. Именно поэтому в приведенном выше примере они необходимы для достижения требуемого результата.

Приоритет логических операций следующий:

НЕ (~) – самый высокий приоритет;
И (&) – средний приоритет;
ИЛИ (|) – самый низкий приоритет.

2.2. Условный оператор switch

В некоторых задачах программирования требуется выполнять проверку на равенство некоторой переменной константным значениям. Например, нужно преобразовать малые буквы в заглавные. В этом случае необходимо произвести проверку текущего символа со всеми возможными буквами алфавита и при равенстве с одной из них, заменить ее на заглавную. Для решения таких задач удобнее пользоваться условным оператором switch, который имеет следующий синтаксис:

```
switch expr
          case case_expr,
            <операторы1>
          case {case_expr1, case_expr2, case_expr3,...}
            <операторы2>
         ...
          otherwise,
            <операторы>
end
```

Здесь expr – переменная, значение которой проверяется на равенство тем или иным константам; case_expr – константы, с которым сравнивается значение переменной; otherwise – ключевое слово, для выполнения операторов, при всех ложных условиях.

Приведем пример работы данного оператора для преобразования малых букв латинского алфавита в заглавные.

```
function upper_symbol
ch='c';
switch ch
    case 'a', ch='A';
    case 'b', ch='B';
    case 'c', ch='C';
    case 'd', ch='D';
    case 'e', ch='E';
   ...
    case 'z', ch='Z';
end

disp(ch);
```

В данной программе задается символьная переменная ch со значением c. Затем, с помощью оператора switch проверяется ее значение со всеми возможными малыми буквами латинского алфавита от a до z. Как только одно из условий сработало, оператор switch завершает свою работу и выполнение программы переходит на функцию disp(), которая отображает значение переменной ch на экран.

2.3. Оператор цикла while

Язык программирования MatLab имеет два оператора цикла: while и for. С их помощью, например, выполняется программирование рекуррентных алгоритмов, подсчета суммы ряда, перебора элементов массива и многое другое.

В самом простом случае цикл в программе организуется с помощью оператора while, который имеет следующий синтаксис:

```
while <условие>
      <операторы>
end
```

Здесь <условие> означает условное выражение подобное тому, которое применяется в операторе if, и цикл while работает до тех пор, пока это условие истинно.

Следует обратить внимание на то, что если условие будет ложным до начала выполнения цикла, то операторы, входящие в цикл, не будут выполнены ни разу.

Приведем пример работы цикла while для подсчета суммы ряда $S=\sum_{i=1}^{20} i$:

```
function sum_i
S = 0;                   % начальное значение суммы
i=1;                     % счетчик суммы
while i <= 20            % цикл (работает пока i <= 20)
     S=S+i;              % подсчитывается сумма
     i=i+1;              % увеличивается счетчик на 1
end                      % конец цикла
disp(S);                 % отображение суммы 210 на экране
```

Теперь усложним задачу и будем подсчитывать сумму ряда $S=\sum_{i=1}^{20} i$, пока $S \le 20$. Здесь в операторе цикла получается два условия: либо счетчик по i доходит до 20, либо значение суммы S превысит 20. Данную логику можно реализовать с помощью составного условного выражения в операторе цикла while:

```
function sum_i
S = 0;                   % начальное значение суммы
i=1;                      % счетчик суммы
while i <= 20 & S <= 20 % цикл (работает пока i<=10 и S<=20
     S=S+i;              % подсчитывается сумма
     i=i+1;              % увеличивается счетчик на 1
end                      % конец цикла
disp(S);                 % отображение суммы 21 на экране
```

Приведенный пример показывает возможность использования составных условий в цикле while. В общем случае в качестве условного выражения можно записывать такие же условия, что и в условном операторе if.

Работу любого оператора цикла, в том числе и while, можно принудительно завершить с помощью оператора break. Например, предыдущую программу можно переписать следующим образом с использованием оператора break:

```
function sum_i
S = 0;                   % начальное значение суммы
i=1;                      % счетчик суммы
while i <= 20             % цикл (работает пока i<=10
     S=S+i;              % подсчитывается сумма
     i=i+1;              % увеличивается счетчик на 1
    if S > 20               % если S > 20,
        break;               % то цикл завершается
     end
end                      % конец цикла
disp(S);                 % отображение суммы 21 на экране
```

В данном примере второе условие завершения цикла, когда S будет больше 20, записано в самом цикле и с помощью оператора break осуществляется выход из цикла на функцию disp(), стоящую сразу после цикла while.

Второй оператор управления выполнением цикла continue позволяет пропускать выполнение фрагмента программы, стоящий после него. Например, требуется подсчитать сумму элементов массива

a = [1 2 3 4 5 6 7 8 9];

исключая элемент с индексом 5. Такую программу можно записать следующим образом:

```
function sum_array
S = 0;                   % начальное значение суммы
a = [1 2 3 4 5 6 7 8 9]; % массив
i=0;                      % счетчик индексов массива
while i < length(a)      % цикл (работает пока i меньше
                         % длины массива a)
     i=i+1;              % увеличивается счетчик индексов на 1
```

```
    if i == 5               % если индекс равен 5
         continue;       % то его не подсчитываем
     end
     S=S+a(i);           % подсчитывается сумма элементов
end                      % конец цикла
disp(S);                 % отображение суммы 40 на экране
```

Следует отметить, что в данной программе увеличение индекса массива i происходит до проверки условия. Это сделано для того, чтобы значение индекса увеличивалось на 1 на каждой итерации работы цикла. Если увеличение счетчика i записать как в предыдущих примерах, т.е. после подсчета суммы, то из-за оператора continue его значение остановилось бы на 5 и цикл while работал бы «вечно».

2.4. Оператор цикла for

Часто при организации цикла требуется перебирать значение счетчика в заданном диапазоне значений и с заданным шагом изменения. Например, чтобы перебрать элементы вектора (массива), нужно организовать счетчик от 1 до N с шагом 1, где N – число элементов вектора. Чтобы вычислить сумму ряда, также задается счетчик от a до b с требуемым шагом изменения step. И так далее. В связи с тем, что подобные задачи часто встречаются в практике программирования, для их реализации был предложен свой оператор цикла for, который позволяет проще и нагляднее реализовывать цикл со счетчиком.

Синтаксис оператора цикла for имеет следующий вид:

```
for <счетчик> = <начальное значение>:<шаг>:<конечное значение>
      <операторы цикла>
end
```

Рассмотрим работу данного цикла на примере реализации алгоритма поиска максимального значения элемента в векторе:

```
function search_max
a = [3 6 5 3 6 9 5 3 1 0];
m = a(1);                     % текущее максимальное значение
for i=1:length(a)             % цикл от 1 до конца вектора с
                              % шагом 1 (по умолчанию)
    if m < a(i)                   % если a(i) > m,
         m = a(i);            % то m = a(i)
    end
end                           % конец цикла for
disp(m);
```

В данном примере цикл for задает счетчик i и меняет его значение от 1 до 10 с шагом 1. Обратите внимание, что если величина шага не указывается явно, то он берется по умолчанию равным 1.

В следующем примере рассмотрим реализацию алгоритма смещения элементов вектора вправо, т.е. предпоследний элемент ставится на место последнего, следующий – на место предпоследнего, и т.д. до первого элемента:

```
function queue
a = [3 6 5 3 6 9 5 3 1 0];
disp(a);
for i=length(a):-1:2              % цикл от 10 до 2 с шагом -1
     a(i)=a(i-1);             % смещаем элементы вектора a
end                           % конец цикла for
disp(a);
```

Результат работы программы

```
3 6 5 3 6 9 5 3 1 0
3 3 6 5 3 6 9 5 3 1
```

Приведенный пример показывает, что для реализации цикла со счетчиком от большего значения к меньшему, нужно явно указывать шаг, в данном случае, -1. Если этого не сделать, то цикл сразу завершит свою работу и программа будет работать некорректно.

В заключении рассмотрим работу оператора цикла for на примере моделирования случайной последовательности с законом изменения

$$x_i = r x_{i-1} + \xi_i,$$

где r - коэффициент от -1 до 1; ξ_i - нормальная случайная величина с нулевым математическим ожиданием и дисперсией

$$\sigma_\xi^2 = \sigma_x^2(1-r^2),$$

где σ_x^2 - дисперсия моделируемого случайного процесса. При этом первый отсчет x_1 моделируется как нормальная случайная величина с нулевым математическим ожиданием и дисперсией σ_x^2. Программа моделирования имеет следующий вид:

```
function modeling_x
r = 0.95;                     % коэффициент модели
N = 100;                      % число моделируемых точек
ex = 100;                     % дисперсия процесса

et = ex*(1-r^2);              % дисперсия случайной добавки ξi
x = zeros(N,1);                    % инициализация вектора x
x(1) = sqrt(ex)*randn;        % моделирование 1-го отсчета
for i=2:N                     % цикл от 2 до N
     x(i)=r*x(i-1)+sqrt(et)*randn;          % моделирование СП
end                           % конец цикла
plot(x);                      % отображение СП в виде графика
```

При выполнении данной программы будет показана реализация смоделированной случайной последовательности $\mathbf{x}$.

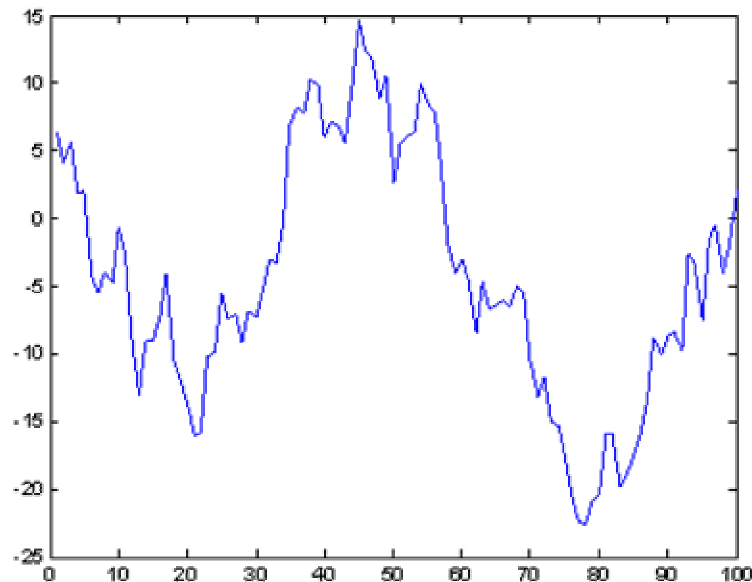


Рис. 2.1. Результат моделирования случайной последовательности.

Работа программы начинается с определения переменных r, σ_x^2 (в программе переменная ex) и N для реализации указанной модели. Затем вычисляется дисперсия $\sigma_\xi^2 = \sigma_x^2(1-r^2)$ и моделируется первый отсчет случайного процесса с помощью функции randn. Функция randn выполняет генерацию нормальных случайных величин с нулевым средним и единичной дисперсией. Чтобы сгенерировать случайную величину с дисперсией σ_x^2 достаточно случайную величину с единичной дисперсией умножить на $\sqrt{\sigma_x^2} = \sigma_x$, т.к. дисперсия – это средний квадрат случайной величины относительно математического ожидания. В результате имеем программную строчку

```
x(1) = sqrt(ex)*randn;
```

Затем, реализуется цикл for со счетчиком i от 2 до N с шагом 1. Внутри цикла выполняется моделирование оставшихся N-1 отсчетов случайного процесса в соответствии с приведенной выше формулой. В последней строчке программы записана функция plot(), которая выводит смоделированную последовательность на экран в виде графика. Более подробно работа с выводом графиков на экран будет рассмотрена в следующей главе.

Глава 3. Работа с графиками в MatLab

MatLab предоставляет богатый инструментарий по визуализации данных. Используя внутренний язык, можно выводить двумерные и трехмерные графики в декартовых и полярных координатах, выполнять отображение изображений с разной глубиной цвета и разными цветовыми картами, создавать простую анимацию результатов моделирования в процессе вычислений и многое другое.

3.1. Функция plot

Рассмотрение возможностей MatLab по визуализации данных начнем с двумерных графиков, которые обычно строятся с помощью функции plot(). Множество вариантов работы данной функции лучше всего рассмотреть на конкретных примерах.

Предположим, что требуется вывести график функции синуса в диапазоне от 0 до π. Для этого зададим вектор (множество) точек по оси Ox, в которых будут отображаться значения функции синуса:

```
x = 0:0.01:pi;
```

В результате получится вектор столбец со множеством значений от 0 до π и с шагом 0,01. Затем, вычислим множество значений функции синуса в этих точках:

```
y = sin(x);
```

и выведем результат на экран

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

В результате получим график, представленный на рис. 3.1.

Представленная запись функции plot() показывает, что сначала записывается аргумент со множеством точек оси Ox, а затем, аргумент со множеством точек оси Oy. Зная эти значения, функция plot() имеет возможность построить точки на плоскости и линейно их интерполировать для придания непрерывного вида графика.

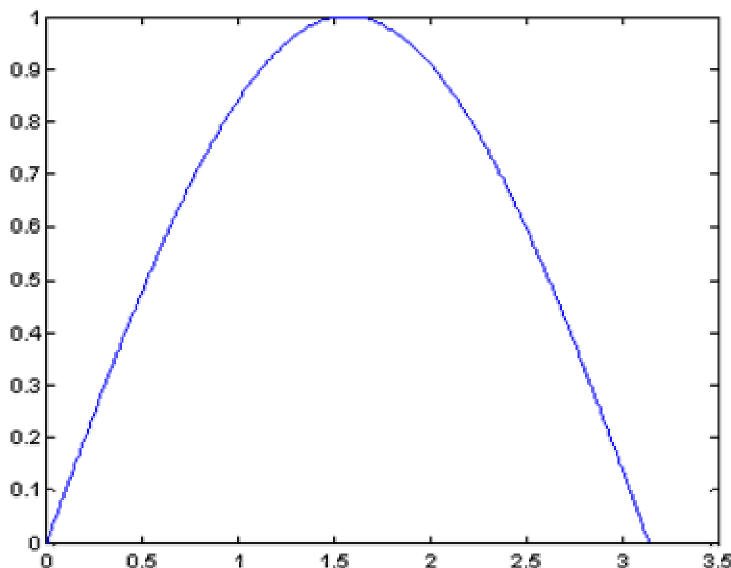


Рис. 3.1. Отображение функции синуса с помощью функции plot().

Функцию plot() можно записать и с одним аргументом x или y:

```
plot(x);
plot(y);
```

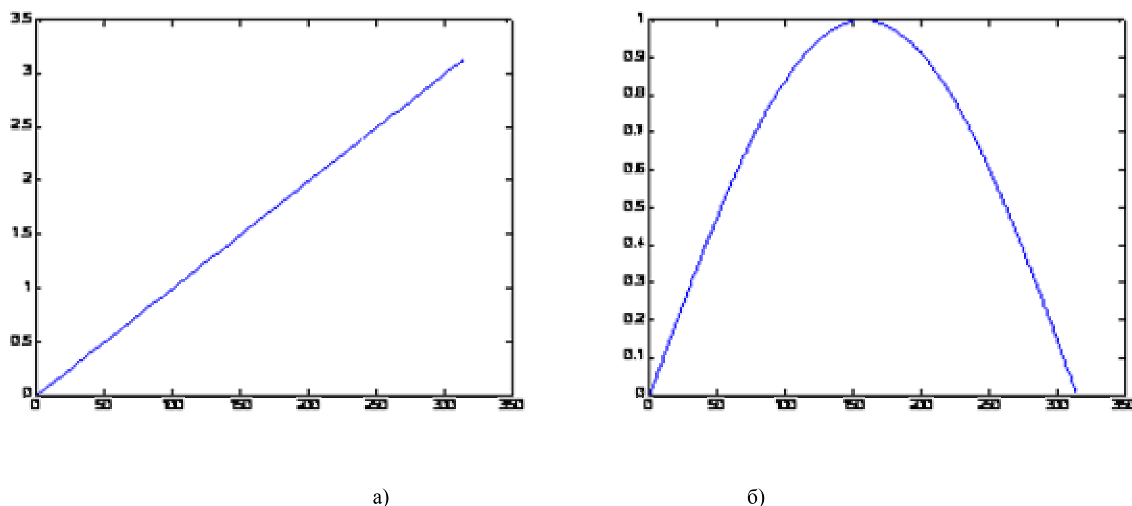
в результате получим два разных графика, представленные на рис. 3.2.

Анализ рис. 3.2 показывает, что в случае одного аргумента функция plot() отображает множество точек по оси Oy, а по оси Ox происходит автоматическая генерация множества точек с единичным шагом. Следовательно, для простой визуализации вектора в виде двумерного графика достаточно воспользоваться функцией plot() с одним аргументом.

Для построения нескольких графиков в одних и тех же координатных осях, функция plot() записывается следующим образом:

```
x = 0:0.01:pi;
y1 = sin(x);
y2 = cos(x);
plot(x,y1,x,y2);
```

Результат работы данного фрагмента программы представлен на рис. 3.3.



а) б)

Рис. 3.2. Результаты работы функции plot() с одним аргументом:

а – plot(x); б – plot(y).

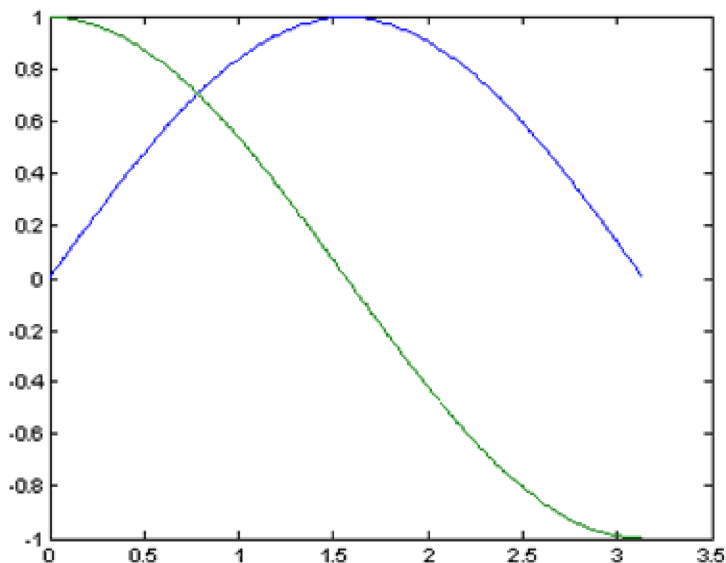


Рис. 3.3. Отображение двух графиков в одних координатных осях.

Аналогичным образом можно построить два графика, используя один аргумент функции plot(). Предположим, что есть два вектора значений

```
y1 = sin(x);
y2 = cos(x);
```

которые требуется отобразить на экране. Для этого объединим их в двумерную матрицу

$$[y1'y2'] = \begin{vmatrix} y1_1 & y2_1 \\ y1_2 & y2_2 \\ \dots & \dots \\ y1_N & y2_N \end{vmatrix},$$

в которой столбцы составлены из векторов y1 и y2 соответственно. Такая матрица будет отображена функцией

```
plot([y1' y2']);     % апострофы переводят вектор-строку
% в вектор-столбец
```

в виде двух графиков (рис. 3.4).

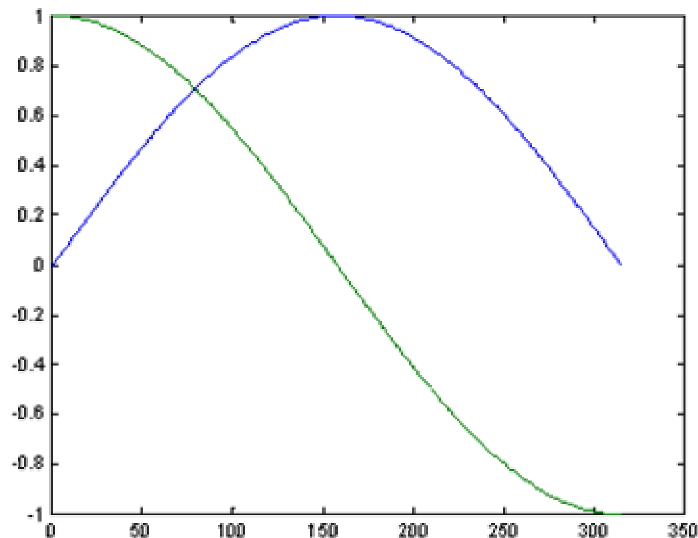


Рис. 3.4. Отображение двумерной матрицы в виде двух графиков.

Два вектора в одних осях можно отобразить только в том случае, если их размерности совпадают. Когда же выполняется работа с векторами разных размерностей, то они либо должны быть приведены друг к другу по числу элементов, либо отображены на разных графиках. Отобразить графики в разных координатных осях можно несколькими способами. В самом простом случае можно создать два графических окна и в них отобразить нужные графики. Это делается следующим образом:

```
x1 = 0:0.01:2*pi;
y1 = sin(x1);

x2 = 0:0.01:pi;
y2 = cos(x2);

plot(x1, y1);            % рисование первого графика
figure;                  % создание 2-го графического окна
plot(x2, y2);            % рисование 2-го графика во 2-м окне
```

Функция figure, используемая в данной программе, создает новое графическое окно и делает его активным. Функция plot(), вызываемая сразу после функции figure, отобразит график в текущем активном графическом окне. В результате на экране будут показаны два окна с двумя графиками.

Неудобство работы приведенного фрагмента программы заключается в том, что повторный вызов функции figure отобразит на экране еще одно новое окно и если программа будет выполнена дважды, то на экране окажется три графических окна, но только в двух из них будут актуальные данные. В этом случае было бы лучше построить программу так, чтобы на экране всегда отображалось два окна с нужными графиками. Этого можно достичь, если при вызове функции figure в качестве аргумента указывать номер графического окна, которое необходимо создать или сделать активным, если оно уже создано. Таким образом, вышеприведенную программу можно записать так.

```
x1 = 0:0.01:2*pi;
y1 = sin(x1);

x2 = 0:0.01:pi;
```

```
y2 = cos(x2);

figure(1);           %создание окна с номером 1
plot(x1, y1);        % рисование первого графика
figure(2);           % создание графического окна с номером 2
plot(x2, y2);        % рисование 2-го графика во 2-м окне
```

При выполнении данной программы на экране всегда будут отображены только два графических окна с номерами 1 и 2, и в них показаны графики функций синуса и косинуса соответственно.

В некоторых случаях большего удобства представления информации можно достичь, отображая два графика в одном графическом окне. Это достигается путем использования функции subplot(), имеющая следующий синтаксис:

```
subplot(<число строк>, <число столбцов>, <номер координатной оси>)
```

Рассмотрим пример отображения двух графиков друг под другом вышеприведенных функций синуса и косинуса.

```
x1 = 0:0.01:2*pi;
y1 = sin(x1);

x2 = 0:0.01:pi;
y2 = cos(x2);

figure(1);
subplot(2,1,1);          % делим окно на 2 строки и один столбец
plot(x1,y1);         % отображение первого графика
subplot(2,1,2);          % строим 2-ю координатную ось
plot(x2,y2);         % отображаем 2-й график в новых осях
```

Результат работы программы показан на рис. 3.5.

Аналогичным образом можно выводить два и более графиков в столбец, в виде таблицы и т.п. Кроме того, можно указывать точные координаты расположения графика в графическом окне. Для этого используется параметр position в функции subplot():

```
subplot('position', [left bottom width height]);
```

где left – смещение от левой стороны окна; bottom – смещение от нижней стороны окна; width, height – ширина и высота графика в окне. Все эти переменные изменяются в пределах от 0 до 1.

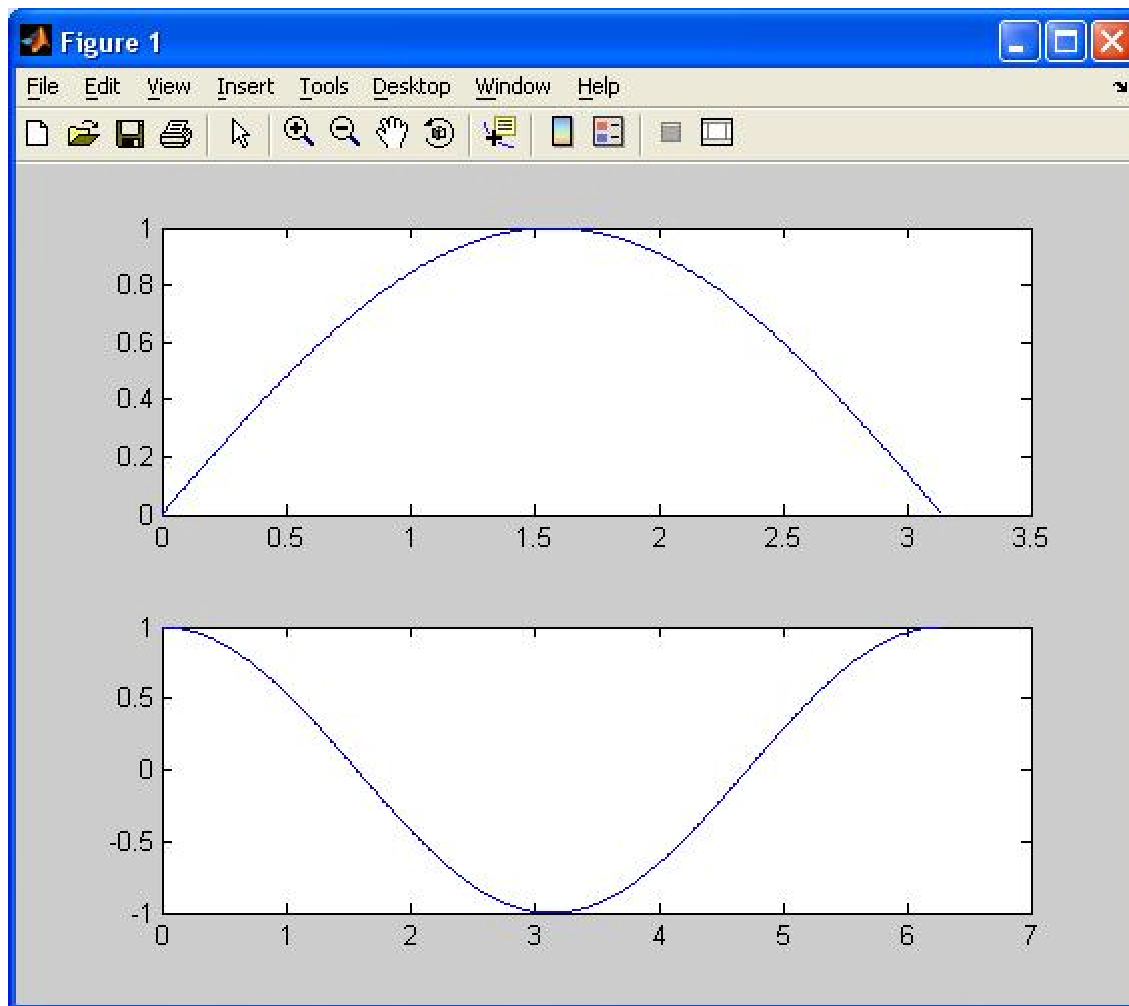


Рис. 3.5. Пример работы функции subplot.

Ниже представлен фрагмент программы отображения графика функции синуса в центре графического окна. Результат работы показан на рис. 3.6.

```
x1 = 0:0.01:2*pi;
y1 = sin(x1);

subplot('position', [0.33 0.33 0.33 0.33]);
plot(x1,y1);
```

В данном примере функция subplot() смещает график на треть от левой и нижней границ окна и рисует график с шириной и высотой в треть графического окна. В результате, получается эффект рисования функции синуса по центру основного окна.

Таким образом, используя параметр position можно произвольно размещать графические элементы в плоскости окна.

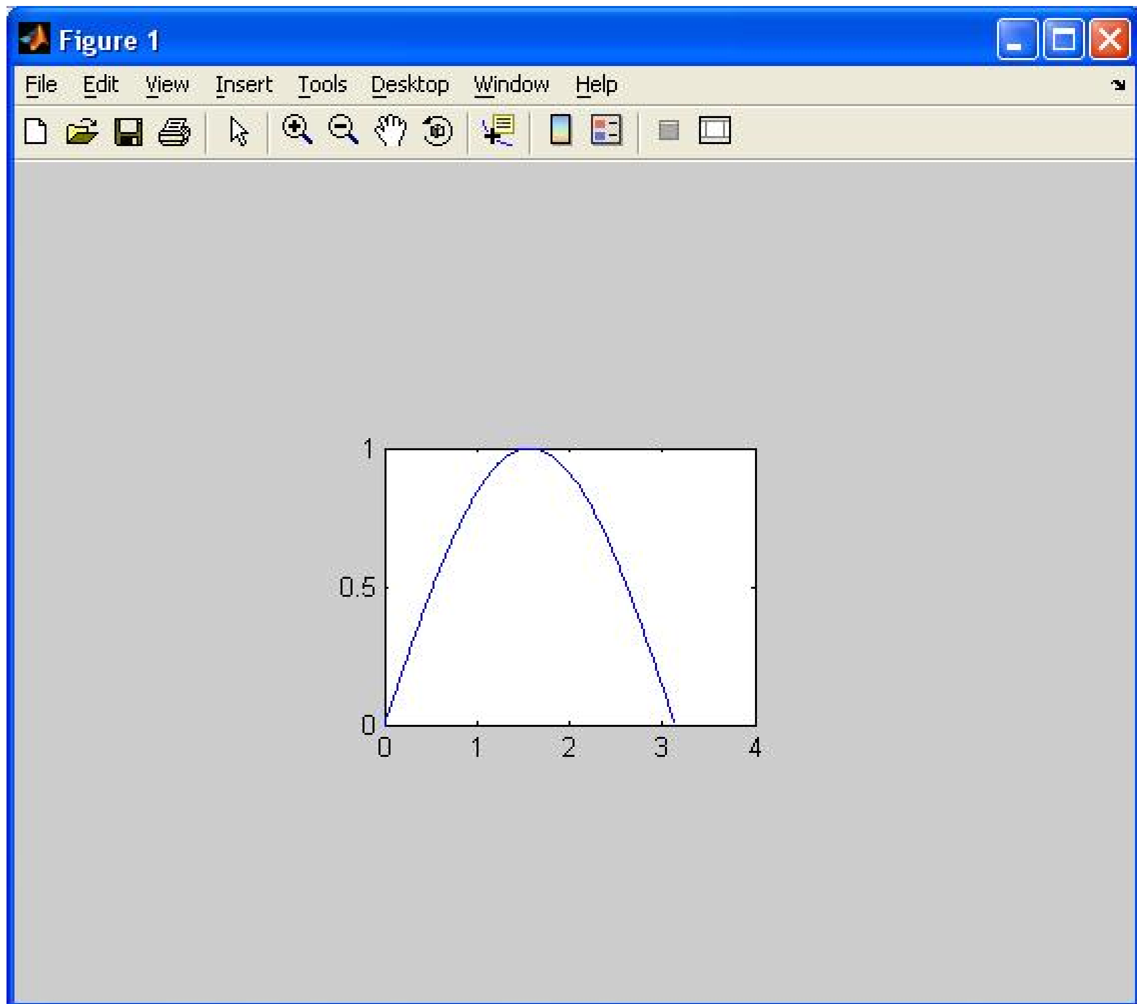


Рис. 3.6. Пример работы функции subplot с параметром position.

3.2. Оформление графиков

Пакет MatLab позволяет отображать графики с разным цветом и типом линий, показывать или скрывать сетку на графике, выполнять подпись осей и графика в целом, создавать легенду и многое другое. В данном параграфе рассмотрим наиболее важные функции, позволяющие делать такие оформления на примере двумерных графиков.

Функция plot() позволяет менять цвет и тип отображаемой линии. Для этого, используются дополнительные параметры, которые записываются следующим образом:

```
plot(<x>, <y>, <'цвет линии, тип линии, маркер точек'>);
```

Обратите внимание, что третий параметр записывается в апострофах и имеет обозначения, приведенные в таблицах 3.1-3.3. Маркеры, указанные ниже записываются подряд друг за другом, например,

```
'ko' – на графике отображает черными кружками точки графика,
'ko-' - рисует график черной линией и проставляет точки в виде кружков.
```

Табл. 3.1. Обозначение цвета линии графика

Маркер	Цвет линии
c	голубой
m	фиолетовый
y	желтый
r	красный
g	зеленый
b	синий

w	белый
k	черный

Табл. 3.2. Обозначение типа линии графика

Маркер	Цвет линии
-	непрерывная
--	штриховая
:	пунктирная
-.	штрих-пунктирная

Табл. 3.3. Обозначение типа точек графика

Маркер	Цвет линии
.	точка
+	плюс
*	звездочка
o	кружок
x	крестик

Ниже показаны примеры записи функции plot() с разным набором маркеров.

```
x = 0:0.1:2*pi;
y = sin(x);

subplot(2,2,1); plot(x,y,'r-');
subplot(2,2,2); plot(x,y,'r-',x,y,'ko');
subplot(2,2,3); plot(y,'b--');
subplot(2,2,4); plot(y,'b--+');
```

Результат работы фрагмента программы приведен на рис. 3.7. Представленный пример показывает, каким образом можно комбинировать маркеры для достижения требуемого результата. А на рис. 3.7 наглядно видно к каким визуальным эффектам приводят разные маркеры, используемые в программе. Следует особо отметить, что в четвертой строчке программы по сути отображаются два графика: первый рисуется красным цветом и непрерывной линией, а второй черными кружками заданных точек графика. Остальные варианты записи маркеров очевидны.

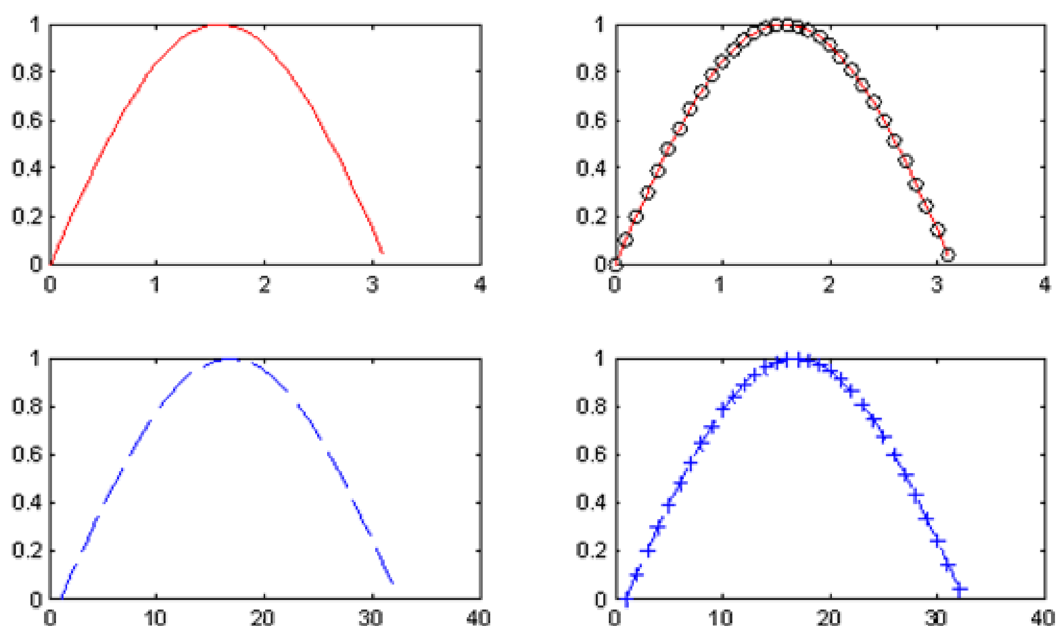


Рис. 3.7. Примеры отображения графиков с разными типами маркеров

Из примеров рис. 3.7 видно, что масштаб графиков по оси Ox несколько больше реальных значений. Дело в том, что система MatLab автоматически масштабирует систему координат для полного представления данных. Однако такая автоматическая настройка не всегда может удовлетворять интересам пользователя. Иногда требуется выделить отдельный фрагмент графика и только его показать целиком. Для этого используется функция axis() языка MatLab, которая имеет следующий синтаксис:

```
axis( [ xmin, xmax, ymin, ymax ] ),
```

где название указанных параметров говорят сами за себя.

Воспользуемся данной функцией для отображения графика функции синуса в пределах от 0 до 2π:

```
x = 0:0.1:2*pi;
y = sin(x);

subplot(1,2,1);
plot(x,y);
axis([0 2*pi -1 1]);

subplot(1,2,2);
plot(x,y);
axis([0 pi 0 1]);
```

Из результата работы программы (рис. 3.8) видно, что несмотря на то, что функция синуса задана в диапазоне от 0 до 2π, с помощью функции axis() можно отобразить как весь график, так и его фрагмент в пределах от 0 до π.

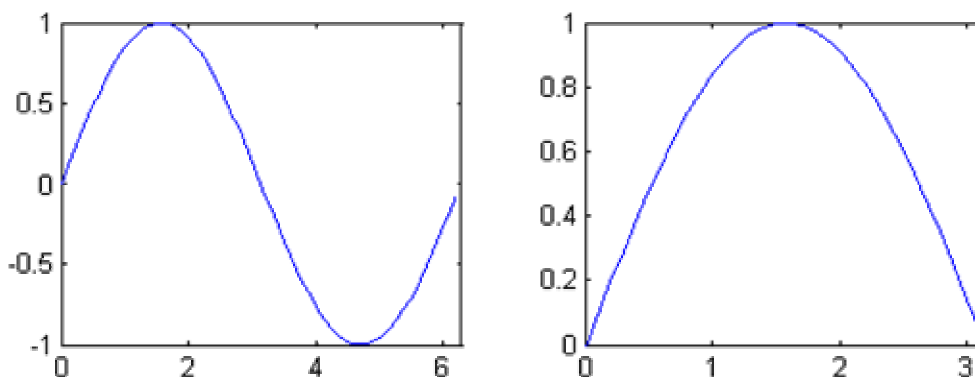


Рис. 3.8. Пример работы функции axis()

В заключении данного параграфа рассмотрим возможности создания подписей графиков, осей и отображения сетки на графике. Для этого используются функции языка MatLab, перечисленные в табл. 3.4.

Таблица 3.4. Функции оформления графиков

Название	Описание
grid [on, off]	Включает/выключает сетку на графике
title('заголовок графика')	Создает надпись заголовка графика
xlabel('подпись оси Ox')	Создает подпись оси Ox
ylabel('подпись оси Oy')	Создает подпись оси Oy
text(x,y,'текст')	Создает текстовую надпись в координатах (x,y).

Рассмотрим работу данных функций в следующем примере:

```
x = 0:0.1:2*pi;
y = sin(x);

plot(x,y);
```

```
axis([0 2*pi -1 1]);
grid on;
title('The graphic of sin(x) function');
xlabel('The coordinate of Ox');
ylabel('The coordinate of Oy');
text(3.05,0.16,'\leftarrow sin(x)');
```

Из результата работы данной программы, представленного на рис. 3.9, видно каким образом работают функции создания подписей на графике, а также отображение сетки графика.

Таким образом, используя описанный набор функций и параметров, можно достичь желаемого способа оформления графиков в системе MatLab.

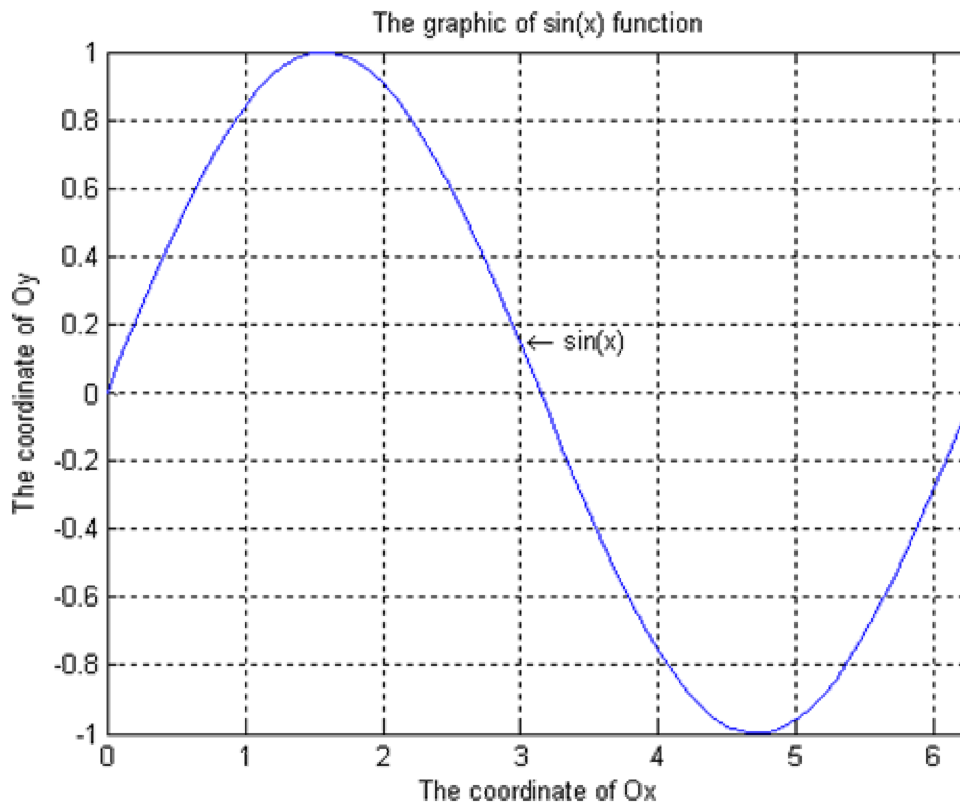


Рис. 3.9. Пример работы функций оформления графика

3.3. Отображение трехмерных графиков

Программа MatLab обладает рядом инструментов для визуализации графиков в трехмерном пространстве. Такие задачи обычно возникают при отображении графиков функций типа $z = f(x, y)$.

В самом простом случае, для визуализации графика в трехмерных координатных осях, используется функция

```
plot3(X,Y,Z);
```

которая в качестве первых двух аргументов принимает матрицы с координатами точек по осям Ox и Oy соответственно, а в качестве третьего аргумента передается матрица значений точек по оси Oz. Рассмотрим работу данной функции на примере отображения графика функции

$$z(x, y) = \exp\left(-x^2 - y^2\right),$$

при $x = -1, -0.9, \ldots, 1$ и $y = -2, -1.9, \ldots, 2$.

Сформируем матрицы X и Y, содержащие координаты точек данного графика по осям Ox и Oy соответственно. Данные матрицы нужны для того, чтобы функция plot3() «знала» какие реальные координаты соответствуют точке Z(i,j) матрицы значений по оси Oz. Для этого достаточно взять i-ю и j-ю компоненту матриц

$$x = X(i, j)$$
$$y = Y(i, j)$$

Формирование матриц X и Y можно осуществить с помощью функции

```
[X,Y]=meshgrid(x,y);
```

языка MatLab. Здесь x и y – одномерные векторы значений координат по осям Ox и Oy соответственно, которые можно сформировать как

```
x=-1:0.1:1;               % координаты точек по оси Ox
y=-2:0.1:2;               % координаты точек по оси Oy
```

и, затем, вычислить матрицы

```
[X,Y]=meshgrid(x,y); % матрицы координат точек по осям Ox и Oy
```

В результате, матрицы X и Y будут содержать следующие первые восемь значений по строкам и столбцам:

```
Матрица X:

-1      -0,9   -0,8   -0,7   -0,6   -0,5   -0,4   -0,3

-1      -0,9   -0,8   -0,7   -0,6   -0,5   -0,4   -0,3

-1      -0,9   -0,8   -0,7   -0,6   -0,5   -0,4   -0,3

-1      -0,9   -0,8   -0,7   -0,6   -0,5   -0,4   -0,3

-1      -0,9   -0,8   -0,7   -0,6   -0,5   -0,4   -0,3

-1      -0,9   -0,8   -0,7   -0,6   -0,5   -0,4   -0,3

-1      -0,9   -0,8   -0,7   -0,6   -0,5   -0,4   -0,3

-1      -0,9   -0,8   -0,7   -0,6   -0,5   -0,4   -0,3

Матрица Y:

-2      -2      -2      -2      -2      -2      -2      -2

-1,9   -1,9   -1,9   -1,9   -1,9   -1,9   -1,9   -1,9

-1,8   -1,8   -1,8   -1,8   -1,8   -1,8   -1,8   -1,8

-1,7   -1,7   -1,7   -1,7   -1,7   -1,7   -1,7   -1,7

-1,6   -1,6   -1,6   -1,6   -1,6   -1,6   -1,6   -1,6

-1,5   -1,5   -1,5   -1,5   -1,5   -1,5   -1,5   -1,5

-1,4   -1,4   -1,4   -1,4   -1,4   -1,4   -1,4   -1,4

-1,3   -1,3   -1,3   -1,3   -1,3   -1,3   -1,3   -1,3
```

Используя данные матрицы, можно вычислить значения матрицы Z, следующим образом:

```
Z=exp(-X.^2-Y.^2);
```

и отобразить результат на экране

```
plot3(X,Y,Z);
```

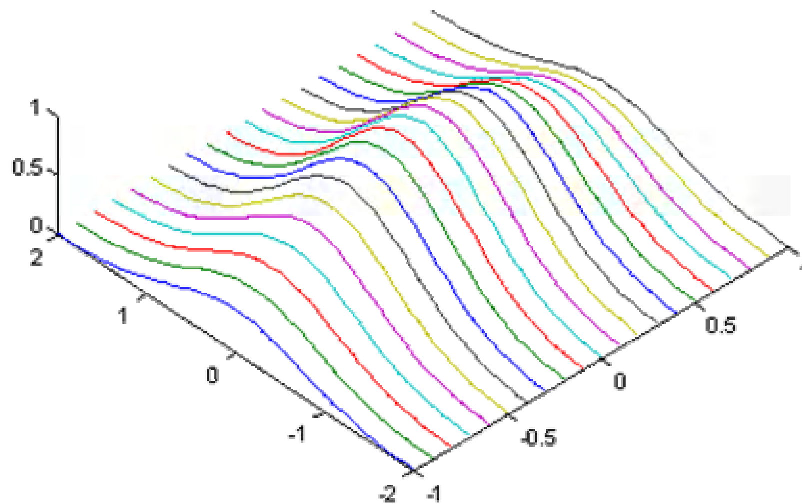


Рис. 3.10. Пример отображения графика с помощью функции plot3()

Из приведенного рисунка видно, что функция plot3() отображает график в виде набора линий, каждая из которых соответствует сечению графика функции $z(x,y)=\exp\left(-x^2-y^2\right)$ вдоль оси Oy.

Такое представление графика не всегда удобно, т.к. набор одномерных не дает полное представление о характере двумерной плоскости. Более лучшей визуализации можно получить, используя функцию

```
mesh(X,Y,Z);    % отображение графика в виде сетки
```

В результате получим следующий вид трехмерного графика (рис. 3.11).

Благодаря использованию функции mesh() получается график, образованный интерполяцией точек массивов X, Y и Z линиями по осям Ox и Oy. Кроме того, цветом указывается уровень точки по оси Oz: от самого малого значения (синего) до самого большого (красного) и производится удаление «невидимых» линий. Это позволяет лучше визуально оценивать структуру трехмерного графика по сравнению с функцией plot3(). Если же необходимо отобразить «прозрачный» график, то следует выключить режим удаления «невидимых» линий:

```
hidden off;          % скрытые линии рисуются
```

В системе MatLab предусмотрена функция визуализации непрерывной поверхности в трехмерных осях

```
surf(X,Y,Z);         % отображение непрерывной поверхности
```

В результате получается график, представленный на рис. 3.12.

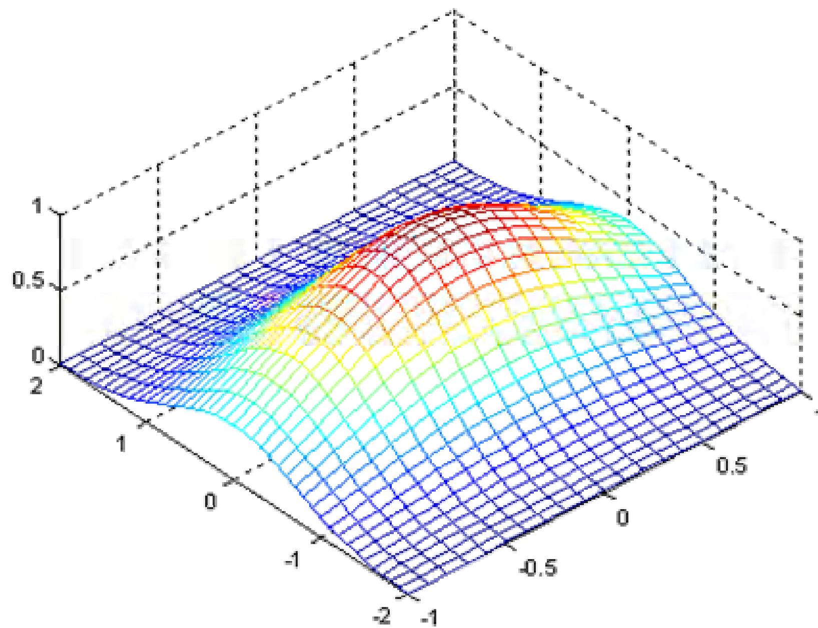


Рис. 3.11. Результат работы функции mesh()

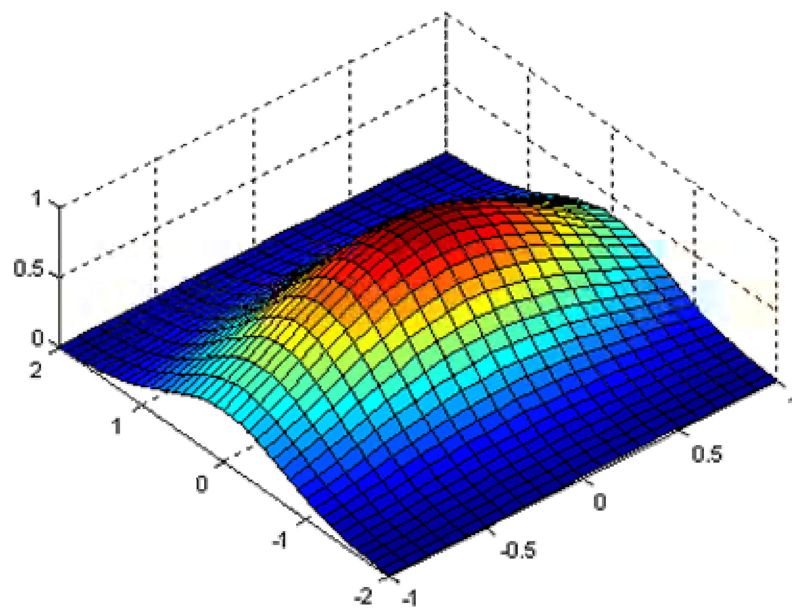


Рис. 3.12. Результат работы функции surf()

Функция surf() может использоваться в режиме

```
shading interp;          % интерполяция тени на гранях графика
```

которая интерполирует цвет на гранях для получения более гладкого изображения поверхности (рис. 3.13). Также существует возможность менять цветовую карту отображения графика с помощью функции

```
colormap( <карта> );     % установка цветовой карты
```

Например, карта с именем hot, используемая по умолчанию может быть заменена на любую другую доступную (hot, hsv, gray, pink, cool, bone copper) или созданную самостоятельно.

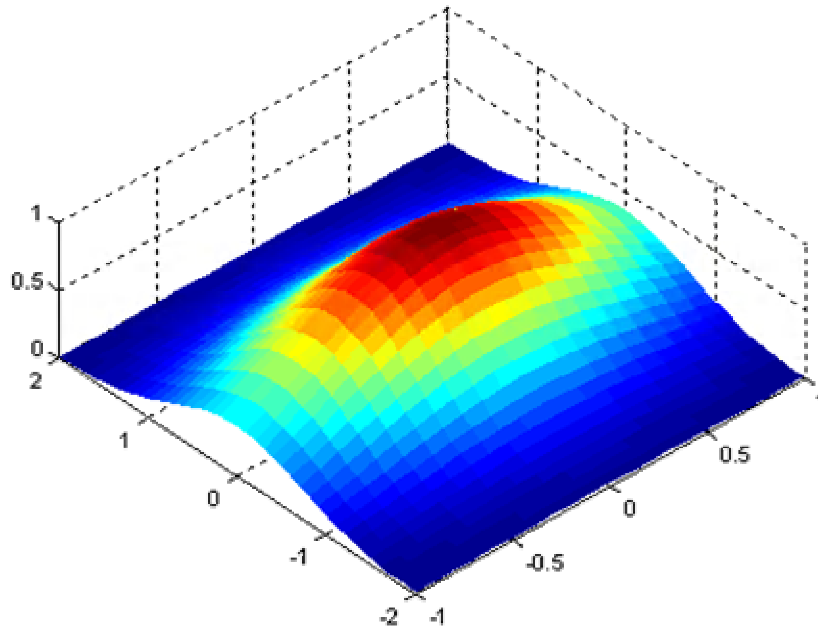


Рис. 3.13. Результат работы функции surf() в режиме shading interp

Следует отметить, что все три функции plot3(), mesh() и surf() могут быть использованы и с одним аргументом Z, который интерпретируется как матрица со значениями точек по оси Oz.

Для масштабирования отдельных участков трехмерных графиков, также как и в случае с двумерными графиками, используется функция

```
axis([xmin xmax ymin ymax zmin zmax]);
```

с очевидным набором параметров.

Для оформления трехмерных графиков можно пользоваться описанными ранее функциями: text, xlabel, ylabel, zlabel, title, grid [on/off], subplot.

Наконец, для трёхмерных графиков существует возможность изменять точку их обзора, т.е. положение виртуальной камеры с помощью функции

```
view([az el]);
```

где az – угол азимута; el – угол возвышения. Изменение первого угла означает вращение плоскости xOy вокруг оси Oz против часовой стрелки. Угол возвышения есть угол между направлением на камеру и плоскостью xOy.

3.4. Отображение растровых изображений

Система MatLab позволяет загружать, отображать и сохранять растровые изображения, представленные в известных графических форматах: bmp, jpg, tiff, gif, png и т.д. Для загрузки изображений используется функция

```
imread(FILENAME,FMT);
```

которой в качестве параметров передается путь с указанием графического файла и формат. На выходе дается массив, соответствующий размерности изображения со значениями в формате uint8:

```
A=imread('1024.bmp','bmp'); % A - 1024x1024 матрица uint8
```

Если исходное изображение имеет глубину цвета не более 256, то матрица A будет двумерной и каждое ее значение будет представлять соответствующий уровень яркости точки. Если же точка исходного изображения представляется, например, 24 битами (3 байта), то матрица A будет иметь размерность 1024x1024x3, где третья размерность будет соответствовать цветовым составляющим исходного изображения. Поэтому для обработки изображений в экспериментальных целях часто используют формат с 256 градациями серого без индексации цвета.

Формат uint8 не очень удобен для последующей обработки точек изображения, поэтому обычно его приводят к вещественному следующим образом:

```
A=double(imread('1024.bmp','bmp'));
```

В результате матрица А будет состоять из элементов типа double.

Для отображения растровых изображений в графическом окне используется функция

```
image(A);
```

результат действия которой показан на рис. 3.14.

Неверное отображение изображения объясняется несоответствием палитры цветов по умолчанию (hot), заданное в MatLab, с палитрой цветов загруженного изображения (gray(256)). Для замены одной палитры на другую используется функция

```
colormap(gray(256));
```

которая в данном случае задает палитру 256 градаций серого и результат отображения принимает вид, представленный на рис. 3.15.

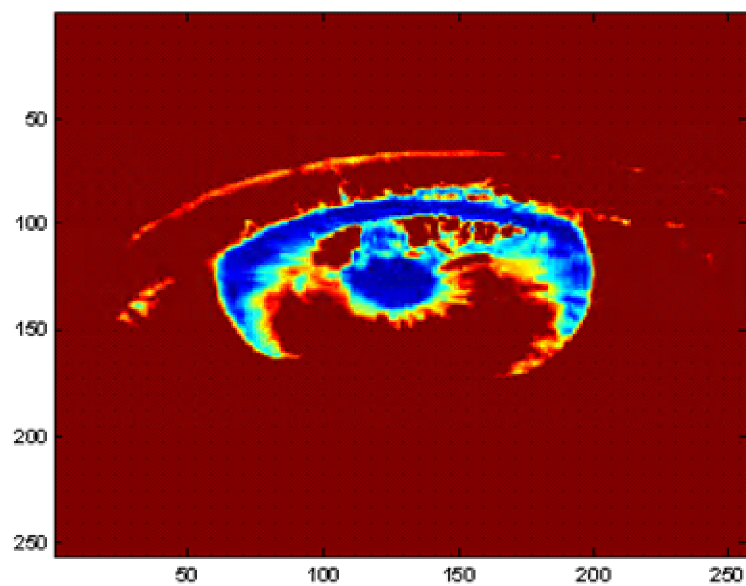


Рис. 3.14. Результат работы функции image

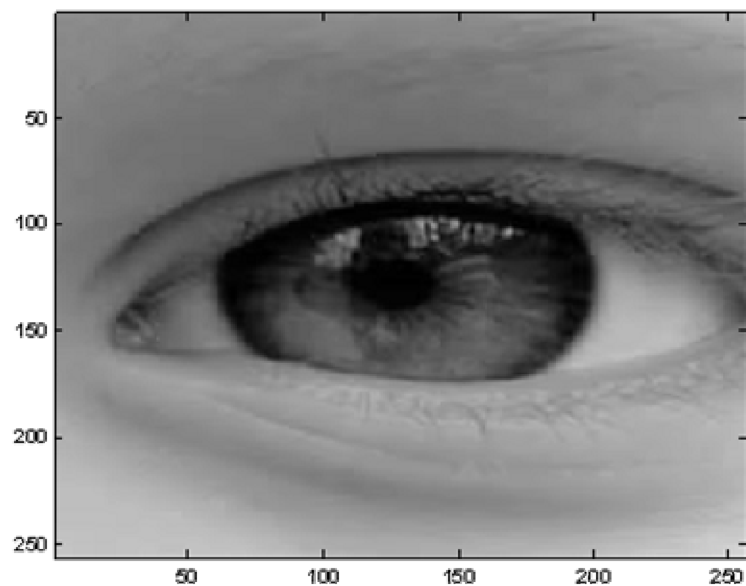


Рис. 3.15. Коррекция цветовой палитры функцией colormap

Если цветовая палитра заранее неизвестна на момент загрузки изображения, то ее можно узнать, используя второй возвращаемый параметр функции imread:

```
[A, map]=imread('1024.bmp','bmp');
image(A);
colormap(map);
```

где map – цветовая карта текущего изображения.

При работе с изображениями возникают ситуации, когда диапазон значений элементов матрицы A может не соответствовать диапазону значений цветовой карты, например, gray(256). В результате отображения такой матрицы на экране монитора изображение будет показываться некорректно и некоторые его детали будут незаметны. Чтобы избежать такой ситуации, диапазон значений и диапазон цветовой карты должны совпадать. Это можно сделать искусственно, масштабируя соответствующим образом значения элементов матрицы A. Однако MatLab предоставляет функцию

```
imagesc(A);
```

которая делает это автоматически. Благодаря ее использованию, масштаб значений матрицы A всегда будет приведен к масштабу цветовой карты.

Глава 4. Программирование функций в MatLab

Часто при программировании приходится много раз повторять одни и те же вычисления, например, определение модуля числа или расчет евклидового расстояния между точками и т.п. Для реализации таких повторяющихся вычислений целесообразно создавать функции и вызывать их по мере необходимости.

4.1. Порядок определения и вызова функций

В первой главе данного пособия был показан порядок определения собственных функций в программе MatLab. В данном параграфе остановимся на более подробном описании программирования пользовательских функций.

Синтаксис для определения собственных функций в MatLab имеет следующий вид:

```
function [ RetVal1, RetVal2,… ] = FunctionName( arg1, arg2,… )
<тело функции>
```

где RetVal1, RetVal2,... – набор возвращаемых значений функцией (результаты работы); arg1, arg2,... – набор входных аргументов; тело функции – набор операторов (программа), которые выполняются при вызове функции.

Рассмотрим пример реализации функции для вычисления евклидового расстояния:

```
function length = euqlid(x1, y1, x2, y2)
length = sqrt((x1-x2)^2+(y1-y2)^2);
```

Продемонстрируем возможность возвращения нескольких параметров на примере вычисления ширины и высоты прямоугольника, заданного координатами левого верхнего угла (x1,y1) и правого нижнего (x2,y2):

```
function [width, height] = RectangleHW(x1,y1,x2,y2)
width = abs(x1-x2);
height = abs(y1-y2);
```

Данную функцию можно записать еще и с таким набором параметров:

```
function [width, height] = RectangleHW(P1, P2)
width = abs(P1(1)-P1(2));
height = abs(P2(1)-P2(2));
```

где P1 и P2 – векторы (массивы) размером в 2 элемента и описывают точку в двумерном пространстве. В этом случае при вызове функции, значения координат точек можно передавать таким образом:

```
[W, H] = RectangleHW([0 0], [10 20]);
```

Если же программист сделает ошибку и при вызове функции передаст неверный размер вектора, например, так

```
[W, H] = RectangleHW(0, [10 20]);
```

то выполнение функции завершится с ошибкой и выполнение всего алгоритма остановится. Чтобы избежать этой ситуации MatLab позволяет проводить проверку корректности переданных аргументов и корректно завершать работу функции без остановки работы всего алгоритма. Следующий пример записи функции демонстрирует работу такой проверки:

```
function [width, height] = RectangleHW(P1, P2)
if length(P1) < 2 | length(P2) < 2
     error( 'Bad 1st or 2nd parameter' );
end

width = abs(P1(1)-P1(2));
height = abs(P2(1)-P2(2));
```

При выполнении данной функции с неверными параметрами, функция выдаст сообщение об ошибке в командное окно MatLab, но программа продолжит свою работу.

Предложенная проверка осуществляет контроль за корректностью переданных аргументов. Однако важной является также проверка числа переданных входных аргументов и числа возвращаемых значений функцией. Например, если вместо двух аргументов, был передан только один, то функция ошибочно завершит свою работу. Аналогично, если функция ожидает возврата трех аргументов, в то время как она определена лишь для двух, то также возникнет ошибочная ситуация.

Для проверки числа переданных аргументов и числа ожидающих возвращаемых значений используются переменные nargin и nargout. Ниже приведен пример функции, использующей проверку корректности числа входных и выходных аргументов.

```
function [width, height] = RectangleHW(P1, P2)
if nargin ~= 2
    error( 'Bad number of parameters' );
end
if nargout ~= 2
    error(  'Must be 2 return values' );
end
if length(P1) < 2 | length(P2) < 2
    error( 'Bad 1st or 2nd parameter' );
end

width = abs(P1(1)-P1(2));
height = abs(P2(1)-P2(2));
```

При этом проверки корректности параметров функции будут срабатывать в следующих ситуациях:

```
[W, H] = RectangleHW([0 0]);      % Bad number of parameters
[W, H, V] = RectangleHW([0 0], [10 20]); % Must be 2 return
                                         % values
[W, H] = RectangleHW(0, [10 20]); % Bad 1st or 2nd parameter
```

4.2. Область видимости переменных

Следует отметить, что переменные, объявленные внутри функций, имеют область видимости только в пределах функции, и за ее пределами уже не доступны (не видны). Следующий пример программы демонстрирует механизм области видимости имен переменных в MatLab:

```
function MyFunc
x = 10;
disp(x);
MyFunc2();

function MyFunc2()
disp(x);
```

В результате на экране будет отображено

```
10
??? Undefined function or variable 'x'.
```

Этот пример показывает, что переменная с именем x, объявленная в функции MyFunc, не доступна в функции MyFunc2. Это сделано с расчетом, чтобы переменные в разных функциях не влияли друг на друга даже если они имеют одни и те же имена. Однако в некоторых случаях требуется, чтобы переменная была видна за пределами функции, в которой объявлена. Это достигается путем обращения к переменной как к глобальной с помощью ключевого слова global, за которым следует имя глобальной переменной. Перепишем пример, представленный выше с использованием глобальной переменной:

```
function MyFunc
x = 10;
disp(x);
MyFunc2();

function MyFunc2()
global x;
disp(x);
```

Обратите внимание, что ключевое слово global написано в функции MyFunc2 и говорит о том, что переменная x уже объявлена ранее и нужно ее использовать внутри текущей функции.

Глава 5. Работа с файлами в MatLab

Создание программ часто предполагает сохранение результатов расчетов в файлы для их дальнейшего анализа, обработки, хранения и т.п. В связи с этим в MatLab реализованы различные функции по работе с файлами, содержащие данные в самых разных форматах. В этой главе рассмотрим наиболее полезные функции для сохранения и загрузки результатов работы алгоритмов из файлов.

5.1. Функции save и load

В самом простом случае для сохранения и последующей загрузки каких-либо данных в MatLab предусмотрены две функции

```
save <имя файла> <имена переменных>  % сохранение данных
load <имя файла> <имена переменных>       % загрузка данных
```

Функция save позволяет сохранять произвольные переменные программы в файл, который будет (по умолчанию) располагаться в рабочем каталоге (обычно поддиректория work) и иметь расширение mat. Соответственно функция load позволяет загрузить из указанного mat-файла ранее сохраненные переменные. Ниже представлен пример использования данных функций:

```
function save_load
x = ones(5);
y = 5;
s = 'hello';

save params x y s;
x = zeros(5);
y = 0;
s = '';

load params x y s;
disp(x);
disp(y);
disp(s);
```

В данной программе сначала выполняется инициализация переменных x, y, s, затем, они сохраняются в файл params.mat, заменяются другими значениями и после загрузки отображаются на экране. При выполнении этой программы на экране будут показаны те же значения переменных, которые они принимали в самом начале. Таким образом демонстрируется работа функций save и load.

Следует обратить внимание, что функция load позволяет загружать из mat-файла не все, а только указанные программистом переменные, например

```
load params x;       % загружает только значение переменной x
load params y;       % загружает только значение переменной y
load params x s;     % загружает значения переменных x и s
```

5.2. Функции fwrite и fread

Недостатком рассмотренных функций save и load является то, что они работают с определенными форматами файлов (обычно mat-файлы) и не позволяют загружать или сохранять данные в других форматах. Между тем бывает необходимость загружать информацию, например, из бинарных файлов, созданных другими программными продуктами для дальнейшей обработки результатов в MatLab. С этой целью были разработаны функции

```
fwrite(<идентификатор файла>, <переменная>, <тип данных>);
```

и

```
<переменная>=fread(<идентификатор файла>);
<переменная>=fread(<идентификатор файла>, <размер>);
<переменная>=fread(<идентификатор файла>, <размер>, <точность>);
```

Здесь <идентификатор файла> - это указатель на файл, с которым предполагается работать. Чтобы получить идентификатор, используется функция

```
<идентификатор файла> = fopen(<имя файла>,<режим работы>);
```

где параметр <режим работы> может принимать значения, приведенные в табл. 5.1.

Таблица 5.1. Режимы работы с файлами в MatLab

параметр <режим работы>	описание
'r'	чтение
'w'	запись (стирает предыдущее содержимое файла)
'a'	добавление (создает файл, если его нет)
'r+'	чтение и запись (не создает файл, если его нет)
'w+'	чтение и запись (очищает прежнее содержимое или создает файл, если его нет)
'a+'	чтение и добавление (создает файл, если его нет)
'b'	дополнительный параметр, означающий работу с бинарными файлами, например, ‘wb’, ‘rb’ ‘rb+’, ‘ab’ и т.п.

Если функция fopen() по каким-либо причинам не может корректно открыть файл, то она возвращает значение -1. Ниже представлен фрагмент программы записи и считывания данных из бинарного файла:

```
A = [1 2 3 4 5];

fid = fopen('my_file.dat', 'wb');     % открытие файла на запись
if fid == -1                         % проверка корректности открытия
    error('File is not opened');
end

fwrite(fid, A, 'double');   % запись матрицы в файл (40 байт)
fclose(fid);                % закрытие файла

fid = fopen('my_file.dat', 'rb');     % открытие файла на чтение
if fid == -1                         % проверка корректности открытия
    error('File is not opened');
end

B = fread(fid, 5, 'double');         % чтение 5 значений double
disp(B);                             % отображение на экране
fclose(fid);                         % закрытие файла
```

В результате работы данной программы в рабочем каталоге будет создан файл my_file.dat размером 40 байт, в котором будут содержаться 5 значений типа double, записанных в виде последовательности байт (по 8 байт на каждое значение). Функция fread() считывает последовательно сохраненные байты и автоматически преобразовывает их к типу double, т.е. каждые 8 байт интерпретируются как одно значение типа double.

В приведенном примере явно указывалось число элементов (пять) для считывания из файла. Однако часто общее количество элементов бывает наперед неизвестным, либо оно меняется в процессе работы программы. В этом случае было бы лучше считывать данные из файла до тех пор, пока не будет достигнут его конец. В MatLab существует функция для проверки достижения конца файла

```
feof(<идентификатор файла>)
```

которая возвращает 1 при достижении конца файла и 0 в других случаях. Перепишем программу для считывания произвольного числа элементов типа double из входного файла.

```
fid = fopen('my_file.dat', 'rb');  % открытие файла на чтение
if fid == -1
    error('File is not opened');
end

B=0;                   % инициализация переменной
cnt=1;                 % инициализация счетчика
while ~feof(fid)     % цикл, пока не достигнут конец файла
    [V,N] = fread(fid, 1, 'double');  %считывание одного
% значения double (V содержит значение
% элемента, N – число считанных элементов)
    if N > 0           % если элемент был прочитан успешно, то
        B(cnt)=V;    % формируем вектор-строку из значений V
        cnt=cnt+1;  % увеличиваем счетчик на 1
    end
end
disp(B);               % отображение результата на экран
fclose(fid);          % закрытие файла
```

В данной программе динамически формируется вектор-строка по мере считывания элементов из входного файла. MatLab автоматически увеличивает размерность векторов, если индекс следующего элемента на 1 больше максимального. Однако на такую процедуру тратится много машинного времени и программа начинает работать заметно медленнее, чем если бы размерность вектора B с самого начала была определена равным 5 элементам, например, так

```
B = zeros(5,1);
```

Следует также отметить, что функция fread() записана с двумя выходными параметрами V и N. Первый параметр содержит значение считанного элемента, а второй – число считанных элементов. В данном случае значение N будет равно 1 каждый раз при корректном считывании информации из файла, и 0 при считывании служебного символа EOF, означающий конец файла. Приведенная ниже проверка позволяет корректно сформировать вектор значений B.

С помощью функций fwrite() и fread() можно сохранять и строковые данные. Например, пусть дана строка

```
str = 'Hello MatLab';
```

которую требуется сохранить в файл. В этом случае функция fwrite() будет иметь следующую запись:

```
fwrite(fid, str, 'int16');
```

Здесь используется тип int16, т.к. при работе с русскими буквами система MatLab использует двухбайтовое представление каждого символа. Ниже представлена программа записи и чтения строковых данных, используя функции fwrite() и fread():

```
fid = fopen('my_file.dat', 'wb');
if fid == -1
    error('File is not opened');
end

str='Привет MatLab';                   % строка для записи
fwrite(fid, str, 'int16');         % запись в файл
fclose(fid);

fid = fopen('my_file.dat', 'rb');
if fid == -1
    error('File is not opened');
end

B='';                                  % инициализация строки
cnt=1;
while ~feof(fid)
    [V,N] = fread(fid, 1, 'int16=>char');  % чтение текущего
% символа и преобразование
% его в тип char
    if N > 0
        B(cnt)=V;
        cnt=cnt+1;
    end
```

```
end
disp(B);                        % отображение строки на экране
fclose(fid);
```

Результат выполнения программы будет иметь вид

```
Привет MatLab
```

5.3. Функции fscanf и fprintf

Описанные выше функции работы с файлами позволяют записывать и считывать информацию по байтам, которые затем требуется правильно интерпретировать для преобразования их в числа или строки. В то же время выходными результатами многих программ являются текстовые файлы, в которых явным образом записаны те или числа или текст. Например, при экспорте данных из MS Excel можно получить файл формата

```
174500,1.63820,1.63840,1.63660,1.63750,288
180000,1.63740,1.63950,1.63660,1.63820,361
181500,1.63830,1.63850,1.63680,1.63740,223
183000,1.63720,1.64030,1.63720,1.64020,220
```

где числа записаны в столбик и разделены запятой.

Прочитать такой файл побайтно, а затем интерпретировать полученные данные довольно трудоемкая задача, поэтому для этих целей были специально разработаны функции чтения

```
[value, count] = fscanf(fid, format, size)
```

и записи

```
count = fprintf(fid, format, a,b,...)
```

таких данных в файл. Здесь value – результат считывания данных из файла; count – число прочитанных (записанных) данных; fid – указатель на файл; format – формат чтения (записи) данных; size – максимальное число считываемых данных; a,b,.. – переменные для записи в файл.

Приведем пример чтения данных из файла, приведенного выше с помощью функции fscanf():

```
function fscanf_ex

fid = fopen('my_excel.dat', 'r');
if fid == -1
    error('File is not opened');
end

S = fscanf(fid, '%d,%f,%f,%f,%f,%d');
fclose(fid);
```

Здесь форматная строка состоит из спецификаторов

%d – работа с целочисленными значениями;
%f – работа с вещественными значениями

и записана в виде '%d,%f,%f,%f,%f,%d'. Это означает, что сначала должно быть прочитано целочисленное значение из файла, затем, через запятую должно читаться второе вещественное значение, затем третье и так далее до последнего целочисленного значения. Полный список возможных спецификаторов приведен в табл. 5.2.

В результате работы программы переменная S будет представлять собой вектор-столбец, состоящий из 24 элементов:

```
S = [174500 1,6382 1,6384 1,6366 1,6375 288 180000 1,6374 1,6395 1,6366 1,6382 361
181500 1,6383 1,6385 1,6368 1,6374 223 183000 1,6372 1,6403 1,6372 1,6402 220]';
```

Несмотря на то, что данные были корректно считаны из файла, они из таблицы были преобразованы в вектор-столбец, что не соответствует исходному формату представления данных. Чтобы сохранить верный формат данных, функцию fscanf() в приведенном примере следует записать так:

```
S = fscanf(fid, '%d,%f,%f,%f,%f,%d', [6 4]);
```

Тогда на выходе получится матрица S размером в 6 столбцов и 4 строки с соответствующими числовыми значениями.

Таблица 5.2. Список основных спецификаторов для функций fscanf() и fprintf()

Спецификатор	Описание
%d	целочисленные значения
%f	вещественные значения
%s	строковые данные
%c	символьные данные
%u	беззнаковые целые значения

Для записи данных в текстовый файл в заданном формате используется функция fprintf(). Ниже представлен пример записи матрицы чисел

```
180000    1.28210    1.28240    1.28100    1.28120    490
190000    1.28100    1.28150    1.27980    1.28070    444
200000    1.28050    1.28080    1.27980    1.28000    399
210000    1.27990    1.28020    1.27880    1.27970    408
220000    1.27980    1.28060    1.27880    1.28030    437
230000    1.28040    1.28170    1.28020    1.28130    419
000000    1.28140    1.28140    1.28010    1.28100    294
010000    1.28080    1.28190    1.28030    1.28180    458
020000    1.28190    1.28210    1.28080    1.28140    384
030000    1.28130    1.28170    1.28080    1.28140    313
```

в файл, в котором числовые значения должны разделяться точкой с запятой. Будем также предполагать, что данная матрица хранится в переменной Y.

```
function fprintf_ex

fid = fopen('my_excel.txt', 'w');
if fid == -1
    error('File is not opened');
end

fprintf(fid, '%6d;%.4f;%.4f;%.4f;%.4f;%d\r\n', Y');
fclose(fid);
```

Следует отметить, что в функции fprintf() переменная Y имеет знак транспонирования ‘, т.к. данные в файл записываются по столбцам матрицы. Кроме того, перед спецификаторами стоят числа, которые указывают сколько значащих цифр числа должно быть записано в файл. Например, спецификатор %6d говорит о том, что целые числа должны иметь 6 значащих цифр, а спецификатор %.4f означает, что после запятой будет отображено только 4 цифры. Наконец, в форматной строке были использованы управляющие символы

\r – возврат каретки;
\n – переход на новую строку

которые необходимы для формирования строк в выходном файле. В итоге, содержимое файла будет иметь вид:

```
180000;1.2821;1.2824;1.2810;1.2812;490
190000;1.2810;1.2815;1.2798;1.2807;444
200000;1.2805;1.2808;1.2798;1.2800;399
210000;1.2799;1.2802;1.2788;1.2797;408
220000;1.2798;1.2806;1.2788;1.2803;437
230000;1.2804;1.2817;1.2802;1.2813;419
     0;1.2814;1.2814;1.2801;1.2810;294
 10000;1.2808;1.2819;1.2803;1.2818;458
 20000;1.2819;1.2821;1.2808;1.2814;384
 30000;1.2813;1.2817;1.2808;1.2814;313
```

С помощью функции fprintf() можно записать значения двух и более переменных разного формата. Например, для записи числа и строки можно воспользоваться следующей записью:

```
str = 'Hello';
y = 10;
count = fprintf(fid, '%d\r\n%s\r\n', y, str);
```

и содержимое файла будет иметь вид:

```
10
Hello
```

Таким образом можно осуществлять запись разнородных данных в файл в требуемом формате.

5.4. Функции imread и imwrite

При работе с файлами изображений, представленных в форматах bmp, png, gif, jpeg, tif и т.д., используются функции чтения

```
[X, map] = imread(filename, fmt)
```

и записи

```
imwrite(X, map, filename, fmt)
```

Здесь X – матрица точек изображения; map – цветовая карта изображения; filename – путь к файлу; fmt – графический формат файла изображения.

Работа функции imread() была подробно рассмотрена в п. 3.4. Ниже приведен пример ее использования для загрузки растрового изображения

```
[A, map]=imread('1024.bmp','bmp');
```

где A – матрица размером 1024x1024xN точек; map – цветовая карта загруженного изображения. Значение N показывает число байт, расходуемых на представление точки изображения. Например, если изображение представляется в формате RGB с 24 бит/пиксел, то N=3. Если же загружается изображение с 256 градациями серого (8 бит/пиксел), то N=1.

После обработки изображение A можно обратно сохранить в файл, используя следующую запись:

```
imwrite(A, map, 'out_img.bmp', 'bmp');
```

В результате в рабочем каталоге MatLab будет сохранено изображение в формате bmp с исходной цветовой картой. Однако следует отметить, что если загруженное изображение A было преобразовано, например, в формат double

```
A = double(A);
```

то непосредственная запись такой матрицы как изображение невозможно. Дело в том, что значения матрицы A должны соответствовать целым числам в диапазоне от 0 до 255, т.е. являться байтовыми числами. Этого можно добиться преобразованием типов при записи изображения в файл следующим образом:

```
imwrite(uint8(A), map, 'out_img.bmp', 'bmp');
```

Здесь uint8 – беззнаковый целый тип в 8 бит.

В качестве переменной map можно указывать любые другие цветовые карты (hot, hsv, gray, pink, cool, bone copper) отличные от исходной. Например, для записи изображения в 256 градациях серого можно записать

```
imwrite(uint8(A), gray(256), 'out_img.bmp', 'bmp');
```

При этом матрица A должна иметь размерность MxNx1, т.е. один байт на пиксел.