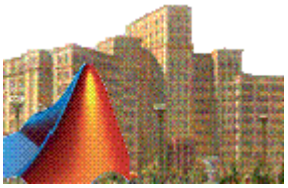


Доля П.Г.
Харьковский Национальный Университет
механико – математический факультет
кафедра геометрии им. А.В. Погорелова

Часть 2. Особенности использования MatLab. Типы данных и программирование.

В части 1 пособия были рассмотрены основные элементы, необходимые для первого знакомства с системой. После его прочтения можно решать довольно сложные математические задачи. Во второй части «Особенности использования MatLab» мы приводим более глубокие сведения о типах данных, программировании и графических возможностях системы. Часть 2 состоит из двух брошюр. Первая называется «Типы данных и программирование», вторая – «Обычная и специальная графика».

Оглавление

2.1 МНОГОМЕРНЫЕ МАССИВЫ И ДРУГИЕ ТИПЫ ДАННЫХ

В системе MatLAB используются несколько основных встроенных типов вычислительных объектов: `double` – числовые массивы и матрицы действительных или комплексных чисел с плавающей запятой; `logical` – логические массивы; `char` – массивы символов; `struct` – массивы записей (структуры); `cell` – массивы ячеек; `sparse` – двумерные действительные или комплексные разреженные матрицы. Имеется еще несколько встроенных типов в основном предназначенных для экономного хранения данных, а в пакетах расширения определено еще несколько дополнительных типов. Имеется возможность создавать собственные типы.

Тип `double` определяет наиболее распространенный класс переменных, с которыми оперирует большинство функций. Для типа

sparse предусмотрен экономный способ хранения данных (хранятся только ненулевые элементы). Для двумерных массивов обоих типов определен одинаковый набор матричных операций. Тип logical предназначен для хранения массивов, элементы которых могут принимать только два значения true и false. Тип char определяет переменные, которые являются совокупностью символов. Их принято называть строкой. Каждый символ занимает в памяти 16 битов. Переменные типа cell (ячейки) являются совокупностью массивов разных типов и размеров. Объекты типа struct состоят из нескольких составляющих, которые называются полями, каждое из которых носит собственное имя. Поля сами могут содержать массивы. Обычно структуры объединяют связанные по смыслу разнотипные данные.

Каждому типу данных соответствуют собственные функции и операторы. Здесь мы рассмотрим матрицы и числовые массивы типа double, а также часто используемые типы logical, char, cell и struct.

2.1.1 Вектора, матрицы и массивы

Все элементы данных в MatLab – это массивы. Числовые массивы по умолчанию имеют тип double. Одно число также считается массивом размерности 1 x 1. Матрица это двумерный числовой массив, для которого, помимо поэлементных операций, определены операции линейной алгебры.

Числа, матрицы и другие данные хранятся в переменных. Имя переменной (идентификатор) может содержать до 63-х символов. Оно не должно совпадать с именами функций системы. Имя должно начинаться с буквы, может содержать буквы, цифры и символ подчеркивания. Недопустимо включать в имена переменных пробелы и специальные знаки, например +, .(точка), *, / и т. д. MATLAB не допускает использование кириллицы в именах файлов и именах переменных.

Операции над числами MATLAB выполняет в формате двойной точности, вычисляя 16 значащих цифр. Однако в командном окне числа могут отображаться в различных видах. Для выбора способа представления числа используется функция **format.** Она меняет только представление чисел на экране, но не меняет сами числа.

Числа, вектора, матрицы – это массивы. Положение элементов массивов определяется индексами. Индексация в MATLAB начинается с единицы. Для многомерных массивов MATLAB поддерживает еще их одномерную индексацию, выбирая данные в постолбцовом порядке,

Числа – это матрицы размера 1 x 1. Для чисел определены все обычные арифметические операции. Вектора – это тоже матрицы размера 1 x n или n x 1. Кроме стандартных матричных операций для них определены операции скалярного произведения (функция **dot**) и векторного произведения (функция **cross**)

```
a=[1 2 3]; b=[4 5 6];
dot(a,b)
```

```
ans =
    32
```

Скалярное произведение можно использовать для определения длины вектора

c=[8 10 6 6 4 2]
d=sqrt(dot(c,c))

```
d =
    16
```

Векторное произведение **a** x **b** определяется для векторов состоящих из трех элементов. Результатом является тоже трехмерный вектор. Для вычисления векторного произведения в MATLAB служит функция **cross**

e=cross(a,b)

```
e =
    -3     6    -3
```

Смешанное произведение векторов (**a, b**, **c)** определяется формулой **(a,b,c)= a • b x c**, где точка обозначает скалярное произведение векторов.

c=[-1 1 2]; v=dot(a,cross(b,c))

```
v =
     3
```

В случае если имеются несколько векторов (строк или столбцов) одинаковой длины, их можно объединить в матрицу оператором объединения **[ ]**. Оператор **[ ]** может объединять и матрицы. Для команды горизонтального объединения **[A B]** требуется равенство количества строк матриц **A** и **B**

A=ones(3,2); B=magic(3); F=[A B]

```
F =
     1     1     8     1     6
     1     1     3     5     7
     1     1     4     9     2
```

А для команды вертикального объединения **[C;B]** требуется равенство количества столбцов матриц **C** и **B**

C=ones(2, 3); D=[C;B]

```
D =
     1     1     1
     1     1     1
     8     1     6
     3     5     7
     4     9     2
```

Оператор **[ ]** горизонтального объединения имеет функциональную версию **horzcat**. Например, команды **V=[A,B]** и **V= horzcat(A, B)** дают одинаковый результат. Оператор **[;]** вертикального объединения может употребляться в виде функции **vertcat**. Команды **V=[B;C]** и **V= vertcat(B,C)** дают одинаковый результат.

Для формирования таблицы значений можно использовать оба оператора

x=[0.1 1.7 1.6 0.5 1.9]; y=exp(x);
[x' y']
[x ; y]'

```
ans =
    0.1000    1.1052
    1.7000    5.4739
    1.6000    4.9530
    0.5000    1.6487
```

```
    1.9000    6.6859
```

Пустой массив задается символом **[ ]**. Он используется также для удаления элементов массивов. Например

B(2,:)=[ ]

```
B =
     8     1     6
     4     9     2
```

Заметим, что здесь мы не создали новый массив из первой и третьей строк матрицы **B**, а удалили из матрицы вторую строку, т.е. изменили исходную матрицу **B**.

В двумерной индексации **Z(n,m)** первый индекс – это номер строки, а второй – номер столбца массива **Z**. Индексы можно использовать как в левой, так и в правой части операции присваивания.

A=magic(4);

A(2,1)=1 (изменяем элемент 2,1 матрицы)

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

B = A - 8.5 (скаляр вычитается из каждого элемента матрицы)

B(1:2,2:3)=0 (обнуляем элементы указанного диапазона матрицы **B**)

```
B =
          7.5            0            0          4.5
         -3.5            0            0         -0.5
          0.5         -1.5         -2.5          3.5
         -4.5          5.5          6.5         -7.5
```

Матричные операции обозначаются стандартно. Для того чтобы отметить, что производится поэлементная операция, перед знаком операции ставится точка. Каждой матричной операции соответствует функция.

Операция	Функция	Действие
A + B	**plus(A,B)**	Сложение массивов одинаковой размерности
A - B	**minus(A,B)**	Вычитание массивов одинаковой размерности
A * B	**mtimes(A,B)**	Матричное умножение
B / A	**mrdivide(B,A)**	Матричное деление (правое) **B / A = B*inv(A)**
A \ B	**mldivide(A,B)**	Матричное деление (левое) **A \ B = inv(A)*B**
A'	**ctranspose(A)**	Комплексное сопряжение и транспонирование
A .* B	**times(A,B)**	Умножение элементов массивов **A(i,j)*B(i,j)**
A ./ B	**rdivide(A,B)**	Деление элементов массивов **A(i,j) / B(i,j)**
A .\ B	**ldivide(A,B)**	Деление элементов массивов **B(i,j) / A(i,j)**
A .'	**transpose(A)**	Транспонирование матрицы (без

		сопряжения)
A .^ B	**power(A,B)**	Поэлементное возведение в степень **A(i,j) ^ B(i,j)**
A ^ n	**mpower(A,n)**	Возведение матрицы **A** в числовую степень **n**

Операции вида **N*A** или **A+N**, где **N** – число, означают умножение всех элементов матрицы **A** на число **N** и, соответственно, прибавление числа **N** ко всем элементам матрицы **A**. В MATLAB принято, что скаляр расширяется до размеров другого операнда и заданная операция применяется к каждому элементу.

Поэлементное сложение и вычитание массивов такое же, как для матриц. Однако матричное умножение и возведение в степень отлично от поэлементного

A^2 % матричное возведение в степень

```
ans =
   345   257   281   273
   257   313   305   281
   281   305   313   257
   273   281   257   345
```

A.^2 % поэлементное возведение в степень

MatLab *использует точку для указания поэлементного умножения, деления и возведения в степень. Точка ставится перед знаком соответствующей операции.*

A .*A (поэлементное умножение)

```
ans =
   256     4     9   169
    25   121   100    64
    81    49    36   144
    16   196   225     1
```

A ./ A (поэлементное деление)

Операции над массивами (поэлементные операции) удобны для создания таблиц и матриц

n = (0:4)'; pows = [n n.^2 2.^n]

```
pows =
     0     0     1
     1     1     2
     2     4     4
     3     9     8
     4    16    16
```

Большинство математических функций MATLAB являются поэлементными, т.е. вычисляют значения функции для каждого элемента массива.

x = (1:0.1:1.5)'; logs = [x log10(x)]

```
logs =
            1            0
          1.1     0.041393
          1.2     0.079181
          1.3      0.11394
          1.4      0.14613
          1.5      0.17609
```

Но есть функции, которые допускают матричный аргумент, например матричная экспонента

$$e^A = I + A + \frac{1}{2!}A^2 + \ldots + \frac{1}{n!}A^n + \ldots$$

Матрица **A** должна быть квадратной. Обращение к функции с именем `fun` выполняется по правилу `funm(A,@fun)`. Например, матричный синус может быть вычислен так

A=magic(3); funm(A,@sin)

```
ans =
   -0.3850    1.0191    0.0162
    0.6179    0.2168   -0.1844
    0.4173   -0.5856    0.8185
```

В то время как поэлементное вычисление синуса дает

sin(A)

```
ans =
    0.9894    0.8415   -0.2794
    0.1411   -0.9589    0.6570
   -0.7568    0.4121    0.9093
```

Для матричной экспоненты, матричного логарифма и матричного квадратного корня имеются специальные функции **expm(A), logm(A), sqrtm(A).**

Отметим некоторые функции, относящиеся к матрицам.

Функция	Описание
det(A)	Определитель матрицы
inv(A)	Обратная матрица
[n,m] = size(A)	Размерности матрицы (к-во строк и столбцов)
S = length(A)	Максимальный размер матрицы, **S=max(size(A))**
trace(A)	След матрицы, сумма диагональных элементов, матрица может быть не квадратной
sum(A)	Вектор, состоящий из сумм элементов столбцов
mean(A)	Вектор, состоящий из средних значений элементов столбцов матрицы
prod(A)	Вектор, состоящий из произведений элементов столбцов
diag(A)	Вектор - столбец элементов главной диагонали матрицы **A**
diag(V)	Квадратная матрица с ненулевыми элементами вектора **V** на ее главной диагонали
triu(A)	Верхняя треугольная часть матрицы
tril(A)	Нижняя треугольная часть матрицы
poly(A)	Вектор коэффициентов характеристического полинома матрицы
jordan(A)	Жорданова форма матрицы
d=eig(A)	Вектор собственных чисел матрицы
[V,D]=eig(A)	Собственные векторы и собственные числа матрицы. Столбцы матрицы **V** есть собственные векторы единичной нормы, а матрица **D** – диагональная, с собственными значениями на диагонали.

rref(A)	приведение матрицы к ступенчатому виду

Приведем несколько примеров с использование описанных выше функций.

A=magic(4)

d = det(A) % определитель матрицы равен нулю

R = rref(A)

```
R =
     1     0     0     1
     0     1     0     3
     0     0     1    -3
     0     0     0     0
```

У матрицы **A** нет обратной. Попытка вычислить $\mathbf{A}^{-1}$ приводит к ошибке

X = inv(A)

```
Warning: Matrix is close to singular or badly scaled.
         Results may be inaccurate. RCOND = 1.306145e-017.
X = . . .
```

Изменим немного матрицу

C=A; C(1,1)=17

Теперь $\mathbf{C}^{-1}$ существует

inv(C)

```
ans =
    1.0000    3.0000   -3.0000   -1.0000
    3.0000   10.2794   -9.8088   -3.5294
   -3.0000  -10.1838    9.7426    3.5588
   -1.0000   -3.1544    3.1838    1.0294
```

eig(A)' % вычисление собственных значений

```
ans =
   34.0000    8.9443   -8.9443    0.0000
```

[V,D]=eig(A) % вычисление собственных векторов и чисел

```
V =
   -0.5000   -0.8236    0.3764   -0.2236
   -0.5000    0.4236    0.0236   -0.6708
   -0.5000    0.0236    0.4236    0.6708
   -0.5000    0.3764   -0.8236    0.2236
D =
   34.0000         0         0         0
         0    8.9443         0         0
         0         0   -8.9443         0
         0         0         0    0.0000
```

Одно из собственных значений равно 0, что является следствием сингулярности матрицы **A**. Из первого столбца матрицы **V** можем заключить, что любой вектор с одинаковыми координатами, является собственным. В частности вектор, составленный из единиц, является собственным. Проверим это

v=ones(4,1); (A*v)'

```
ans =
    34    34    34    34
```

Вектор коэффициентов характеристического полинома $\det(A-\lambda\cdot I)$ возвращает функция **poly**.

poly(A)

```
ans =
  1.0e+003 *
    0.0010   -0.0340   -0.0800    2.7200   -0.0000
```

Записать полином в символьном виде мы можем командой

syms x; r = poly2sym([1 - 34 - 64 217 0], x)

```
r =
x^4-34*x^3-64*x^2+217*x
```

Т.о. характеристический полином магической матрицы **A = magic(4)** равен

$$\det(A-\lambda\cdot I)=\lambda^4-34\lambda^3-64\lambda^2+2176\lambda.$$

Вычисление средних значений столбцов матрицы можно выполнить с помощью функции **mean**

A=magic(4); mean(A)

```
ans =
         8.5          8.5          8.5          8.5
```

mu = mean(A), sigma = std(A) (вычисление среднего значения и среднеквадратичного отклонения по столбцам).

Имеется большое количество команд создания векторов и матриц. Приведем команды создания некоторых стандартных матриц

- **zeros(n,m)** – матрица из нулей размера `n x m`;
- **ones(n,m)** – матрица из единиц размера `n x m`;
- **rand(n,m)** – матрица размера `n x m` из случайных чисел от 0 до 1;
- **eye(n)** – единичная матрица порядка `n`;
- **eye(n,m)** – матрица размера `n x m` из единиц на главной диагонали;
- **magic(n)** – магическая матрица размера n x n.

Команда

linspace(2,8,5)

```
ans =
     2    3.5    5   6.5   8
```

создает вектор с начальным значением 2, конечным значением 8, всего 5 членов арифметической прогрессии.

linspace(2,8,7)

```
ans =
     2     3     4     5     6     7     8
```

Функция **logspace(a,b,n)** создает вектор – строку элементов геометрической прогрессии, начиная от 10^a до 10^b всего **n** членов геометрической прогрессии.

logspace(0,4,5)

```
ans =
     1    10   100  1000  10000
```

Функция **eye(n)** создает единичную матрицу `n x n`.

eye(3)

```
ans =
     1     0     0
     0     1     0
     0     0     1
```

Функция **diag(V,k)** создает матрицу, большинство элементов которой равно нулю. Если смещение **k=0**, то элементы вектора **V** располагаются на главной диагонали матрицы. Если **k=1**, то элементы вектора располагаются выше главной диагонали, образуя верхнедиагональную матрицу. Если **k=-1**, то элементы вектора располагаются ниже главной диагонали, образуя нижнедиагональную матрицу.

diag([3,4,5],0)

```
ans =
     3     0     0
     0     4     0
     0     0     5
```

diag([3,4,5],1)

```
ans =
     0     3     0     0
     0     0     4     0
     0     0     0     5
     0     0     0     0
```

Смещение **k** может быть больше единицы или меньше минус единицы.

diag([3,4,5],-2)

```
ans =
     0     0     0     0     0
     0     0     0     0     0
     3     0     0     0     0
     0     4     0     0     0
     0     0     5     0     0
```

A=magic(4)

B=triu(A) % выделение верхней треугольной матрицы

C=tril(A) % выделение нижней треугольной матрицы

Из стандартных блоков можно создать новую матрицу

B=[triu(ones(2)), zeros(2,1); ones(1,3)]

```
B =
     1     1     0
     0     1     0
     1     1     1
```

Функция **reshape** реорганизовывает матрицу. Число элементов новой матрицы должно совпадать с числом элементов исходной.

A2=[1 2 3 4; 2 3 4 5; 3 4 5 6]

```
A2 =
     1     2     3     4
     2     3     4     5
     3     4     5     6
```

A3=reshape(A2,2,6)

```
A3 =
     1     3     3     3     5     5
     2     2     4     4     4     6
```

Многомерные массивы

Массивы с числом размерностей более двух считаются многомерными. Создать такие массивы можно, например, функциями **zeros**, **ones**, **rand**.

A=ones(2,3,2)

```
A(:,:,1) =                  A(:,:,2) =
     1     1     1              1     1     1
     1     1     1              1     1     1
```

r=rand(2,2,2) (массив 2 x 2 x 2 случайных чисел от 0 до 1)

```
r(:,:,1) =                  r(:,:,2) =
      0.58279      0.51551          0.43291      0.57981
       0.4235      0.33395          0.22595      0.76037
```

size(A) (возвращает размерности массива)

```
ans =
     2     3     2
```

Команда **size(A,n)** возвращает число строк (n=1) или число столбцов (n=2) матрицы **A**

A=[1 2 3 4; 2 3 4 5; 3 4 5 6]; size(A,2)

```
ans =
     4
```

Создать многомерный массив можно путем присваивания значений его элементам

A(:,:,1)=[1 2 3; 4 5 6; 7 8 9];
A(:,:,2)=[-1 -2 5; 0 -5 8; -7 -10 -12];
A

```
A(:,:,1) =                A(:,:,2) =
     1     2     3          -1    -2     5
     4     5     6           0    -5     8
     7     8     9          -7   -10   -12
```

Для формирования многомерных массивов можно использовать функцию **cat**. Она позволяет задать размещение двумерных массивов вдоль указанной размерности, используя синтаксис **B=cat(dim,A1,A2,…)**, где **dim** номер размерности, вдоль которой размещаются массивы, **A1, A2, …** список двумерных массивов.

A1=[1 2; 3 4];
A2=[4 5; 6 7];
B=cat(3,A1,A2)

```
B(:,:,1) =                B(:,:,2) =
     1     2                   4     5
     3     4                   6     7
```

A=cat(3,[9 -1; 5 4],[1 3; 0 2]);
B=cat(3,[3 6; 2 4],[1 1; -1 -2]);
D=cat(4,A,B,cat(3,A1,A2))

```
D(:,:,1,1) =              D(:,:,2,1) =
     9    -1                   1     3
     5     4                   0     2
D(:,:,1,2) =              D(:,:,2,2) =
     3     6                   1     1
     2     4                  -1    -2
D(:,:,1,3) =              D(:,:,2,3) =
     1     2                   4     5
     3     4                   6     7
```

Функция **ndims** возвращает количество размерностей массива

ndims(D)

```
ans =
     4
```

Функция **sum(A,d)** вычисляет суммы, изменяя индекс **d**.

sum(A,1) (то же, что просто **sum(A)**)

```
ans(:,:,1) =
    12    15    18
ans(:,:,2) =
    -8   -17     1
```

sum(A,2)

```
ans(:,:,1) =
     6
    15
    24
ans(:,:,2) =
     2
     3
   -29
```

sum(A,3)

```
ans =
     0     0     8
```

```
     4     0    14
     0    -2    -3
```

Логические данные и функции. В MatLab есть логический тип данных – `logical`.

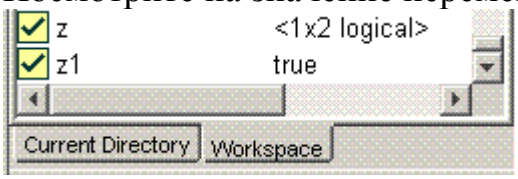
z1=1>0

```
z1 =
     1
```

z=1>0; z(2)=1<0

```
z =
     1     0
```

Посмотрите на значение переменных **z** и **z1** в окне Workspace.



Переменная **z1** имеет значение `true` (`истина`), а **z** является массивом `1 x 2` с типом `logical` (логический массив). Если посмотреть на значение **z** в командном окне, то увидим, что истинному условию в логическом массиве соответствует единица, а ложному – ноль.

Выражения, значение которых равняется `true` или `false`, называются логическими. Значению `true` соответствует логическая единица, а `false` – логический ноль. Логические выражения конструируются из операций отношения и логических операций

Операции отношения используют следующие знаки или эквивалентные им функции

Операция	Функция	Название
<	**lt**	меньше
<=	**le**	меньше или равно
>	**gt**	больше
>=	**ge**	больше или равно
==	**eq**	равно
~=	**ne**	не равно

Операции **==** и **~=** выполняют сравнение вещественных и мнимых частей комплексных чисел, а операции **>, >=, <, <=** – только вещественных частей.

Логические операции используют следующие знаки или эквивалентные им функции

Функция	Операция	Название
and	**&**	Логическое И
or	**|**	Логическое ИЛИ
not	**~**	Логическое НЕ
xor	отсутствует	Исключающее ИЛИ
any	отсутствует	`true`, если какой то элемент вектора или столбца матрицы не равен нулю
all	отсутствует	`true`, если все элементы вектора или столбца матрицы не равны нулю

```
a=1; b=2; c=-1; a==b
ans =
     0
a~=b
ans =
     1
ge(a,b)
ans =
     0
le(a,b)
ans =
     1
( a ~= b ) & (a>c)    % операция логического И
ans =
     1
a<c | a<b             % операция логического ИЛИ
not(a<c | a<b)        % функция отрицания
ans =
     1
ans =
     0
```

Результат логического выражения может быть присвоен переменной

```
a=-1; b=3; f = a <= b
f =
     1
```

Арифметические переменные могут использоваться в одном выражении с логическими. При этом арифметические операции всегда имеют приоритет по отношению к логическим. С точки зрения арифметических операций логические единица и ноль совпадают с арифметическими единицей и нулем (но не наоборот, арифметические единица и ноль не всегда совпадают с логическими единицей и нулем!). Например, выражения

```
2+(3>0)
ans =
     3
2+(3<0)
ans =
     2
```

не является ошибочными.

Поскольку числа в MatLab являются массивами (размера `1 x 1`), то естественно ожидать, что операции отношения можно использовать с массивами произвольного размера

```
A=[1 -2; -3 4]; B=A<0
B =
     0     1
     1     0
```

Посмотрите тип переменной **B** в окне Workspace – она являются логическим массивом размера `2 x 2`.

Операции отношения применимы к массивам одинакового размера. Происходит поэлементное сравнение и результатом является логический массив, состоящий из логических нулей и единиц, того же размера, что и исходные массивы. Единицы соответствуют тем элементам, для которых условие выполняется, а ноль означает невыполнение условия.

A=[1 2; -1 3]; B=ones(2,2);

A<B **A<=B** **A==B**

```
ans =             ans =             ans =
     0     0           1     0           1     0
     1     0           1     0           0     0
```

Отметим, что кроме поэлементной функции сравнения `eq` есть функция матричного сравнения `isequal`. Когда функция **eq(A,B)** сравнивает два массива, то она возвращает массив из логических единиц и нулей. А функция **isequal(A,B)** сравнивая два массива, возвращает только одно логическое значение (единицу, если все элементы массивов **A** и **B** совпадают по типу и значению и ноль – в противном случае)

A=[1 2; 3 4]; B=A;

eq(A,B)

```
ans =
     1     1
     1     1
```

isequal(A,B)

```
ans =
     1
```

isequal(A,ones(2,2))

```
ans =
     0
```

Часто полезна матричная функция `isempty`. Она возвращает логическую единицу, если массив пустой.

isempty(A)

```
ans =
     0
```

B=[ ]; isempty(B)

```
ans =
     1
```

но, если переменная еще не определена, то функция возвращает ошибку

clear A; isempty(A)

```
??? Undefined function or variable 'A'.
```

Функция **all** возвращает (одну) логическую единицу, если все элементы вектора ее аргумента не равны нулю

all(B)

```
ans =
     0
```

Если ее аргументом является матрица, то логическая единица или ноль возвращаются для каждого столбца

C=[1 2 3; 4 5 0; 0 2 5];

```
C =
     1     2     3
     4     5     0
     0     2     5
```

all(C)

```
ans =
     0     1     0
```

Логические массивы можно присваивать переменным

A=[1 2 3; 4 5 6; 7 8 9]; B=magic(3); C=A==B

```
C =
     0     0     0
     0     1     0
```

```
     0     0     0
```

D=A>B

```
D =
     0     1     0
     1     0     0
     1     0     1
```

Аргументами логических операторов могут быть не только логические единица и ноль, но также числа и строки. Числовой ноль воспринимается как логический, а любое отличное от нуля число воспринимается как логическая единица.

A=[1 2 3]; B=[1 0 0];

and(A,B)

```
ans =
     1     0     0
```

or(A,B)

```
ans =
     1     1     1
```

not(B)

```
ans =
     0     1     1
```

A & B

```
ans =
     1     0     0
```

A | B

```
ans =
     1     1     1
```

~B

```
ans =
     0     1     1
```

A = [1 3; -1 0]; B = [0 4; 8 8]

C=A & B

```
C =
     0     1
     1     0
```

D=A | B

```
D =
     1     1
     1     1
```

E=~A

```
E =
     0     0
     0     1
```

Для строк действует то же правило, только каждый символ строки представляется своим кодом

and('abc','efg')

```
ans =
     1     1     1
```

Функция «исключающее или» `xor`, сравнивает массивы одинакового размера. Если один из элементов входного массива не равен нулю, а соответствующий элемент другого равен, то `xor` записывает единицу на соответствующее место выходного массива. Во всех остальных случаях (ноль и ноль, не ноль и не ноль) `xor` возвращает логический ноль. Аргументами `xor` могут быть массив и число и, конечно, два числа.

A=[1 2 3]; B=[1 0 0]; xor(A,B)

```
ans =
     0     1     1
```

В MatLab определены однотипные операции **&**, **|** и **&&**, **||** . Логические операции **&** и **|** учитывают оба операнда для вычисления результата. Но значение логического выражения в ряде случаев определяется значением только первого операнда. Если первый операнд логического «или» является `истиной`, то результат всегда будет `истина`. Если первый операнд логического И – `ложь`, то результат – `ложь`. Операции **&&** и **||** отличаются от **&** и **|** тем, что в перечисленных двух ситуациях не проверяют значение второго операнда.

Приоритет выполнения операций следующий

- логические операторы **and, or, not, xor** (поскольку они являются функциями);
- отрицание **~**;

- транспонирование, возведение в степень (в том числе поэлементное), знак плюс или минус перед числом;
- Умножение и деление (в том числе поэлементное);
- Сложение и вычитание;
- Операции отношения: **>, >=, <, <=, == ;**
- Логическое И **& ;**
- Логическое ИЛИ **| ;**
- Логическое И **&& ;**
- Логическое ИЛИ **|| .**

Например, в соответствии с вышесказанным выражения **and(A, B)+C** и **A & B+C** не эквивалентны!

Логическое индексирование

Использование логических массивов весьма удобно в матричной операции, называемой логическим индексированием. Она имеет следующий формат

имя_матрицы (*логический_массив*),

где логический массив должен иметь тот же размер, что и матрица. Результатом этой операции является вектор, составленный из элементов матрицы, для которых в логическом массиве соответствующие элементы равны логической единице.

Пусть из матрицы **B** требуется выбрать все отрицательные элементы и записать их в вектор **f**.

B=[4 3 -1; 2 7 0; -5 1 2]

Сначала создадим логическую матрицу

ind=B<0

```
ind =
     0     0     1
     0     0     0
     1     0     0
```

Число ноль в логической операции, как и в арифметических операциях с матрицами, расширяется до размеров матрицы другого операнда. Использование логического массива **ind** в качестве единственного индекса исходного массива **B** позволяет решить поставленную задачу

f=B(ind)

```
f =
    -5
    -1
```

Можно было обойтись и без вспомогательного массива **ind**, написав сразу **f = B(B<0)**.

Аналогично

A=[1 2 3; 4 5 6; 7 8 9]; B=magic(3); A(A>B)'

```
ans =
     4     7     2     9
```

Таким образом, логическое индексирование позволяет выбрать из массива элементы, удовлетворяющие определенным условиям, которые заданы логическим выражением.

Если требуется присвоить новое значение элементам массива, удовлетворяющим определенному условию, то выражение **B(B < 0)** должно войти в левую часть оператора присваивания.

A=[1 2 3; 4 5 6; 7 8 9]; B=magic(3); C=A<=B

```
C =
     1     0     1
     0     1     1
     0     1     0
```

A(C)=0

```
A =
     0     2     0
     4     0     0
     7     0     9
```

В этом примере логический массив **C** был создан при выполнении операции сравнения **A<=B**. Однако было бы ошибкой считать, что для выделения нужных элементов достаточно просто создать массив из нулей и единиц и указать его в качестве индекса массива. Создайте, например, массив **C1** с теми же элементами, что и **C.**

C1=[1 0 1; 0 1 1; 0 1 0]

и попытайтесь использовать его для логического индексирования:

A(C1)

```
??? Subscript indices must either be real positive integers or logicals.
```

Получаем сообщение об ошибке. Выход состоит в преобразовании числового массива **C1** в логический массив **C2** при помощи функции **logical**, который затем можно использовать для индексирования

C2=logical(C1)

```
C2 =
     1     0     1
     0     1     1
     0     1     0
```

A(C2)'

```
ans =
     1     5     8     3     6
```

Убедитесь, что **C2** – логический массив, изучив информацию о нем в окне Workspace.

Логическое индексирование позволяет получить значения нужных элементов матрицы или вектора или изменить их, но индексы элементов остаются неизвестными. Для поиска индексов элементов, удовлетворяющих определенному условию, служит функция **find**.

Рассмотрим пример. Требуется найти номера всех элементов вектора, равных максимальному значению. Вызов функции **max** с двумя выходными аргументами не решает эту задачу, поскольку определяется только один элемент и его номер

x=[1 2 5 3 4 5 1 5];

[m,k]=max(x)

```
m =
     5
k =
     3
```

Но теперь мы знаем значение m максимального элемента, оно равно 5, и могли бы использовать логическое индексирование для записи всех максимальных значений

xm=x(x==5)

```
xm =
     5     5     5
```

но номера максимальных элементов все равно неизвестны. Вместо логического индексирования используем функцию f**ind**, указав во входном аргументе логическое выражение **x == 5**

km=find(x==5)

```
km =
     3     6     8
```

Функция **find** вернула номера элементов вектора, которые совпадают с максимальным значением.

Поиск номеров в матрице так же осуществляется при помощи функции **find**. Найдем, например, индексы всех неположительных элементов матрицы.

B=[4 3 -1; 2 7 0; -5 1 2]

Вызовем **find** с двумя выходными аргументами – векторами, в которые требуется записать номера строк и столбцов искомых элементов.

[i,j]=find(B<=0)

```
i =
     3
     1
     2
j =
     1
     3
     3
```

Действительно, все эти элементы **B(3,1), B(1,3), B(2,3)** меньше или равны нулю. К функции **find** можно обратиться и с одним выходным аргументом:

k=find(B<=0)

```
k =
     3
     7
     8
```

В этом случае вектор **k** содержит порядковые номера требуемых элементов матрицы с учетом одномерной схемы хранения элементов по столбцам матрицы.

Рассмотрим еще пример. Пусть, например, дан вектор

x=[2.5 1.3 1.6 1.5 NaN 1.9 1.8 1.5 3.1 1.8 1.1 1.8];

Функция **NaN** возвращает не число. Например, она представляет метку для недостающего наблюдения или заменяет ошибку при ответе на анкету. Функция **finite(x)** дает истину для всех конечных значений, иначе ложь (для **NaN** и **Inf**). Тогда

x = x(finite(x))

```
x =
2.5  1.3  1.6  1.5  1.9  1.8  1.5  3.1  1.8  1.1  1.8
```

Следующей командой мы отфильтровываем элементы, которые отличаются от среднего на значение среднеквадратичного отклонения.

x = x(abs(x-mean(x)) <= std(x))

```
x =
    1.3    1.6    1.5    1.9    1.8    1.5    1.8    1.8
```

Выполним еще несколько примеров. Обнулим элементы матрицы **A**, которые не являются простыми числами

A=magic(4); A(~isprime(A))=0

```
A =
     0     2     3    13
     5    11     0     0
     0     7     0     0
     0     0     0     0
```

A=magic(4); k = find(isprime(A))' %определяем индексы простых чисел

```
k =
     2     5     6     7     9    13
```

A(k) % выводим элементы, являющиеся простыми числами

```
ans =
     5     2    11     7     3    13
```

B=magic(4); k = find(isprime(B)); B(k) = -1 % заменяем простые числа на -1

```
B =
    16    -1    -1    -1
    -1    -1    10     8
     9    -1     6    12
     4    14    15     1
```

2.1.2 Символьные строки

В MatLab строки вводятся в одинарных кавычках

s = 'Hello'

В результате переменная **s** хранит символьный массив. Команда

a = double(s)

```
a =    72   101   108   108   111
```

преобразует символьный массив в числовой, содержащий коды символов. Команда

c=char(a)

```
c = Hello
```

выполняет обратное преобразование (числового вектора в строку). Коды прописных букв английского алфавита начинаются с номера 65 и идут в порядке возрастания, соответствуя порядку букв в алфавите

ac=[65 66 67 68 69 70 71 72]

```
ac =    65    66    67    68    69    70    71    72
```

char(ac)

```
ans = ABCDEFGH
```

В кодировке ASCII печатаемые символы имеют код от 32 до 127. Целые числа, меньшие 32, представляют непечатаемые символы.

Функция **reshape(A,m,n)** используется для отображения массива A в виде таблицы `m x n`. Сформируем таблицу чисел от 32 до 127

F = reshape(32:127,12,8)'

```
F =
 32  33  34  35  36  37  38  39  40  41  42  43
 44  45  46  47  48  49  50  51  52  53  54  55
 56  57  58  59  60  61  62  63  64  65  66  67
 68  69  70  71  72  73  74  75  76  77  78  79
 80  81  82  83  84  85  86  87  88  89  90  91
 92  93  94  95  96  97  98  99 100 101 102 103
104 105 106 107 108 109 110 111 112 113 114 115
116 117 118 119 120 121 122 123 124 125 126 127
```

Следующая команда сформирует таблицу символов, имеющих эти числа своими кодами

char(F)

```
ans =
 !"#$%&'()*+
,-./01234567
89:;<=>?@ABC
DEFGHIJKLMNO
PQRSTUVWXYZ[
\]^_`abcdefg
hijklmnopqrs
tuvwxyz{|}~□
```

Команда

char(F+1008)

```
ans =
АБВГДЕЖЗИЙКЛ
МНОПРСТУФХЦЧ
ШЩЪЫЬЭЮЯабвг
дежзийклмноп
рстуфхцчшщъы
ьэюя□ёђѓєsiї
jљњћќ□ўџ□□□□
□□□□□□□□□□□□
```

выводит таблицу символов с кодами от 1040 до 1135 . Как видим, коды символов русского алфавита попали в этот диапазон (зависит от кодовой таблицы, установленной на вашем компьютере). Чтобы узнать, какая кодовая страница установлена на вашем компьютере, следует выполнить команду

slCharacterEncoding

```
ans =
windows-1251
```

Оператор объединения (квадратные скобки [ ]) соединяет две строки в одну.

h=[s, ' world']

```
h =
Hello world
```

Если один из элементов в операторе объединения является строкой, а другие – целыми числами, то все конвертируется в строку (массив символов)

A=['Hello' 32 119 111 114 108 100]

```
A =
Hello world
```

Другая возможность объединить строки – использование функции **strcat(s1,s2,...sn)**. Она производит сцепление заданных строк **s1, s2, ... sn** в единую строку

s1=' You'; s2=' are'; s3=' beautiful.';
st = strcat(h,'.', s1, s2, s3)

```
st =
Hello world. You are beautiful.
```

Функция **strcat** отличается от оператора объединения **[ ]** тем, что отбрасывает хвостовые пробелы своих аргументов.

s1='Мир '; s2='прекрасен.';
ss1=[s1 s2]

```
ss1 =
Мир прекрасен.
```

ss2=strcat(s1, s2)

```
ss2 =
Мирпрекрасен.
```

Команда

v = [s; 'world']

```
v =
Hello
World
```

создает текстовую переменную, состоящую из двух строк. Результат во всех случаях является массивом символом. Но переменная **h=[s, ' world']** является массивом `1 x 11`, а переменная **v** – массивом `2 x 5` символов.

Отметим, что вертикально объединяемые строковые переменные должны быть одинаковой длины.

w = [s; 'world123']

```
??? Error using ==> vertcat ...
```

Другой способ объединить строки символов в массив из нескольких отдельных строк состоит в использовании функции вертикальной конкатенации **strvcat**. Она дополняет короткие строки пробелами, чтобы результирующий объект стал матрицей символов.

stv = strvcat(s1,s2)

```
stv =
Мир
прекрасен.
```

Отдельная символьная строка представляет собой вектор-строку, элементами которой являются отдельные символы, включая символы пробелов. Поэтому информацию о любом символе в строке можно получить, указав номер этого символа от начала строки. Аналогично можно обращаться к элементам массива из отдельных строк, используя два индекса

Например

st(7)

```
ans =
w
```

stv(2,4)

```
ans =
к
```

st(7:11)

```
ans =
world
```

stv(2,:)

```
ans =
прекрасен.
```

Есть еще два способа сформировать массив строк. Функция **char(…)** принимает любое число строк, добавляет пробелы и формирует массив строк.

S = char('MATLAB' , 'очень' , 'крутая' , 'программа.')

```
S =
MATLAB
очень
крутая
программа.
```

Другой способ – это хранение массива строк в массиве ячеек (о них мы поговорим в следующем параграфе)

C = {'Привет' ; 'мир.' ; 'Ты' ; 'прекрасен' }

```
C =
    'Привет'
    'мир.'
    'Ты'
    'прекрасен'
```

Преобразовать уже имеющийся массив строк в массив ячеек можно с помощью функции **cellstr**.

cellstr(S)

Обратное преобразование выполняет функция **char**.

char(C)

Вообще то основное назначение функции **char** преобразовывать код символа или нескольких символов в сам символ или строку символов так, как это было показано выше

char(65)

```
ans =
A
```

При работе со строками возникают задачи замены части строки, вставки куска одной строки в другую, вставки числового значения в середину строки и т.д. Для их решения имеются специальные функции. Некоторые из них мы здесь опишем.

Процедура **strrep(s1, s2, s3)** формирует строку из строки **s1** путем замены всех ее фрагментов, которые совпадают со строкой **s2** на строку **s3**

st1='Это '; st2='строка '; st3='символов.';

st = [st1 st2 st3]

```
st =
Это строка символов.
```

y = strrep(st,'о','у') % заменяет вхождение всех символов 'о' на 'у'

```
y =
Эту струка симвулув.
```

x = strrep(st,'с','н')

```
x =
Это нтрока нимволов.
```

Функция **upper(str)** переводит все символы строки **str** в верхний регистр

z1 = upper(st)

```
z1 =
ЭТО СТРОКА СИМВОЛОВ.
```

Функция **lower(str)** переводит все символы строки **str** в нижний регистр

z2 = lower(z1)

```
z2 =
это строка символов.
```

Функция **findstr(st,st1)** выдает номер элемента строки **st**, с которого начинается первое вхождение строки **st1**

findstr(st,'сим')

```
ans =
   12
```

Чтобы вставить в строку символов числовое значение его вначале можно преобразовать в строку, а затем конкатенировать с другими строками. Такое преобразование выполняет функция **num2str**. Ее входным аргументом является числовая переменная, а выходным – символьная строка. Формат представления задается вторым аргументом функции. Он задается в виде строки, начинающейся с символа % (процент) и заканчивается преобразующим символом **f, g, e** и другими. Между ними могут стоять числа, определяющие ширину поля вывода и точность. Подробно форматы вывода описаны на странице справки функции **fprintf**.

num2str(pi)

```
ans =
3.1416
```
num2str(pi,'%16.14f')
```
ans =
3.14159265358979
```
Аргументом функции **num2str** может быть матрица

x = rand(2,3) * 10 % Create a 2-by-3 matrix
```
x =
   4.1027   0.5789   8.1317
   8.9365   3.5287   0.0986
```
num2str(x,'%12.6f')
```
ans =
4.102702    0.578913    8.131665
8.936495    3.528681    0.098613
```
Аналогичным образом, при помощи функции **mat2str(A)** можно получить значение матрицы **A** в виде символьной строки

A=[1 2 3
4 5 6
7 8 9]
```
A =
     1     2     3
     4     5     6
     7     8     9
```
Astr=mat2str(A)
```
Astr =
[1 2 3;4 5 6;7 8 9]
```
На первый взгляд **Astr** – это числовая матрица. Однако это не так. На самом деле это строка, состоящая из квадратных скобок, чисел, пробелов и точек с запятой. Всего 19 символов. Для проверки выполним команду сложения единицы с переменной **Astr**

Astr+1
```
ans =
  92  50  33  51  33  52  60  53  33  54  33  55  60  56  33  57  33  58  94
```
Полученный вектор не похож на результат сложения матрицы с единицей. Это объясняется тем, что строка символов **Astr** при включении ее в арифметическую операцию автоматически перестраивается в числовой вектор, т. е. все символы, составляющие ее, заменяются целыми числами. После этого сложение полученного числового вектора с числом 1 происходит по обычным правилам, т. е. единица суммируется с каждым из кодов символов.

Обратной к функциям **num2str** и **mat2str** является функция **str2num**. Она преобразует строку символов в число, вектор или матрицу.

str2num(Astr)
```
ans =
     1     2     3
     4     5     6
     7     8     9
```
Функция **str2mat(st1,st2,... ,stn)** действует аналогично функции **strvcat,** т. е. образует символьную матрицу, располагая строки **st1, st2, ... ,stn** одна под другой, дополняя их, если нужно, пробелами, чтобы количество символов во всех строках было одинаковым

st1='Это '; st2='строка '; st3='символов.';
B=str2mat(st1,st2,st3)

```
B =
Это
строка
символов.
```
B(1,3)
```
ans =
о
```
B(1,9)
```
ans =
```
% возвращается пробел

B(1,10)
```
??? Index exceeds matrix dimensions.
```

Строковые выражения обычно не вычисляются. Однако строка, представляющая математическое выражение, может быть вычислена с помощью функции `eval('строковое выражение')`. Например,

eval('2*sin(pi/3)+(1/3)^(1/5)')
```
ans =
      2.5348
```
Еще один пример. Сначала задаются значения переменных, а затем вычисляется символьное выражение, содержащее эти переменные,

a=2; b=4;
eval('a^2 – sqrt(b) + a*b – a/b')
```
ans =
      9.5000
```

2.1.3 Структуры

Структуры – это многомерные массивы MATLAB с элементами, доступ к которым выполняется через поля. Структуры, как и переменные других типов в MatLAB, не объявляются. Они создаются автоматически при задании конкретных значений полям записи. Например

S.name = 'Николай';
S.syrname='Иванов';
S.age = 53;
S.salary = 2500;
S
```
S =
       name: 'Николай'
    syrname: 'Иванов'
        age: 53
     salary: 2500
```
Структуры, как и все в MatLab, являются массивами. Поэтому для формирования массива из двух элементов у идентификатора существующей структуры достаточно поставить индекс второго элемента.

S(2).name = 'Петр';
S(2).syrname='Шевченко';
S(2).age = 33;
S(2).salary = 1500;
S
```
S =
1x2 struct array with fields:
    name
    syrname
    age
```

```
    salary
```
Добавление элементов в массив структур можно также выполнить следующим способом

S(3) = struct('name', 'Геннадий','syrname','Матвиенко','age', 40, 'salary', 2100)
```
S =
1x3 struct array with fields:
    name
    syrname
    age
    salary
```
В случае массива структур, содержимое полей уже не выводится на экран. Отображается лишь информация о структуре массива, его размерах и именах полей.

Если к какому-либо из элементов массива записей (структуры) добавляется значение нового поля, то же поле автоматически появляется во всех остальных элементах, хотя значение этого поля у других элементов при этом остается пустым.

S(1).SecName='Сергеевич';
S(3).SecName
```
ans =
     [ ]
```
Чтобы удалить некоторое поле из всех элементов массива структур, надо использовать процедуру **rmfield** по схеме **S = rmfield (S, 'имя поля ')**, где **S** – имя массива структур, который корректируется.

S=rmfield(S,'SecName')
```
S =
1x3 struct array with fields:
    name
    syrname
    age
    salary
```
Следующая команда выводит значения заданного поля всех элементов массива структур

S.age
```
ans =
    53
ans =
    33
ans =
    40
```
Однако удобнее использовать оператор объединения [ ] (квадратные скобки)

A=[S.age]
```
A =
    53    33    40
```
A(2)
```
ans =
    33
```
Из всех значений заданного поля можно сконструировать массив ячеек

{S.name}
```
ans =
    'Николай'    'Петр'    'Геннадий'
```
Функция **char** создает массив строк из всех значений указанного поля

char(S.name)
```
ans =
Николай
```

```
Петр
Геннадий
```
Если вы хотите узнать только имена полей массива структур, то можно использовать функцию **fieldnames** с одним аргументом – именем массива структур

fieldnames(S)
```
ans =
    'name'
    'syrname'
    'age'
    'salary'
```
Поле структуры может само включать другую структуру

P.name='Массив';
msize.first=2;
msize.second=3;
P.size=msize
```
P =
    name: 'Массив'
    size: [1x1 struct]
```
массив или матрицу

P.arr=[1 2; 3 4]
```
P =
    name: 'Массив'
    size: [1x1 struct]
     arr: [2x2 double]
```
или даже массив структур

ars(1).f='f1'; ars(1).g='g1'; ars(2).f='f2'; ars(2).g='g2';
ars
```
ars =
1x2 struct array with fields:
    f
    g
```
P.ars=ars
```
P =
    name: 'Массив'
    size: [1x1 struct]
     arr: [2x2 double]
     ars: [1x2 struct]
```
Если структура уже существует, то с помощью операторов присваивания или функции **struct** можно вложить другие структуры в существующие поля

P.name=struct('name', 'Геннадий','syrname','Матвиенко','age', 40, 'salary', 2100);
P.name
P

2.1.4 Массивы ячеек

Массив ячеек - это массив, элементами которого есть другие массивы. Эти элементы могут быть массивами различных типов, в том числе быть другими массивами ячеек. Например, одна из ячеек может содержать матрицу действительных чисел, вторая – массив символьных строк, третья – вектор комплексных чисел.

Массивы ячеек создаются путем заключения разнообразных объектов в фигурные скобки

A={3,[1 2 3],[1 2 3; 4 5 6; 7 8 9],'Hello'}
```
A =
```

```
    [3]    [1x3 double]    [3x3 double]    'Hello'
```

Конструктор **{ }** действует подобно оператору **[ ]** для числовых массивов. Он объединяет данные в ячейки.

Для хранения матриц одинакового размера может использоваться трехмерный массив, а для хранения матриц различного размера – массив ячеек.

```
M{1}=magic(4);
M{2}=magic(3);
M{3}=[1 2 3 4 5];
M{4}=[1 2 3; 4 5 6];
M
M =
    [4x4 double]
    [3x3 double]
    [1x5 double]
    [2x3 double]
```

В п. 2.1.2 мы видели, что создание массива строк предполагает выравнивание длины строк путем дополнения их пробелами. Там же мы видели, что есть способ хранения строк в массиве ячеек (без выравнивания длины строк)

```
C={'One','Three','Five'}
C =
    'One'    'Three'    'Five'
```

Функция **char** преобразует массив ячеек в массив строк

```
CC=char(C)
CC =
     One
     Three
     Five
```

При этом, функция **char** выравнивает длину строк. Это можно увидеть, затребовав данные о размерностях массива

```
size(CC)
ans =
     3     5
```

Массивы ячеек могут быть любой размерности, а элементы иметь различный тип

```
C{1,1}='Hello';
C{1,2}=[1 2; 3 4];
C{2,1}=3+4*i;
C{2,2}=pi;
C
C =
    'Hello'               [2x2 double]
    [3.0000 + 4.0000i]    [    3.1416]
```

Из примеров видно, что MatLAB отображает массив ячеек в сокращенной форме. Чтобы отобразить содержимое ячеек, нужно применить функцию **celldisp**

```
celldisp(C)
C{1,1} =
Hello
C{2,1} =
   3.0000 + 4.0000i
C{1,2} =
     1     2
```

```
     3     4
C{2,2} =
    3.1416
```

Извлечение содержимого отдельных элементов определенной ячейки, если она является массивом ячеек, производится дополнительным указанием в фигурных скобках индексов элемента

C{1,2}

```
ans =
     1     2
     3     4
```

C{1,2}(2,1)

```
ans =
     3
```

или

C{1,1}

```
ans =
     Hello
```

C{1,1}(1:3)

```
ans =
     Hel
```

или

A={3,[1 2 3],[1 2 3; 4 5 6; 7 8 9],'Hello'}; B2=A{2}

```
B2 =
     1     2     3
```

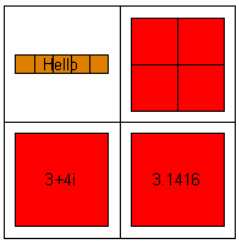
B3=A{3}

```
B3 =
     1     2     3
     4     5     6
     7     8     9
```

Здесь мы видим, что содержимое второго элемента массива ячеек **A** является вектор **[1 2 3],** а третьего – матрица **[1 2 3; 4 5 6; 7 8 9].**

Для отображения структуры массива ячеек в виде графического изображения предназначена функция **cellplot**

cellplot(C)



Функция **cell** позволяет создать шаблон массива ячеек, заполненный пустыми ячейками

M=cell(1,4)

```
M =
     [ ]     [ ]     [ ]     [ ]
```

Заполним одну из ячеек, используя оператор присваивания

M{2}='Hello'

```
M =
     [ ]    'Hello'     [ ]     [ ]
```

Таким же способом можно заполнить остальные ячейки.

Для образования массива ячеек можно использовать привычные операторы горизонтального и вертикального объединения **[ ]**. Например, следующая команда создает массив 2 x 2 ячеек строк символов

B=[{'МатАнализ'}, {'Геометрия'}; {'Алгебра'}, {'ДифУры'}]

```
B =
    'МатАнализ'    'Геометрия'
    'Алгебра'      'ДифУры'
```

Используя индексацию в массиве ячеек (круглые скобки), можно получить доступ к подмножествам ячеек внутри массива ячеек. Результат будет массивом ячеек. Например, следующая команда выбирает первую строку в массиве ячеек.

B(1,:)

```
ans =
    'МатАнализ'    'Геометрия'
```

Использование с именем массива ячеек индекса в круглых скобках **C(j)** определяет отдельную ячейку (точнее массив 1 x 1 ячеек)

C={3,[1 2 3],[1 2 3; 4 5 6; 7 8 9],'Hello'}

```
C =
    [3]    [1x3 double]    [3x3 double]    'Hello'
```

C(2)

```
ans =
    [1x3 double]
```

C{2}

```
ans =
     1     2     3
```

C(2:3)

```
ans =
    [1x3 double] [3x3 double]
```

C{2:3}

```
ans =1     2     3
ans =
     1     2     3
     4     5     6
     7     8     9
```

Повторим еще раз, индекс в фигурных скобках **C{ j }** обозначает содержимое соответствующей ячейки, а в круглых **C(j)** – ячейку (массив ячеек 1 x 1) с соответствующим содержимым.

D={[1 2 3],[4 5 6 7 8]} % здесь **D{1}, D{2}** есть векторы – содержимое

```
D =
    [1x3 double]    [1x5 double]
```

Следующая операция объединяет **D{1}, D{2}** в один вектор

E=[D{:}]

```
E =
     1     2     3     4     5     6     7     8
```

Символ **{ }** соответствует пустому массиву ячеек точно так же, как **[ ]** соответствует пустому числовому массиву. Используя **[ ]** можно удалить ячейки из массива. При этом, удалять можно либо целую строку, либо столбец. Например, следующая команда удаляет первую строку

B=[{'МатАнализ'},{'Геометрия'};{'Алгебра'},{'ДифУры'}]; B(1, :)=[ ]

```
B =
    'Алгебра'    'ДифУры'
```

Массив ячеек стал одномерным. Команда

B(2)=[ ]

```
B =
    'Алгебра'
```

удаляет вторую ячейку.

Фигурные скобки **{ }** являются конструктором массива ячеек, а квадратные **[ ]** – конструктором числового массива. Фигурные скобки аналогичны квадратным скобкам, за исключением того, что они могут быть еще и вложенными.

Система MATLAB не очищает массив ячеек при выполнении оператора присваивания. Могут остаться старые данные в незаполненных ячейках. Полезно удалять массив перед выполнением оператора присваивания командой **clear имя**.

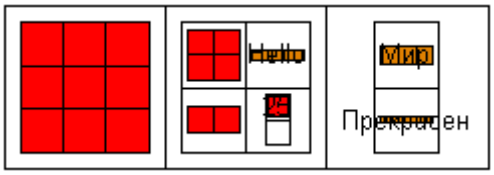
Так же, как и в случае числового массива, если данные присваиваются ячейке, которая находится вне пределов текущего массива, MATLAB автоматически расширяет массив ячеек. При этом ячейки, которым не присвоено значений, заполняются пустыми массивами.

Допускается, что ячейка может содержать массив ячеек и даже массив массивов ячеек. Массивы, составленные из таких ячеек, называются вложенными. Сформировать вложенные массивы ячеек можно с помощью последовательности фигурных скобок, функции **cell** или операторов присваивания

```
clear A
A(1, 1) = {magic(3)};
A(1, 2) = {{[1 2; 3 4] 'Hello'; [1-4i 1+i] {25 ; [ ]}}}
A(1, 3) = {'Мир'}
A =
    [3x3 double]    {2x2 cell}    'Мир'
```

Можно переопределить содержимое ячейки

```
A(1, 3) = {{'Мир';'Прекрасен'}}
A =
    [3x3 double]    {2x2 cell}    {2x1 cell}
cellplot(A)
```



Заметим, что в правой части оператора присваивания **A(1, 2) = {{[1 2; 3 4] 'Hello'; [1-4i 1+i] {25 ; [ ]}}}** использовано 3 пары фигурных скобок: первая пара определяет ячейку **A(2, 1)**, вторая пара говорит, что ее содержимым является массив ячеек размера 2 x 2, а третья – говорит, что элемент (2, 1) этого массива ячеек сам является ячейкой **{25 ; [ ]}**.

Для образования многомерного массива ячеек можно использовать функцию **cat**.

```
A{1,1}='Вася'; A{1,2}='Петренко'; A{2,1}=123; A{2,2}=4-i;
B{1,1}='Коля'; B{1,2}='Сидоров'; B{2,1}=4321; B{2,2}=logical([1 0;0 1]);
C=cat(3,A,B)
C(:,:,1) =
    'Вася'    'Петренко'
    [ 123]    [4.0000 - 1.0000i]
C(:,:,2) =
    'Коля'    'Сидоров'
    [4321]    [2x2 logical]
```

В заключение заметим, что для того, чтобы установить, какому типу принадлежит тот или другой вычислительный объект, к имени этого объекта следует применить функцию **class**

```
x=pi; class(pi)
ans =
      double
st='Hello'; class(st)
ans =
      char
S.name='Peter'; S.age=52; class(S)
ans =
      struct
class(C)
ans =
      cell
```

2.2 ОСНОВЫ ПРОГРАММИРОВАНИЯ В MATLAB

Программы MATLAB состоят из последовательности команд, записанных в текстовые файлы с расширением *.m. Поэтому их часто называют m – файлами. Есть два типа таких файлов. В части 1 п. 1.2 мы говорили о файлах – сценариях и файл – функциях. Если требуется, то прочитайте этот параграф еще раз. Здесь мы поговорим о некоторых особенностях создания сценариев, файл – функций и их разновидностях, а также об управляющих конструкциях языка программирования MatLab.

2.2.1 Файлы – сценарии.

Сценарий – это файл, составленный из команд MatLab. Когда он вызывается (по имени файла), MatLab просто выполняет команды, содержащиеся в файле. Сценарии берут данные из рабочего пространства и создают переменные, которые остаются в рабочем пространстве.

Команды в файле отделяются друг от друга так же как и в командном окне – символами новой строки или знаками **;** (точка с запятой).

При запуске m-файла на выполнение в рабочем пространстве могут храниться результаты предыдущих вычислений, причем имена переменных и массивов могут совпасть с именами переменных или матриц запускаемого m-файла, что при определенном стечении обстоятельств может привести к неверному результату производимых вычислений. Это возможно, поскольку все переменные и массивы, используемые в файлах - сценариях, являются глобальными. Во избежание подобных нежелательных моментов в начале файлов – сценариев желательно произвести очистку рабочего пространства от результатов предыдущих вычислений, закрыть все графические окна (если они были ранее открыты) и очистить экран от ранее выведенной информации. Поэтому вначале любого сценария рекомендуем выполнять следующие команды

```
clear all        % очистка рабочего пространства
close all        % Закрытие всех графических окон
clc              % Очистка командного окна
```

После этого можно задать некоторые исходные данные с клавиатуры с помощью функции input. Она выводит в командное окно текст приглашения, позволяет пользователю ввести с клавиатуры произвольный набор символов и после нажатия клавиши Enter записывает введенные символы в специальную переменную ans. Ее значение затем можно присвоить другой переменной.

```
input('Введите первое значение: ');
x0=ans;
input('Введите второе значение: ');
x2=ans;
...
```

Можно вводит и код функции как строку.

```
input('Введите функцию в одинарных кавычках: ');
Введите функцию в одинарных кавычках: 'x.^2-2*x'
```

Здесь в ответ на приглашение мы ввели строку **'x.^2-2*x'** . Затем запоминаем эту строку в переменную f.

```
f=ans
```

Позже в любом месте сценария из строковой переменной f можно создать inline – функцию

```
F=inline(f,'x')
F =
     Inline function:
     F(x) = x.^2 -2*x
```

с которой можно работать как с обычной функцией, например, вычислять ее значение в точке или строить график. Впрочем, вычислять значение можно и, не создавая inline – функции, используя функцию **eval('строковое_выражение').** Например

```
x=4; eval(f)
ans =
     8
```

Досрочный выход из программы может быть выполнен командой **return**.

Приведем пример сценария, в котором ищется корень полинома на отрезке [a,b] методом половинного деления. Обратите внимание на функции input. Управляющие конструкции if и while будут описаны в п. 2.2.3

```
% сценарий определения корня полинома на отрезке [a,b]
% методом половинного деления
input('Введите a: ');
a = ans;
fa = -Inf;
input('Введите b: ');
b = ans;
fb = Inf;
input('Введите функцию в одинарных кавычках: ');
F=ans;
while b-a > 0.0001
    x = (a+b)/2;
    fx = eval(F);
    if fx ==0
        break
    elseif sign(fx) == sign(fa)
        a = x; fa = fx;
    else
        b = x; fb = fx;
```

```
    end
end
x
```

Использование сценария `scrFindRootHalf.m` возможно следующим образом

scrFindRootHalf

```
Введите a: 0
Введите b: 3
Введите функцию в одинарных кавычках: 'x.^3-x.^2'
x =
    1.0000
```

2.2.2 Файл – функции

Функции оформляются в отдельных m – файлах. Имена функций должны совпадать именами этих файлов. При первом обращении к функции MatLab ищет ее на диске по имени файла. После того как программа найдена, производится ее синтаксический анализ. Если в его результате обнаружены ошибки, то сообщение выводится в командное окно. В случае успеха, создается псевдокод, который загружается в рабочее пространство и выполняется. При повторном обращении к функции она будет найдена в рабочем пространстве и поэтому синтаксический анализ не выполняется, и функция выполняется быстрее. Удалить псевдокод из рабочего пространства можно командой

clear *имя_функции*

После завершения работы функции ее выходные значения копируются в переменные вызывающей команды. Количество фактических выходных переменных должно быть не больше количества формальных, но может быть меньше.

Оформление алгоритма в виде одной функции не всегда удобно, а разбрасывание его по функциям, созданным в разных файлах, также не всегда желательно. Для решения этой проблемы в MatLab есть несколько способов – создание приватных функций, подфункций и вложенных функций.

Подфункции. В одном файле можно поместить определения сразу нескольких функций. Первая из функций является основной, и вызывать из командного окна можно только ее. Остальные функции называются подфункциями. Они могут вызываться основной и другими вспомогательными функциями текущего m – файла.

Основная функция может быть только одна. Заголовок подфункции одновременно является признаком конца основной функции или предыдущей подфункции. Основная функция обменивается информацией с подфункциями только при помощи входных и выходных параметров. Переменные, определенные в подфункциях и в основной функции, являются локальными, они доступны в пределах своей функции.

Структура m – файла с подфункциями имеет следующий вид

function […]=main(…)

```
    a=…;

function […]=sub1(…)

    a=…;

function […]=sub2(…)

    a=…;
```

Видимость переменных распространяется только на тело своей функции. Так в выше приведенном примере переменная с именем **a** в основной функции и ее подфункциях различна.

Один из возможных вариантов использования переменных, которые являются общими для всех функций М-файла, состоит в объявлении переменных в начале основной функции и подфункциях глобальными, при помощи **global** со списком имен переменных, разделенных пробелом.

Вложенные функции. Другой разновидностью функций, доступных в одном М-файле, являются вложенные функции. Если подфункция является внешней по отношению к основной функции, то вложенная функция является внутренней. В силу этого обстоятельства переменные из рабочей среды основной функции доступны во вложенной функции.

При написании вложенных функций следует использовать оператор **end** для закрытия тела функции. Поэтому вложенная функция может размещаться в любом месте тела содержащей ее функции. Основная функция также завершается оператором **end**. В одном М-файле допускается использование подфункций и вложенных функций одновременно, но тогда последним оператором подфункции должен быть **end**.

Уровень вложенности функций не ограничен. Функция может обратиться к своей вложенной функции, но не может использовать вложенную функцию нижнего уровня.

Структура m – файла с вложенными функциями имеет следующий вид

```
function […]=main(…)

    a=…;

    function […]=sub(…)

        c=…;

        function […]=subsub(…)

        end;

    end;
```

```
    function […]= sub2(…)

        c=…;

    end;

end;
```

Область видимости переменных основной функции распространяется на все вложенные в нее функции, а также вложенные во вложенные функции. В приведенном выше примере во вложенных функциях **sub, subsub** и **sub2** есть доступ к переменной **a** из основной функции. Во вложенной функции **subsub** есть доступ к переменной **c** из **sub**. Но переменные с именем **c** в функции **sub** и **sub2** различны.

Приватные функции. *Приватные функции* – это функции, размещенные в каталоге с именем `private`. Эти функции доступны только из функций, расположенных в этом и его родительском каталоге. Каталог `private` не следует указывать в путях доступа.

Допускается использование функций и переменных с одинаковыми именами. Но тогда следует учитывать следующий порядок выбора системой таких имен:

- имя ищется в текущей рабочей среде (переменная с этим именем, вложенная функция, анонимная или `inline` функция);
- ищется подфункция в текущем m – файле;
- ищется приватная функция;
- ищется встроенная функция MatLab;
- производится поиск файл – функции в текущем каталоге и путях поиска MatLab.

Чтобы узнать, является ли функция встроенной, можно использовать функцию **exist(имя_функции).** Она возвращает числовое значение: 1 – если имя занято под переменную рабочей среды, 2 – функция расположена в путях поиска файлов, 5 – встроенная функция. Приватные функции и подфункции не идентифицируются.

Функции от функций. Иногда возникает необходимость в качестве аргументов одной функции использовать другие функции. Например, функция численного интегрирования **quad** первым аргументом принимает имя функции, интеграл которой вычисляется. MatLab предоставляет возможности создания функций, аргументами которых выступают другие функции.

В MatLab любая функция, например, с именем FUN1, может быть выполнена не только с помощью обычного обращения **FUN1(x1,x2,... ,xn),** а и при помощи специальной процедуры **feval('FUN1',x1,x2,... ,xn),** где имя функции `FUN1` является одним из входных аргументов (текстовым).

x=pi/2; feval('sin',x)
```
ans =
     1
```
Первый входной аргумент **feval** может быть указателем на inline-функцию или быть анонимной функцией
feval(@sin,x)
```
ans =
     1
```
или
feval(@prod,[1 2 3])
```
ans =
      6
```
В общем случае все входные аргументы исследуемой функции задаются в списке аргументов **feval** через запятую после имени функции. Например, следующие три вызова некоторой функции `myfun` эквивалентны
a = myfun(x, y, z)
a = feval('myfun', x, y, z)
a = feval (@myfun, x, y, z)
Так как при вызове функции с помощью процедуры **feval** имя функции рассматривается как текстовый аргумент, то его (имя функции) можно использовать как переменную или оформлять как аргумент функции, вызывающей внутри себя передаваемую функцию.

Вот пример упрощенного варианта функции, вычисляющей определенный интеграл $\int_a^b f(x)\,d\,x$.

```
function s=rectint(fcn,a,b, N)
% RECTINT(FCN,A,B)
% вычисление определенного интеграла методом прямоугольников
% fcn - имя функции (строка), a, b - пределы интегрирования
% N - количество отрезков разбиения (необязательный аргумент)
if nargin<3
    error('Число аргументов не может быть меньше трех');
elseif nargin==3
    dlt=(b-a)/100;
else
    dlt=(b-a)/N;
end;
x=a:dlt:b;
y=feval(fcn,x);
y1=y;
y2=y;
y1(1)=[];
y2(end)=[];
z=(y1+y2)/2;
s=sum(z)*dlt;
```

rectint('sin',0)
```
??? Error using ==> rectint
Число аргументов не может быть меньше трех
```
rectint('sin',0,pi)
```
ans =
    1.9998
```
rectint('sin',0,pi,1000)
```
ans =
    2.0000
```
quad('sin',0,pi)

```
ans =
    2.0000
```

Отметим отличие функций **eval** и **feval**. Первая принимает строковое выражение, которое следует вычислить, а вторая – имя функции в виде строки, значение которой следует вычислить.

Рекурсивные функции. MatLab допускает рекурсивный вызов функций. Например, следующая функция вычисляет факториал целого положительного числа

```
function z=fact(n)
% вычисление факториала числа n
if n==1
    z=1;
else
    z=n*fact(n-1);
end
```

fact(5)

```
ans =
   120
```

Алгоритмы некоторых стандартных функций MatLab являются рекурсивными. Например, функции **quad** и **quadi** имеют открытый код, и вы можете самостоятельно изучить их. Они находятся в подкаталоге `\toolbox\matlab\funfun\` основного каталога MATLAB.

Функции с переменным числом аргументов. Большинство стандартных функций MATLAB допускают обращение к ним с различным числом входных и выходных аргументов. Для этого все входные аргументы упаковываются в специальный массив (вектор) ячеек с предопределенным именем `varargin`, каждый аргумент занимает ровно одну ячейку.

Переменную `varargin` можно использовать только внутри m – файлов. Она содержит набор необязательных аргументов функции и должна быть объявлена как последний входной аргумент. `varargin` будет вбирать в себя все аргументы по порядку, начиная с места на котором она стоит.

Процедура `plotAboveDomain`, приведенная в следующем примере, строит каркасный график функции двух переменных `funz(x,y)` над плоской областью $a \le x \le b$, $f_d(x) \le y \le f_u(x)$. Первые три ее аргумента являются функциями. Четвертый – вектор из двух элементов, задает интервал изменения переменной `x`. Все аргументы, начиная с пятого, собираются в массив ячеек `varargin`, а функция `mesh(...,varargin{:})` принимает его, как свой список аргументов.

```
function plotAboveDomain(funz,fund, funu,V,varargin)
% PLOT DOMAIN
% PLOTABOVEDOMAIN(FUNZ,FUND,FUNU,V, список_аргументов)
% построение графика функции funz(x,y) двух переменных над областью,
% ограниченной снизу и сверху кривыми y=fund(x) и y=funu(x)
% на отрезке V=[a,b] (a<=x<=b)
```

```
a=V(1);
b=V(2);
dlt=(b-a)/20;
t=a:dlt:b;
maxY=max(feval(funu,t));
minY=min(feval(fund,t));
[U,V]=meshgrid(t,minY:dlt:maxY);
X=U;
Y=feval(fund,X)+PR(V,feval(fund,X),feval(funu,X)-feval(fund,X));
Z=feval(funz,X,Y);
mesh(X,Y,Z,varargin{:});

% подфункция
function z=PR(x,a,w)
z=(w+abs(x-a)-abs(x-a-w))/2;
```

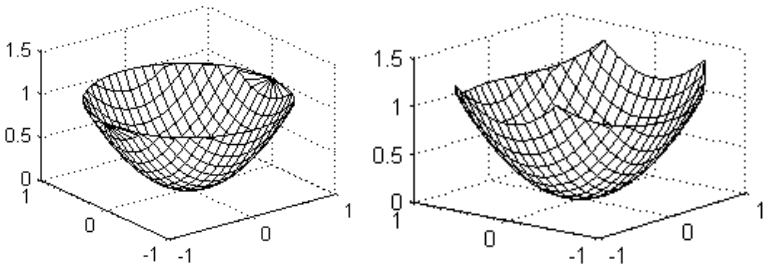
Отметим, что здесь мы также создали подфункцию `PR(x,a,w)` и то, что функция `plotAboveDomain` не возвращает никаких значений.

Чтобы вызвать функцию `plotAboveDomain`, создадим анонимную функцию **myfun,** график которой будем строить, и две функции **fund**, **funu**, которые будут ограничивать сверху и снизу область изменения независимых переменных

```
myfun=@(x,y) x.^2+y.^2;
fund=@(x) -sqrt(1-x.^2);
funu=@(x) sqrt(1-x.^2);
plotAboveDomain(myfun,fund, funu,[-1 1], 'EdgeColor','k')
```

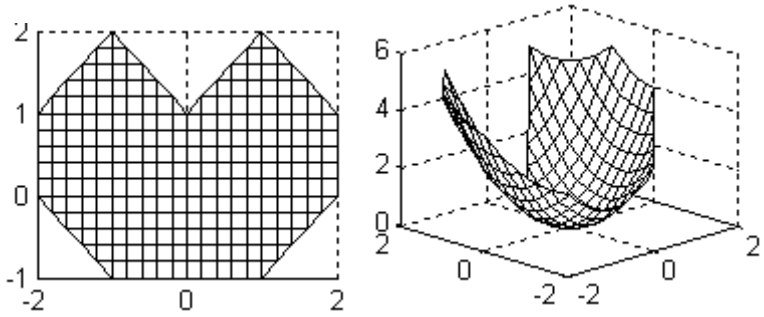
График поверхности $z = x^2 + y^2$ над единичным кругом показан на следующем рисунке слева. На рисунке справа та же поверхность построена над областью $-1 \le x \le 1$, $-\cos x \le y \le \cos x$.

```
fund=@(x) -cos(x)
plotAboveDomain(myfun,fund, @cos,[-1 1])
```



Вот пример построения графика поверхности над более сложной областью. Область показана на следующем рисунке слева (вид поверхности сверху), а сама поверхность – справа. Здесь графики функций `funu(x)` и `fund(x)` ограничивают область сверху и снизу

```
funu=@(x) 3-abs(x+1)+abs(x)-abs(x-1)
fund=@(x) -2 +0.5*abs(x+1)+0.5*abs(x-1)
myfun=@(x,y) x.^2+y.^2;
plotAboveDomain(myfun,fund, funu,[-2 2], 'EdgeColor','k')
```



Как видим, в качестве дополнительных аргументов мы можем использовать любые допустимые имена свойств и их значений.

plotAboveDomain(myfun, fund, funu, [-1 1], 'EdgeColor', [1 0 1], 'LineWidth', 3, 'LineStyle',':')

Массив `varargin` может быть единственным входным аргументом функции. Тогда ее заголовок будет выглядеть так: `function z = myfun(varargin)`

Внутри тела функции доступ к входным аргументам, т. е. ячейкам, производится при помощи заключения индекса в фигурные скобки и последующим обращении с содержимым в зависимости от типа хранимых данных. Например, `varargin{1}` содержит первый аргумент, `varargin{2}` – второй, и т.д. Вот простой пример

```
function z= vectsum(varargin)
% вычисление суммы элементов любого количества векторов
% произвольной длины
N = length(varargin);
s=0;
for i = 1:N
    s=s+sum(varargin{i});
end
z=s;
```

vectsum([1 2 3],[12 1 5],[5 1], [1; 3; 4; 5; 6])

```
ans =
    49
```

Напомним, что в теле функции можно использовать переменную с именем `nargin`. Она хранит число входных аргументов при каждом использовании функции.

Функции с переменным числом возвращаемых параметров. Для создания такой функции в строке ее заголовка следует использовать переменную с предопределенным именем `varargout`, например так,

`function varargout=myfun(x,y,…).`

или так

`function [a,b,…,varargout]=myfun(x,y,…).`

Переменная `varargout` должна быть объявлена последней (или единственной). Она собирает все выходные значения по порядку, начиная с места ее вхождения. Кроме того, в теле функции можно использовать переменную с именем `nargout`. Она хранит число принимающих переменных при каждом использовании функции.

Для примера создадим функцию, которая находит максимальное значение элементов одномерного вектора и возвращает (если надо) вектор номеров элементов, на которых достигается это максимальное значение.

```
function [val, varargout]=mymax(x)
% функция находит максимальное значение val одномерного массива x
% и возвращает вектор номеров элементов, на которых
% достигается это значение
val=max(x);
nums=find(x==val);
varargout(1)={nums};
```

X=[1 3 5 -2 7 0 7 2 5 7 -3];
v=mymax(X)

```
v =
     7
```

[v,n]=mymax(X)

```
v =
     7
n =
     5     7    10
```

Но

[v,n,k]=mymax(X)

```
??? Error using ==> mymax. Too many output arguments.
```

Количество фактических выходных параметров должно быть не больше количества формальных, но может быть меньше!

Для примера приведем еще одну функцию из справочной системы MatLab

```
function [s,varargout]=mysize(x)
nout = max(nargout,1)-1;
s = size(x);
for k=1:nout, varargout(k)={s(k)}; end;
```

Она возвращает вектор из количеств элементов массива по каждому из индексов и, если требуется, возвращает отдельно каждый из этих размеров.

s=mysize(ones(4,3))

```
s =
     4     3
```

[s,r,c]=mysize(ones(4,3))

```
s =
     4     3
r =
     4
c =
     3
```

Иногда внутри функции желательно знать имя фактического параметра, подставленного на место формального аргумента. Для этого используется функция `inputname(номер_аргумента)`. Для примера создадим функцию

```
function firstArgName(a,b)
disp(sprintf('Имя первой переменной  "%s".',inputname(1)));
```

Тогда

x=4; y=7;
firstArgName(x, y)

```
Имя первой переменной  "x".
```

firstArgName(y, x)

```
Имя первой переменной  "y".
```

Но

firstArgName(5, x)

```
Имя первой переменной  " ".
```

Вызов любой функции можно осуществить, написав выражение в командном окне. Но всегда существует вероятность допустить ошибку, связанную с несовпадением количества или типов фактических и формальных параметров. MATLAB не выполняет никаких проверок на эту тему. Чтобы избегать ошибочных ситуаций нужно в тексте m – файлов выполнять проверку входных аргументов. Примеры функций, приведенные выше, такой проверки не выполняли. Для написания правильного кода требуется полный набор конструкций управления, к рассмотрению которых мы переходим в следующем параграфе.

2.2.3 Операторы управления вычислительным процессом

Существует 4 основных оператора управления последовательностью выполнения команд:

- оператор цикла `for` выполняет группу команд фиксированное число раз;
- оператор `while` выполняет группу команд до тех пор, пока не будет выполнено условие, указанное в его заголовке;
- оператор условия `if` … `elseif` … `else` выполняет группу команд в зависимости от значения некоторого логического выражения;
- оператор `switch` … `case` … `otherwise` выполняет группу команд в зависимости от значения некоторого логического условия.

Использование `for` осуществляется следующим образом:

```
for cnt = start: step: final
     команды
end
```

Здесь `cnt` – переменная цикла, `start` – ее начальное значение, `final` – конечное значение, `step` – шаг, на который увеличивается `cnt` при каждой следующей итерации. Цикл заканчивается, когда значение `cnt` становится больше `final`. Переменная цикла может принимать не только целые, но и вещественные значения любого знака.

```
A=[1 2 3; 4 5 6; 7 8 9];
for k=1:9
      A(k)=rand(1);
end
A
A =
      0.74679      0.46599      0.52515
       0.4451      0.41865      0.20265
      0.93181      0.84622      0.67214
```

Заметим, что оператор цикла можно вводить в командном окне и завершать набор каждой строки нажатием клавиши Enter. При этом код не будет выполняться, пока вы не введете завершающую команду **end**.

Цикл можно вводить в одну строку, отделяя инструкцию for от тела цикла запятой

```
v=[ ]; for i=1:6, v=[v i.^2]; end; v
v =
     1     4     9    16    25    36
```

Оператор цикла в форме

```
for i=A          % A – вектор
     команды
end
```

определяет переменную цикла i как элемент вектора **A**. Для первого шага цикла i равно первому элементу вектора, для второго – второму и т.д. Например

```
A=[1 5 7 8 11 21]; a=[ ];
for i=A
      a=[a i.^2];
end;
a
a =
     1    25    49    64   121   441
```

Если **A** будет матрицей, то запись for i=A будет определять переменную цикла i как вектор **A(:, k)**. Для первого шага цикла k=1, для второго k=2 и т.д. Цикл выполняется столько раз, сколько столбцов в матрице A. Для каждого шага переменная цикла i – это вектор, содержащий один из столбцов матрицы **A**. Например

```
A=[1 2 3; 4 5 6; 7 8 9]; v=[ ];
for i=A
      v=[v sum(i)];
end
v
v =
    12    15    18
```

Цикл **for** оказывается полезным при выполнении определенного конечного числа итераций. Существуют алгоритмы с заранее неизвестным количеством повторений, реализовать которые позволяет цикл **while**. Он служит для организации повторений группы команд до тех пор, пока выполняется некоторое условие

```
while условие
     команды
end
```

Рассмотрим пример создания функции вычисления значения синуса разложением в степенной ряд $\sin x = \sum_{k=0}^{\infty} (-1)^k \frac{x^{2k+1}}{(2k+1)!}$

```
function z=mysin(x)
% Вычисление синуса разложением в степенной ряд
% Использование: y = mysin(x)
```

```
k=0;
u=x;
s=u;
x2=x*x;
while any(abs(u)>1e-10)
    k=k+1;
    u=-u*x2/(2*k)/(2*k+1);
    s=s+u;
end
z=s;
```

Здесь в теле цикла проверяется условие `abs(u)>1e-10` и, если оно выполняется, вычисления в теле продолжаются. Использование функции **any** потребовалось для нормальной работы алгоритма для случая, когда аргумент **x** функции является вектором. Дело в том, что когда проверяется условие с массивом, все его элементы должны удовлетворять условию. Только тогда выполнение будет продолжаться. Чтобы получить скалярное условие следует использовать функции `any` или `all`.

Использовать нашу функцию можно разными способами: вычислять ее значение в точках или строить график

```
mysin([0, pi/6, pi/4, pi/3, pi/2 ,pi])
ans =
         0    0.5000    0.7071    0.8660    1.0000    0.0000
```

График можно построить следующим способом

```
x=-pi:pi/20: pi; y=mysin(x); plot(x,y)
```

или, используя функцию **fplot**

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

Условие цикла `while` может содержать логическое выражение, составленное из операций отношения. Сложные условия записываются с применением логических операций. О них мы говорили в п.2.1.1. Например

```
x=10*rand(1,5)
x =
    7.6210    4.5647    0.1850    8.2141    4.4470
and(x>=2,x<5)
ans =
     0     1     0     0     1
```

Возможен досрочный выход из тела цикла с помощью функции `break`, которая прерывает выполнение циклов `for` и `while`. Циклы `for` и `while` могут быть вложенными. В случае вложенных циклов прерывание возможно только из самого внутреннего цикла.

При программировании алгоритмов кроме организации повторяющихся действий в виде циклов часто требуется выполнить тот или иной блок команд в зависимости от некоторых условий, т. е. использовать ветвление алгоритма. Это позволяют сделать оператор `if` и оператор переключения `switch`.

В самом простом случае оператор `if` выглядит так

```
if условие
```

```
     команды
end
```

Оператор `if` вычисляет условие и, если оно истинно, выполняет группу команд, стоящую в его теле до завершающей инструкции `end`. Если условие неверно, то происходит переход к командам, расположенным после `end`. Условие является логическим выражением и записывается по общим правилам составления таких выражений. Более развернутый вариант команды имеет вид

```
If условие
     команды 1
else
     команды 2
end
```

В этом случае проверяется условие и если оно истинно, то выполняется блок команд 1, а если нет – блок команд 2. Общая схема использования оператора `if` имеет вид

```
If условие1
     Команды 1
elseif условие2
     команды 2
elseif условие3
     команды 3
     . . .
else
     команды
end
```

Вначале проверяется условие1 и если оно истинно, то выполняется блок команд 1. Если условие не выполняется, то проверяется условие2 и, если оно истинно, то выполняется блок команд 2. Если нет, то проверяется условие3 и т.д. Если ни одно условие не выполняется, то выполняется блок команд стоящий после ключевого слова `else`. Возможен вариант без последнего блока команд `else`. Операторы `if` могут быть вложенными.

Напишем функцию, вычисляющую значения следующей кусочной функции

$$y = \begin{cases} \sqrt{1-x^2} & -1 \le x \le 1 \\ 0, & \textit{иначе} \end{cases}.$$

```
function z=mysqrt(x)
% Вычисление кусочной функции по формуле:
% sqrt(1-x^2) при -1<=x<=1 и 0 иначе
% Использование: y = mysqrt(x)
z=[];
for xx=x
    if and(xx>=-1,xx<=1)
        zz=sqrt(1-xx.^2);
    else
        zz=0;
    end;
    z=[z zz];
end
```

В этом примере мы использовали второй вариант оператора `if`. Можно построить график нашей функции так

x=-2:0.1:2; y=mysqrt(x); plot(x,y)

или так

fplot(@mysqrt,[-2 2])

Если логическое условие включает переменную, не являющуюся скаляром, то утверждение будет истинным, если все элементы удовлетворяют этому условию. Например, для матрицы X условие

```
if X>0
     команды
end
```

равносильно следующему

```
if all(X(:))
     команды
end
```

Например

X=[1 2; 3 0]; X(:)'

```
ans =
     1     3     2     0
all(X(:))
ans =
     0
```

При составлении кода функций всегда следует обращать внимание на это обстоятельство. Если аргумент операции сравнения является массивом, а не скаляром, то условие может работать совсем по-другому. Например, в последней функции `mysqrt`, казалось бы, можно было использовать следующий код

```
function z=mysqrt(x)
if and(x>=-1,x<=1)
    z=sqrt(1-x.^2);
else
    z=0;
end
```

Но этот код в случае, когда **x** вектор, будет работать неправильно!

Оператор `switch` - это оператор выбора. Схема его использования следующая

```
switch выражение
case {список_значений_1}
     команды_1
case {список_значений_2}
     команды_2
     . . .
otherwise
     команды
end
```

Вычисляется выражение и по очереди сравнивается со значениями, перечисленными после ключевых слов `case`. Если найдено совпадение, то выполняется соответствующий блок команд и управление передается на команду, следующую за завершающим ключевым словом `end`. Если

совпадений не найдено, то выполняются команды блока за ключевым словом otherwise. Блок команд otherwise может отсутствовать. Значения в фигурных скобках разделяются запятыми. Если какой либо список содержит одно значение, то фигурные скобки можно опускать.

Оператор switch работает, сравнивая значения выражения со значениями групп case. Для числового выражения оператор case выполняется, если выражение == значение. Для строковых выражений оператор case истинен, если strcmp(выражение, значение) истинно.

Кроме операторов управления, важным элементом любого языка программирования являются операторы обработки исключительных ситуаций. В MatLab таким оператором является try. . .catch. Схема его использования выглядит следующим образом:

```
try
     % операторы, выполнение которых
     % может привести к ошибке
catch
     % операторы, которые следует выполнить
     % при возникновении ошибки
end
```

Например, следующий код читает данные из файла и в случае возникновения ошибки доступа к файлу выводит сообщение, а затем управление передает на команды следующие за ключевым словом end.

```
try
     A = load('my.dat');
     pie(A)
catch
     disp('не могу найти файл my.dat')
end
```

Здесь мы использовали функцию disp. Команда disp(x) выводит в командное окно значение переменной x, а команда disp('текст') выводит в командное окно строку 'текст'. После каждой команды disp происходит перевод на новую строку.

Иногда в случае возникновения ошибки желательно вывести текст сообщения и завершить выполнение программы. Это можно сделать командой error('сообщение').

Команда warning('сообщение') используется для выдачи предупреждения. Вывод предупреждений можно отключить командой warning off и снова включить командой warning on.

При написании программ полезными могут быть следующие функции проверки и сравнения

Имя	Назначение
isempty	Выявление пустого массива
isequal	Проверка равенства матриц
nonzeros	Вывод вектора из ненулевых элементов массива

isfinite	Определение конечных элементов числового массива (единицы соответствуют числам, нули – inf, NaN)
isnumeric	Проверка, является ли массив числовым
isinf	Определение бесконечных элементов массива (единицы соответствуют +inf, -inf, нули – числам)
isnan	Выявление элементов нечислового типа (единицы соответствуют NaN, нули – числам)
isletter	Проверка на символ
isstr	Проверка на строковую переменную
strcmp	Сравнение двух строк
isreal	единицы соответствуют вещественным числам, нули – комплексным

Векторизация. Удобным приемом программирования в MatLab является векторизация. Составим сценарий TicToc1.m основу которого представляет цикл

```
tic;
x = 0.01;
for k = 1:1001
      y(k) = log10(x);
      x = x + .01;
end;
t=toc; t
```

В начале и конце этого файла мы используем функции tic и toc. Функция tic начинает отсчет времени, а функция toc возвращает время в секундах, прошедшее после последней засечки времени функцией tic.

TicToc1

```
t =
    0.0200081295248418
```

Под векторизацией понимают замену цикла кодом, содержащим матричные и векторные операции. Следующий сценарий TicToc2.m выполняет те же вычисления, что и предыдущий

```
tic;
x = 0.01: .01:10;
y = log10(x);
t=toc; t
```

Но время выполнения этого кода значительно меньше

TicToc2

```
t =
      0.000521854034521147
```

Когда важна скорость, следует искать способы замены циклов на участки кода, использующие векторные и матричные операции.

Заметим, что время выполнения предыдущих двух сценариев существенно различается только после их второго вызова. При первом вызове время работы сценариев почти одинаково, поскольку происходит создание псевдокода программы.

Если не удается векторизовать часть кода, то следует использовать предварительное выделение памяти по вектор или матрицу.

r = zeros(32,1);

```
for n = 1:32
        r(n) = rank(magic(n));
end
```

Без предварительного создания вектора **r** интерпретатор MatLab будет выделять дополнительное место под `r(n)` на каждом шаге цикла, что снижает производительность.