

ПРОГРАММИРОВАНИЕ В СРЕДЕ МАТЛАВ.

Программами в системе MatLab являются *m*-файлы текстового формата, содержащие запись программ в виде программных кодов. Язык программирования системы MatLab имеет следующие средства:

- данные различного типа;
- константы и переменные;
- операторы, включая операторы математических выражений;
- встроенные команды и функции;
- функции пользователя;
- управляющие структуры;
- системные операторы и функции;
- средства расширения языка.

В MatLab определены следующие основные типы данных, в общем случае представляющих собой многомерные массивы:

- **single** — числовые массивы с числами одинарной точности;
- **double** — числовые массивы с числами удвоенной точности;
- **char** — строчные массивы с элементами-символами;
- **sparse** — наследует свойства **double**, разреженные матрицы с элементами-числами удвоенной точности;
- **cell** — массивы ячеек; ячейки, в свою очередь, тоже могут быть массивами;
- **struct** — массивы структур с полями, которые также могут содержать массивы;
- **function_handle** — дескрипторы функций;
- **int32, uint32** — массивы 32-разрядных чисел со знаком и без знаков;
- **int16, uint16** — массивы 16-разрядных целых чисел со знаком и без знаков;
- **int8, uint8** — массивы 8-разрядных целых чисел со знаками и без знаков.

Язык программирования системы MatLab вобрал в себя почти все средства, необходимые для реализации различных технологий программирования:

- *процедурного;*
- *операторного;*
- *функционального;*
- *логического;*
- *структурного (модульного);*
- *объектно-ориентированного;*
- *визуально-ориентированного.*

В основе процедурной, операторной и функциональной технологии программирования лежат процедуры, операторы и функции, используемые как основные объекты языка. Эти типы объектов присутствуют в MatLab. Логическое программирование реализуется в MatLab с помощью логических операторов и функций.

Наиболее ярко в MatLab представлены идеи структурного программирования. Подавляющее большинство функций и команд языка представляют собой вполне законченные модули, обмен данными между которыми происходит через их входные параметры, хотя возможен обмен информацией и через глобальные переменные.

Объектно-ориентированное программирование также широко представлено в системе MatLab. Оно особенно актуально при программировании задач графики. Что касается визуально-ориентированного программирования, то в MatLab оно представлено в основном в пакете моделирования заданных блоками устройств и систем Simulink.

Для более сложных задач число команд возрастает, и работа в командной строке становится непродуктивной. Использование истории команд, сохранение переменных рабочей среды или ведение дневника при помощи diary незначительно повышают производительность работы. Эффективное решение состоит в оформлении собственных алгоритмов в виде программ (М-файлов), которые можно запустить из рабочей среды или из редактора. Встроенный в MATLAB редактор М-файлов позволяет не только набирать текст программы и запускать ее целиком или частями, но и отлаживать алгоритм. Подробная классификация М-файлов приведена ниже.

Типы М-файлов. Файл-программы

М-файлы в MATLAB бывают двух типов: файл-программы (Script M-Files), содержащие последовательность команд, и файл-функции (Function M-Files), в которых описываются функции, определяемые пользователем.

Файл-программы представляют собой простейший тип М-файлов. Они не имеют входных и выходных аргументов и оперируют переменными, существующими в рабочей среде, или могут создавать новые переменные. Все переменные, объявленные в файл-программе, становятся доступными в рабочей среде после ее выполнения.

Последовательность поиска MATLAB говорит о том, что очень важно правильно задавать имя собственной файл-программы при сохранении ее в М-файле. Во-первых, ее имя не должно совпадать с именем существующих функций в MATLAB. Узнать, занято имя или нет можно при помощи функции `exist`.

Во-вторых, имя файла не должно начинаться с цифры, знаков "+" или "-", словом с тех символов, которые могут быть интерпретированы MATLAB как ошибка при вводе выражения.

Очень распространена еще одна ошибка при задании имени файл-программы, которая на первый взгляд имеет необъяснимые последствия: программа запускается только один раз. Повторный запуск не приводит к выполнению программы.

Для решения вычислительных задач и написания собственных приложений в MATLAB часто требуется программировать файл-функции, которые производят необходимые действия с входными аргументами и возвращают результат в выходных аргументах. Число входных и выходных аргументов зависит от решаемой задачи – может быть только один входной и один выходной аргумент, несколько и тех и других, или только входные аргументы.

Файл-функции с одним входным аргументом.

Предположим, что в вычислениях часто необходимо использовать значение функции:

$$e^{-x} \sqrt{\frac{x^2 + 1}{x^4 + 0.1}}$$

Имеет смысл один раз написать файл-функцию, а потом вызывать ее всюду, где необходимо вычисление этой функции для заданного аргумента. Для этого необходимо открыть в редакторе М-файлов новый файл и набрать текст:

```
function f = myfun(x)
```

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

Слово *function* в первой строке определяет, что данный файл содержит файл-функцию. Первая строка является заголовком функции, в которой размещаются имя функции и списки входных и выходных аргументов. Входные аргументы записываются в круглых скобках после имени функции. В нашем примере есть только один входной аргумент – *x*. Выходной аргумент *f* указывается слева от знака равенства в заголовке функции.

Теперь созданную функцию можно использовать так же, как и встроенные *sin*, *cos* и другие, например, из командной строки:

```
>> y=myfun(1.3)
```

```
y =
```

```
0.2600
```

При создании файл-функции *myfun* мы подавили вывод значения *f* в командное окно, завершив оператор присваивания точкой с запятой. Если этого не сделать, то оно выведется при обращении *y=myfun(1.3)*. Как правило, лучше избегать вывода в командное окно результатов промежуточных вычислений внутри файл-функции.

Файл-функция, приведенная в предыдущем примере, имеет один существенный недостаток.

Попытка вычисления значений функции от массива приводит к ошибке, а не к массиву значений так, как это происходит при использовании встроенных функций.

```
>> x=[1.3 7.2];
```

```
>> y=myfun(x)
```

```
??? Error using ==> ^
```

```
Matrix must be square.
```

```
Error in ==> C:\MATLAB6p5\work\myfun.m
```

```
On line 2 ==> f = exp(-x)*sqrt((x^2 + 1)/(x^4 + 0.1));
```

Очевидно, что для избежания этой ошибки необходимо использовать поэлементные операции. В частности, для правильной работы нашей функции необходимо текст функции переписать в следующем виде:

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

Теперь аргументом функции `myfun` может быть как число, так и вектор или матрица значений, например:

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

Переменная `y`, в которую записывается результат вызова функции `myfun`, автоматически становится вектором нужного размера.

Рассмотрим пример использования функций. Строим график функции `myfun` на отрезке $[0,4]$ при помощи файл-программы или из командной строки:

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


Рассмотрим сейчас только один простой пример того, как использование файл-функций упрощает визуализацию математических функций. Только что мы построили график при помощи `plot`. Что для вычисления вектора `y` не обязательно было вызывать `myfun` – можно сразу записать выражение для него и потом указать пару `x` и `y` в `plot`. Имеющаяся в нашем распоряжении файл-функция `myfun` позволяет обратиться к специальной функции `fplot`, которой требуется указать имя нашей файл-функции (в апострофах) или указатель на нее (с оператором `@` перед именем функции) и границы отрезка для построения графика (в векторе из двух элементов)

```
>> fplot('myfun', [0 4])
```

или

```
>> fplot(@myfun, [0 4])
```

Следует добавить алгоритм функции `fplot` автоматически подбирает шаг аргумента, уменьшая его на участках быстрого изменения исследуемой функции, что дает пользователю хорошее отображение данных.

Файл-функции с несколькими входными аргументами.

Написание файл-функций с несколькими входными аргументами практически не отличается от случая одного аргумента. Все входные аргументы размещаются в списке через запятую. Следующий пример содержит файл-функцию, вычисляющую длину радиус-вектора точки трехмерного пространства $(x^2 + y^2 + z^2)^{1/2}$.

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

Для вычисления длины радиус-вектора теперь можно использовать функцию `radius3`, например:

```
>> R = radius3(1, 1, 1)
R = 1.7321
```

Кроме функций с несколькими аргументами, MATLAB позволяет создавать функции, возвращающие несколько значений, т. е. имеющих несколько выходных аргументов.

Файл-функции с несколькими выходными аргументами.

Файл-функции с несколькими выходными аргументами удобны при вычислении функций, возвращающих несколько значений (в математике они называются вектор-функции). Выходные аргументы добавляются через запятую в список выходных аргументов, а сам список заключается в квадратные скобки. Следующий пример приводит файл-функцию *hms* для перевода времени, заданного в секундах, в часы, минуты и секунды:

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

При вызове файл-функций с несколькими выходными аргументами результат следует записывать в вектор соответствующей длины:

```
>> [H, M, S] = hms(10000)
```

```
H =
```

```
2
```

```
M =
```

```
46
```

```
S =
```

```
40
```

Если при использовании данной функции явно не указывать выходные параметры, то результатом вызова функции будет только первый выходной аргумент:

```
>> hms(10000)
```

```
ans =
```

```
2
```

Если список выходных аргументов пуст, т. е. заголовок выглядит так:

```
function myfun(a, b) или function [ ] = myfun(a, b),
```

то файл-функция не будет возвращать никаких значений. Такие функции тоже иногда оказываются полезными.

Функции MATLAB обладают еще одним полезным качеством — возможностью получения информации о них при помощи команды `help`, например, `help fplot`. Собственные файл-функции так же можно наделить этим свойством, используя строки комментариев. Все строки комментариев после заголовка и до тела функции или пустой строки выводятся в командное окно командой `help`. Например для нашей функции можно создать подсказку:

```
function [hour, minute, second] = hms(sec)
```

```
%hms - перевод секунд в часы, минуты и секунды
```

```
% Функция hms предназначена для перевода секунд
```

```
% в часы минуты и секунды.
```

```
% [hour, minute, second] = hms(sec)
```

```
hour = floor(sec/3600);
```

```
minute = floor((sec - hour*3600)/60);
```

```
second = sec - hour*3600 - minute*60;
```

Рассмотрим еще одну разновидность функций – подфункции. Использование подфункций основано на выделении части алгоритма в самостоятельную функцию, текст которой содержится в том же файле, что и основная функция. Рассмотрим это на примере.

```
function simple;  
% Основная функция  
a = 2*pi;  
f1 = f(1.1, 2.1)  
f2 = f(3.1, 4.2)-a  
f3 = f(-2.8, 0.7)+a  
function z = f(x, y)  
% Подфункция  
z = x^3 - 2*y^3 - x*y + 9;
```

Первая функция *simple* является основной функцией в *simple.m*, именно ее операторы выполняются, если пользователь вызывает *simple*, например, из командной строки. Каждое обращение к подфункции *f* в основной функции приводит к переходу к размещенным в подфункции операторам и последующему возврату в основную функцию.

Файл-функция может содержать одну или несколько подфункций со своими входными и выходными параметрами, но основная функция может быть только одна. Заголовок новой подфункции одновременно является признаком конца предыдущей. 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.

Один из возможных вариантов использования переменных, которые являются общими для всех функций M-файла, состоит в объявлении данных переменных в начале основной функции и подфункции как глобальных, при помощи `global` со списком имен переменных, разделяемых пробелом.

ОСНОВНЫЕ ОПЕРАТОРЫ М-ЯЗЫКА

```
% Вычисление длины окружности с диалоговым вводом радиуса
r=0;
while r>=0,
r=input('Введите радиус окружности r=');
if r>=0 disp(' Длина окружности l='); disp(2*pi*r), end
end
```

Эта программа служит для многократного вычисления длины окружности по вводимому пользователем значению радиуса r . Обратите внимание на то, что здесь мы впервые показываем пример организации простейшего диалога. Он реализован с помощью команды `input`:

`input('Введите радиус окружности r=');`

Следующая строка

`if r>=0 disp(' Длина окружности l = '); disp(2*pi*r);end`

с помощью команды `disp` при $r \geq 0$ выводит надпись «Длина окружности $l =$ » и вычисленное значение длины окружности. Она представляет собой одну из наиболее простых управляющих структур типа `if...end`.

Если данная программа записана в виде *m*-файла, то работа с ней будет выглядеть следующим образом:

```
Введите радиус окружности R=1
Длина окружности l=
6.2832
Введите радиус окружности R=2
Длина окружности l=
12.5664
Введите радиус окружности R=-1
»
```

Функция `input` может использоваться и для ввода произвольных строковых выражений. При этом она задается в следующем виде:

`input('Комментарий', V)`

При выполнении этой функции она останавливает вычисления и ожидает ввода строкового комментария. После ввода возвращается набранная строка. Это иллюстрирует следующий пример:

```
» S=input('Введите выражение ','s')
Введите выражение (Вводим) 2*sin(1)
S =
    2*sin(1)
» eval(S)
ans =
    1.6829
>>
```

Схожие и повторяющиеся действия выполняются при помощи операторов цикла **for** и **while**. Цикл **for** предназначен для выполнения заданного числа повторяющихся действий, а **while** – для действий, число которых заранее не известно, но известно условие продолжения цикла.

Цикл **for**.

Использование **for** осуществляется следующим образом:

Оператор цикла типа **for...end** обычно используются для организации вычислений с заданным числом повторяющихся циклов. Конструкция такого цикла имеет следующий вид:

```
for count = start:step:final  
    команды MATLAB  
end
```

Здесь **count**— переменная цикла, **start** – ее начальное значение, **final** – конечное значение, а **step** – шаг, на который увеличивается **count** при каждом следующем заходе в цикл. Цикл заканчивается, как только значение **count** становится больше **final**. Переменная цикла может принимать не только целые, но и вещественные значения любого знака.

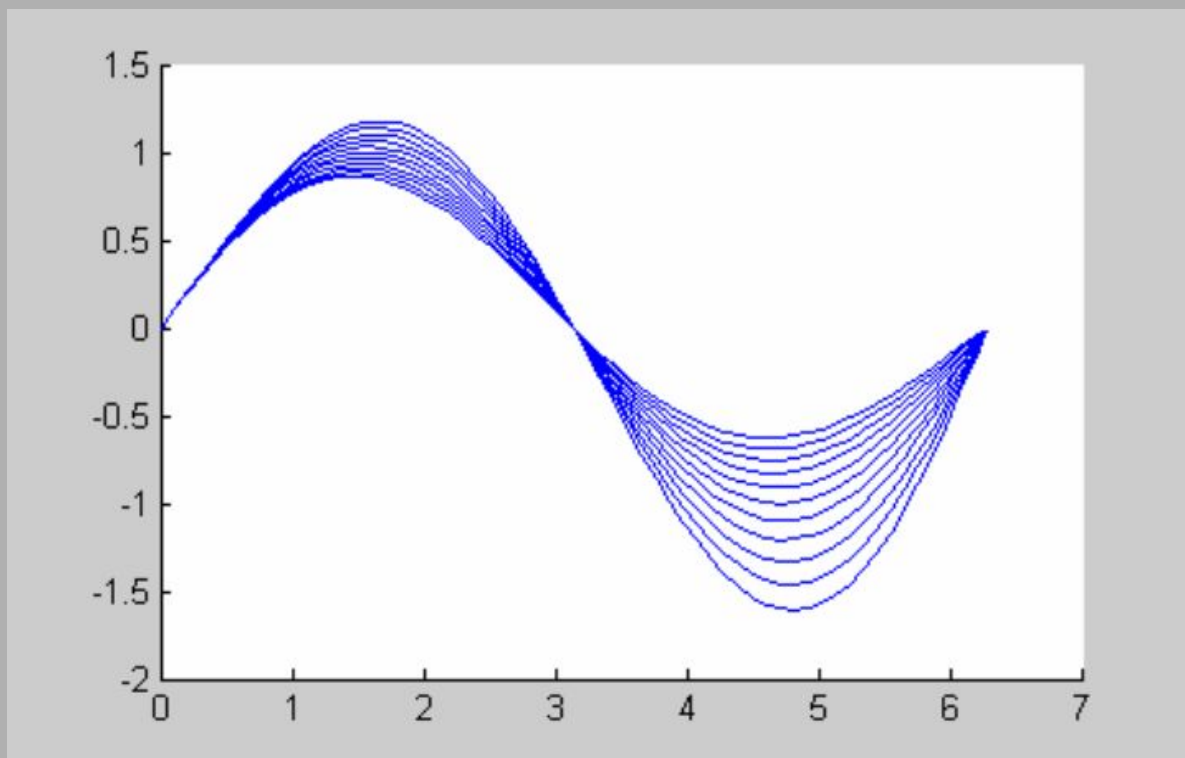
Приведем пример применения цикла `for`. Пусть требуется вывести графики семейства кривых, которое задано функцией, зависящей от параметра a , для значений параметра a от -0.1 до 0.1 с шагом 0.02 . Можно, конечно, последовательно вычислять $y(x,a)$ и строить ее графики для различных значений a , но гораздо удобнее использовать цикл `for`.

Текст файл-программы:

```
figure % создание графического окна
x = 0:pi/30:2*pi; % вычисление вектора значений аргумента

% перебор значений параметра в цикле
for a = -0.1:0.02:0.1
% вычисление вектора значений функции для текущего значения ...
параметра
    y = exp(-a*x).*sin(x); % добавление графика функции
    hold on
    plot(x, y)
end
```

В результате выполнения этой файл-программы появится графическое окно, которое содержит требуемое семейство кривых.



Операторы цикла.

Циклы *for* могут быть вложены друг в друга, при этом переменные вложенных циклов должны быть разными. Вложенные циклы удобны для заполнения матриц. Пример создания матрицы Гильберта:

```
n = 4;  
a = zeros(n);  
    for i = 1:n  
        for j = 1:n  
            a(i, j) = 1/(i+j-1);  
        end  
    end
```

```
%  
for i=1:3  
for j=1:3  
A(i,j)=i+j;  
end  
end
```

В результате выполнения этого цикла формируется матрица *A*:

```
>> Пример4-3-12  
%%  
>>A  
A =  
    2    3    4  
    3    4    5  
    4    5    6  
>>
```

Операторы цикла.

Следует отметить, что формирование матриц с помощью оператора : (двоеточие) обычно занимает намного меньше времени, чем с помощью цикла. Однако применение цикла нередко оказывается более наглядным и понятным. *MatLab* допускает использование в качестве переменной цикла массива A размера $m \times n$. При этом цикл выполняется столько раз, сколько столбцов в массиве A , и на каждом шаге переменная var представляет собой вектор, соответствующий текущему столбцу массива A :

```
» A=[1 2 3:4 5 6]
A =
     1 2 3
     4 5 6
» for var=A; var, end
var =     1     4
var =     2     5
var=     3     6
>>
```

В качестве значений переменной цикла допускается использование массива значений:

for count = A
команды MATLAB
end

Если A — вектор-строка, то $count$ последовательно принимает значение ее элементов при каждом заходе в цикл. В случае двумерного массива A на i -ом шаге цикла $count$ содержит столбец $A(:,i)$. Разумеется, если A является вектор-столбцом, то цикл выполнится всего один раз со значением $count$, равным A .

Цикл for оказывается полезным при выполнении определенного конечного числа действий. Существуют алгоритмы с заранее неизвестным количеством повторений, реализовать которые позволяет более гибкий цикл *while*.

Цикл while служит для организации повторений одностипных действий в случае, когда число повторений заранее неизвестно и определяется выполнением некоторого условия. Рассмотрим пример разложение $\sin(x)$ в ряд:

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

Конечно, до бесконечности суммировать не удастся, но можно накапливать сумму с заданной точностью, например, 10^{-10} . Очевидно, что число членов ряда в данном случае неизвестно, поэтому использование оператора `for` невозможно. Выход состоит в применении цикла `while`, который работает, пока выполняется условие цикла:

```
while условие повторения цикла
    команды MATLAB
end
```

В данном примере условием повторения цикла является то, что модуль текущего слагаемого $x^{2k+1}/(2k+1)!$ больше 10^{-10} . Текст файл-функции `mysin`, вычисляющей сумму ряда на основе рекуррентного соотношения:

$$a_k = \frac{x^2}{2k(2k+1)} a_{k-1}$$


```
function s = mysin(x)
```

```
% Вычисление синуса разложением в ряд
```

```
% Использование: y = mysin(x),  $-pi < x < pi$ 
```

```
% вычисление первого слагаемого суммы для  $k = 0$ 
```

```
k = 0;
```

```
u = x;
```

```
s = u;
```

```
% вычисление вспомогательной переменной
```

```
x2 = x*x;
```

```
while abs(u) > 1.0e-10
```

```
k = k + 1;
```

```
u = -u* x2/(2*k)/(2*k + 1);
```

```
s = s + u;
```

```
end
```

Условие цикла `while` может содержать логическое выражение, составленное из операций отношения и логических операций или операторов. Для задания условия повторения цикла допустимы операции отношения

<code>==</code>	Равенство
<code><</code>	Меньше
<code><=</code>	Меньше или равно
<code>></code>	Больше
<code>>=</code>	Больше или равно
<code>~=</code>	Не равно

Операторы цикла.

Матрицы в операциях сравнения должны иметь одинаковые размеры. Все операции сравнения производят поэлементное сравнение матричных элементов. Результатом операции сравнения является матрица из нулей и единиц и того же размера, что и сравниваемые матрицы.

Задание более сложных условий производится с применением логических операторов или операций

Тип выражения	Выражение	Логический оператор	Логическая операция
Логическое “и”	$x < 3$ и $4 = k$	<code>and(x<3,k==4)</code>	<code>(x<3)&(k==4)</code>
Логическое “или”	$x = 1$ или 2	<code>or(x==1,x==2)</code>	<code>(x==1) (x==2)</code>
Логическое “не”	$a \neq 1.9$	<code>not(a==1.9)</code>	<code>-(a==1.9)</code>

Операнды логических операций и операторов должны иметь одинаковые размеры. Логические операции производят поэлементные операции над матричными элементами. Результатом логической операции всегда является матрица из нулей и единиц и того же размера, что и операнды.

Условный оператор *if* и оператор переключения *switch* позволяют создать гибкий разветвляющийся алгоритм, в котором при выполнении определенных условий выполняется соответствующий блок операторов или команд MATLAB. Практически во всех языках программирования имеются аналогичные операторы.

Условный оператор *if*.

Условный оператор *if* в общем виде записывается следующим образом:

```
if Условие1
    Инструкции_1
else if Условие2
    Инструкции_2
else
    Инструкции_3
End
```

Эта конструкция допускает несколько частных вариантов. В простейшем случае

```
if Условие
    Инструкции
end
```

Пока Условие возвращает логическое значение 1 (то есть «истина»), выполняются Инструкции, составляющие тело структуры *if...end*. При этом оператор *end* указывает на конец перечня инструкций. Инструкции в списке разделяются оператором, (запятая) или ; (точка с запятой). Если Условие не выполняется (дает логическое значение 0, «ложь»), то Инструкции также не выполняются.

Еще одна конструкция

if Условие

Инструкции_1

else

Инструкции_2

end

выполняет Инструкции_1, если выполняется Условие, или Инструкции_2 в противном случае.

Условия записываются в виде:

Выражение_1 Оператор_отношения Выражение_2,

причем в качестве Операторов_отношения используются следующие операторы:

==, <, >, <=, >= или ~=.

В общем случае применение этих структур достаточно очевидное. Приведем только один общий пример:

```
function ifdem(a)  
% пример использования структуры if-elseif-else  
if (a == 0)  
disp('a - ноль')  
elseif a == 1  
disp('a - единица')  
elseif a >= 2  
disp('a - двойка или больше')  
else  
disp('a меньше двух, но не ноль и не единица')  
end
```

Следует обратить внимание, что в данном примере мы использовали специальную функцию *disp*, которая позволяет выводить текстовую информацию.

Оператор switch.

Для осуществления множественного выбора (или ветвления) используется конструкция с переключателем типа **switch**:

switch *Выражение*

case *Значение*

Список инструкций

case {*Значение1*, *Значение2*, *Значение3*, ...}

Список инструкций

otherwise,

Список инструкций

end

Каждая ветвь определяется оператором *case*, переход в нее выполняется тогда, когда '*Выражение*' оператора *switch* принимает значение, указанное после *case*, или одно из значений списка *case*. После выполнения какой-либо из ветвей происходит выход из *switch*, при этом значения, заданные в других ветвях *case*, уже не проверяются. Если подходящих значений не нашлось, то выполняется ветвь оператора переключения, соответствующая *otherwise*.

```
function switchdem(a)
% пример использования
оператора switch
switch a
case 3
disp('Март')
case 4
disp('Апрель')
case 5
disp('Май')
case {1, 2, 6, 7, 8, 9, 10, 11, 12}
disp('Не весенние месяцы')
otherwise
disp('Ошибка задания')
end
```

Поясним применение оператора switch на примере

```
switch var
  case {1,2,3}
    disp('Первый квартал')
  case {4,5,6}
    disp('Второй квартал')
  case {7,8,9}
    disp('Третий квартал')
  case {10,11,12}
    disp('Четвертый квартал')
  otherwise
    disp('Ошибка в задании')
end
```

*Эта программа в ответ на значения переменной **var** – номера месяца – вычисляет, к какому кварталу относится заданный месяц, и выводит соответствующее сообщение:*

```
>> var=2; sw1
Первый квартал
>>var=4;sw1
Второй квартал
>> var=7:sw1
Третий квартал
>> var=12;sw1
Четвертый квартал
>var=-1;sw1
Ошибка в задании
>>
```

В управляющих структурах, в частности в циклах *for* и *while*, часто используются операторы, влияющие на их выполнение.

Так, оператор ***break*** может использоваться для досрочного прерывания выполнения цикла. Как только он встречается в программе, цикл прерывается.

Оператор ***continue*** передает управление в следующую итерацию цикла, пропуская операторы, которые записаны за ним, причем во вложенном цикле он передает управление на следующую итерацию основного цикла.

Оператор ***return*** обеспечивает нормальный возврат в вызывающую функцию или в режим работы с клавиатурой.

Операторы *break*, *continue* и *return*. .

В управляющих структурах, в частности в циклах ***for*** и ***while***, часто используются операторы, влияющие на их выполнение.

Так, оператор ***break*** может использоваться для досрочного прерывания выполнения цикла. Как только он встречается в программе, цикл прерывается.

Оператор ***continue*** передает управление в следующую итерацию цикла, пропуская операторы, которые записаны за ним, причем во вложенном цикле он передает управление на следующую итерацию основного цикла.

Оператор ***return*** обеспечивает нормальный возврат в вызывающую функцию или в режим работы с клавиатурой.

Операторы *break*, *continue* и *return*. .

Для остановки программы используется оператор *pause*. Он используется в следующих формах:

- *pause* — останавливает вычисления до нажатия любой клавиши;
- *pause(N)* — останавливает вычисления на *N* секунд;
- *pause on* — включает режим отработки пауз;
- *pause off* — выключает режим отработки пауз.

Следующий пример поясняет применение команды *pause*:

```
for i=1:20;  
x =rand(1,40); y =rand(1,40); z = sin(x.*y);  
tri = delaunay(x,y);  
trisurf(tri,x,y,z)  
pause;  
end
```

Команда *pause* обеспечивает показ 20 рисунков - построений трехмерных поверхностей из треугольных окрашенных областей со случайными параметрами.

Рациональная техника программирования.

MATLAB интерпретирует команды, записанные в *M*-файлах, в машинный код и последовательно выполняет их. Процесс интерпретации занимает много времени в том случае, когда алгоритм обработки большого объема данных содержит циклы, поскольку каждая строка цикла интерпретируется столько раз, сколько выполняется цикл. Следовательно, при разработке приложений *MATLAB* необходимо свести использование циклов к минимуму. Эффективность приложений также определяется распределением памяти под создаваемые большие массивы.

```
clear all  
tic  
x=0:2*pi/10000:2*pi;  
y=exp(-x.^2).*cos(x);  
toc  
clear all  
tic  
for i=1:10001  
x(i)=2*pi/10000*(i-1);  
y(i)=exp(-x(i)^2).*cos(x(i));  
end  
toc
```

Рациональная техника программирования.

Работа этой файл-программы состоит из двух частей. Первая часть начинается с вызова функции `tic` - таймера, затем вычисляются значения двух массивов `x` и `y`, и вызывается функция `toc` – вывести на экран время в секундах, прошедшее с момента включения таймера.

Вторая часть файл-программы выполняет такие же действия начинается вновь с включения таймера. Затем в цикле `for` вычисляются значения массивов `x` и `y`. Обе части файл-программы делают практически одно и то же, но используя разные операторы. После запуска файл-программы на моем компьютере я получил, что время выполнения первой части 0.016 секунды, а второй - 1.36 секунды.

использование циклов снижает быстродействие программы более чем на два порядка.

Старайтесь свести использование циклов в программах к минимуму. Иногда бывает полезно делать вставки на `C`, `C++` или `Fortran` в тех местах программы, где циклы существенно снижают быстродействие.