Доля П.Г.
Харківський національний університет імені В. Н. Каразіна
факультет математики і інформатики
кафедра теоретичної і прикладної інформатики
2012 р.

Основы работы в системе MATLAB

> Прикладные математики, способные к глубокому общению с другими учеными и инженерами и знакомые с мощью и ограничениями цифровых машин … способны стать вождями завтрашнего математического мира …!
>
> Из книги Гаррета Бирхгоффа "Математика и психология".

Программа MATLAB представляет собой высокоуровневый технический вычислительный язык и интерактивную среду для разработки алгоритмов, визуализации и анализа числовых расчетов. В некотором смысле MATLAB это мощный калькулятор. Даже если вы только поверхностно знакомы с программой, вы можете использовать ее для выполнения невероятных вещей. Его название является сокращением от Matrix Laboratory. В MATLAB реализованы классические численные алгоритмы решения уравнений, задач линейной алгебры, вычисления определенных интегралов, аппроксимации, решения дифференциальных уравнений и их систем. MATLAB обладает хорошо развитыми возможностями визуализации двумерных и трехмерных данных. Его основой являются численные расчеты, но и символьные вычисления MATLAB умеет делать хорошо.

Символьные вычисления в MATLAB основаны на библиотеке, являющейся ядром пакета Maple. Решение уравнений и систем, интегрирование и дифференцирование, вычисление пределов, разложение в ряд и суммирование рядов, поиск решения дифференциальных уравнений и систем, упрощение выражений — вот далеко не полный перечень возможностей MATLAB для проведения аналитических выкладок и расчетов. Обязательным элементом символьной математики является поддержка численных расчетов произвольной точности.

В настоящем пособии мы даем описание только основных возможностей, которыми, как нам кажется, должен владеть каждый пользователь-математик системы MATLAB. Пособие предназначено в первую очередь для знакомства с математическими возможностями программы. Читайте текст и выполняйте примеры. Во многих случаях все пояснения дает само решение задачи. Надеемся, что по окончании выполнения последнего примера вы начнете применять систему MATLAB для решения ваших задач.

Оглавление

1. Знакомство с MatLab

1.1 Числа, векторы, матрицы, операции, выражения.

1.1.1 Знакомство

После запуска пакета MatLab вы увидите стандартное окно программы. Его основным элементом является командное окно, расположенное справа. Оно предназначено для ввода команд и отображения результатов вычислений. На первых порах мы будем работать в нем.

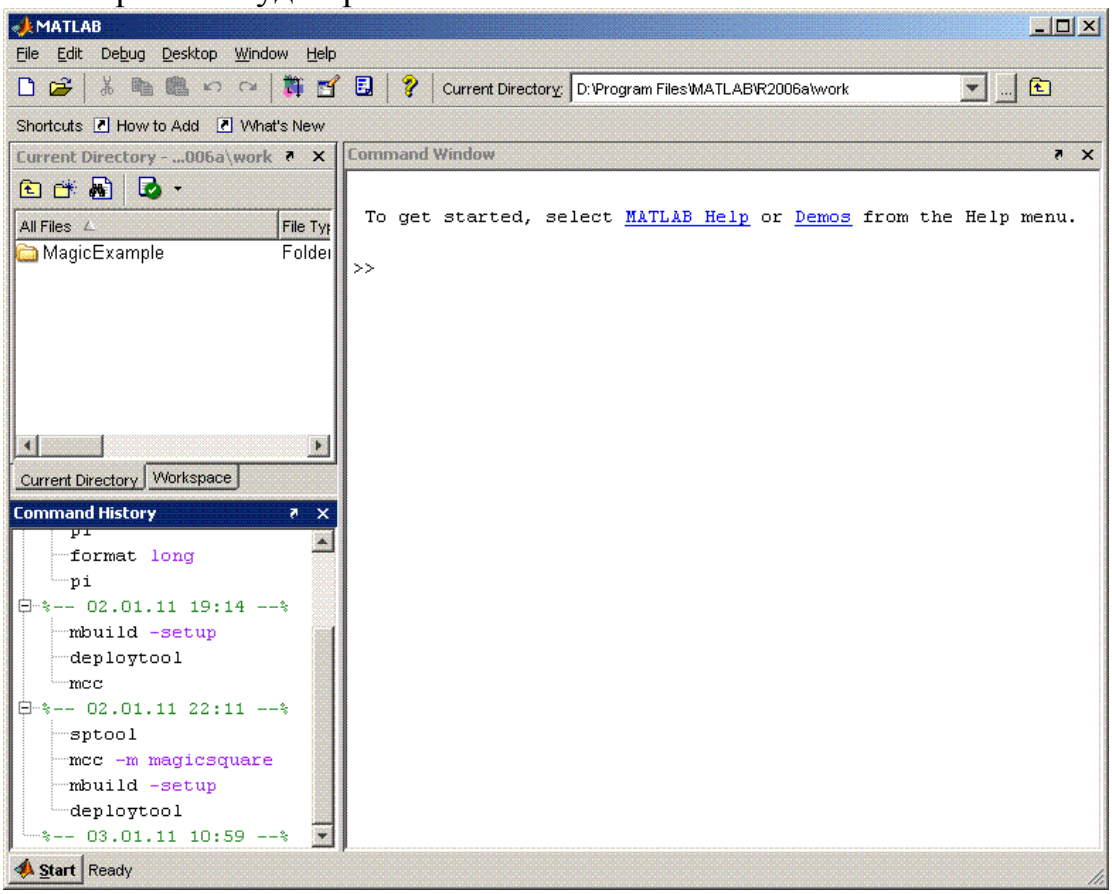


Рис. 1.1 Внешний вид окна программы

Наберите в командном окне после символа **>>: 2+2**. Теперь нажмите клавишу **Enter**. Вы увидите:

```
>> 2+2
ans =
4
```

Вероятно строки ввода и вывода будут разделены пустыми строками. Их мы в нашем пособии не печатаем. Символ **>>**, стоящий слева, является символом приглашения на ввод команд. Команды можно вводить справа от него или под ним. Символ **ans** представляет идентификатор результата (answer – ответ).

В нашем пособии текст, набранный шрифтом **Arial-жирный** или **Courier-Bold,** будет представлять команды (то, что мы вводим). Мы будем его печатать

без символа приглашения. Текст напечатанный шрифтом Curier – будет представлять то, что мы получим в результате вычисления. Иногда результаты вычисления мы печатать не будем.

MatLab это сокращение от **Mat**rix **Lab**oratory (матричная лаборатория). Числа в MatLab это матрицы. Чтобы ввести матрицу надо ввести ее как список: окружить квадратными скобками, разделять строки точкой с запятой, отделять числа одной строки пробелами. Введите после символа приглашения **>>**

A = [16 3 2 13; 5 10 11 8; 9 6 7 12; 4 15 14 1]

Выполним строку, нажав Enter. Получим

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

Теперь идентификатор A есть в рабочем пространстве и к матрице можно обращаться по ее имени A.

Операции + – * / между матрицей и числом означает прибавление (вычитание) числа к каждому элементу матрицы, умножение (деление) на число каждого элемента матрицы.

A+15
A*2
15-A

(но возведение в степень работает по-другому!).

Команды MatLab – это, как правило, вызов одной или нескольких функций по их имени. Аргументы функций набираются в круглых скобках и разделяются запятой. Например

sum(A)

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

Ответу всегда присваивается идентификатор. Если вы не ввели его, то MatLab сделает это сам (ans). Здесь мы подсчитали сумму элементов столбцов. MatLab предпочитает работать со столбцами. Чтобы просуммировать строки матрицу надо транспонировать.

A'

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

sum(A')'

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

Вектор (матрица – столбец) составленный из чисел, стоящих на диагонали матрицы, мы получим с помощью функции **diag.**

diag(A)
sum(diag(A))

Последняя команда просуммировала элементы, стоящие на главной диагонали матрицы.

Функция **fliplr(A)** зеркально отображает матрицу слева направо

fliplr(A)

и делает побочную матрицу главной. Тогда команда

sum(diag(fliplr(A)))

дает сумму элементов побочной диагонали.

Чтобы получить элемент матрицы надо написать **A(i,j)** – первый индекс номер строки, второй номер столбца. Например,

A(1,4) + A(2,4) + A(3,4) + A(4,4)

даст сумму элементов 4-го столбца

Можно ссылаться к элементам двумерной матрицы по одному индексу **A(k).** В этом случае она рассматривается как одномерный вектор, сформированный из столбцов матрицы. Нумерация индексов начинается с единицы. Например

A(15)

```
ans =
    12
```

это другой способ ссылаться к элементу A(3,4).

Если индекс неправильный, то получим сообщение об ошибке

A(3,5)

```
???  Index exceeds matrix dimensions.
```

Но если такому элементу присвоить значение, то размер матрицы увеличивается

A(3,5)=17

```
A =
    16     3     2    13     0
     5    10    11     8     0
     9     6     7    12    17
     4    15    14     1     0
```

Оператор двоеточие:

Выражение

1:10

```
ans =
     1     2     3     4     5     6     7     8     9    10
```

это вектор строка, содержащая целые числа от 1 до 10. Между начальным значением и конечным можно ввести приращение

1:2:10

Для получения обращенного интервала вводят отрицательное приращение

100:-10:50

```
ans =
   100    90    80    70    60    50
```

100:-7:50

или

0:pi/4:pi

```
0 0.7854 1.5708 2.3562 3.1416
```

Индекс с двоеточием **A(1:k, j)** выделяет часть матрицы

A(1:3,2)

```
ans =
     3
    10
     6
```

(из 2 – го столбца берем элементы от 1 – го до 3 – го). Например,

sum(A(1:4,4))

суммирует элементы 4 – го столбца.

Слово **end** в индексе матрицы означает выбор последнего столбца (или строки), а двоеточие – выбор всех элементов столбца (или строки).

A(end, :)

```
ans =
     4    15    14     1     0
```

sum(A(:,end)) – суммирует все элементы последнего столбца матрицы.

В MatLab есть функция, которая создает «магическую матрицу» любого размера (матрица у которой сумма элементов по столбцам, строкам, по диагонали и по побочной диагонали одинакова). Создадим такую матрицу размера 4 x 4

B=magic(4)

и переставим местами столбцы

A=B(:,[1 3 2 4])

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

Эта запись означает, что для каждой строки элементы переписываются в порядке 1 3 2 4.

Чтобы отменить вывод результатов вычисления в командное окно в конце выражения надо поставить точку с запятой.

B=magic(4);
A=B(:,[1 3 2 4])

1.1.2 Выражения

В MatLab нет необходимости определять тип переменной или размерности матрицы. Например

std=5

создает матрицу размером 1 x 1 с именем **std.** Имена переменных состоят из цифр, букв и символов подчеркивания. Заглавные и строчные буквы различаются. Различаются первые 31 символ в имени переменной.

В числах целая часть от дробной отделяется точкой. Научная система использует букву **e** для определения множителя степени десяти. Мнимые числа используют **i** и **j** как суффикс. Вот примеры правильной записи чисел

```
3    -99      0.0001
9.6397238   1.60210e-20   6.02252e23
1i     -3.14159j     2+4*i
```

Числа хранятся в формате с плавающей точкой, который обладает ограниченной точностью – примерно 16 значащих цифр, и ограничены диапазоном от 10^{-308} до 10^{308} .

Выражения используют обычные арифметические операции и правила старшинства: + – * / \(левое деление) ^(степень) '(комплексно сопряженное транспонирование). Это матричные операции, например, **A*A** это матричное умножение матрицы на себя. Для выполнения поэлементных операций перед знаком операции следует поставить точку, например, **A.*A** – поэлементное умножение матрицы на себя. Сравните

A = [16 3 2 13; 5 10 11 8; 9 6 7 12; 4 15 14 1];
A*A
A.*A

Чтобы ввести и выполнить несколько выражений или команд за один раз в командном окне после ввода первой команды следует нажать комбинацию клавиш Shift – Enter, ввести вторую команду, нажать снова комбинацию клавиш Shift – Enter и т.д. После ввода последней команды следует нажать Enter. Все команды будут выполнены и те, которые не заканчивались точкой с запятой, напечатают результат своей работы.

Изменить порядок выполнения операций можно с помощью круглых скобок ( ), которые также используются для указания аргументов функций. Если аргументом функции является вектор (матрица), то функция возвращает вектор или матрицу все элементы которых получены применением этой функции к каждому элементу вектора (матрицы).

x=0:pi/4:2*pi;
y=sin(x);
[x', y']

```
ans =
         0         0
    0.7854    0.7071
    1.5708    1.0000
    2.3562    0.7071
    3.1416    0.0000
    3.9270   -0.7071
    4.7124   -1.0000
    5.4978   -0.7071
    6.2832   -0.0000
```

Здесь мы напечатали таблицу значений функции **sin(x)** от 0 до 2π с интервалом π/4. Математические функции всегда выполняются поэлементно!

Стандартные функции имеют имена: abs sqrt exp sin и т.д. Список всех элементарных функций можно вывести командой **help elfun**. Команды

help specfun
help elmat

выводят список более сложных математических и матричных функций.

Стандартные функции встроены, более сложные, например **sinh**, реализованы в M – файлах и можно даже посмотреть их код. Несколько специальных функций представляют значения часто используемых констант, например, **pi** или **realmax** (наибольшее число с плавающей точкой; на моем компьютере оно равно 1.7977e+308), **Inf** (бесконечность), **eps** – относительная точность числа с плавающей точкой 2^{-52}, **NaN** используется для представления не числового значения, например, результат вычисления 0/0 или `Inf - Inf`.

Имена функций не защищены, поэтому можно менять их значения **pi=5.6**. Начальное значение можно восстановить следующим образом **clear pi**.

Приведем несколько примеров выражений MatLab

rho = (1+sqrt(5))/2

```
rho =
1.6180
```

a = abs(3+4i)

```
a =
5
```

z = sqrt(besselk(4/3,rho-i))

```
z =
0.3730 + 0.3214i
```
huge = exp(log(realmax))
```
huge =
1.7977e+308
```
toobig = pi*huge
```
toobig =
Inf
```
Набранное выражение, как правило, не заканчивается никаким символом. В этом случае результат вычисления будет отображаться в командном окне следом за введенной командой. Если вы желаете выполнить вычисления, а результат не выводить, то завершайте такие выражения точкой с запятой.

1.1.3 Работа с матрицами.

Есть функции, которые генерируют матрицы предопределенного размера и структуры. Например

Z = zeros(2,4) (**zeros** генерирует матрицу со всеми нулями)
```
Z =
0 0 0 0
0 0 0 0
```
F = 5*ones(3,3) (**ones** генерирует матрицу со всеми единицами)
```
F =
5 5 5
5 5 5
5 5 5
```
N = fix(10*rand(1,10)) (**rand** – генерирует матрицу со случайными числами на отрезке [0, 1]; **fix** округление до целого числа в сторону нуля)
```
N =
9 2 6 4 8 7 4 0 8 4
```
Аргументы функций **zeros, ones, rand** задают количество строк и столбцов генерируемой матрицы.

Справку по функции можно получить командой **help** *имя функции*

help fix
```
FIX    Round towards zero.
```
. . .

help rand
```
 RAND   Uniformly distributed random numbers.
```
. . .

Матрицы можно загружать из файлов. Команда **load** может загрузить текстовый файл, содержащий числовые данные, или двоичные файлы с матрицами, созданные в MatLab ранее. Текстовый файл должен быть сформирован в виде прямоугольной таблицы чисел, разделенных пробелами с равным количеством чисел в строке. Например, создадим в «Блокнот»–е файл `Matr1.dat` с таблицей чисел

```
1  2    3    4
5     6     7    8
9    10   11   12
```

```
13   14   15   16
```
и выполним команду

load Matr1.dat

Файл будет прочитан и в рабочем пространстве MatLab будет создана переменная с именем Matr1, содержащая матрицу. Содержимое рабочего пространства можно увидеть в окне Workspace. На рис.1.1 его можно увидеть слева от командного окна (левое верхнее). Для этого надо щелкнуть на закладку – корешок Workspace.

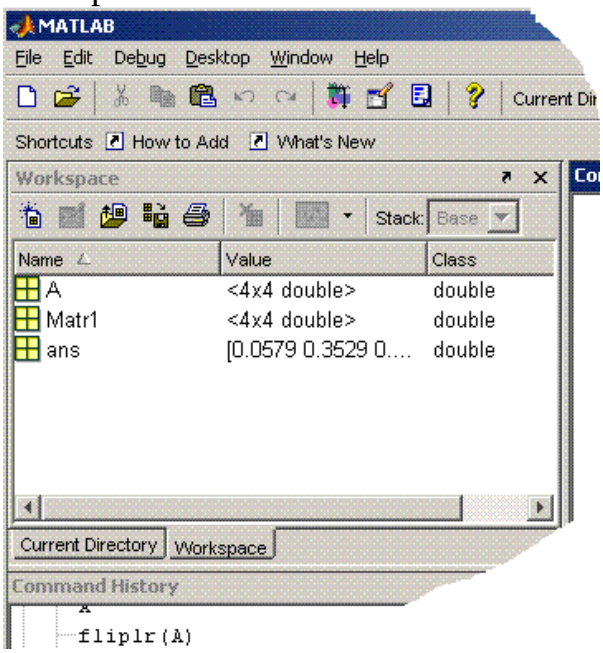


Рис. 1.2

В этом окне отображаются имена, тип и значения (если это возможно) переменных, доступных в текущем сеансе работы MatLab.

Чтобы проверить, что матрица с именем `Matr1` создана, просто наберите ее имя в командном окне

Matr1

Можно создавать матрицы, используя M-файлы, которые представляют текстовые файлы, содержащие код MatLab. Для создания такого файла вызовем встроенный текстовый редактор из меню `File - New - M-file`. В открывшемся окне введем

```
C=[16   3  2  0
   5       10   11   8
   9       6  7  12
   4       15   14   1];
```

Сохраним файл под именем W1.m . Тогда команда

W1

прочитает файл и создаст переменную с именем **C**, которая будет содержать матрицу. Вы можете увидеть имя загруженной матрицы в окне рабочего пространства или, набрав имя матрицы, напечатать ее содержимое.

Заметим, что для того, чтобы не указывать каталог расположения m файлов его название (каталога) должно быть указано в настройках MatLab. Это делается из меню `File - Set Path...`

Матрицы можно конструировать путем объединения уже имеющихся матриц. Это процесс создания матриц из маленьких. Пара квадратных скобок – это оператор объединения. Например

A=magic(4);

B = [A A+32; A+48 A+16]

```
B =
    16     2     3    13    48    34    35    45
     5    11    10     8    37    43    42    40
     9     7     6    12    41    39    38    44
     4    14    15     1    36    46    47    33
    64    50    51    61    32    18    19    29
    53    59    58    56    21    27    26    24
    57    55    54    60    25    23    22    28
    52    62    63    49    20    30    31    17
```

Результатом является матрица 8 x 8, получаемая соединением четырех матриц. Суммы в столбцах

sum(B)

```
ans =
   260   260   260   260   260   260   260   260
```

Сумма по строкам

sum(B')

```
ans =
   196   196   196   196   324   324   324   324
```

Удаление строк и столбцов. Чтобы удалить какой – то элемент матрицы (строку, столбец) его надо присвоить пустым квадратным скобкам.

X=A;

Удалим второй столбец командой

X(:,2) = [ ]

```
X =
    16     3    13
     5    10     8
     9     6    12
     4    15     1
```

Удаление одного элемента приводит к ошибке

X(1,2)=[ ]

```
???  Indexed empty matrix assignment is not allowed.
```

Однако использование одного индекса удаляет отдельный элемент и преобразует матрицу в строку.

X(3)=[ ]

```
X =
    16     5     4     3    10     6    15    13     8    12     1
```

или

X=A

X(2:2:10) = [ ]

```
X =
    16     9     2     7     3     6    15    13     8    12     1
```

1.1.4 Командная строка MatLab

Все приказы программе вводятся в командной строке командного окна, которая начинается символом **>>.** Форматом выводимых результатов можно управлять с помощью команды **format**, используя подходящие аргументы. Выполнив следующие примеры, вы легко поймете смысл аргументов этой команды.

x = [4/3 1.2345e-6]
format short; x
```
x =
1.3333 0.0000
```
format short e ; x
```
1.3333e+000 1.2345e-006
```
format short g ; x
```
1.3333 1.2345e-006
```
format long ; x
```
1.33333333333333 0.00000123450000
```
format bank ; x
```
1.33 0.00
```
format rat ; x
```
4/3 1/810045
```
format hex ; x
```
3ff5555555555555 3eb4b6231abfd271
```
Команда
format compact
отменяет вывод большого количества пустых линий при выводе.

Нажатие `Enter` в конце строки выполняет команду и выводит результат. Если закончить команду точкой с запятой, то результат в командное окно не выводится. Например, не обязательно смотреть на большую матрицу
A = magic(100);
Если выражение не умещается в одной строке, то используйте троеточие и `Enter` для обозначения продолжения на следующую строку.
```
s = 1 -1/2 + 1/3 -1/4 + 1/5 - 1/6 + 1/7 ...
-1/8 + 1/9 - 1/10 + 1/11 - 1/12;
```
Пробелы вокруг знаков + и – необязательны, но улучшают читаемость строки.

Если вы допустили ошибку в строке, то вместо того, чтобы ее набирать заново, нажмите стрелку ↑. Затем используйте стрелки ← →, чтобы перейти к ошибке и исправить ее. Повторное использование стрелки ↑ вызовет предыдущую команду. Наберите несколько первых символов, тогда клавиша ↑ найдет команду, начинающуюся с них.

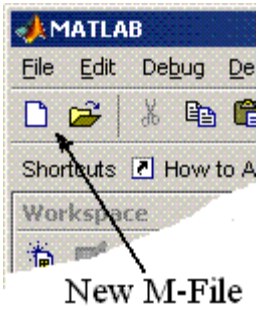
1.2 Сценарии и функции

Выше мы рассмотрели достаточно простые примеры, для выполнения которых требуется набрать несколько команд в командной строке. Для более сложных задач число команд возрастает, и работа в командной строке становится непродуктивной. 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.

Файлы, которые содержат код MatLab, называются М – файлами. Вы создаете такие файлы, используя текстовый редактор (можно использовать даже 'Блокнот'). Затем эти файлы используются как любая функция или команда. Существует два вида М-файлов:

- сценарии, которые не имеют входных и выходных аргументов; они используют данные (переменные) из рабочего пространства;
- функции, которые имеют входные и выходные аргументы; они используют локальные переменные.

Раскройте меню `File` рабочей среды MATLAB и в пункте `New` выберите подпункт `M-File` или нажмите кнопку `New M-file` на панели инструментов рабочей среды.



Новый файл открывается в окне редактора М-файлов. Вводите в каждой строке по одной команде MatLab, затем сохраните файл, например, под именем W1.m Этот файл называется сценарием. Чтобы увидеть содержимое М – файла в командном окне надо выполнить команду

type W1

Когда вызывается сценарий, MatLab просто выполняет команды, содержащиеся в файле. Сценарии берут данные из рабочего пространства и создают переменные, которые остаются в рабочем пространстве. В качестве примера создадим сценарий, исследующий ранг магических матриц разного размера. Выберем пункт меню `File – New – M-file`. Откроется текстовый редактор, в котором введите следующий код

```
% Исследование ранга магических матриц
r = zeros(1,32);
for n = 3:32
r(n) = rank(magic(n));
end
r
bar(r)
```

Как видим, строка с комментарием начинается знаком % (процент). Сохраните файл под именем SC_01.m. в одном из рабочих каталогов MatLab (напомним, что добавление/удаление таких каталогов выполняется из пункта меню `File – Set Path…`). Для выполнения сценария в командной строке наберите

SC_01

Сценарий выводит одномерный массив значений рангов матриц и строит столбчатую диаграмму этих значений.

Когда курсор находится в теле сценария, выполнить его код можно нажатием клавиши `F5`.

В теле файлов сценариев и функций можно использовать управляющие конструкции, такие как **for**, которая организует циклическое вычисление команд, стоящих после нее до первой команды **end.**

Команда **return** выполняет досрочное завершение сценария. Все команды, стоящие в коде после нее, не будут выполняться.

Команды управления потоками (элементы языка программирования MatLab) будут описаны во второй части нашего пособия.

Функции (по – другому называемые файл – функциями) – это М – файлы, которые могут иметь входные аргументы и возвращать значения. Имя М – файла и функции должно быть одинаковым. Функции работают с переменными в пределах собственного рабочего пространства, отличного от того с которым вы оперируете в командной строке.

Хорошим примером является функция **rank**. Ее код находится в файле `toolbox/matlab/matfun/rank.m` Чтобы посмотреть содержимое этого файла выполните команду

type rank

```
function r = rank(A,tol)
%RANK   Matrix rank.
%   RANK(A) provides an estimate of the number of linearly
%   independent rows or columns of a matrix A.
%   RANK(A,tol) is the number of singular values of A
%   that are larger than tol.
%   RANK(A) uses the default tol = max(size(A)) * norm(A) * eps.

%   Copyright 1984-2000 The MathWorks, Inc.
%   $Revision: 5.9 $  $Date: 2000/06/01 02:04:15 $

s = svd(A);
if nargin==1
   tol = max(size(A)') * max(s) * eps;
end
r = sum(s > tol);
```

Первая строка функции в М – файле начинается со слова `function`, после которого задаются выходные переменные, имя функции и входные аргументы. В нашем примере используется один выходной параметр `r` и два входных аргумента `A` и `tol`. Следующие несколько строк до первой пустой строки или первой выполняемой строки являются комментариями, которые выводятся по команде

help rank

При этом самая первая строка комментария несет еще одну нагрузку. Она используется для поиска функций по ключевым словам командой

lookfor rank

Остальное содержимое М – файла представляет исполняемый код. Все переменные: `s` - внутренняя, `r` - возвращаемая, `A` и `tol` – аргументы являются локальными. Функция **rank** может быть использована несколькими способами

rank(A)
r = rank(A)
r = rank(A, 1.e-6)

Обратите внимание, что функция **rank** может вызываться с одним или двумя аргументами. Если нет выходного аргумента, то результат сохраняется в переменной `ans`. Если нет второго входного аргумента, то функция присваивает ему значение по умолчанию. Внутри тела функции можно использовать переменные `nargin` и `nargout`, которые хранят число входных

и выходных аргументов при каждом использовании функции. Рассмотренная функция **rank** использует только переменную `nargin`.

Если надо использовать одну копию какой-то переменной в нескольких функциях, то ее надо объявить как `global` во всех функциях. То же надо сделать в командной строке, если вы хотите, чтобы основное рабочее пространство получило доступ к этой переменной. Например, создадим файл `falling.m`, вычисляющий пройденный путь свободно падающего тела за указанное время

```
function h = falling(t)
global GRAVITY
h = 1/2*GRAVITY*t.^2;
```

Затем в командной строке введем следующий код

```
global GRAVITY
GRAVITY = 10;
y = falling((0: .1: 5)' );
```

Определение `global` должно стоять до первого использования ее в теле функции или рабочем пространстве.

Имя файл-функции не обязательно должно совпадать с именем файла, однако обращение к ней происходит по имени файла. Например, если в файле `f22.m` содержится функция с заголовком `g = init(z)`, то ее следует вызывать так:

```
f = f22(- 0.9)
```

а вовсе не

```
f = init(- 0.9)
```

(не работает !)

Чтобы не было путаницы, всегда давайте имя m - файлу такое же, как у функции.

На сеанс можно создавать временные функции с помощью команды inline, которой в качестве аргумента надо передать выражение, заключенное в кавычки

```
ff=inline('x^2/2-x^4/4')
ff =
     Inline function:
     ff(x) = x^2/2-x^4/4
```

С такой функцией можно выполнять любые допустимые операции, например, можно вычислять ее значение в точке

```
ff(1)
ans =
         0.25
```

или строить график с помощью функции **fplot**

```
fplot(ff,[-2 2])
```

О графических функциях мы подробнее поговорим в следующем параграфе.

С помощью `inline` можно создавать функции 2-х переменных

```
fff=inline('x.^2+y.^2')
fff =
     Inline function:
     fff(x,y) = x.^2+y.^2
```

Тогда график этой функции можно построить следующим образом

```
ezsurf(fff,[-1 1 -1 1]);
```

Замечание. *При создании функций следует всегда ставить точку для указания поэлементных операций. Точку можно не ставить, когда арифметическая операция выполняется между переменной и числом (константой).*

Функции можно создавать с помощью дескриптора @.

zz=@(x) exp(-x).*sin(10.*x)./(10*x)

```
zz =
    @(x) exp(-x).*sin(10.*x)./(10*x)
```

Идентификатор, стоящий в левой части равенства становится именем функции. Справа после знака @ в скобках следует перечислить аргументы функции. С такой функцией также можно выполнять любые допустимые операции – вычислять ее значение в точке, строить график, находить корни, интегрировать и т.д.

zz(0.01)

```
ans =
    0.9884
```

fplot(zz,[-1 3])

Вместо аргументов функциям можно передавать вектора и матрицы. Так выражение

g=@(x,y) x.^2+y.^2

g([1 2], [3 4])

```
ans =
    10    20
```

дает значения функции в точках (1, 3) и (2, 4).

Команды и функции взаимозаменяемы. Пример команд

help

Многие команды используют управляющий параметр, который определяет последующее действие

load имя файла

help magic

type rank

Другой способ использования командных параметров состоит в создании строки аргументов функции

load('имя_файла.dat')

help('magic')

type('rank')

Любая команда типа **command argument** может быть записана в виде **command('argument').** Это называется командно – функциональной двойственностью. Преимущества функционального подхода проявляется, когда командная строка создается из нескольких частей.

1.3 Графика

Графические возможности пакета весьма обширны. Для первого знакомства мы приведем краткое описание простейших функций.

1.3.1 Базовые графические объекты

Все графики, кривые и поверхности, которые можно построить в MatLab, являются комбинацией некоторого количества простейших – отрезков,

многоугольников и т.д. Комбинируя их определенным образом, мы можем получить практически любой рисунок. Изучение всех возможностей мы отложим до второй части нашего пособия. Здесь же мы рассмотрим основные примитивы, которые могут понадобиться в самом начале вашей работы.

Простейшей функцией является **line.** Она строит отрезки и ломанные. Ее первым аргументом является вектор *x* координат узлов ломаной, а вторым – вектор *y* координат узлов. Для построения пространственной ломаной используется третий аргумент – вектор *z* координат узлов. После задания векторов можно задать свойства – толщину линий, цвет и т.д. в формате

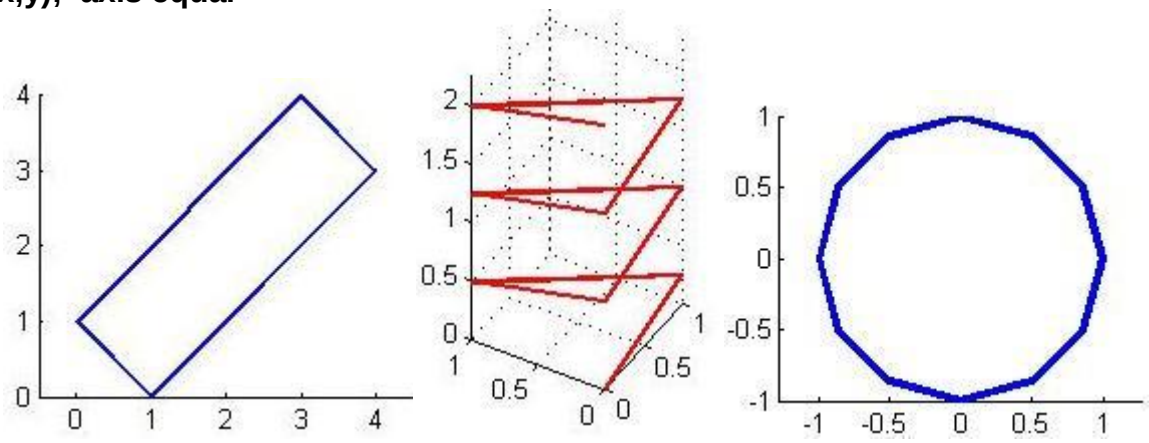
`line(X,Y,Z,'PropertyName',PropertyValue,...).`

Например

```
line([1 4 3 0 1],[0 3 4 1 0]); axis equal
line([0 1 1 0 0 1 1 0 0],[0 0 1 1 0 0 1 1 0],[0 1 2 3 4 5 6 7 8], 'Color','r','LineWidth',2)
grid on;      % рисует сетку на координатных плоскостях
```

или

```
t=0:pi/6:2*pi; x=cos(t); y=sin(t);
line(x,y);  axis equal
```



Команда **axis equal** делает отметки приращений осей одинаковой длины. Каждая команда **line** добавляет графический объект ломаной в текущее графическое окно. Очистка окна от старого рисунка может быть выполнена командой

```
clf
```

Можно также перед построением нового рисунка закрыть графическое окно кнопкой – крестиком в правом верхнем углу окна или командой

```
close
```

Команда **rectangle** рисует прямоугольник, эллипс или фигуру близкую им по форме. Команда `rectangle('Position',[x,y,w,h])` рисует прямоугольник с вершиной в точке (*x*,*y*) шириной *w* высотой *h*. Свойство **Curvature** определяет кривизны сторон.

`rectangle('Position',[x,y,w,h], 'Curvature',[`c_g`,`c_v`])`

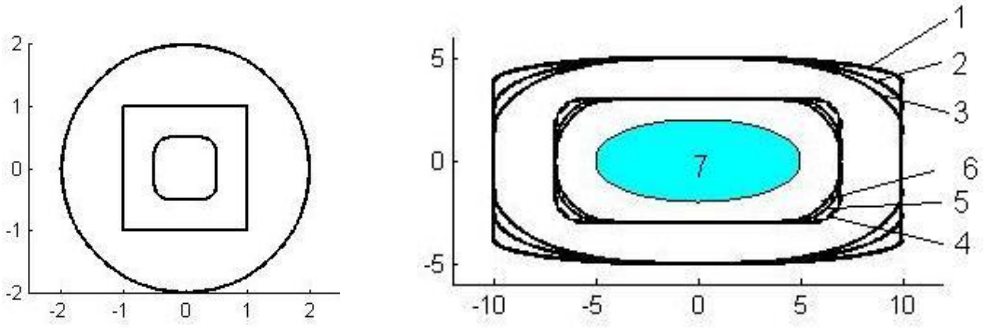
Если оно опущено или задано равным [0 0], то отрезки сторон будут прямые, и мы получаем прямоугольник. Задание значений кривизны в интервале от 0 до 1 искривляет стороны фигуры – от прямолинейных (при c_g=0 или c_v=0) до эллиптических (при c_g=1 или c_v=1). Первое число в паре [c_g,c_v] управляет

кривизной горизонтальных сторон, второе – вертикальных. Слово кривизна используется только для обозначения параметра функции и не является кривизной в точном математическом смысле этого значения. Можно использовать одно число, тогда оно относится к меньшей из сторон фигуры.

```
clf
rectangle('Position',[-2 -2 4 4], 'Curvature',[1,1],'LineWidth',2);
rectangle('Position',[-1 -1 2 2], 'LineWidth',2);
rectangle('Position',[-1/2 -1/2 1 1], 'Curvature',[0.5,0.5],'LineWidth',2);
axis equal
```

Построенные этими командами фигуры показаны на следующем рисунке слева. Фигуры, построенные следующими командами, показаны на рисунке справа. Номера кривых соответствуют номеру строки команды **rectangle** в коде.

```
clf
rectangle('Position',[-10,-5,20,10],'Curvature',[0.8,0.25], 'LineWidth',2);
rectangle('Position',[-10,-5,20,10],'Curvature',[0.8,0.5], 'LineWidth',2);
rectangle('Position',[-10,-5,20,10],'Curvature',[0.8,0.75], 'LineWidth',2);
 rectangle('Position',[-7,-3,14,6],'Curvature',0.2, 'LineWidth',2);
rectangle('Position',[-7,-3,14,6],'Curvature',0.5, 'LineWidth',2);
rectangle('Position',[-7,-3,14,6],'Curvature',1, 'LineWidth',2);
rectangle('Position',[-5,-2,10,4],'Curvature',[1,1],'FaceColor','c')
daspect([1,1,1])
xlim([-12,12])
ylim([-6,6]);
```



Команда **daspect([1,1,1])** определяет одинаковый масштаб единиц по осям. А команда **daspect([1 3 1])** означает, что одна единица по оси *x* равна по длине трем единицам по оси *y* и одной единице по оси *z*, т.е. по оси *y* происходит сжатие.

Опция **'FaceColor','c'** приводит к построению закрашенной фигуры цветом **'c'** – cyan. Команды **xlim([-12,12])** и **ylim([-6,6])** задают границы отображаемой области.

Команда **fill(x,y)** – рисует закрашенный многоугольник с *x* координатами вершин, заданными в первом векторе, и *y* координатами – во втором. При этом последняя вершина не обязана совпадать с первой. Третий аргумент задает цвет заливки многоугольника

```
clf; fill([1 2 1 0],[0 1 2 1],'g'); daspect([1 1 1])
```

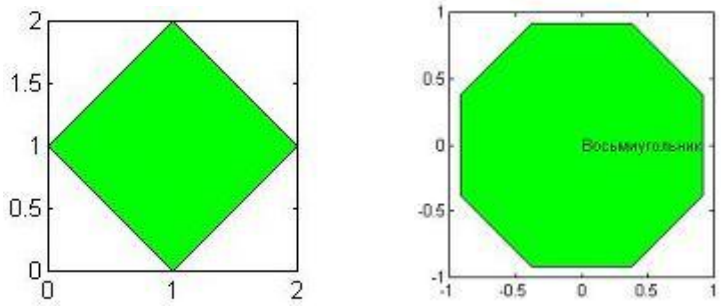
Фигура показана на следующем рисунке слева.

Код, приведенный далее, строит восьмиугольник, в середину графического окна выводит текст '**Восьмиугольник**', а в командное окно выводит форматированный текст с координатами одной из его вершин.

```
clf
```

```
t= (1/16:1/8:1)'*2*pi; x = sin(t); y = cos(t);
fill(x,y,'g'); axis square
text(0,0,'Восьмиугольник')
str=sprintf('Координаты второй вершины: x(2)=%6.2f y(2)=%6.2f',x(2),y(2));
disp(str);
```

```
Координаты второй вершины: x(2)=  0.92 y(2)=  0.38
```



Команда **text(x,y,'текст')** в точку с координатами `(x,y)` выводит заданный текст.

Функция **sprintf** создает строку, составленную из текста и отформатированных чисел. Формат задается символами **%** и буквой (в данном примере **f**). Между процентом и буквой могут стоять уточняющие символы – флаги и числа, определяющие ширину поля вывода. Переменные, значения которых будет записываться в строку, являются вторым, третьим и т.д. аргументами функции **sprintf**. Первой форматирующей последовательности символов **%...f** соответствует второй аргумент функции **sprintf** (имя первой переменной), второй последовательности **%...f** – третий аргумент (имя второй переменной) и т.д. В нашем примере мы сказали, что значения `x(2)` и `y(2)` должны быть отображены шестью цифрами, из которых две должны стоять после десятичной точки. Неиспользованные места цифр заполняются пробелами. Кроме буквы **f** для отображения чисел используется большое количество других символов – **g, d, e** и т.д. С ними вы можете познакомиться в справочной системе, например, выполнив команду

doc sprintf

Команда **disp(str)** выводит строку **str** в командное окно. При формировании текстовой константы ее следует заключать в одинарные кавычки.

1.3.2 Функции и кривые

Самая простая функция **plot** имеет много форм. Для построения одного графика она использует два вектора – первый вектор значений независимой переменной и второй – вектор значений функции в этих точках. Например, для построения графика функции `sin` от 0 до 2π выполним команды

```
t = 0:pi/100:2*pi;
y = sin(t);
plot(t,y)
```

Мы построили два вектора **t** и **y**, которые передали в качестве аргументов функции **plot.** Обратите внимание, что для графика создается отдельное окно. MatLab автоматически присваивает каждому графику свой цвет. Вот как строится 3 графика.

```
y2 = sin(t-.25);
y3 = sin(t-.5);
plot( t, y, t, y2, t, y3)
```

Можно задавать цвет, стиль линий и маркеры следующим образом

plot(x, y, 'цвет стиль маркер')

(пробелы не обязательны). Цвета определяются символами c , m , y , r , g , b, w, k (черный). Тип линии определяются символами: – (минус) сплошная линия, **--** (два минуса) разрывная линия, **:** (двоеточие) пунктирная линия, **–.** (минус точка) штрих пунктирная, '**none**' нет линии. Наиболее часто встречающиеся маркеры + , h , * , o.

```
plot(t,y, 'y:+')
```

Функция **plot** автоматически создает новое окно изображений, если до этого оно еще не было открыто. Повторные графики заменяют изображение в существующем окне.

Если надо создать новое окно, то используйте команду **figure** или **figure(n)**, которое n-е окно делает текущим. Номер окна отображается в его заголовке. Команда

```
hold on
```

включает режим добавления графиков в текущее окно

```
x=0:0.01:6;
y=1.2*sqrt(x);
plot(x,y)
```

Команда **hold off** снова включает режим замены графиков.

Команды

```
t=[0:0.1:2*pi];
plot(cos(t),sin(t))
```

рисуют окружность по ее параметрическому уравнению, но она выглядит как овал. Чтобы это исправить, надо управлять осями графика командой axis.

axis square - создает оси одинаковой длины (делает круг кругом)

axis equal - делает отметки приращений осей одинаковой длины (круг тоже будет кругом)

axis([xmin xmax ymin ymax]) - определяет диапазон отображаемой области

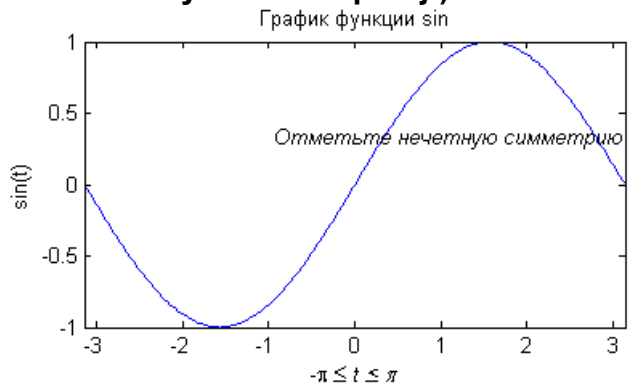
axis auto - возвращает значения по умолчанию

axis on, axis off - включают и выключают отображение осей

grid on, grid off - включают и выключают сетку координат

Функции **xlabel, ylabel, zlabel** добавляют подписи к осям, а **title** – добавляет заголовок в верхней части окна.

```
t = -pi:pi/100:pi;
y = sin(t);
plot(t,y)
axis([-pi pi -1 1])
xlabel( ' -\pi \leq \itt \leq \pi ' )
ylabel( ' sin(t) ' )
title('График функции sin')
text(-1,1/3,'\it {Отметьте нечетную симметрию}')
```



Обратите внимание, что функции, выводящие текст – **text, xlabel, ylabel,** распознают теги TeX – а.

Если вы имеете файл – функцию, то для построения ее графика можно использовать функцию **fplot.** При обращении к ней требуется указать имя этой файл – функции (в апострофах) или указатель на нее (с оператором @ перед именем функции) и границы отрезка для построения графика (в векторе из двух элементов).

Создадим в редакторе m – файлов функцию `myfun`

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

и построим ее график командой

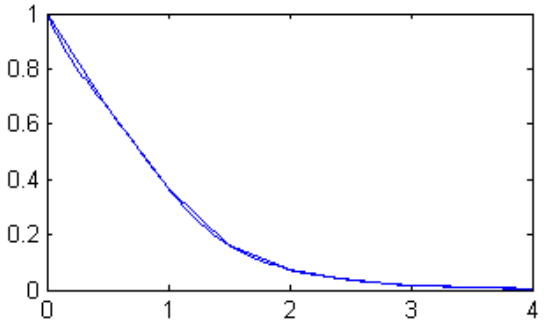
```
fplot('myfun',[0 4])
```

или командой

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

Постройте графики **plot** и **fplot** на одних осях (используйте **hold on**) так, как показано на следующем рисунке.

```
x=0:0.5:4;
y=myfun(x);
plot(x,y)
hold on
fplot('myfun',[0 4])
```



График, построенный **fplot**, более точно отражает поведение функции, т. к. алгоритм **fplot** автоматически подбирает шаг аргумента, уменьшая его на участках быстрого изменения исследуемой функции.

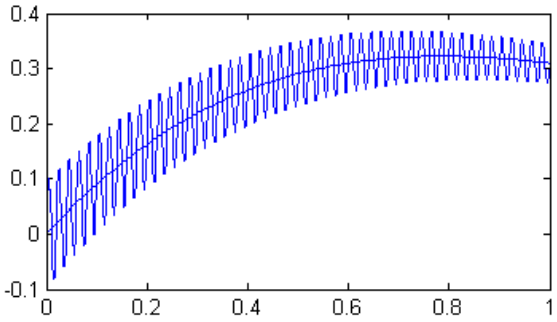
Повторим еще раз. Имеющаяся в нашем распоряжении файл-функция **myfun** позволяет обратиться к специальной функции **fplot**, которой требуется указать имя нашей файл – функции (в апострофах) или указатель на нее (с оператором @ перед именем функции).

При построении графика существенную роль играет выбор шага. Неудачный выбор шага может привести к неверному результату. Создадим, например, следующую файл – функцию

```
function z=myfun2(t)
z=exp(-t).*(sin(t)+0.1*sin(100*pi*t));
```

и построим ее графики с помощью **plot** и **fplot** на отрезке [0, 1]

```
x=0:0.01:1;
y=myfun2(x);
plot(x,y)
hold on
fplot('myfun2',[0 1])
```



Как видим, правильный выбор шага иногда важен.

Третьим аргументом функции **fplot** можно указать тип линии и маркера, так же, как и в **plot.** Положение маркеров покажет те точки, которые выбрала функция **fplot** для построения графика.

```
fplot('myfun',[0 4],'.-')
```

Аргументом функции **fplot** может выступать выражение, заключенное в кавычки

```
fplot('exp(-0.2*x)*sin(x^2)',[0,4*pi])
fplot('sin(x)^2',[0 12])
```

Можно использовать имена стандартных функций

```
fplot('sin',[0 12])
```

Вот пример одновременного построения двух графиков

```
fplot('[cos(x)^2,sin(x)]',[0,4*pi])
```

Отметим, что функция **fplot** бедна на опции форматирования вывода. Однако если мы принимаем результат функции **fplot**, то получаем пару векторов и функция **fplot** не рисует график. Потом график можно нарисовать обычным способом и задать любые опции, которые не поддерживает **fplot.**

```
[X,Y]=fplot('sin(x)/x',[-10,10]);
plot(X,Y,'LineWidth',3)
```

1.3.3 Поверхности

Для построения поверхностей в MatLab есть функции аналогичные **plot**, которые более сложны в использовании. Мы опишем их во второй части нашего пособия.

Есть простые в использовании функции, аналогичные функции **fplot.** Большинство из них начинается с префикса `ez`. Аналогом функции **fplot** в этой группе функций является функция **ezplot.** Ее первым аргументом является строковое выражение функции (в кавычках), а вторым – диапазон изменения независимой переменной.

ezplot('x^2*sin(x)',[0 40])

Отметим, что в выражениях для этой функции (и всех функций этой группы) следует использовать знаки арифметических операций без точек.

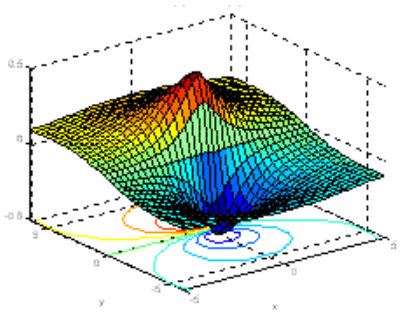
Для построения поверхностей в этой группе имеются функции **ezsurf, ezsurfc, ezmesh** и другие. Первым аргументом они принимают строковое выражение функции (в кавычках) двух переменных, а вторым – диапазон изменения этих переменных. Если диапазона нет, то он назначается системой автоматически. Вот несколько примеров построения поверхностей

ezsurfc('sqrt(x^2 + y^2)');
ezsurfc('sqrt(x^2 + y^2)',[-1 1]);
ezsurfc('sqrt(x^2 + y^2)',[-1 1 0 2]);

Отметим, что интервалы **[-1 1 0 2]** изменения переменных **[-1 1]** и **[0 2]** определяются порядком следования в алфавите имен аргументов **x** и **y.**

Следующий график строится над областью $-5 < x < 5, -2\pi < y < 2\pi$ на сетке 35 x 35

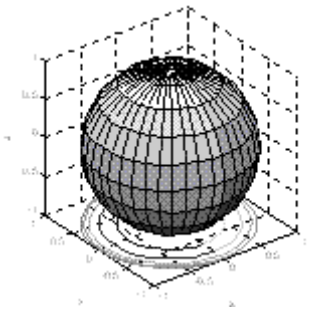
ezsurfc('y/(1 + x^2 + y^2)',[-5,5,-2*pi,2*pi],35)
colormap('default')



Функция **colormap('default')** определяет цветовую гамму рисунка, в данном случае цветовая гамма устанавливается стандартной.

Поверхность сферы по ее параметрическому уравнению можно построить так

ezsurfc('cos(v)*cos(u)','cos(v)*sin(u)','sin(v)',[-pi/2,pi/2,0,2*pi],21);
colormap(gray(128))

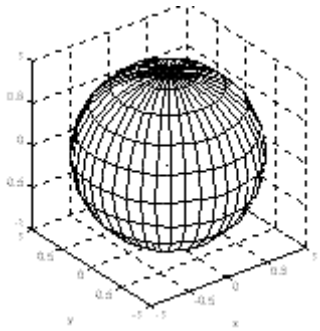


Обратите внимание, что на плоскости XY построены контурные линии. Использование функции **ezsurf** (без последней буквы c) построит ту же поверхность без контурных линий на плоскости XY

ezsurf('cos(v)*cos(u)','cos(v)*sin(u)','sin(v)',[-pi/2,pi/2,0,2*pi],21)

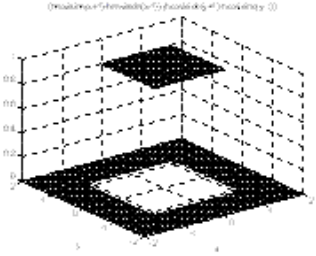
Каркасную поверхность сферы с удаленными невидимыми линиями можно послучить с помощью функции **ezmesh** следующим образом

ezmesh('cos(v)*cos(u)','cos(v)*sin(u)','sin(v)',[-pi/2,pi/2,0,2*pi],21)
colormap([0 0 0])
axis equal



Можно строить даже разрывные поверхности

ezsurfc('(heaviside(x+1)-heaviside(x-1))*(heaviside(y+1)-heaviside(y-1))',[-2,2],51)
ezmesh('(heaviside(x+1)-heaviside(x-1))*(heaviside(y+1)-heaviside(y-1))',[-2,2],51)
colormap([0 0 0])

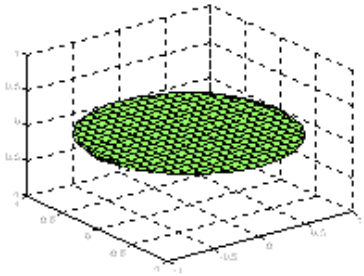


Можно нарисовать плоскую область как плоскую поверхность с краем, например, круг

ezmesh('u','(abs(sqrt(1-u^2)+v)-abs(sqrt(1-u^2)-v))/2','0',[-1,1,-1,1],21)

или

ezsurf('u','(abs(sqrt(1-u^2)+v)-abs(sqrt(1-u^2)-v))/2','0',[-1,1,-1,1],21)



или поверхность над ним

ezmesh('u','(abs(sqrt(1-u^2)+v)-abs(sqrt(1-u^2)-v))/2','u^2+((abs(sqrt(1-u^2)+v)-abs(sqrt(1-u^2)-v))/2)^2',[-1,1,-1,1],21)

Но если областью изменения аргументов функции является круг, то проще использовать опцию этих функций **'circ'**

ezmesh('x*y','circ')

Функции группы ez... можно использовать и со стандартными файл – функциями MatLab, но при этом в формулах следует использовать знаки поэлементных арифметических операций (с предшествующей точкой). Для примера создадим следующую m – функцию

```
function x=xellipse(u,v)
x=2*cos(v).*cos(u);
```

Ее график можно построить следующим образом

ezsurf('xellipse',[0 2*pi -pi/2 pi/2])

У этой функции есть один недостаток – вместо числа 2 хотелось бы иметь аргумент, которому можно было бы присваивать произвольные значения. Но использование функции с тремя аргументами при вызове функции **ezsurf** приведет к ошибке. Изменим тело функции следующим образом

```
function x=xel(u,v,a)
x=a.*cos(v).*cos(u);
```

Чтобы построить ее график надо создать локально функцию двух переменных **@(u,v)xel(u,v,2)** и использовать ее при вызове графической функции

ezsurf(@(u,v)xel(u,v,2), [0 2*pi -pi/2 pi/2])

Создадим еще две аналогичные функции

```
function y=yel(u,v,b)
y=b.*cos(v).*sin(u);
```

и

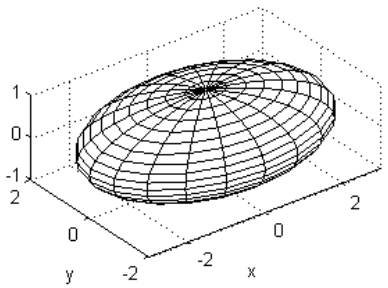
```
function z=zel(u,v,c)
z=c.*sin(v);
```

Тогда можно построить поверхность эллипсоида по его параметрическому уравнению, подставляя произвольные значения полуосей

ezsurf(@(u,v)xel(u,v,3),@(u,v)yel(u,v,2),@(u,v)zel(u,v,2), [0 2*pi -pi/2 pi/2])

или каркасную поверхность эллипса

ezmesh(@(u,v)xel(u,v,3),@(u,v)yel(u,v,2),@(u,v)zel(u,v,1), [0 2*pi -pi/2 pi/2])
axis equal
colormap([0 0 0])



Функции, созданные с помощью дескриптора @, можно присваивать идентификаторам и использовать их имена при вызове графических функций

x=@(u,v)xel(u,v,3) ;
y=@(u,v)yel(u,v,2);
z=@(u,v)zel(u,v,1);
ezmesh(x,y,z, [0 2*pi -pi/2 pi/2],21)

Функции можно создавать сразу по месту, без создания m – файлов и строить графики поверхностей

x=@(u,v) u ;
y=@(u,v) (abs(sqrt(1-u.^2)+v)-abs(sqrt(1-u.^2)-v))/2;
z=@(u,v) u.^2+v.^2;
ezmesh(x,y,z,[-1 1],21)

или, используя построенные функции `x(u,v)`, `y(u,v)`,

ezmesh(x,y,'0',[-1 1],21)

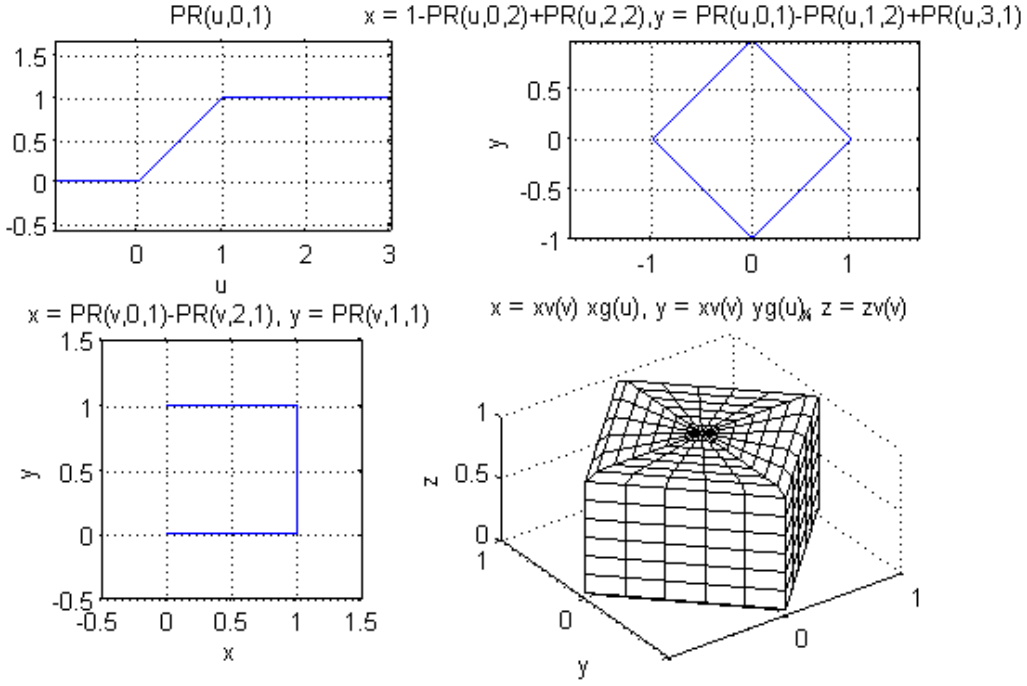
В следующем упражнении построим параллелепипед по его параметрическим уравнениям. Каждые несколько строк будем пояснять рисунком. Чтобы рядом построить несколько графиков используется функция **subplot(m,n,k)**, где **m** – количество рядов графиков, **n** – количество столбцов, **k** – номер текущего графика (тот в который следующая графическая функция будет выводить результат). Графики нумеруются построчно. Результат всех построений мы собрали в следующий файл сценарий

```
PR=@(x,a,w) (w+abs(x-a)-abs(x-a-w))/2;
subplot(2,2,1);ezplot(@(u)PR(u,0,1),[-1 3]);
xg=@(u) 1-PR(u,0,2)+PR(u,2,2);
yg=@(u) PR(u,0,1)-PR(u,1,2)+PR(u,3,1);
subplot(2,2,2);ezplot(xg,yg,[0 4]); axis equal;
xv=@(v) PR(v,0,1)-PR(v,2,1);
zv=@(v) PR(v,1,1);
subplot(2,2,3); ezplot(xv,zv,[0 3]); axis equal;
xs=@(u,v) xv(v).*xg(u);
ys=@(u,v) xv(v).*yg(u);
zs=@(u,v) zv(v);
subplot(2,3,6);ezmesh(xs,ys,zs,[0 4 0 3],21);colormap([0 0 0]);axis equal;
```

Мы рекомендуем вам вначале набирать и выполнять этот код по несколько строк. Вначале введите и выполните первые две строки – получите график функции **PR(u,0,1)**, затем еще три строки – получите квадрат, построенный по его параметрическому уравнению x=xg(u), y=yg(u). Заметим, что полезно построить отдельно график каждой из координатных функций xg(u) и yg(u). Следующие три строки вводят и строят ломаную в форме скобы. Последние четыре строчки вводят координатные функции параметрического уравнения поверхности параллелепипеда, и по ним функция **ezmesh** строит каркас поверхности. После отладки строк кода скопируйте их в редактор m – файлов и сохраните под именем parallelepiped.m. Затем выполните команду

parallelepiped

В результате вы увидите графическое окно с четырьмя рисунками



Использование функции двух переменных f(x,y) при вызове функции **ezplot** позволяет построить кривую f(x,y)=0 по ее неявному уравнению

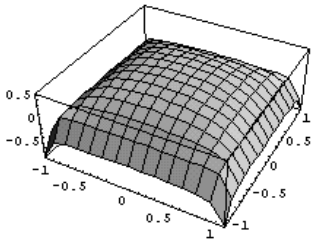
```
w=@(x,y) 1-x.^2-y.^2;
ezplot(w,[-1 1 -1 1]); axis equal
```

В следующем примере строится квадрат по его неявному уравнению w(x,y)=0

```
w=@(x,y) 2-x.^2-y.^2-sqrt((1-x.^2).^2+(1-y.^2).^2);
ezplot(w,[-1.2,1.2])
```

Заодно, посмотрите как выглядит поверхность функции z=w(x,y)

```
ezmesh(w,[-1.2 1.2 -1.2 1.2])
colormap([0 0 0])
```



Несколько полезных графических команд

Команда

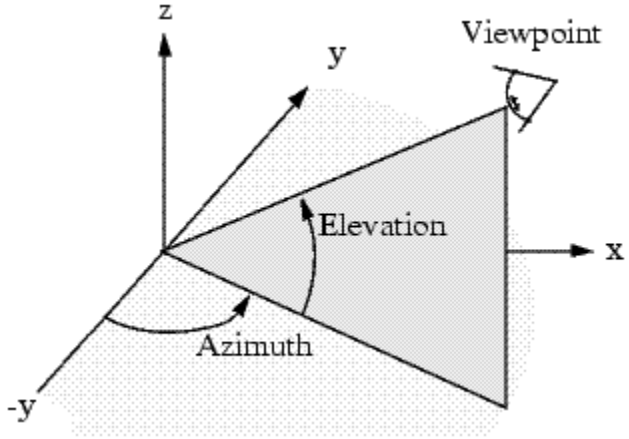
newplot

создает пустое графическое окно. Команда **view** позволяет посмотреть на трехмерный рисунок с другого направления. Например

view(2) % вид со стандартного направления 2

view(3) % вид со стандартного направления 3

Команда **view(Azimuth,Elevation)** определяет вид с направления, задаваемого углом **Azimuth,** отсчитываемым вокруг оси **z** и углом **Elevation**, отсчитываемым от плоскости XY (см. рисунок). Углы задаются в градусах.



Например

view(60,-45)

Команда

colorbar

выводит рядом с рисунком столбик цветовой палитры. Команда

shading interp

сглаживает цвета на трехмерном графике.

Команда **box on** отображает прямоугольник или прямой параллелепипед вокруг графической области. Команда **box off** стирает охватывающий

прямоугольник или параллелепипед. Обычно команду **box** полезно использовать, когда строятся трехмерные фигуры.

1.3.4 Простейшая анимация

При изучении движения точки на плоскости или в трехмерном пространстве полезно не только построить траекторию точки, но и следить за движением точки по траектории. MATLAB предоставляет возможность получить анимированный график, на котором кружок, обозначающий точку, перемещается на плоскости или в пространстве, оставляя за собой след в виде линии — траектории движения. График похож на летящую комету с хвостом. Поэтому для построения таких анимированных графиков применяются функции **comet** и **comet3**.

```
t=[0:0.001:10];
x=sin(t)./(t+1);
y=cos(t)./(t+1);
comet(x,y)
```

При выполнении последней команды следите за тем, чтобы окно с графиком было поверх остальных окон. Обратите внимание, что при изменении размеров графического окна или при его минимизации и последующем восстановлении траектория движения пропадает. Это связано со способом, который применяет MATLAB для построения графика.

Скоростью движения кружка можно управлять, задавая различные шаги при создании вектора. Использование **comet** с одним аргументом (вектором) приводит к построению динамически рисующегося графика значений компонентов вектора в зависимости от их номеров.

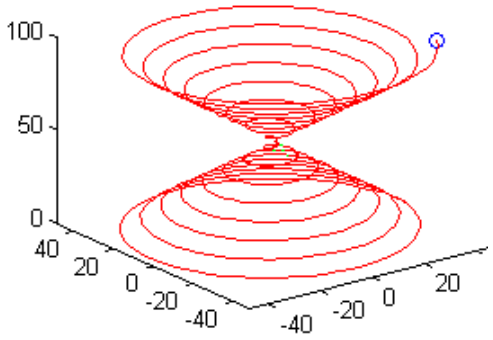
Для построения траектории точки, перемещающейся в пространстве, используется функция **comet3**. Для примера создадим следующий M - файл

```
t=[0:0.1:100];
x=cos(t).*abs(t-50);
y=sin(t).*abs(t-50);
z=t;
comet3(x,y,z)
```

и сохраним его под именем `mycomet.m`. Затем в командном окне выполним команду

```
mycomet
```

Последний кадр этой анимации показан на следующем рисунке



Графическая функция **ezplot3** имеет опцию '**animate**', которая приводит к движению небольшого кружка по кривой, построенной с ее использованием
ezplot3('sin(t)', 'cos(t)', 't',[0,6*pi],'animate')

Для построения анимации в общем случае MatLab предполагает использование функции **getframe**, которая захватывает графические кадры и может сохранять их в массив. Другая функция **movie** проигрывает такой массив в графическом окне, создавая, тем самым, анимацию. Вот простой пример

```
figure
t=0:0.01:2*pi;
set(gca,'nextplot','replacechildren');   % устанавливает режим замены кадров
                                         в графическом окне
for j = 1:20
    x=sin(j.*t).*exp(-t);
    plot(t,x);
    F(j)=getframe;        % запоминает текущий графический кадр
end
movie(F,10)          % Проигрывает массив F десять раз
```

Внутри цикла может стоять любая графическая функция. Созданные с ее помощью кадры будут сохраняться в массив F и проигрываться функцией **movie**.

Вот пример из справочной системы MatLab. Создайте следующий файл – сценарий с именем `mymovie.m`

```
Z= peaks;
surf(Z);
axis tight
set(gca,'nextplot','replacechildren');
% Record the movie
for j = 1:20
    surf(sin(2*pi*j/20)*Z,Z)
    F(j) = getframe;
end
movie(F,20)    % Play the movie twenty times
```

и выполните команду
mymovie

Некоторые другие графические возможности системы MatLab мы опишем в следующем параграфе. Об операторах циклов и функции **set** мы будем подробно говорить во второй части пособия.

1.4 Символьные вычисления

1.4.1 Использование символьных переменных

Перед использованием символьные переменные и функции в MatLab должны быть объявлены как символьные. Это отличает MatLab от таких программ как Mathematica или Maple, где переменные, если им не присвоено никаких значений или присвоены значения выражений, составленных из числовых констант, рассматриваются как символьные. Объявление символьных переменных выполняется командой **syms** . Например

```
syms x a b
```

Здесь мы создали три символьных переменных. Символьные выражения конструируются из символьных переменных

```
f=(sin(x)+a)^2*(cos(x)+b)^2/sqrt(abs(a+b))
f =
(sin(x)+a)^2*(cos(x)+b)^2/abs(a+b)^(1/2)
```

Выражение **f** автоматически становится символьным. Чтобы увидеть выражение в более привычном виде можно использовать функцию **pretty**

```
pretty(f)
                  2              2
          (sin(x) + a)  (cos(x) + b)
          --------------------------
                        1/2
                | a + b |
```

Символьную функцию можно создать без предварительного объявления переменных при помощи команды **sym**, входным аргументом которой является строка с выражением, заключенная в апострофы

```
z = sym( 'c^2 / (d + 1 ) ' )
```

Эта же функция **sym** может быть использована для объявления символьных переменных. Команда **syms a, b, c** эквивалентна последовательности

```
a=sym('a');b=sym('b');c=sym('c');
```

Функция **sym** может использоваться для работы с символьными константами

```
sym(2) / sym(5) + sym(1) / sym(3)
ans =
11/15
syms x; x+3.1
ans =
x+31/10
```

Символьные переменные могут являться элементами матриц и векторов. Элементы строк матриц при вводе отделяются пробелами или запятыми, а столбцов — точкой с запятой, так же как и для обычных матриц. В результате образуются символьные матрицы и векторы, к которым применимы матричные и поэлементные операции.

```
syms a b c d e f g h
A=[a b; c d]
A =
[ a, b]
[ c, d]
B=[e f; g h]
B =
[ e, f]
[ g, h]
C=A*B
```

```
C =
[ a*e+b*g, a*f+b*h]
[ c*e+d*g, c*f+d*h]
```
F=A.*B
```
F =
[ a*e, b*f]
[ c*g, d*h]
```
Конструирование блочных символьных матриц не отличается от числовых, требуется следить за размерами соответствующих блоков:
D=[C A; B C]
```
D =
[ a*e+b*g, a*f+b*h,       a,       b]
[ c*e+d*g, c*f+d*h,       c,       d]
[       e,       f, a*e+b*g, a*f+b*h]
[       g,       h, c*e+d*g, c*f+d*h]
```
Обращение к элементам символьных матриц и векторов производится при помощи индексации, в том числе двоеточием и вектором со значениями индексов. Например, выделим вторую строку предыдущей матрицы
d2=D(2,:)
```
d2 =
[ c*e+d*g, c*f+d*h, c, d]
```
Из полученного вектора выберем 1 – й, 3 – й и 4 – й элементы
d=d2([1 3 4])
```
d =
[ c*e+d*g,  c,  d]
```
Для удаления строк или столбцов символьных матриц используется пустой массив
D(1:2,:)=[ ]
```
D =
[   e,   f, a*e+b*g, a*f+b*h]
[   g,   h, c*e+d*g, c*f+d*h]
```

Для преобразования значения числовой переменной в символьную служит функция **sym**. Введите массив типа *double array* (обычная матрица MatLab)
A=[1.3 -2.1 4.9
6.9 3.7 8.5]
```
A =
  1.3    -2.1   4.9
  6.9     3.7   8.5
```
Здесь перенос ввода во вторую строку (в командном окне) выполняется нажатием комбинации клавиш Shift – Enter. Затем образуйте соответствующий символьный массив
B=sym(A)
```
B =
[  13/10, -21/10,  49/10]
[  69/10,  37/10,   17/2]
```
Создайте символьный вектор – столбец **d**
c=[3.2; 0.4; -2.1]
```
c =
    3.2
    0.4
   -2.1
```
d=sym(c)
```
d =
   16/5
    2/5
```

```
 -21/10
```
Умножьте матрицу **B** на вектор **d** – результат является символьной переменной, причем все вычисления проделаны над рациональными дробями.

e=B*d
```
e =
 -697/100
  571/100
```

Символьные операции позволяют находить значение выражений и символьных констант со сколь угодно большой точностью. В частности значение рациональных дробей, функций от целых и рациональных аргументов, от числа π и т.д. можно получить с любой степенью точности, т.е. найти сколь угодно много значащих цифр результата. Для этого предназначена функция **vpa:**

c=sym('sqrt(2)'); cn=vpa(c)
```
cn =
1.4142135623730950488016887242097
```
По умолчанию удерживается 32 значащие цифры. Второй дополнительный входной параметр **vpa** служит для задания точности:

cn=vpa(c,60)
```
cn =
  1.41421356237309504880168872420969807856967187537694807317668
```
pp=sym('pi'); vpa(pp,60)
```
ans =
  3.14159265358979323846264338327950288419716939937510582097494
```
Второй аргумент **vpa** задает удерживаемое число значащих цифр только для данного вызова **vpa**. Важно понимать, что выходной аргумент функции **vpa** является символьной переменной. Для перевода символьных переменных в числовые, т. е. в переменные типа *double array*, используется функция **double**

cnum=double(cn)
```
cnum =
  1.4142135623731
```
Переменная **cn** выглядела как число. Однако она являлась символьной переменной и ее можно использовать в арифметических операциях только с другими символьными переменными. Переменную **cnum** можно использовать в обычных выражениях MatLab.

При необходимости очистить рабочее пространство от всех символьных переменных следует использовать команду

maple restart

1.4.2 Графическое представление символьных функций

Построение графика символьной функции одной переменной осуществляется при помощи функции **ezplot**. Самый простой вариант использования **ezplot** состоит в указании символьной функции в качестве единственного входного аргумента. В этом случае в графическое окно выводится график функции на отрезке [-2л, 2л]

f=sym('x^2*sin(x)');
ezplot(f)

Обратите внимание, что автоматически создается соответствующий заголовок. По умолчанию в качестве отрезка, на котором строится график, принимается

промежуток пересечения области определения функции и интервала [- 2 π, 2 π]. Вторым аргументом может быть задан вектор с границами отрезка, на котором требуется построить график функции

ezplot(f,[-3,2])

Другие графические функции группы ez… также можно использовать для построения графиков символьных функций. Вот несколько примеров

syms s t; ezmesh(exp(-s)*cos(t),exp(-s)*sin(t),t,[0,8,0,4*pi])

syms u v; ezmesh(u,(abs(sqrt(1-u^2)+v)-abs(sqrt(1-u^2)-v))/2,u,[-1,1,-1,1],21)

syms x; ezplot(erf(x)); grid

syms t; ezplot3(sin(t), cos(t), t,[0,6*pi])

syms t; ezpolar(1+cos(t)) % график в полярных координатах

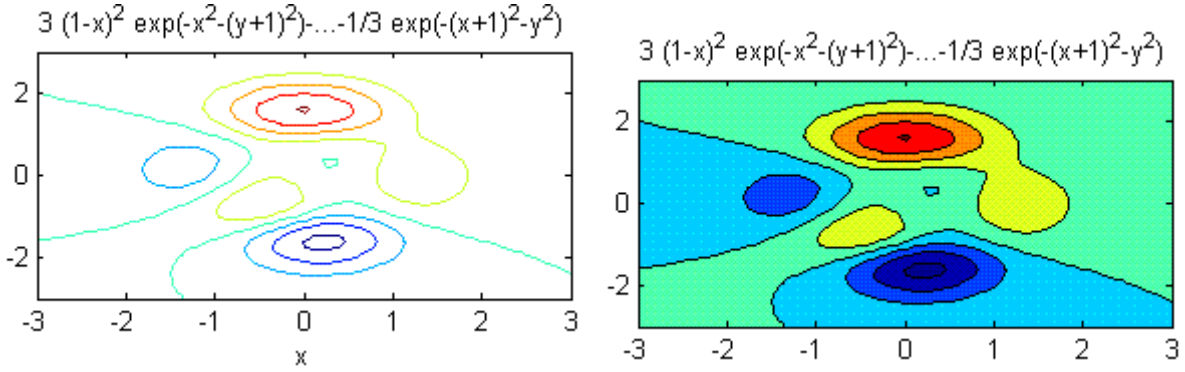
f=sym('x^2 +y^2'); ezsurf(f,[-1 1 -1 1])

Функция **ezsurf** отображает график символьной функции только для допустимых значений аргументов, остальные значения отбрасываются, что позволяет исследовать область определения функции двух переменных. Например

ezsurf('asin(x^y)',[0 6 -7 7])

Функции **ezcontour, ezcontourf** используются для построения контурных графиков – линий уровня функций двух переменных. Второй параметр задает прямоугольную область изменения аргументов, третий `n` – количество ячеек сети `n x n` на которую разбивается область изменения аргументов (по – умолчанию `n=60`)

```
syms x y
f = 3*(1-x)^2*exp(-(x^2)-(y+1)^2) ...
- 10*(x/5 - x^3 - y^5)*exp(-x^2-y^2) ...
- 1/3*exp(-(x+1)^2 - y^2);
ezcontour(f,[-3,3],49)
ezcontourf( f, [-3,3, -3, 3],60)
```



1.4.3 Алгебраические преобразования

Алгебраические преобразования с символьными переменными рассмотрим на примерах.

syms x; 27*x+12*x % упрощение символьного выражения

```
ans =
39*x
```

Операции с полиномами реализуют функции **collect**, **expand**, и **factor**. Функция **expand** раскрывает скобки

p=sym('(x+a)^4+(x-1)^3-(x-a)^2-a*x+x-3'); expand(p)

```
ans =
x^4+4*a*x^3+6*x^2*a^2+4*x*a^3+a^4+x^3-4*x^2+4*x-4+a*x-a^2
```

syms a b; expand((a+b)^5)

```
ans =
a^5+5*a^4*b+10*a^3*b^2+10*a^2*b^3+5*a*b^4+b^5
```

Аргумент **expand** может быть не только полином, но и символьным выражением, содержащим тригонометрические, экспоненциальную и логарифмическую функции

f=sym('sin(arccos(3*x))+exp(2*log(x))'); fe=expand(f)

```
fe =
(1-9*x^2)^(1/2)+x^2
```

Вычисление коэффициентов при степенях независимой переменной производится с использованием функции **collect**. Введите следующий полином

p=sym('(x+a)^4+(x-1)^3-(x-a)^2-a*x+x-3')

```
p =
(x+a)^4+(x-1)^3-(x-a)^2-a*x+x-3
```

pc=collect(p)

```
pc =
x^4+(1+4*a)*x^3+(-4+6*a^2)*x^2+(4+a+4*a^3)*x+a^4-4-a^2
```

По умолчанию в качестве переменной выбирается *x*, однако можно было считать, что *a* — независимая переменная, а *x* входит в коэффициенты полинома, зависящего от *a*. Второй аргумент функции **collect** предназначен для указания переменной, при степенях которой следует найти коэффициенты

pca=collect(p,'a')

```
pca =
a^4+4*x*a^3+(-1+6*x^2)*a^2+(4*x^3+x)*a+x^4+(x-1)^3-x^2-3+x
```

Символьные полиномы разлагаются на множители функцией **factor**, если получающиеся множители имеют рациональные коэффициенты

p=sym('x^5+13*x^4+215/4*x^3+275/4*x^2-27/2*x-18');
pf=factor(p)

```
pf =
1/4*(2*x+1)*(2*x-1)*(x+6)*(x+4)*(x+3)
```

syms a b; factor(a^2+2*a*b+b^2)

```
ans =
(a+b)^2
```

factor(a^2-4/9)

```
ans =
1/9*(3*a-2)*(3*a+2)
```

Функция **factor** умеет работать и с выражениями некоторых других типов, например, с тригонометрическими выражениями

syms x
factor(sin(x)^2-cos(x)^2)

```
ans =
(sin(x)-cos(x))*(sin(x)+cos(x))
```

Преобразование выражений общего вида производится при помощи функции **simplify**, которая реализует алгоритм упрощения выражений, содержащих тригонометрические, экспоненциальную, логарифмическую функции, и некоторые специальные функции. Кроме того, **simplify** способна преобразовывать выражения, содержащие суммирование и интегрирование.

```
rho = sym('(1 + sqrt(5))/2'); f = rho^2 - rho - 1
f =
(1/2+1/2*5^(1/2))^2-3/2-1/2*5^(1/2)
simplify(f)
ans=
0
simplify(sin(x)^2+cos(x)^2)
ans =
1
simplify((x^2+a*x+b*x+a*b)/(a+x))
ans =
x+b
```

Для вычисления значения выражения в точке следует использовать функцию подстановки в виде **R = subs(S,old,new)**

```
syms x; f = 2*x^2 - 3*x + 1;subs(f,2)
ans =
3
```

Здесь MatLab сам определяет имя символьной переменной в выражении **f** и заменяет ее заданным значением 2. Если символьных переменных несколько, то следует указать имя переменной, которая должна быть заменена значением

```
syms a x; f=a*x^2-x; subs(f,x,2)
ans =
4*a-2
```

Одновременная замена нескольких символьных переменных может быть выполнена следующим образом

```
syms a b; subs(cos(a)+sin(b),{a,b},{sym('c'),2})
ans =
cos(c)+sin(2)
```

В символьное выражение вместо символьной переменной можно подставить вектор или матрицу. Результатом будет объект той же размерности (вектор или матрица). Вот примеры табулирования функции

```
y=subs('x^2','x',[0 1 2 3])
y=
    0   1   4   9
syms x y; subs(x^2+y^2,x,[0 1 2])
ans =
[   y^2, 1+y^2, 4+y^2]
syms x y ; subs(x^2+y^2,{x,y},{[0 1 2],[3 4 1]})
ans =
     9    17     5
subs(x*y,{x,y},{[0 1;-1 0],[1 -1;-2 1]})
ans =
     0    -1
     2     0
```

Для вычисления композиции двух функций используется функция **compose**

```
syms x y z t u; f = 1/(1 + x^2); g = sin(y); compose(f,g)
ans =
1/(1+sin(y)^2)
compose(f,g,z)   % возвращает f(g(z))
```

```
ans =
1/(1+sin(z)^2)
```

Если задана функция *f(x)*, то обратная функция $g = f^{-1}$ удовлетворяет равенству *g(f(x)) = x*. Символьное представление обратной функции может быть получено с помощью функции **finverse**

syms u v ; finverse(exp(u-2*v),u)

```
ans =
2*v+log(u)
```

Для вычисления целой и дробной части символьного числа можно использовать следующие функции: **fix** – округление до целого в сторону нуля; **floor** – вычисление наибольшего целого, не превосходящего данное; **round** – округление до ближайшего целого; **ceil** – вычисление ближайшего большего целого; **frac** – вычисление дробной части числа **frac(X) = X - fix(X)**

x = sym(-5/2); [fix(x) floor(x) round(x) ceil(x) frac(x)]

```
ans =
[   -2,   -3,   -3,   -2, -1/2]
```

MatLab хорошо умеет работать с разрывными функциями

F = 2/3*atan(1/3*tan(1/2*x))
ezplot(F)
J = sym(2*pi/3)*sym('round(x/(2*pi))');
ezplot(J,[-6 6]) % ступенчатая функция
F1 = F+J
ezplot(F1) % плавно возрастающая

1.4.4 Элементы математического анализа

MatLab умеет выполнять основные операции математического анализа, в частности, дифференцирование и интегрирование символьных выражений

diff(x^4,x)

```
ans =
4*x^3
```

diff(x^4,x,3) % вычислить третью производную

```
ans =
24*x
```

y1=sin(x)/x; limit(y1,x,0) % $\lim_{x\to 0}\frac{\sin x}{x}$

```
ans=1
```

y2=(1-exp(-x))/x; limit(y2,x,inf) % $\lim_{x\to \infty}\frac{1-e^{-1}}{x}$

```
ans=0
```

Для дифференцирования постоянной ее следует объявить символьной

c = sym('5'); diff(c)

```
ans =
0
```

Однако, простое дифференцирование числа дает

diff(5)

```
ans =
     [ ]
```

int((x+a)^3,x) % вычислить неопределенный интеграл

```
ans =
```

```
1/4*(x+a)^4
```
int(-2*x/(1+x^2)^2)
```
ans =
1/(1+x^2)
```
syms x z; int(x/(1+z^2),z) % интегрирование по переменной z
```
ans =
x*atan(z)
```
sym s t; int(2*x, sin(t), 1) % здесь `sin(t)` нижний предел интегрирования
```
ans =
1-sin(t)^2
```
При вычислении интегралов может получиться следующая ситуация
syms a x; f = exp(-a*x^2); int(f,x,-inf,inf)
```
ans =
PIECEWISE([1/a^(1/2)*pi^(1/2), csgn(a) = 1],[Inf, otherwise])
```
Не зная ничего о символьной переменной **a**, MatLab вернул два результата – первый получается, когда интеграл сходится, второй – когда расходится. Но если объявить переменную **a** положительной, то сразу получаем результат
syms a positive; f = exp(-a*x^2); int(f,x,-inf,inf)
```
ans =
1/a^(1/2)*pi^(1/2)
```
Функция **diff** умеет дифференцировать выражения любой степени сложности. Однако она не всегда возвращает результат в самой простой форме. Тогда результат следует упростить
diff(sin(x)^2-cos(x)^2,x)
```
ans =
4*sin(x)*cos(x)
```
simplify(diff(sin(x)^2-cos(x)^2,x))
```
ans =
4*sin(x)*cos(x)
```
Функция **simplify** не справилась со своей задачей упрощения. Однако функция **simple**, которая пытается найти представление выражения с меньшим числом символов, чем исходное, получила более простой результат
simple(diff(sin(x)^2-cos(x)^2,x))
```
. . .
ans =
2*sin(2*x)
```
Функция **simplify** срабатывает, когда в ее аргументе стоит неопределенный интеграл
simplify(x^3-int(x^2,x))
```
ans =
2/3*x^3
```
Символьное ядро MatLab хорошо умеет работать с конечными и бесконечными суммами и рядами. Вот примеры суммирования рядов
syms x k ; s1 = symsum(1/k^2,1,inf)
```
s1 =
1/6*pi^2
```
s2 = symsum(x^k,k,0,inf)
```
s2 =
-1/(x-1)
```
symsum(k^2,0,10)
```
ans =
385
```
symsum(x^k/sym('k!'), k, 0,inf)
```
ans =
exp(x)
```

Разложение функций в ряд Тейлора также возможно

syms x; f = 1/(5+4*cos(x)); T = taylor(f,8)

```
T =
1/9+2/81*x^2+5/1458*x^4+49/131220*x^6
```

С некоторыми символьными функциями работать не очень удобно. Например, чтобы вычислить факториал (символьный, следовательно с любым количеством значащих цифр) надо выполнить пару команд

kfac = sym('k!');
subs(kfac,k,50)

```
ans =
30414093201713378043612608166064768844377641568960512000000000000
```

В этом случае можно напрямую обратиться к ядру Maple

maple('50!')

```
ans =
30414093201713378043612608166064768844377641568960512000000000000
```

Для использования стандартных функций Maple их следует передавать функции **maple** как текстовую строку

maple('gcd(14, 21)') % наибольший общий делитель

```
ans =
7
```

Обычно функция **maple** возвращает результат в символьном виде. Однако, если в результате выполнения возникает ошибка, то функция возвращает вектор из двух элементов: [результат сообщение], где сообщение – это строка с информацией о возникшей ошибке.

Команду **maple** допускается вводить по – разному. Следующие три команды эквивалентны

maple('evalf(Pi,50)')
maple evalf Pi 50
maple('evalf','Pi',50)

Отметим еще одну функцию, которая переводит символьное выражение в код LaTex

syms x; f = taylor(log(1+x)); latex(f)

```
ans =
x-1/2\,{x}^{2}+1/3\,{x}^{3}-1/4\,{x}^{4}+1/5\,{x}^{5}
```

Возможности символьных вычислений в MatLab весьма обширны. Более подробно с ними вы можете познакомиться по справочной системе. Получить справку по функциям Maple, встроенным в MatLab, или какой – либо теме можно командой

mhelp имя_функции
mhelp тема
mhelp('тема')

Список функций Maple возвращает команда

help mfunlist

1.5 Алгебраические задачи

1.5.1. Полиномы

Полином в MATLAB задается вектором его коэффициентов, начиная с коэффициента при старшей степени. Например, для определения полинома $p = x^7 + 3.2x^5 - 5.2x^4 + 0.5x^2 + x - 3$ следует использовать команду

p=[1 0 3.2 -5.2 0 0.5 1 - 3];

Промежуточные нулевые коэффициенты должны содержаться в векторе.

Функция **polyval** предназначена для вычисления значения полинома от некоторого аргумента

polyval(p,1)

```
ans =
   -2.5000
```

Аргумент может быть матрицей или вектором, в этом случае производится поэлементное вычисление значений полинома и результат представляет матрицу или вектор того же размера, что и аргумент

polyval(p,[1 2;3 4])

```
ans =
  1.0e+003 *
   -0.0025    0.0726
    0.8473    4.5824
```

Функция **roots** возвращает вектор корней полинома, в том числе и комплексных. Аргументом функции **roots** является вектор коэффициентов полинома

roots(p)

```
ans =
  -0.5962 + 2.0565i
  -0.5962 - 2.0565i
   1.2450
   0.3103 + 0.6513i
   0.3103 - 0.6513i
  -0.6732
```

Убедимся в правильности работы функции roots, вычислив значение полинома в одном из корней

polyval(p,r(3))

```
ans =
 -1.3323e-015
```

Умножение двух полиномов осуществляется при помощи функции **conv.** Например, для вычисления произведения полиномов $p(x) = x^5 + x^3 + 1$ и $q(x) = x^2 + 2x + 3$ следует создать два вектора коэффициентов и использовать их в качестве аргументов **conv:**

p= [1 0 1 0 0 1] ;
q = [1 2 3] ;
s=conv(p,q)

```
s =
     1     2     4     2     3     1     2     3
```

Функция **deconv** осуществляет деление полиномов с остатком. Она вызывается с двумя выходными аргументами — частным и остатком от деления:

[d r]=deconv(p,q)

```
d =
     1    -2     2     2
r =
     0     0     0     0   -10    -5
```

Размер вектора, содержащего коэффициенты остатка, равен максимальному из размеров векторов, соответствующих делимому полиному и его делителю.

Для сложения и вычитания полиномов в MATLAB нет специальной функции. В то же время использование знака "+" для нахождения суммы полиномов разной степени приведет к ошибке, т. к. нельзя складывать векторы разных размеров. Вы можете самостоятельно написать функцию сложения полиномов. Она должна приводить полиномы к общей степени, и затем вектора одинакового размера, представляющие полиномы, можно сложить.

Функция **polyder** предназначена для вычисления производной не только от полинома, но и от произведения и частного двух полиномов. Вызов **polyder** с одним аргументом — вектором, соответствующим полиному, приводит к вычислению вектора коэффициентов производной полинома:

polyder(p)

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

Для вычисления производной от произведения полиномов следует использовать **polyder** с двумя входными аргументами:

polyder(p,q)

```
ans =
     7    12    20     8     9     2     2
```

Для нахождения производной отношения двух полиномов, следует вызвать **polyder** с двумя выходными аргументами **[q, d] = polyder(b,a).** Производная частного возвращается в виде двух полиномов `q` – числителя и `d` - знаменателя отношения **q / d**

b= [1 0 1 0 0 1] ;
a = [1 2 3] ;
[q,d]=polyder(b,a)

```
q =
     3     8    16     4     9    -2    -2
d =
     1     4    10    12     9
```

Для задания полинома в символьном виде используется команда **poly2sym**. Она принимает вектор коэффициентов и возможно имя символьной переменной

syms x
r = poly2sym([1 3 2],x)

```
ans =
x^2+3*x+2
```

и возвращает символьное представление полинома.

Функция **poly** возвращает вектор (представление полинома в виде вектора) по его корням

pp=poly([1 2])

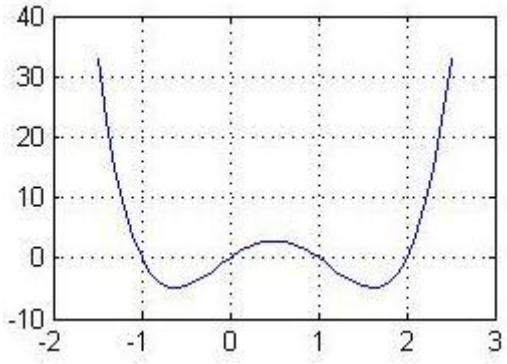
```
pp =
     1    -3     2
```

syms x; poly2sym(pp,x)

```
ans =
x^2-3*x+2
```

Чтобы нарисовать график полинома надо задать вектор значений независимой переменной, вычислить его значения в этих точках и обратиться к команде **plot**

```
p=poly([-1 0 1 2])
x=-2:0.1:3;
y=polyval(p,x);
plot(x,y)
```



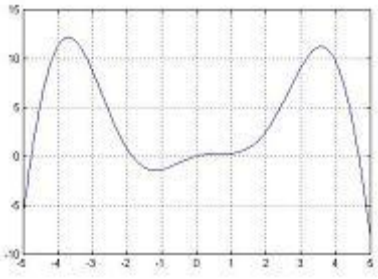
1.5.2. Нахождение корней и решение алгебраических уравнений

Функция **fzero** позволяет приближенно вычислить корень уравнения на некотором интервале или ближайший к заданному начальному приближению. В простейшем варианте **fzero** вызывается с двумя входными и одним выходным аргументом **x=fzero('funcname',x_0)**, где `funcname` имя файл – функции (левая часть уравнения *f(x)=0*), x_0 – начальное приближение корня. Для примера решим уравнение $\sin x - x^2 \cos x = 0$. Создадим m - функцию

```
function y = myf(x)
y = sin(x)-x.^2.*cos(x);
```

Посмотрим на ее график

```
fplot('myf',[-5 5])
```



Видим, что на отрезке [- 5 5] имеется 4 корня. Чтобы уточнить значение корня, например, вблизи 5 выполним команду

```
fzero('myf',5)
ans =
    4.6665
```

Алгоритм функции **fzero** по умолчанию находит корень уравнения с точностью **eps**

```
eps
ans =
  2.2204e-016
```

т.е. с точностью до двойки в шестнадцатом знаке после десятичной точки. Чтобы убедиться в этом, измените формат представления данных на **long e** и выведите значение корня

x=fzero('myf',5);
format long e
x
```
x =
    4.666499563444645e+000
```
Вместо начального приближения вторым параметром **fzero** можно задать интервал, на котором следует найти корень
x2=fzero('myf',[-3 -1])
```
x2 =
   -1.853927459696150e+000
```
На границах указываемого интервала функция должна принимать значения разных знаков, иначе появится сообщение об ошибке!

Функцию, корень которой определяется, можно задать при помощи указателя на нее
format long
x1 = fzero(@myf, [-5 -4])
```
x1 =
  -4.75655940570290
```
Вместо создания m – функции можно создать анонимную функцию
fun=@(x) sin(x)-x.^2.*cos(x)
x2 = fzero(fun, -2)
```
x2 =
  -1.85392745969615
```

На сеанс можно создавать временные функции одной переменной
ff=inline('x^2/2-x^4/4')
```
ff =
     Inline function:
     ff(x) = x^2/2-x^4/4
```
fplot(ff,[-2 2])
и определять корень с помощью функции **fzero**
fzero(ff,1)
```
ans =
    1.4142
```
Важной особенностью **fzero** является то, что она вычисляет только те корни, в которых функция меняет знак, а не касается оси абсцисс. Найти корень уравнения $x^2=0$ при помощи **fzero** не удается
fzero(@(x) x^2,0.001)
```
Exiting fzero: aborting search for an interval ...
```

Функция **fsolve** решает нелинейные алгебраические уравнения и их системы, заданные в виде $F(x)=0$, где F вектор – функция, а x – неизвестный вектор. Функция **fsolve(fun,x_0)** принимает имя функции **fun** и начальное значение искомого вектора x_0. Для примера сравните корни функции **fun=@(x) sin(x)-x.^2.*cos(x)**, полученные с помощью **fzero** и **fsolve.**
x1=fzero(fun,5)
x2=fsolve(fun,5)
```
x1 =
    4.666499563444645e+000
Optimization terminated: first-order optimality is less than options. TolFun.
x2 =
    4.666499563444647e+000
```

Для решения системы следует создать функцию, возвращаемым значением которой является вектор, т.е. вектор – функцию. Решим, например, систему

$$\begin{cases} x(2-y)-e^{y}\cdot\cos x=0 \\ x-y-\cos x-e^{y}=-2 \end{cases}$$

Создаем функцию, которая возвращает вектор из двух значений

```
function F=FN(x)
% x  вектор аргументов, F - вектор значения
F(1)=x(1)*(2-x(2))-cos(x(1))*exp(x(2));
F(2)=2+x(1)-x(2)-cos(x(1))-exp(x(2));
```

Решаем систему командой

[x,f]=fsolve(@FN, [0 0])

```
Optimization terminated: first-order optimality is less than options. TolFun.
x =
    0.7391    0.4429
f =
  1.0e-011 *
   -0.4702   -0.6404
```

Здесь **[0 0]** начальное приближение. Найденное решение возвращается в векторе *x*, а значение – в векторе *f*. Посмотреть параметры настроек (точность и т.д.) функции **fsolve** можно командой

optimset fsolve

```
ans =
        Display: 'final'
    MaxFunEvals: '100*numberofvariables'
        MaxIter: 400
         TolFun: 1.000000000000000e-006
           TolX: 1.000000000000000e-006
       ...
```

Здесь мы не приводим всех опций, поскольку их более 50. Это могут быть указания на метод решения, точность, количество итераций и т.д. За подробным описанием смысловой нагрузки опций обращайтесь к справочной системе.

Для изменения одной или нескольких опций мы также используем функцию **optimset** в формате **opt** = **optimset('OptionName','OptionValue',…).** Возвращенное функцией множество параметров **opt** нужно передать третьим аргументом функции **fsolve.**

Найдем матрицу, удовлетворяющую уравнению

$$X\cdot X\cdot X=\begin{bmatrix} 37 & 54 \\ 81 & 118 \end{bmatrix}$$

Создаем m – функцию

```
function F = FNM(x)
F = x*x*x-[37,54;81,118];
```

Следующая последовательность команд решает нашу задачу

x0 = ones(2,2) % Начальное приближение

opt = optimset('Display','off'); % Отключить вывод сообщения

[x,F] = fsolve(@FNM,x0,opt)

```
x0 =
     1     1
     1     1
x =
    9.999999999041618e-001    2.000000000043856e+000
    3.000000000065749e+000    3.999999999969842e+000
```

```
F =
   -4.013145371573046e-011    1.733724275254645e-011
    2.506794771761634e-011   -1.691091711109038e-011
```

Полученный вектор **x** близок к точному значению $\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$ (установлен формат вывода **long e**). Если вы установите формат вывода **short**, то увидите точное решение

format short

x

```
x =
    1.0000    2.0000
    3.0000    4.0000
```

Можно решать системы линейных уравнений с использованием функции **fsolve**. Решим для примера систему

$$\begin{cases} x_1 + x_2 + x_3 = 6 \\ 2x_1 - x_2 - x_3 = -3 \\ 3x_1 - 2x_2 + x_3 = 2 \end{cases}$$

Создадим m – функцию

```
function F = FNL(x)
% x  вектор аргументов, F - вектор значения
F(1)=x(1)+x(2)+x(3)-6;
F(2)=2*x(1)-x(2)-x(3)+3;
F(3)=3*x(1)-2*x(2)+x(3)-2;
```

Тогда последовательность команд напечатает нам решение задачи

x0=[0 0 0]

opt = optimset('Display','off'); % Отключить вывод сообщения

x = fsolve(@FNL,x0,opt)

```
x =
    1.0000    2.0000    3.0000
```

Однако для решения систем линейных уравнений лучше использовать другие возможности MatLab, о которых мы поговорим в следующем параграфе.

Для нахождения корней системы нелинейных уравнений можно использовать команду **solve** из пакета символьной математики

syms a b c x

S = a*x^2 + b*x + c;

solve(S)

```
ans =
 1/2/a*(-b+(b^2-4*a*c)^(1/2))
 1/2/a*(-b-(b^2-4*a*c)^(1/2))
```

t = solve('x^2-3*x+2 = 0') % находит оба корня

```
t =
 2
 1
```

solve(x^2-5*a*x+4*a^2,x) % находит оба корня

```
ans =
   a
 4*a
```

syms a x

s = solve(x^3+a*x+1)

```
s =. . .
```

Здесь мы не выписали длинную формулу решения кубического уравнения.

Функция **solve** находит все корни (если это возможно) и возвращает решение в символьном виде. В частности это значит, что можно получить любую, сколь угодно большую точность решения. Однако следует помнить, что не для всех уравнений можно построить решение в символьном виде.

Функция **solve** возвращает решение в виде структуры

```
R=solve(2*x+3*y-13,3*x-2*y,x,y)
R =
    x: [1x1 sym]
    y: [1x1 sym]
```

Чтобы увидеть значения надо выполнить команды

```
R.x
ans =
2
R.y
ans =
3
```

Вот пример решения нелинейной системы

```
R=solve(x^2+y^2-16,x^2-y-4,x,y);
R.x'
ans =
[  0,  0, 7^(1/2), -7^(1/2)]
R.y'
ans =
[ -4, -4,  3,  3]
```

Такой способ возврата результата удобен, если у системы несколько решений

```
S = solve('u^2-v^2 = a^2','u + v = 1','a^2-2*a = 3')
S =
    a: [2x1 sym]
    u: [2x1 sym]
    v: [2x1 sym]
```

Чтобы их увидеть можно выполнить следующие команды

```
S.a'            % напечатать оба значения a
ans =
[  3, -1]
s2 = [S.a(2), S.u(2), S.v(2)]  % напечатать второе решение
s2 =
[ -1,  1,  0]
```

Вот, что получается, если решения нет

```
R=solve(x^2-3,x^2-4,x,y)
R =
[ empty sym ]
```

В следующем примере мы подсказываем MatLab, что переменная **a** вещественное число

```
syms a real; syms x y
[x,y] = solve(x^2*y^2, x-y/2-a);
[x';y']
ans =
[    0,    0,    a,    a]
[ -2*a, -2*a,    0,    0]
```

Уравнения для функции **solve** можно передавать в виде строки

```
eq = 'x^2*y^2=1, x-y/2-a';
[x,y] = solve(eq)
```

С помощью функции **solve** можно решать линейные системы

```
clear u v x y; syms u v x y
S = solve(x+2*y-u, 4*x+5*y-v);
sol = [S.x;S.y]
sol =
 -5/3*u+2/3*v
  4/3*u-1/3*v
```

Функция **solve** сама определяет имена переменных, относительно которых нужно решать уравнение или систему. Но если символьных переменных много, то вторым аргументом можно явно указать имена, относительно которых следует решать задачу.

```
solve('a*x^2 + b*x + c','b')
ans =
-(a*x^2+c)/x
solve('a*x^2 + b*x + c','x')
ans =
 1/2/a*(-b+(b^2-4*a*c)^(1/2))
 1/2/a*(-b-(b^2-4*a*c)^(1/2))
```

1.5.3 Решение систем линейных уравнений

Если надо решить большую систему линейных уравнений, лучше всего использовать встроенные возможности MatLab для решения линейных систем. Функция **X=A \ B** находит решение системы уравнений вида **A·X = B**, где **A** – прямоугольная матрица размера `m x n` и **B** – матрица размера `n x k`. В частности, если **b** столбец длины `n`, то выражение **A \ b** возвращает решение системы уравнений вида **A·X = b**. Решим для примера систему вида

$$\begin{bmatrix} 3 & 2 & 1 \\ 0 & 4 & -1 \\ -2 & -5 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 10 \\ 5 \\ -9 \end{bmatrix}$$

```
A=[3 2 1; 0 4 -1; -2 -5 1];
b=[10; 5; -9]
x=A \ b
x =
    1.0000
    2.0000
    3.0000
```

Операцию **** называют операцией левого деления (умножения слева на обратную матрицу **inv(A)**). Она эквивалентна вызову функции **A \ B = mldivide(A,B)**. Рассмотрим еще пример

```
A=[1, -2, 3, -1;2, 3, -4, 4; 3, 1, -2, -2;1, -3, 7, 6]
A =
     1    -2     3    -1
     2     3    -4     4
     3     1    -2    -2
     1    -3     7     6
b=[6;-7;9;-7];
y=A \ b
y =
    2.0000
   -1.0000
         0
   -2.0000
```

Проверим, что вектор **y** удовлетворяет системе

A*y

```
ans =
    6.0000
   -7.0000
    9.0000
   -7.0000
```

Решить систему можно командой **inv(A)*B**, где функция **inv** вычисляет обратную матрицу.

inv(A)

```
ans =
    0.2857    0.1429    0.1429    0.0000
   -5.0286   -2.3143    2.6857    1.6000
   -2.8571   -1.4286    1.5714    1.0000
    0.7714    0.4857   -0.5143   -0.2000
```

Проверим

A*inv(A) % получаем единичную матрицу

Решение системы, используя обратную матрицу, можно получить следующим образом

z=inv(A)*b; z'

```
ans =
    2.0000   -1.0000    0.0000   -2.0000
```

Если результат обращения матрицы не является надежным (определитель матрицы коэффициентов равен или близок к нулю), то выдается сообщение. Например,

A=[1 2 3; 4 5 6; 7 8 9];
b=[1 2 3]';
y=A \ b

```
Warning: Matrix is close to singular or badly scaled.
         Results may be inaccurate. RCOND = 1.541976e-018.
y =
   -0.3333
    0.6667
         0
```

Решение с помощью оператора **** эффективнее, чем использование обратной матрицы. Это объясняется тем, что алгоритм решения систем линейных уравнений при помощи оператора **** определяется структурой матрицы коэффициентов системы. В частности, MATLAB исследует, является ли матрица треугольной, или может быть приведена перестановками строк и столбцов к треугольному виду, симметричная матрица или нет, квадратная или прямоугольная (MATLAB умеет решать системы с прямоугольными матрицами — переопределенные или недоопределенные). Поэтому решать системы при помощи **** разумно, когда выбор алгоритма решения поручается MATLAB. Если же имеется информация о свойствах матрицы системы, то следует использовать специальные методы.

В MatLab есть еще одна операция **/** - это операция правого деления матриц. Выражение **B / A** эквивалентно **B*inv(A).** Значит эта операция позволяет решать системы линейных уравнений вида **Y·A = B**. Вместо операции **B / A** можно вызывать эквивалентную ей функцию **mrdivide(B,A)**. Поясним различие между операциями **** и **/** следующей таблицей

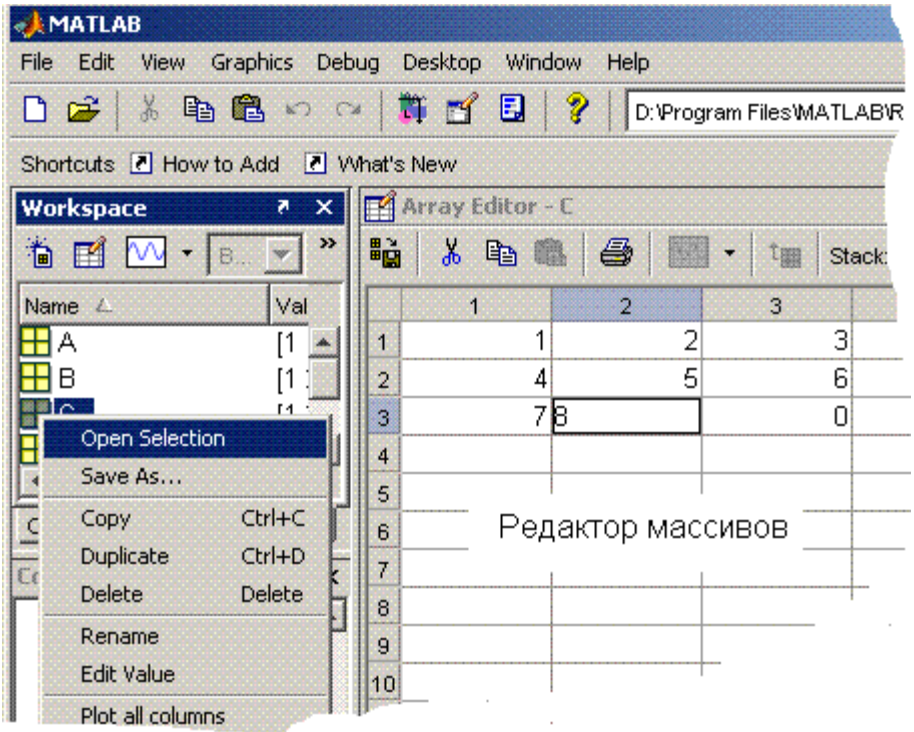
A=[1 1; 0 1]
```
A =
     1     1
     0     1
```
inv(A)
```
ans =
     1    -1
     0     1
```
inv(A)*B
```
ans =
    -2    -2
     3     4
```
A \ B
```
ans =
    -2    -2
     3     4
```
B=[1 2; 3 4]
```
B =
     1     2
     3     4
```
B*inv(A)
```
ans =
     1     1
     3     1
```
B / A
```
ans =
     1     1
     3     1
```

Правило для запоминания: верх значка операции матричного деления **** или **/** всегда наклонен в сторону обращаемой матрицы. При этом размерности массивов при использовании операций **** и **/** должны быть согласованы.

Решение систем небольшой размерности можно выполнить, введя матрицу системы и вектор правой части непосредственно из командной строки. Однако часто требуется найти решение системы, состоящей из большого числа линейных уравнений. Для ввода данных можно воспользоваться редактором массивов. Выполните команду

C=[ ]

Пустой массив появится в окне рабочего пространства (Workspace). Щелкните в этом окне правой кнопкой мыши по имени массива **C** и выберите пункт Open Selection. Можно также выполнить двойной щелчок по имени переменной. Откроется редактор массивов, показанный на следующем рисунке



В этом окне вы можете вводить новый или редактировать уже существующий массив.

Можно также загрузить массив из файла командой **load** так, как мы это делали в п. 1.1.3.

Отметим, что имеются и поэлементные операции левого и правого деления. Операция **A .\ B** эквивалента поэлементному делению **B(i,j) / A(i,j)**, а операция **A ./ B** эквивалентна **A(i,j) / B(i,j)**.

```
A=[1 1; 2 4]   B=[1 2; 3 4]   A .\ B          A ./ B
A =            B =            ans =          ans =
   1   1          1   2          1.0  2.0       1.0   0.5
   2   4          3   4          1.5  1.0     0.6667 1.0
```

Операции **** и **/** могут решать недоопределенные системы. Например, решим систему, матрица коэффициентов которой имеет нулевой определитель

```
A=[1 2 3; 4 5 6; 7 8 9]
det(A)
ans =
     0
b=[6; 15; 24]
x1=A \ b
Warning: Matrix is close to singular or badly scaled.
         Results may be inaccurate. RCOND = 1.541976e-018.
x1 =
     0
     3
     0
```

Проверим, что полученный вектор `x1` является решением системы

```
A*x1-b     % получаем нуль – вектор
```

Но легко проверить, что другой вектор – столбец, например, **[1 1 1]'** также является решением системы.

```
x2=[1 1 1]'
A*x2-b     % получаем нуль – вектор
```

В данном случае решений бесконечно много и MatLab возвращает тот вектор решений, который содержит больше нулевых компонент.

Однако сообщение и возвращаемый вектор могут быть другими. Например

```
A=[1 2 3; 4 5 6; 6 6 6]
det(A)
ans =
     0
b=[6 15 18]';
A \ b
Warning: Matrix is singular to working precision.
ans =
   NaN
   NaN
   NaN
```

Функции **mldivide(A,B)** и **mrdivide(A,B)** (операции **** и **/**) довольно сложны. Для квадратных матриц они находят точное решение, для переопределенных систем находят решение с использованием метода наименьших квадратов, а для недоопределенных находят решение с минимальным числом ненулевых компонент. За более подробной информацией обращайтесь к справочной системе.

Можно решать линейные системы в символьном виде, используя те же операции левого и правого деления

```
syms u v
A = [1 2; 4 5];
b = [u; v];
z = A \ b
z = -5/3*u+2/3*v
     4/3*u-1/3*v
```

1.6 Справочная система и среда MatLab

1.6.1 Справочная система

Простой способ получить справку по какой – либо функции это выполнить команду

help *имя функции*

Например

help magic

Информация о функциях по команде **help** выводится в командное окно. Все имена функций строчные, несмотря на то, что MatLab выводит их заглавными буквами. Можно получить справку по целой теме, если вы знаете ее имя

help matfun

Следующая команда выводит список тем

help

Команда **helpwin** или выбор пункта меню `Help – MatLab Help` открывает справочное окно MatLab с именами тем, а команда **helpwin** *topic* открывает окно справки по соответствующей теме. Например

helpwin matfun

Команда **lookfor** *keyword* позволяет искать функции по ключевому слову. Например

lookfor inverse

Аналогом команды **helpwin** является команда **doc.** Однако справка по одной и той же теме или функции отображается по – разному. Например, команды

doc ezplot
helpwin ezplot

откроют разные страницы справки об одной и той же функции.

1.6.2 Среда MatLab

Среда MatLab включает в себя совокупность переменных, созданных во время сеанса, и набор файлов, содержащих программы и данные, существующие между сеансами.

Рабочее пространство – это область памяти, доступная из командной строки. Команды

who % выводит список переменных, доступных в текущем сеансе
whos % выводит более подробную информацию о переменных,
доступных в текущем сеансе

показывают текущее содержание рабочего пространства. Окно `Workspace` содержит ту же информацию, которая выводится по команде **whos**. Но при этом двойной щелчок по имени переменной открывает окно, в котором можно редактировать содержимое переменной (матрицы) прямо в памяти. Команда

save

сохраняет содержание рабочего пространства в MAT – файле, который может быть прочитан командой **load** в последующих сеансах работы. Например,

save 12_03_2010

сохранит содержание рабочего пространства в файл `12_03_2010.mat`. Можно сохранить только определенные переменные, указывая их имена после имени файла.

MatLab использует маршрут поиска файлов. Когда мы вызываем функцию, то выполняется первый М – файл на этом пути. Команда

path

показывает маршруты поиска файлов. Меню `Set Path…` из меню `File` показывает и позволяет добавлять новые каталоги в маршрут поиска. Команда

clear ValName или **clear('ValName')**

очищает содержимое переменной **ValName.** Команда **clear** без параметров очищает все переменные текущего сеанса. Команда **pack** сохраняет все переменные на диске, очищает память и снова загружает переменные.

Комбинация клавиш `Ctrl – Break` прерывает процесс вычислений.

Ниже мы приводим список основных специальных символов.

; (точка с запятой)	– разделяет строки матрицы; – завершает команды, если вывод результатов в командное окно не нужен;
[ ] (квадратные скобки)	– используются при задании матриц и векторов;
, (запятая)	– разделяет элементы строк матриц; – разделяет операторы в строке ввода;
: (двоеточие)	– используется для указание диапазона (интервала изменения величины);
( ) (круглые скобки)	– задают порядок выполнения матема – тических операций; – указывают аргументы функций; – указывают индексы массивов;
… (три точки и более)	– являются символом продолжения выражения на следующую строку;
. (точка)	– отделяет дробную часть от целой; – применяется в составе комбинирован – ных знаков поэлементных операций (./ .\ .^ .*)
% (процент)	– начало комментария
' (апостроф)	– начинает и завершает символьные строки (для включения самого апострофа в строку надо поставить два апострофа подряд).

Команда **type** `'FileName'` печатает в командном окне текст m – файла. Команда

clc

очищает командное окно, оставляя знак приглашения (переменные в памяти остаются). Команда **home** делает то же самое.

2. Особенности использования MatLab.

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`.

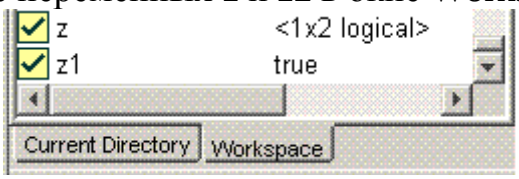
z1=1>0

```
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
```
ans =
     0     0
     1     0
```
A<=B
```
ans =
     1     0
     1     0
```
A==B
```
ans =
     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
ьэюяёђѓєsiї
jљњћќўџ
```

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
```

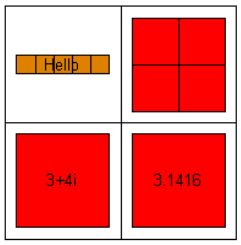
B3=A{3}

```
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'}

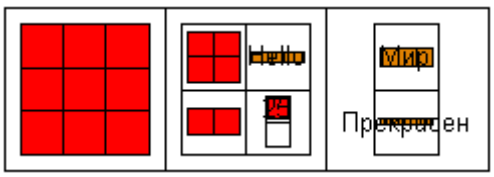
```
A =
    [3x3 double]    {2x2 cell}    'Мир'
```

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 $\int_a^b f(x)\,dx$.

```
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;
```

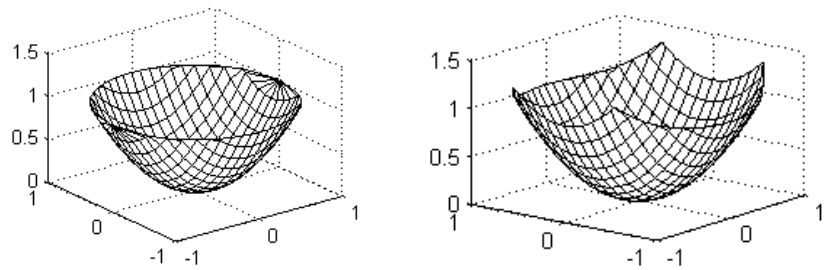
Отметим, что здесь мы также создали подфункцию PR(x,a,w) и то, что функция plotAboveDomain не возвращает никаких значений.

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')
```

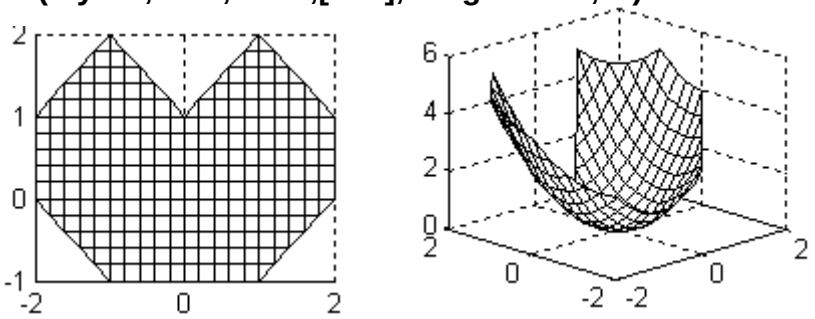
График поверхности $z = x^2 + y^2$ над единичным кругом показан на следующем рисунке слева. На рисунке справа та же поверхность построена над областью $-1 \le x \le 1$, $-\cos x \le y \le \cos x$.

```
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

$$y = \begin{cases} \sqrt{1-x^2} & -1 \le x \le 1 \\ 0, & \text{èíà÷å} \end{cases}.$$

```
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.

3. Решение обыкновенных дифференциальных уравнений.

Многие прикладные задачи сводятся к решению обыкновенных дифференциальных уравнений (ОДУ) или систем таких уравнений. Для некоторых ОДУ можно построить формулы «точного» решения, например, для уравнений и систем с постоянными коэффициентами. Элементы символьной математики, встроенные в MATLAB, позволяют находить аналитический вид решений таких уравнений. Но и для них, если функции внешних воздействий сложны (разрывные, ломанные или неинтегрируемые функции) построение «аналитических» решений затруднительно. Поэтому использование приближенных методов крайне важно и в MatLab реализовано большое количество численных алгоритмов решения ОДУ.

В данной главе мы приводим основные сведения о функциях MATLAB, предназначенных для решения ОДУ и их систем, как в символьном, так и в численном виде. Однако, в данной брошюре рассматриваются только функции, предназначенные для решения задачи Коши для ОДУ, и символьные возможности построения общих решений дифференциальных уравнений в частных производных (ДУЧП). Возможности MATLAB численного решения краевых задач ОДУ и ДУЧП здесь не рассматриваются.

3.1. Символьное решение дифференциальных уравнений

3.1.1 Использование встроенной функции dsolve.

Функцией MATLAB, решающей ОДУ в символьном виде, является функция `dsolve`. Простейший формат ее вызова следующий

```
dsolve('equation'),
```

или

```
var = dsolve('equation'),
```

где `'equation'` – строка символов, задающая уравнение, а независимой переменной, если она не указана, является `t`. Символ `D` в строке уравнения обозначает дифференцирование по независимой переменной, представляя например, оператор $\frac{d}{dt}$. Цифры, следующие за символом `D` (если они указаны), обозначают порядок производной. Например, `D2` обозначает оператор $\frac{d^2}{dt^2}$. Любой символ, следующий без пробела за оператором дифференцирования, представляет имя зависимой переменной (которая не должна содержать символ `D`), например, `D3y` обозначает $\frac{d^3y}{dt^3}$. Арифметические операции и функции в строке, представляющей уравнение, обозначаются обычным образом (без точек). Если начальные условия не заданы, то в решении присутствуют постоянные интегрирования `C1`, `C2` и т.д. Возвращаемое выражение `var` имеет символьный тип `sym` (не путать со строковым типом `char`).

z=dsolve('Dx = -a*x') % решаем уравнение $\frac{d\,x}{d\,t} = -a \cdot x(t)$

z =C1*exp(-a*t) % получаем $x(t) = C_1 e^{-a\,t}$

Если в рабочем пространстве имен MATLAB есть переменные *a* и C1 (пусть, например, *a*=3, C1=10), то их подстановку в решение можно выполнить командой

subs(z)

```
ans = 10*exp(-3*t)
```

Решим уравнение $\frac{d\,z}{d\,t} = 4\,z(t) + 3\sin t$

dsolve('Dz = 4*z + 3*sin(t)')

```
ans = -3/17*cos(t)-12/17*sin(t)+exp(4*t)*C1
```

Поучаем $C_1 e^{4t} - \frac{3}{17}\cos t - \frac{12}{17}\sin t$. Как видим, в решениях используется постоянная C1. Если порядок уравнения более высокий, то в решении появляются и другие независимые постоянные.

dsolve('D2y+4*Dy+3*y+2=0') % решаем уравнение $y'' + 4\,y' + 3\,y + 2 = 0$

ans = exp(-3*t)*C2+exp(-t)*C1-2/3% получаем $C_1 e^{-t} + C_2 e^{-3t} - 2/3$

Функция dsolve ищет решение в символьном виде и пытается вернуть решение в квадратурах. Если уравнение содержит неопределенную функцию, то в решении появляются неопределенные интегралы. Например

dsolve('Dy=f(t)')

```
ans = Int(f(t),t)+C1
```

dsolve('Dy-y=f(t)')

```
ans = (Int(f(t)*exp(-t),t)+C1)*exp(t)
```

dsolve('Dy-g(t)*y=f(t)')

```
ans = (Int(f(t)*exp(-Int(g(t),t)),t)+C1)*exp(Int(g(t),t))
```

Функция dsolve умеет работать с некоторыми разрывными функциями, например

dsolve('D2y-y=heaviside(t)')

```
ans = exp(-t)*C2+exp(t)*C1+1/2*heaviside(t)*(-2+exp(-t)+exp(t))
```

Здесь heaviside – функция Хевисайда, для которой в математической литературе принято обозначение H(x) и которая определяется выражением

$$H(x) = \begin{cases} 0, & x < 0 \\ 1, & x > 0 \end{cases}$$

Ее значение в точке разрыва в различных источниках определяется по-разному. MATLAB в точке разрыва функцию Хевисайда не определяет и возвращает NaN.

В формате вызова функции dsolve можно указать другое имя независимой переменной (не t) следующим образом

```
dsolve('equation','varname')
```

или

```
var = dsolve('equation','varname')
```

Например

dsolve('D2y+4*Dy+2*y+c=0','u')

```
ans = exp((-2+2^(1/2))*u)*C2+exp(-(2+2^(1/2))*u)*C1-1/2*c
```

Решим уравнение Бернулли $y' = y \cdot ctg\, x + \frac{y^2}{\sin x}$. Имеем

y=dsolve('Dy=y*cos(x)/sin(x)+y^2/sin(x)','x')

```
y = -sin(x)/(x-C1)
```

Обратите внимание, что переменная `y` имеет символьный тип, а переменной `x` в рабочем пространстве нет. Чтобы выполнить проверку создадим символьную переменную `x` и символьное выражение `ss`, представляющее дифференциальное уравнение.

syms x

ss=y*cos(x)/sin(x)+y^2/sin(x)-diff(y,x)

```
ss = 0
```

В символьное выражение `ss` значение `y`, которое вернула функция `dsolve`, было подставлено автоматически.

Общий формат вызова функции `dsolve` следующий

```
r = dsolve('eq1,eq2,...', 'cond1,cond2,...', 'v')
r = dsolve('eq1','eq2',...,'cond1','cond2',..., 'v')
```

Здесь `eq1`, `eq2`,... – строковые выражения, представляющие обыкновенные дифференциальные уравнения, использующие для обозначения независимой переменной символ `v`; `cond1`, `cond2`, ... – выражения, задающие начальные и/или граничные условия. Начальные/граничные условия задаются строками вида `'y(a)=b'` или `'Dy(a)=b'`, где `y` имя искомой функции, а `a`, `b` – постоянные. Имя независимой переменной должно задаваться последним аргументом. Если аргумент `'v'` не задан, то имя независимой переменной полагается `t`. Если начальных/граничных условий недостаточно для однозначной разрешимости уравнения(й), то в решении используются произвольные постоянные, обозначаемые `C1`, `C2`, …. Уравнения и начальные/граничные условия можно передавать отдельными строками, однако, всех аргументов не должно быть больше 12. Указывать явно имя искомой функции не нужно.

dsolve('Dy=1+y^2')

```
ans = tan(t+C1)
```

Здесь символ `y` используется в качестве имени искомой функции и `t` – в качестве независимой переменной. Как видим, решением является функция $tg(t + C_1)$. Для задания начального условия используем второй аргумент

y = dsolve('Dy=1+y^2','y(0)=1')

```
y = tan(t+1/4*pi)
```

Обратите внимание, что переменная `y` появилась в рабочем пространстве MATLAB, а независимая переменная `t` – нет. Добавим символьную переменную `t` рабочее пространство, выполнив команду

syms t

Теперь можно проверить, что функция `tan(t+pi/4)` удовлетворяет дифференциальному уравнению, напечатав символьное выражение, представляющее уравнение

diff(y,t)-1-y^2

```
ans = 0
```

Замена символьной переменной y ее выражением произошла автоматически. Для проверки начального условия выполним команду подстановки

subs(y,t,0)

```
ans = 1.0000
```

В следующем примере решим краевую задачу для дифференциального уравнения второго порядка $y'' = -a^2 y,\ y(0) = 1,\ y'\left(\frac{\pi}{a}\right) = 0$. Имеем

dsolve('D2y = -a^2*y', 'y(0) = 1', 'Dy(pi/a) = 0')

```
ans = cos(a*t)
```

Вот другие примеры

dsolve('D2y+4*y=0','y(0)=0')

```
ans = C1*sin(2*t)
```

dsolve('D2y+4*y=0','y(0)=0, Dy(0)=1')

```
ans = 1/2*sin(2*t)
```

dsolve('x*Dy=3*y+x^4*cos(x)','y(2*pi)=0','x')

```
ans = x^3*sin(x)
```

Общее решение ДУ 4 – го порядка включает 4 неопределенные константы

de='D4y+2*D2y-x=1';

dsolve(de,'x')

```
ans =
-1/2*sin(2^(1/2)*x)*C2-1/2*cos(2^(1/2)*x)*C1+1/12*x^3+1/4*x^2+C3*x+C4
```

dsolve('D4y=y')

```
ans = C1*exp(-t)+C2*exp(t)+C3*sin(t)+C4*cos(t)
```

Каждое независимое начальное условие уменьшает на единицу количество независимых констант

dsolve('D4y=y','y(0)=0, Dy(0)=0')

```
ans = (-C2-C4)*exp(t)+C2*exp(-t)+(2*C2+C4)*sin(t)+C4*cos(t)
```

В граничных условиях можно задавать линейную комбинацию значений функции и ее производных.

y=dsolve('D2y + y=0','y(0)=1, y(1)+Dy(1)=0');

pretty(y)

```
                - cos(1) + sin(1)
                  ---------------- sin(t) +  cos(t)
                   cos(1) + sin(1)
```

Строки уравнений и начальных/граничных условий могут содержать неопределенные переменные. В этом случае они интерпретируются как символьные и входят в результирующее символьное решение

dsolve('Dy = a*y', 'y(0) = b') % решаем задачу Коши $y' = a\,y,\ y(0) = b$

```
ans = b*exp(a*t)      % получаем y = b*exp(a*t)
```

f=dsolve('m*D2y-k*Dy=0','y(0)=0,y(1)=1','x')

```
f = -1/(-1+exp(k/m))+1/(-1+exp(k/m))*exp(k/m*x)
```

pretty(f)

```
                                        k x
                                     exp( --- )
                1                         m
            - ------------- + -------------
              -1 + exp(k/m)   -1 + exp(k/m)
```

Функция dsolve может возвратить результат тремя различными способами. Для одного уравнения и одной результирующей переменной dsolve возвращает решение в виде символьного выражения (или вектора символьных

выражений, если уравнение было нелинейным и для него найдено несколько решений). Для системы нескольких уравнений и равного количество выходных переменных `dsolve` лексически упорядочивает решения и присваивает их выходным переменным. Для системы уравнений и одной выходной переменной `dsolve` возвращает структуру с полями, содержащими выражения решений.

Например, для следующего нелинейного дифференциального уравнения функция `dsolve` вернет вектор из 4 – х символьных выражений

z=dsolve('(Dy)^2 + y^2 = 1') `% решаем уравнение` $(y')^2 + y^2 = 1$

```
z =
          1
         -1
  sin(t-C1)
 -sin(t-C1)
```

Если задать начальные условия, то решений будет два

y = dsolve('(Dy)^2 + y^2 = 1','y(0) = 0')

```
y =
 -sin(t)
  sin(t)
```

В следующем примере для системы двух уравнений мы получим решение в виде двух символьных выражений `x` и `y`.

[x y]=dsolve('Dx = y', 'Dy = -x')

```
x = -C1*cos(t)+C2*sin(t)
y = C1*sin(t)+C2*cos(t)
```

Если принимаем решение в одну переменную, то она будет структурой с двумя полями, имена которых совпадут с именами искомых функций

q=dsolve('Du = v', 'Dv = -u')

```
q =
    u: [1x1 sym]
    v: [1x1 sym]
```

q.u

```
ans = -C1*cos(t)+C2*sin(t)
```

q.v

```
ans = C1*sin(t)+C2*cos(t)
```

Обращаться с результатом решения ДУ следует также как с любыми символьными объектами. Для примера решим краевую задачу для дифференциального уравнения второго порядка $y'' = -4y,\ y(0) = 1,\ y'\left(\frac{\pi}{2}\right) = 0$ и построим график решения

z=dsolve('D2y = - 4*y', 'y(0) = 1', 'Dy(pi/2) = 0')

```
z = cos(2*t)
```

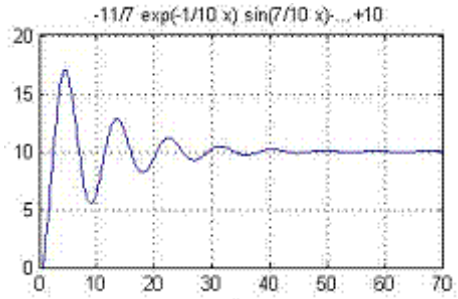
ezplot(z) `% строим график решения`

Напомним, что функция `ezplot` строит график символьного или строкового выражения, переданного ей в качестве первого (и возможно единственного) аргумента.

f=dsolve('4*D2y+0.8*Dy+2*y=20','y(0)=-1, Dy(0)=0','x')

```
f =
-11/7*exp(-1/10*x)*sin(7/10*x)-11*exp(-1/10*x)*cos(7/10*x)+10
```

ezplot(f,[0, 70]); axis([0 70 0 20]); grid on;



Для ДУ 2 – го порядка фазовые траектории строятся на плоскости (y, y').

y=dsolve('D2y + y=0','y(0)=1, Dy(0)=1')

```
y = sin(t)+cos(t)
```

syms t;

z=diff(y,t)

```
z = cos(t)-sin(t)
```

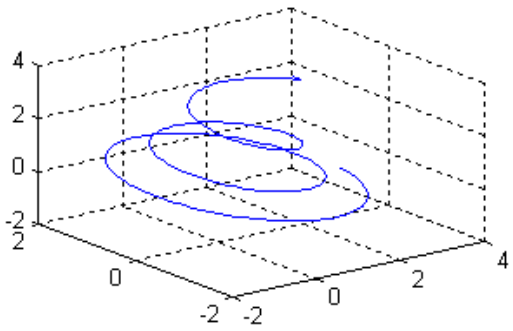
ezplot(y,z,[0,2*pi]) `% строим фазовую траекторию = окружность`

В следующем примере мы строим фазовую траекторию системы

$$\frac{dx}{dt} = y - z, \frac{dy}{dt} = z - x, \frac{dz}{dt} = x - 2 \cdot y,$$

проходящую через точку (1, 0, 2). Для удобства весь код мы собираем в единую функцию `ex005`.

```
function ex005
% пример построения фазовой траектории системы 3-х уравнений
sys='Dx = y-z, Dy = z-x, Dz=x-2*y';
cnd='x(0)=1,y(0)=0, z(0)=2';
[x,y,z]=dsolve(sys,cnd);
ezplot3(x,y,z,[-4 6]);
```



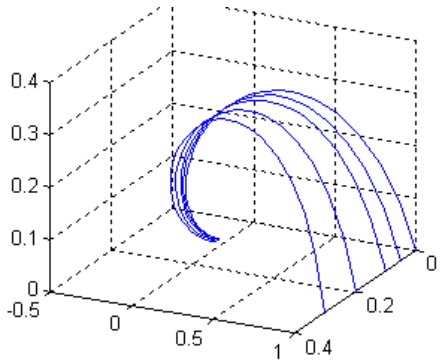
В следующем примере мы строим фазовые траектории системы дифференциальных уравнений

$$\frac{dx}{dt} = -x + z, \frac{dy}{dt} = -y - z, \frac{dz}{dt} = y - z,$$

проходящие при `t=0` через точки (0, 1, 0), (0.05, 1, 0), (0.1, 1, 0), (0.2, 1, 0), (0.3, 1, 0).

```
function ex007
% пример построения фазовых таекторий в E3
sys='Dx = -x+z, Dy = -y-z, Dz=y-z';
cnd={'x(0)=0,y(0)=1, z(0)=0','x(0)=0.05,y(0)=1,...
      z(0)=0','x(0)=0.1,y(0)=1, z(0)=0','x(0)=0.2,y(0)=1,...
      z(0)=0','x(0)=0.3,y(0)=1, z(0)=0'};
for k=1:5
    [x,y,z]=dsolve(sys,char(cnd(k)));
    X=subs(x,t,0:0.1:pi);
    Y=subs(y,t,0:0.1:pi);
    Z=subs(z,t,0:0.1:pi);
    plot3(X,Y,Z);
```

```
    hold on;
end
hold off;
grid on;
```



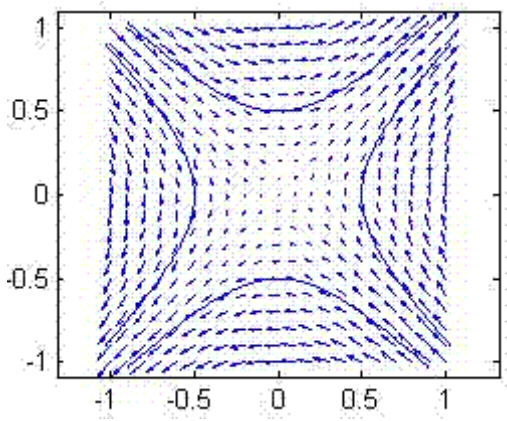
Если система двух дифференциальных уравнений автономная, то на фазовой плоскости можно строить поле направлений. Для этого используется функция `quiver(X,Y,U,V)` для которой X, Y – матрицы координат точек фазовой плоскости, а U,V – матрицы координат векторного поля в этих точках.

В следующем примере мы создали функцию `ex010`, которая решает систему уравнений

$$\frac{dx}{dt} = y, \frac{dy}{dt} = x,$$

строит поле направлений этой системы и строит четыре фазовые траектории, проходящие через точки (0, 0.5), (0.5, 0), (-0.5,0), (0, -0.5).

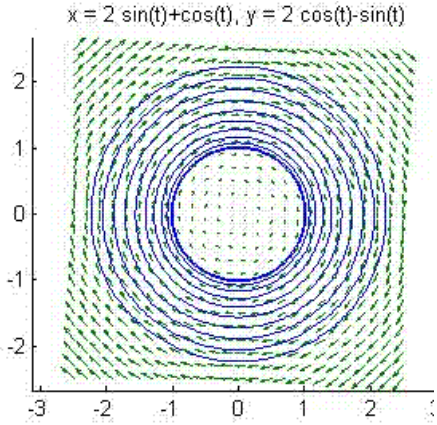
```
function ex010
% пример построения фазовых траекторий и поля направлений системы ДУ
[X,Y] = meshgrid(-1:0.1:1);
quiver(X,Y,Y,X); % строим поле направлений (вектор [Y,X] в точке [X,Y])
hold on;
[x y]=dsolve('Dx = y, Dy = x','x(0)=0,y(0)=0.5');
ezplot(x,y,[-1.35 1.35]);
[x y]=dsolve('Dx = y, Dy = x','x(0)=0.5,y(0)=0');
ezplot(x,y,[-1.35 1.35]);
[x y]=dsolve('Dx = y, Dy = x','x(0)=-0.5,y(0)=0');
ezplot(x,y,[-1.35 1.35]);
[x y]=dsolve('Dx = y, Dy = x','x(0)=0,y(0)=-0.5');
ezplot(x,y,[-1.35 1.35]);
hold off;
```



Вот пример построения поля скоростей и фазовых траекторий для одного ДУ 2 – го порядка $f'' + f = 0$, которое заменой $y = f, z = f'$ может быть

представлено системой $y' = z, z' = -y$, похожей на систему из предыдущего примера.

```
function ex004
% построения фазовых траекторий и поля направлений уравнения 2 - го порядка
syms t;
newplot; hold on;
for a=0: 0.2:2
    x=dsolve('D2f+f=0','f(0)=1', ['Df(0)=' num2str(a)]);
    y=diff(x,t);
    ezplot(x,y,[0,2*pi]);
end;
[X,Y] = meshgrid(-2:0.2:2);
U=Y;
V=-X;
quiver(X,Y,U,V);    % строим поле направлений
hold off;
```



Решение многих ДУ содержит неэлементарные (специальные) функции. Например следующее однородное ДУ разрешимо в элементарных функциях.

dsolve('Dy-x*y=0','x')

```
ans = C1*exp(1/2*x^2)
```

Но добавление неоднородности усложняет задачу и в решении появляется специальная функция – интеграл ошибок.

dsolve('Dy-x*y=1','x')

```
ans =
(1/2*pi^(1/2)*2^(1/2)*erf(1/2*2^(1/2)*x)+C1)*exp(1/2*x^2)
```

Посмотреть график функции `erf(x)` можно следующим способом

ezplot('erf(x)')

Если задать начальное условие, то в решении не будет неопределенных постоянных и можно построить график решения

z=dsolve('Dy-x*y=1','y(0)=0','x')

```
ans =
1/2*exp(1/2*x^2)*pi^(1/2)*2^(1/2)*erf(1/2*2^(1/2)*x)
```

ezplot(z)

В следующем примере решение выражается через функции Бесселя

dsolve('D2y-exp(x)*y=0','x')

```
ans =
C1*besseli(0,2*exp(1/2*x))+C2*besselk(0,2*exp(1/2*x))
```

В следующем примере решение представляется через функции Эйри

dsolve('D2y-x*y=0','x')

```
ans = C1*AiryAi(x)+C2*AiryBi(x)
```

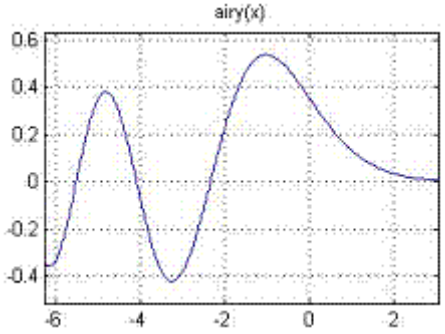
Однако следует иметь в виду, что обозначения функций Эйри в MatLab и MAPLE различается (функция `dsolve` наследует обозначения системы символьных вычислений MAPLE). Поэтому команда

ezplot('AiryAi(x)')

```
??? Error using ==> inlineeval
Error in inline expression ==> AiryAi(x)...
```

возвращает ошибку. Следует в справочной системе MatLab найти правильное обозначение этой функции и в последующих вычислениях использовать его

ezplot('airy(x)') `% график функции Эйри Ai(x)`



Если решение в явном виде найти не удается, то функция `dsolve` пытается найти решение в неявном виде и при этом может вывести соответствующее сообщение. В следующем примере мы решаем уравнение Абеля и решение получаем в неявной форме (в форме алгебраического уравнения)

q=dsolve('Dy+x*y^3+y^2=0','x')

```
Warning: Explicit solution could not be found; implicit solution returned.
In dsolve at 320
q = log(x)-C1+1/2*log(-1+y^2*x^2+x*y)-...
     1/5*5^(1/2)*atanh((2/5*x*y+1/5)*5^(1/2))-log(x*y) = 0
```

Использовать начальные условия для определения решения в таком случае невозможно

dsolve('Dy+x*y^3+y^2=0','y(0)=1','x')

```
??? Error using ==> dsolve
Error, (in dsolve/IC) The 'implicit' option is not available when giving Initial
Conditions.
```

Однако можно выполнить подстановку

syms C1; subs(q,C1,1)

Если функция `dsolve` не может найти решение, то она выводит сообщение и возвращает пустой символьный объект.

dsolve('D2y=sin(y)*sin(t)')

```
Warning: Explicit solution could not be found.
ans =
[ empty sym ]
```

de='D2y+y+y^3-cos(x)=0';

dsolve(de,'x')

```
Warning: Explicit solution could not be found.
In dsolve at 338
ans =
[ empty sym ]
```

3.1.2 Символьное решение дифференциальных уравнений.

В MATLAB встроено ядро системы символьных вычислений MAPLE, в котором можно решать обыкновенные дифференциальные уравнения. Рассмотренная выше встроенная функция `dsolve` наследует большинство возможностей одноименной функции MAPLE, однако, не все. Чтобы использовать дополнительные возможности символьного решения ОДУ следует обращаться к функции `dsolve` ядра MAPLE.

Напомним, что если надо обратиться к стандартной функции MAPLE, то ее (имя и аргументы) следует передать функции `maple` как текстовую строку. Создадим для примера кусочно – постоянную функцию (точнее выражение) `z` в рабочем пространстве ядра MAPLE

```
maple 'z:=piecewise(x<0,0,x<1,1,2)'  %создаем выражение z(x)
ans = z := PIECEWISE([0, x < 0],[1, x < 1],[2, otherwise])
```

Тогда проинтегрировать ее в символьном виде, используя ядро MAPLE, можно, вызывая функцию интегрирования `int`, следующими способами

```
maple('int(z,x)')
maple 'int(z,x)'
maple int z x
maple('int','z','x')
```

Все перечисленные вызовы возвращают одинаковый ответ

```
ans = PIECEWISE([0, x <= 0],[x, x <= 1],[2*x-1, 1 < x])
```

Этот пример иллюстрирует различные способы обращения к функциям MAPLE. Если вы присваиваете результат переменной, то допустимо использовать только способы со скобками

```
v=maple('int(z,x)')
u=maple('int','z','x')
```

Обратите внимание на тип переменных `u,v` – он строковый. Если входной аргумент являлся строкой, то функция `maple` возвращает переменную строкового типа. Если входной аргумент функции `maple` являлся символьным выражением, то возвращается символьное выражение. Например

```
q = sym('int(x^2,x)');  % создаем символьное выражение
w = maple(q)      % переменная w имеет символьный тип
w = 1/3*x^3
```

Для решения дифференциальных уравнений с помощью функции `dsolve` ядра MAPLE следует использовать синтаксис MAPLE для обозначения производных и уравнений. Дифференцирование в MAPLE выполняется функцией `diff` с указанием имени функции с аргументом (т.е. указывается выражение) и переменной, по которой происходит дифференцирование, например, `diff(y(x),x)`. Первым аргументом функции `dsolve` ядра MAPLE является уравнение, вторым – искомая функция, последующие необязательные аргументы могут указывать метод решения или способ представления результата. Справку по функции `dsolve` можно получить командой

```
mhelp dsolve
```

где `mhelp` – команда для получения справки по функциям ядра MAPLE.

Интегрирование, рассмотренное в предыдущем примере, это фактически решение дифференциального уравнения

maple 'dsolve({diff(y(x),x)=piecewise(x<0,0,x<1,1,2)},y(x))'

```
ans =
{y(x) = PIECEWISE([_C1, x < 0],[_C1+x, x < 1],[_C1+2*x-1, 1 <= x])}
```

В примере выше при интегрировании неопределенная константа была отброшена, а при решении ОДУ функция `dsolve` ее оставила. Вот еще примеры

maple('dsolve(diff(y(x),x)*sqrt(1-x^2)=1+y(x)^2,y(x))')

```
ans =
y(x) = tan(asin(x)+_C1)
```

Выполним проверку, используя встроенные возможности MAPLE.

maple 'q:=subs(y(x)=tan(arcsin(x)+_C1),1+y(x)^2-diff(y(x),x)*sqrt(1-x^2))'

```
ans =
q := 1+tan(asin(x)+_C1)^2-diff(tan(asin(x)+_C1),x)*(1-x^2)^(1/2)
```

maple 'simplify(q)'

```
ans = 0
```

Для проверки того, что найденное решение удовлетворяет уравнению, в ядре MAPLE есть функция `odetest`. Она имеет следующий формат вызова

`odetest(sol, eq)`

где `sol` переменная пространства имен MAPLE, содержащая выражение, возвращенное функцией `dsolve`, а `eq` – уравнение. Решение ДУ и проверку решения последнего примера можно выполнить следующим образом

maple 'eq:=diff(y(x),x)*sqrt(1-x^2)-1+y(x)^2';
maple('sol:=dsolve(eq,y(x))')

```
ans =
sol := y(x) = tanh(asin(x)+_C1)
```

maple 'odetest(sol,eq)'

```
ans = 0
```

Способ символьного решения указывается третьим аргументом `type=typename`. Можно искать решение дифференциального уравнения с помощью преобразования Лапласа (`type=laplace`), в виде степенного ряда различной длины (`type=series`), в неявном виде (`type=implicit`), в параметрическом виде (`type=parametric`) или в явном виде (`type=explicit`). По умолчанию используется опция `explicit`.

Создадим в пространстве имен ядра MAPLE переменную `sys`, которая будет содержать выражение системы двух дифференциальных уравнений

maple 'sys:=diff(y(x),x)-2*z(x)-y(x)-x=0, diff(z(x),x)=y(x)'

```
ans =
sys := diff(y(x),x)-2*z(x)-y(x)-x, diff(z(x),x) = y(x)
```

Тогда решить задачу Коши для этой системы можно следующей командой

maple 'dsolve({sys,y(0)=0,z(0)=1},{y(x),z(x)})'

```
ans =
{y(x) = -1/3*exp(-x)+5/6*exp(2*x)-1/2,
 z(x) = 1/3*exp(-x)+5/12*exp(2*x)+1/4-1/2*x}
```

Найти решение этой задачи в виде степенных рядов до третьего порядка можно следующими командами

maple 'Order:=3';
maple 'dsolve({sys,y(0)=0,z(0)=1},{y(x),z(x)},series)'

```
ans = {y(x) = series(2*x+3/2*x^2+O(x^3),x,3),
       z(x) = series(1+1*x^2+O(x^3),x,3)}
```

Здесь переменная `Order` определяет для ядра MAPLE максимальную степень ряда Тейлора для представления решения, а третий аргумент `series` функции `dsolve` определяет метод поиска решения в виде степенного ряда. Решение в виде степенного ряда пятого порядка можно получить следующими командами

maple 'Order:=5';
maple 'dsolve({sys,y(0)=0,z(0)=1},{y(x),z(x)},series)'

```
ans =
{y(x) = series(2*x+3/2*x^2+7/6*x^3+13/24*x^4+O(x^5),x,5),
 z(x) = series(1+1*x^2+1/2*x^3+7/24*x^4+O(x^5),x,5)}
```

Преобразовать ряды в полиномы можно следующим образом

maple 'sol:=dsolve({sys,y(0)=0,z(0)=1},{y(x),z(x)},series)';
maple convert(rhs(sol[1]),polynom)

```
ans = 2*x+3/2*x^2+7/6*x^3+13/24*x^4
```

maple convert(rhs(sol[2]),polynom)

```
ans = 1+x^2+1/2*x^3+7/24*x^4
```

Здесь функция `rhs` (right hand side) возвращает символьное выражение, находящееся в правой части уравнения, которое содержится в переменных `sol[1]` и `sol[2]`. Получить решение той же задачи Коши, используя метод преобразования Лапласа, можно следующим образом

maple restart;
maple 'sys:=diff(y(x),x)-2*z(x)-y(x)-x=0, diff(z(x),x)=y(x)'
maple 'dsolve({sys,y(0)=0,z(0)=1},{y(x),z(x)},laplace)'

```
ans =
{y(x) = -1/3*exp(-x)+5/6*exp(2*x)-1/2,
 z(x) = 1/3*exp(-x)+5/12*exp(2*x)+1/4-1/2*x}
```

Если параметр `type` не указан, то по умолчанию используется значение `explicit` – поиск решения в явном виде (можно использовать `type = exact` – поиск аналитического решения). Явное использование значения `explicit` в данном примере дает то же решение, что и выше.

maple 'dsolve({sys,y(0)=0,z(0)=1},{y(x),z(x)},explicit)'

Если в явном виде решение не существует, то система пытается найти его в неявном виде.

maple 'eq:=diff(y(x),x)=sqrt(x^2-y(x))+2*x' ;
maple 'dsolve(eq,y(x))'

```
ans =
2/(4*y(x)-3*x^2)/(2*(x^2-y(x))^(1/2)+x)*(x^2-y(x))^(1/2)-1/(4*y(x)-
3*x^2)/(2*(x^2-y(x))^(1/2)+x)*x-_C1 = 0
```

maple 'isolate(%,y(x))'

```
ans =
y(x) = 1/4*(5*_C1*x^2+1+2*_C1*x*(-x+1/(-_C1)^(1/2)))/_C1
```

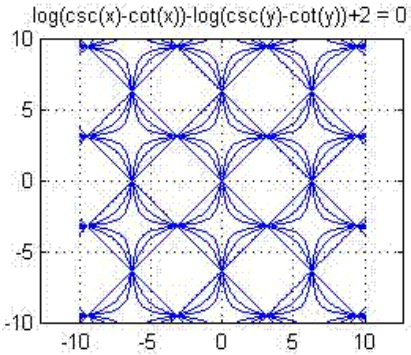
Здесь команда `isolate` ядра MAPLE выражает заданное вторым параметром выражение (`y(x)`) из уравнения, определяемого первым параметром (в нашем случае из неявного вида общего решения дифференциального уравнения). Символ `'%'` (процент) заменяет результат последнего вычисления в ядре MAPLE.

Решение можно сразу искать в неявном виде, передав функции `dsolve` значение параметра `type=implicit` (или просто `implicit`). Этот же параметр можно сразу использовать, если требуется получить не общее решение, а общий интеграл ОДУ.

maple('dsolve(diff(y(x),x)*sqrt(1-x^2)=1+y(x)^2,y(x),implicit)')
```
ans =
asin(x)-atan(y(x))+_C1 = 0
```

Построим, например, интегральные кривые уравнения $\frac{dy}{dx}=\frac{\sin y}{\sin x}$. Для этого надо получить решение ДУ в неявном виде и заменить в нем постоянную `_C1` на конкретные значения. Поскольку функция `ezplot(f(x,y),[xmin, xmax, ymin, ymax])`, строит кривую по ее неявному уравнению в виде `f(x,y)=0`, то для получения выражения `f(x,y)` в требуемом формате следует еще выполнить замену `y(x)->y`. Для выделения левой части решения `q`, которое ядро MAPLE возвращает в форме уравнения, используем еще функцию `lhs` (left hand side).

maple('q:=dsolve(diff(y(x),x)=sin(y(x))/sin(x),y(x),implicit)')
q1=maple('lhs(subs([_C1=0, y(x)=y],q))')
ezplot(q1,[-10 10 -10 10]); axis equal; grid on; hold on;
q2=maple('lhs(subs([_C1=1, y(x)=y],q))');
ezplot(q2,[-10 10 -10 10]);
q3=maple('lhs(subs([_C1=2, y(x)=y],q))');
ezplot(q3,[-10 10 -10 10]); hold off;
```
ans = q := log(csc(x)-cot(x))-log(csc(y(x))-cot(y(x)))+_C1 = 0
q1 =log(csc(x)-cot(x))-log(csc(y)-cot(y))
```



В следующем примере решение в явном и неявном виде найти не удается

maple('dsolve(ln(diff(y(x),x))+sin(diff(y(x),x))-x=0,y(x))')
```
ans =
y(x) = Int(RootOf(-log(_Z)-sin(_Z)+x),x)+_C1
```

Однако в параметрическом виде оно выглядит относительно просто

maple('q:=dsolve(ln(diff(y(x),x))+sin(diff(y(x),x))-x=0,y(x),parametric)')
```
ans =
q := [x(_T) = log(_T)+sin(_T), y(_T) = cos(_T)+_T*sin(_T)+_T+_C1]
```

MATLAB не «хочет» работать с переменными имена которых начинаются с символа подчеркивания. Поэтому, полученное решение, желательно преобразовать в строковое. После этого создадим строковые переменные `x` и `y`, которые будут содержать полученные выражения и с которыми можно работать обычным образом – создавать символьные выражения, функции MATLAB или строить график.

maple('qq:=subs([_T=t, _C1=0],q)')
```
ans = qq := [x(t) = log(t)+sin(t), y(t) = cos(t)+t*sin(t)+t]
```
x=maple('rhs(qq[1])')
```
x = log(t)+sin(t)
```
y=maple('rhs(qq[2])')
```
y = cos(t)+t*sin(t)+t
```

Функция `rhs` (right hand side) возвращает символьное выражение, находящееся в правой части уравнения, которое содержится в переменной `qq[1]`, а функция `maple` возвращает уже строковое выражение.

Аргумент `parametric` Функции `dsolve` полезен при решении классической задачи о брахистохроне, которая требует минимизации функционала

$$T = \int_{x_A}^{x_B} \frac{\sqrt{1+y'(x)^2}}{\sqrt{2g\left(y_A - y(x)\right)}}\,dx$$

Задача сводится к дифференциальному уравнению

$$\left(y_A - y\right)\cdot\left(1+y'(x)^2\right) = Const$$

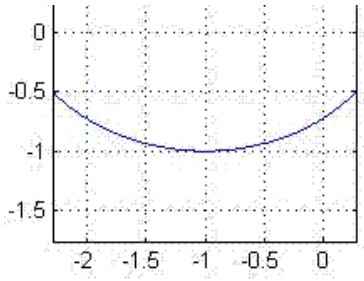
общее решение которого можно получить в параметрическом виде.

```
function ex009
% задача о брахистохроне
maple('q1:=dsolve((yA-y(x))*(1+diff(y(x),x)^2)=C, y(x), parametric)');
maple 'q2:=subs([_T=t,C=1,_C1=-1,yA=0],q1)'; %подстановка
y=maple('rhs(q2[1])');
x=maple('rhs(q2[2])');
ezplot(x,y,[-1,1]);
grid on;
```

Приведенная функция `ex009` возвращает решение в параметрическом виде

```
q2 := [y(t) = -1/(1+t^2), x(t) = (t+atan(t)+atan(t)*t^2-1-t^2)/(1+t^2)]
```

и строит кривую решения



Чтобы построить график мы выполнили замену постоянных интегрирования. В задаче о брахистохроне их следует подбирать из граничных условий.

Функция `dsolve` ядра MAPLE, кроме прочего, предоставляет возможность определения линейно независимых базисных решений дифференциального уравнения. Для этого следует указать способ вывода результата, задав значение параметра `output` равное значению `basis`.

maple('dsolve((D@@2)(y)(x)+3*D(y)(x)+2*y(x)-a*exp(x),y(x),output=basis)')

```
ans =
[[exp(-x), exp(-2*x)], 1/6*a*exp(x)]
```

В MAPLE имеется пакет расширения `DEtools`, который содержит набор функций, расширяющий возможности по исследованию и решению дифференциальных уравнений. При загрузке пакета можно увидеть список всех загружаемых функций.

maple 'with(DEtools)'

```
[DEnormal, DEplot, DEplot3d, DEplot_polygon, DFactor, DFactorLCLM, DFactorsols,
Dchangevar, ...
```

Однако следует иметь в виду, что графические функции пакета работать в MATLAB не будут.

Приведем пример использования одной простой и полезной функции этого пакета `odeadvisor`. Она сообщает тип ДУ (с разделяющимися переменными, 2 – го порядка и т.д). Перед первым использованием ее следует загрузить в ядро MAPLE командой

maple 'with(DEtools,odeadvisor)'

```
ans =
[odeadvisor]
```

или загрузить все функции пакета так, как показано выше. После этого функция готова к работе. Выполняем команду

maple 'odeadvisor(diff(y(x),x)*sqrt(1-x^2)=1+y(x)^2,y(x))'

```
ans =
[_separable]
```

Это уравнение с разделяющимися переменными. Уравнение может принадлежать нескольким типам. В таком случае функция `odeadvisor` возвращает пользователю их список.

maple 'odeadvisor(diff(y(x),x,x)+4*y(x)=1,y(x))'

```
ans =[[_2nd_order, _missing_x]]
```

maple 'odeadvisor(diff(y(x),x$2)+4*y(x)=x,y(x))'

```
ans = [[_2nd_order, _with_linear_symmetries]]
```

maple 'odeadvisor(diff(y(x),x$2)+4*y(x)^2=x,y(x))'

```
ans = [NONE]
```

Для использования функций пакетов следует познакомиться с их описанием. Это можно сделать либо по справочной системе MAPLE, либо командой

`mhelp funcname`

Имена функций можно узнать из списка, отображаемого при загрузке пакета. Список доступных пакетов MAPLE можно узнать из справочной системы MATLAB.

3.1.3 Символьное решение уравнений в частных производных.

Ядро MAPLE «умеет» аналитически решать некоторые дифференциальные уравнения в частных производных (ДУЧП). Это делает функция `pdsolve`. Ее использование разберем на примерах.

Найдем общее решение следующего уравнения

$$a\frac{\partial^2 f}{\partial x^2}+b\frac{\partial^2 f}{\partial x\partial y}=c$$

maple('pdsolve(a*diff(f(x,y),x,x)+b*diff(f(x,y),x,y)=c, f(x,y))')

```
ans = f(x,y) = _F1(y)+_F2(a*y-b*x)+1/2*c*x^2/a
```

В решении присутствуют две произвольные функции _F1 и _F2 одной переменной. Вот еще несколько примеров.

Решим уравнение $a\frac{\partial g}{\partial x}+b\frac{\partial^2 g}{\partial x\partial y}=x\cdot y$.

maple('pdsolve(a*diff(g(x,y),x)+b*diff(g(x,y),x,y)=x*y, g(x,y))')

```
ans = g(x,y) = _F1(y)+exp(-a*y/b)*_F2(x)+1/2*x^2*(a*y-b)/a^2
```

Решим уравнение $\frac{\partial u}{\partial x}+\frac{\partial u}{\partial y}=\frac{1}{x\cdot y}$.

maple('pdsolve(diff(u(x,y),x)+diff(u(x,y),y)=1/(x*y), u(x,y))')

```
ans = u(x,y) = (log(y)-log(x)-_F1(y-x)*y+_F1(y-x)*x)/(-y+x)
```

Пока неизвестная функция и ее производные входят в уравнение линейно можно надеяться получить общее решение. Вот пример решения однородного уравнения

$$y\frac{\partial U}{\partial x}+x\frac{\partial U}{\partial y}=0$$

maple('pdsolve(y*diff(U(x,y),x)+x*diff(U(x,y),y)=0, U(x,y))')

```
ans = U(x,y) = _F1(-x^2+y^2)
```

В решении следующего неоднородного уравнения появляется специальная функция

$$x\frac{\partial U}{\partial x}+y\frac{\partial U}{\partial y}=e^{x\cdot y}$$

maple('pdsolve(x*diff(u(x,y),x)+y*diff(u(x,y),y)=exp(x*y), u(x,y))')

```
ans = u(x,y) = -1/2*Ei(1,-x*y)+_F1(y/x)
```

Отметим, что при использовании этого решения следует сверить обозначения специальной функции в MAPLE и MATLAB. Интегральная показательная функция $Ei(x)=\int_{-\infty}^{x}\frac{e^t}{t}dt$ в MATLAB обозначается как `expint(x)`.

Можно решать ДУЧП относительно функций с тремя переменными. Поучим общее решение следующего уравнения

$$\frac{\partial u}{\partial x}+\frac{\partial u}{\partial y}+\frac{\partial u}{\partial y}=0$$

maple('pdsolve(diff(u(x,y,z),x)+diff(u(x,y,z),y)+ diff(u(x,y,z),z)=0, u(x,y,z))')

```
ans =
u(x,y,z) = _F1(y-x,z-x)
```

Поучим общее решение следующего уравнения

$$\frac{\partial u}{\partial x}+a\cdot\frac{\partial u}{\partial y}+b\cdot\frac{\partial u}{\partial y}=c\cdot x\cdot y\cdot z$$

maple('pdsolve(diff(U(x,y,z),x)+a*diff(U(x,y,z),y)+b*diff(U(x,y,z),z)=c*x*y*z, U(x,y,z))')

```
ans =U(x,y,z) = 1/12*c*a*b*x^4+
 (-1/6*c*b*y-1/6*c*a*z)*x^3+1/2*c*x^2*y*z+_F1(y-a*x, z-b*x)
```

Нелинейные уравнения не часто имеют общее решение. Вот один из редких примеров

$$\frac{\partial u}{\partial x}+\frac{\partial u}{\partial y}=e^{u(x,y)}$$

maple('pdsolve(diff(u(x,y),x)+diff(u(x,y),y)=exp(u(x,y)), u(x,y))')

```
ans =
u(x,y) = log(-1/(x+_F1(y-x)))
```

Покажем, как можно использовать полученные решения. Для примера построим общее решение волнового уравнения

$$\frac{\partial^2 u}{\partial t^2}=c^2\frac{\partial^2 u}{\partial x^2},$$

описывающего распространение волн по бесконечной струне со скоростью `c`.

Ниже мы создаем сценарий `infwave.m`, строящий анимацию – распространение бегущей волны по бесконечной струне. Но вначале поясним его основные команды. На первом шаге строим общее решение ДУЧП

maple('eq:=pdsolve(diff(u(x,t),t,t)-c^2*diff(u(x,t),x,x)=0, u(x,t))')

```
ans = eq := u(x,t) = _F1(c*t+x)+_F2(c*t-x)
```

Здесь _F1 и _F2 произвольные функции одной переменной. Затем выполняем подстановку и выделяем правую часть получаемого выражения

maple 'q2:=subs([_F1=F1,_F2=F1, c=1],eq)'

```
ans = q2 := u(x,t) = F1(t+x)+F1(t-x)
```

u=maple('rhs(q2)')

```
u = F1(t+x)+F1(t-x)   % u имеет строковый тип
```

Теперь создаем функцию MATLAB двух переменных, которая будет вычислять значение смещения `U` точки струны с координатой `x` в момент времени `t`.

U=@(x,t) eval(u); `% функция eval вычисляет строковое выражение`

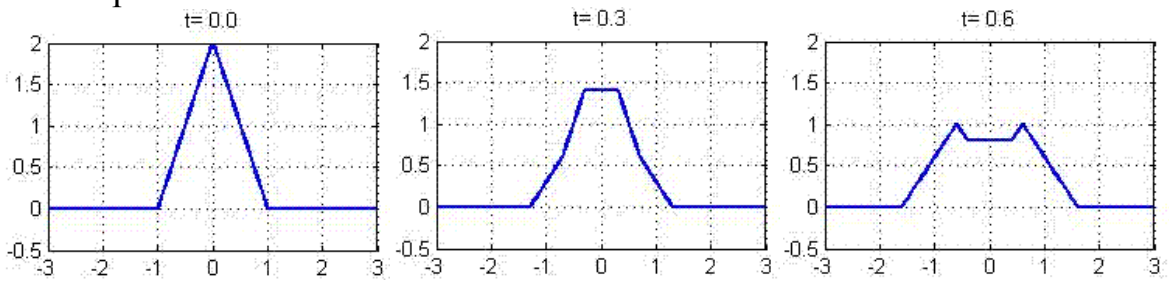
Функция `F1(x)`, используемая в решении, определяет профиль струны в начальный момент времени (`t=0`). Создадим функцию `F1(x)`

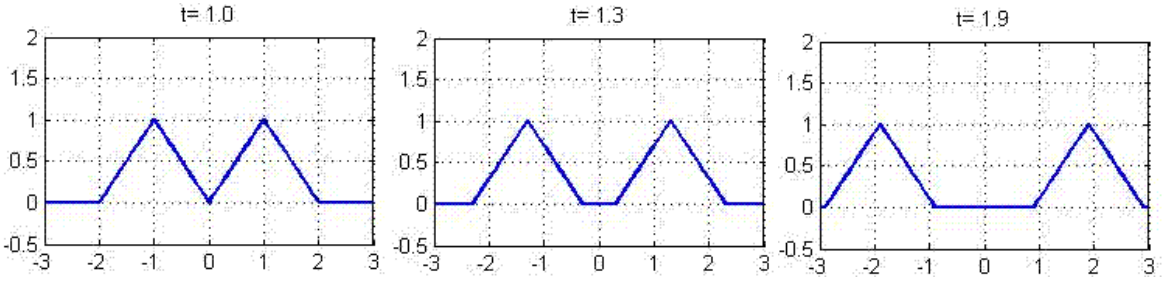
```
function F=F1(x)
F=0.5.*abs(x+1)-abs(x)+0.5.*abs(x-1);
```

Теперь можно создать сценарий, строящий графики функции `U(x,t)` в различные моменты времени. Для наглядности делаем паузу после каждого кадра

```
% сценарий infwave.m - построение графика бегущей волны по бесконечной струне
% находим общее решение одномерного волнового уравнения
maple('eq:=pdsolve(diff(u(x,t),t,t)-c^2*diff(u(x,t),x,x)=0, u(x,t))');
maple 'q2:=subs([_F1=F1,_F2=F1, c=1],eq)'; % выполняем подстановки
u=maple('rhs(q2)');   % u имеет строковый тип
U=@(x,t) eval(u);     % функция eval вычисляет строковое выражение
x=-3:0.1:3;       % задаем диапазон изменения x
i=1;          % номер кадра анимации
for t=0:0.1:3
    y=U(x,t);
    plot(x,y,'LineWidth',2);
    axis([-3 3 -0.5 2]); grid on;
    M(i)=getframe;    % сохраняем график в массив кадров
    i=i+1;
    pause;
end
movie(M);        % проигрываем анимацию
```

На следующих рисунках приведены графики профиля струны в различные моменты времени





Символьное решение дифференциальных уравнений в ядре MAPLE не ограничиваются приведенными примерами и описанными выше функциями. Если вы собираетесь использовать все возможности этой системы, то вам, вероятно, лучше решать дифференциальные уравнения в MAPLE. Однако и в MATLAB вы можете использовать большинство этих возможностей (за исключением графических), а для построения графиков можно использовать функции MATLAB.

3.2. Численное решение задачи Коши

Данный параграф посвящен описанию возможностей, предоставляемых MATLAB, численного решения задачи Коши для обыкновенных дифференциальных уравнений или систем таких уравнений. Для этого предназначены специальные функции, их называют солверы (solver). В MatLab их несколько – для каждого типа задачи и метода решения свой солвер. Изучение методики их применения мы разобьем на несколько шагов. Вначале рассмотрим применение солверов на простых примерах, затем рассмотрим более сложные задачи, «выжимающие» из солверов максимум их возможностей

Задача Коши для одного ОДУ `n` – го порядка состоит в нахождении функции, удовлетворяющей дифференциальному уравнению вида

$$y^{(n)} = f\left(t, y, y', \ldots, y^{(n-1)}\right)$$

и начальным условиям

$$y(t_0) = y_0,\ y'(t_0) = y_1, \ldots, y^{(n-1)}(t_0) = y_{n-1}$$

Перед решением уравнение должно быть записано в виде системы ОДУ первого порядка

$$\mathbf{y}'(t) = \mathbf{F}(t, \mathbf{y}),\ \mathbf{y}(0) = \mathbf{y_0}, \qquad (1)$$

где $\boldsymbol{y}$ – вектор – функция, а $\boldsymbol{y_0}$ – ее начальное значение.

Если задана система ДУ некоторые (или все) уравнения которой имеют порядок второй и выше, то перед решением в MATLAB ее также следует преобразовать к виду (1).

Для решения задачи Коши (1) в MATLAB существует семь функций (солверов): `ode23`, `ode45`, `odell3`, `odel5s`, `ode23s`, `ode23t` и `ode23tb`. Методика их использования одинакова, включая способы задания входных и выходных аргументов. В общем случае вызов солвера для решения задачи Коши производится следующим образом

`[T, Y] = solver(odefun, ,[t0, tend], y0, options)`

Здесь `solver` – имя одной из вышеупомянутых функций, `odefun` – дескриптор функции (или строка с ее именем), реализующей вычисление

вектор – функции **F(*t*,y)** – правой части системы уравнений (1). Функция **F(*t*,y)** должна быть создана заранее, и иметь не менее двух входных аргументов. Ее назначение – вычислять правую часть системы для вектора ***y*** при значении независимой переменной `t` и возвращать результаты в виде вектора столбца. Вектор из двух чисел $[t_0, t_{end}]$ представляет диапазон изменения независимого переменного (начальное и конечное значение). y_0 – вектор начальных значений искомой вектор – функции. Необязательный аргумент структура `options` используется для управления параметрами вычислительного процесса. В ней пользователь может задать абсолютную и/или относительную погрешность, если значения по умолчанию (10^6 и 10^3) его не устраивают.

Солвер возвращает массив `T` с координатами узлов (значений независимой переменной), в которых найдено решение, и матрицу решений `Y`, каждый столбец которой является значением компоненты вектор – функции ***y*** решения в узлах. Значения функций расположены по столбцам матрицы, в первом столбце – значения первой искомой функции, во втором – второй и т. д. Каждая строка матрицы `Y` представляет вектор решения, который отвечает соответствующему значению независимой переменной из вектора `T`. Если выходные параметры не заданы, то MatLab рисует графики найденных решений.

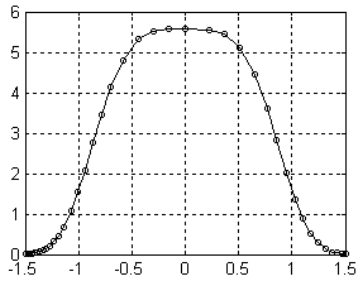
Пример 1. Рассмотрим задачу Коши $y' = -5t^3 \cdot y, \quad y(-1.5) = 0.01.$

Создаем функцию

```
function dydt = odefn1(t,y)
dydt=-5*t.^3.*y;
```

Вызываем солвер и строим график

```
[T,Y]=ode23(@odefn1, [-1.5, 1.5], 0.01);
plot(T,Y,'-ok','MarkerSize',2); grid on
```



Пример 2. Решим задачу Коши $z'' + \frac{1}{5}z' + z = 0,\ z(0) = 0,\ z'(0) = 1.$

Вначале приведем ее к задаче Коши для системы ОДУ 1-го порядка. Обозначим $y_1 = z, y_2 = z'$. Тогда приходим к следующей задаче

$$\frac{dy_1}{dt} = y_2,\ \frac{dy_2}{dt} = -\frac{1}{5}y_2 - y_1,\ y_1(0) = 0,\ y_2(0) = 1$$

Введем вектор – функцию $Y = [y_1, y_2]^T$ и запишем систему в векторном виде $Y' = F(Y), Y(0) = [0,1]^T$, где векторная функция в правой части системы имеет

вид $F(Y)=[y_2,-0.2\cdot y_2-y_1]^T$. В развернутом виде это выглядит следующим образом

$$\begin{pmatrix} y_1' \\ y_2' \end{pmatrix} = F\begin{pmatrix} y_1 \\ y_2 \end{pmatrix}, \text{ где } F\begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = \begin{pmatrix} y_2 \\ -0.2\cdot y_2 - y_1 \end{pmatrix} \text{ и } \begin{pmatrix} y_1(0) \\ y_2(0) \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix},$$

Создадим функцию $F(Y)$ в MatLab. Она обязана содержать первый аргумент – имя независимой переменной, даже если он не используется при формировании функции.

```
function F=odefn2(x,y)
F=[y(2);-0.2*y(2)-y(1)];
```

Для решения полученной системы ОДУ вызываем команду

[T,Y]=ode45('odefn2', [0, 20], [0,1]);

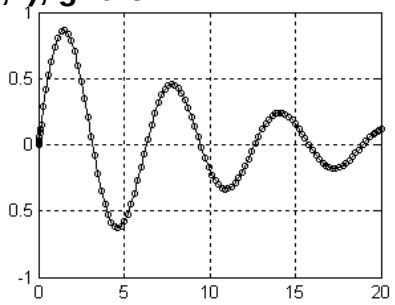
Здесь второй аргумент [0, 20] функции `ode45` определяет диапазон изменения независимого аргумента, а третий [0, 1] - начальные значения вектор – функции. Функция `ode45` возвращает решение в следующем виде

[T Y]

```
ans =
         0         0    1.0000
    0.0001    0.0001    1.0000
    0.0001    0.0001    1.0000
   . . .
   19.7780    0.1025    0.0829
   19.8890    0.1110    0.0694
   20.0000    0.1179    0.0553
```

Первый столбец представляет выбранные значения независимого аргумента `T`, а второй и третий – значения функций y_1 и y_2. Строим график решения (значения искомой функции находятся в первом столбце матрицы Y)

plot(T,Y(:,1),'-ok','MarkerSize',2); grid on



□

Отметим, что солверы `ode23` и `ode45`, использованные нами в примерах 1 и 2, являются наиболее часто используемыми. `ode23` реализует алгоритмы Рунге – Кутта 2 – го и 3 – го порядков, работающие синхронно. Функция автоматически корректирует длину шага, если вычисления обоими методами сильно различаются на каком – либо шаге. `ode23` имеет смысл применять в задачах, в которых требуется получить решение быстро с невысокой степенью точности. Солвер `ode45` основан на формулах Рунге—Кутта четвертого и пятого порядка точности. Он дает лучшие результаты и для большинства задач им стоит воспользоваться в первую очередь. При выборе солвера для решения задачи необходимо учитывать свойства системы дифференциальных уравнений, иначе можно получить неточный результат или затратить слишком много времени на

решение. Все солверы пытаются найти решение с относительной точностью 10^{-3} и абсолютной – 10^{-6}.

Пример 3. Решим систему уравнений

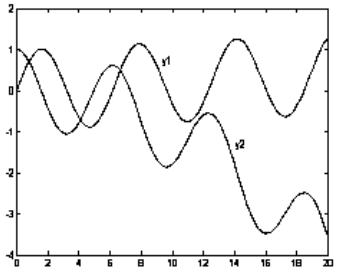
$$\begin{cases} y_1' = y_2 + K x^2 \\ y_2' = -y_1 \end{cases}, \begin{pmatrix} y_1(0) \\ y_2(0) \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

Создаем функцию правой части системы

```
function F=odefn3(x,y)
F=[y(2)+0.01*x.^2;-y(1)];
```

Решаем систему ОДУ и строим график

```
[X Y]=ode45('odefn3',[0,20],[0,1]);
plot(X,Y(:,1)); hold on;
plot(X,Y(:,2));
```



Если вы захотите нанести метки на график, чтобы различить кривые, то можно выполнить команду

```
hold on; gtext('y1'), gtext('y2')
```

и в интерактивном режиме укажите место для отображения первой метки и второй (метки уже показаны на предыдущем рисунке).

Солверы могут найти приближенное решение для заданных значений независимой переменной, если в качестве второго входного аргумента указать вектор с этими значениями, упорядоченными по возрастанию.

```
[X Y]=ode45('odefn3',[0,5,10,15,20],[0,1])
X =
     0
     5
    10
    15
    20
Y =
         0    1.0000
   -0.8396    0.0486
   -0.3342   -1.8015
    0.9356   -2.9757
    1.2954   -3.5830
```

□

Интерфейс солверов допускает обращение к ним с одним выходным аргументом:

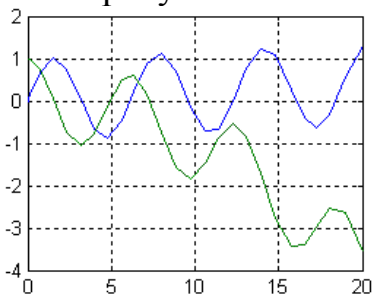
```
sol = ode45(@odefn3,[0 ,20],[0,1]);
```

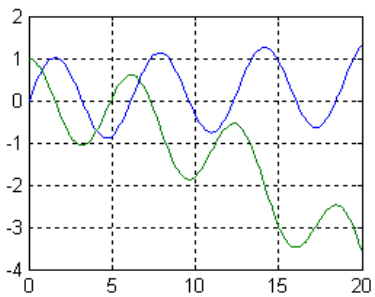
Такой вызов солвера приводит к образованию структуры `sol` с информацией о полученном приближенном решении. Поле `x` структуры `sol` содержит вектор-строку значений независимой переменной, а поле `y` – матрицу значений решения, записанных построчно. Иначе говоря, `sol.x` эквивалентно вектору `T`, а `sol.y` – матрице `Y′` из предыдущего вызова. Тогда, например, для

построения графика решения, полученного в последнем примере, мы должны выполнить команду

plot(sol.x, sol.y(1,:));

График показан на рисунке слева.





Изломы на графике говорят о том, что при таком вызове солвера, узлов, в которых найдено решение, недостаточно. Для определения значений решения в промежуточных точках следует прибегнуть к интерполяции. Это эффективно может выполнить функция `deval`. Ее аргументами является структура `sol` и вектор с координатами независимой переменной, для которых следует вычислить значение. Найденные значения компонент вектор функции построчно записываются в выходном аргументе. Например, в следующем коде в `yval(1,:)` хранятся значения первой искомой функции в точках вектора `xval`, в `yval(2,:)` – второй.

xval= 0:0.1:20;
yval=deval(sol,xval);
plot(xval, yval(1,:), xval, yval(2,:));
grid on;

График показан на предыдущем рисунке справа. Как видим, график решения стал более гладким.

Отметим, что солверы самостоятельно выбирают узлы на заданном отрезке. Кроме того, солвер `ode45` еще сам выполняет интерполяцию, в случае обращения к нему с двумя выходными аргументами.

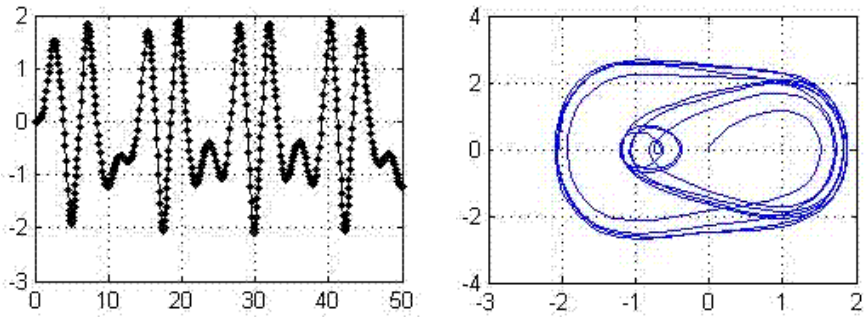
Пример 4. Исследуем решение задачи Коши

$$x'' + x^3 = \sin t,\ x(0) = 0, x'(0) = 0$$

Преобразуем уравнение 2 – го порядка в систему уравнений 1 – го порядка и весь код решения соберем в одну функцию `odefn4`. Она не принимает ни одного аргумента и не возвращает никакого значения, но содержит подфункцию `fun(x,y)`, вычисляющую правую часть системы уравнений.

```
function odefn4
% решение ОДУ 2 - го порядка
Y0=[0; 0];              % вектор начальных условий
[T,Y]=ode45(@fun,[0 50],Y0); % решаем систему
plot(T,Y(:,1),'k.-');    % график решения
grid on;
pause;
plot(Y(:,1),Y(:,2));     % фазовая траектория
function F=fun(x,y)      % подфункция правой части системы
F=[y(2); -y(1)^3+sin(x)];
```

На левом рисунке показан график решения `x(t)`, а на правом – фазовая траектория



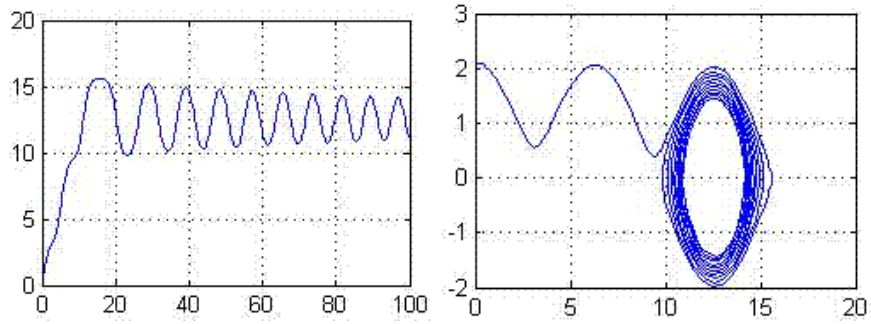
Пример 5. Исследуем решение задачи Коши для системы уравнений

$$x' = y(t),\ y' = -0.01y(t) - \sin x(t),\ x(0) = 0,\ y(0) = 2.1$$

Весь код решения соберем в одну функцию `odefn5` с подфункцией `fun(x,y)`, вычисляющей правую часть системы уравнений.

```
function odefn5
% решение ОДУ 2 - го порядка
Y0=[0; 2.1];                % вектор начальных условий
[T,Y]=ode45(@fun,[0 100],Y0); % решаем систему
plot(T,Y(:,1));
grid on;
pause;
plot(Y(:,1),Y(:,2));
function F=fun(x,y)         % подфункция правой части системы
F=[y(2); -0.01*y(2)-sin(y(1))];
```

На левом рисунке показан график решения `x(t)`, а на правом – фазовая траектория



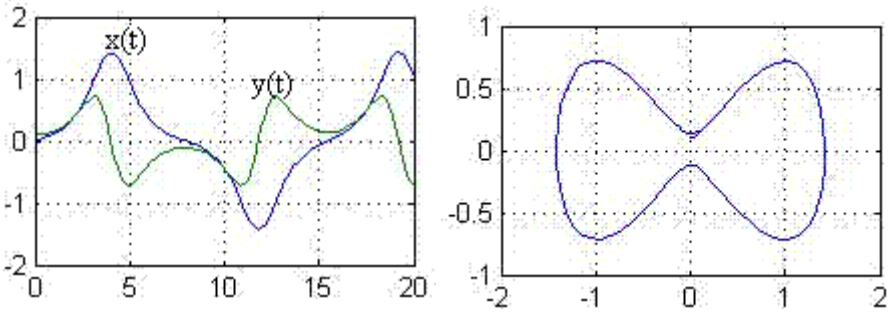
Пример 6. Исследуем решение задачи Коши для системы уравнений

$$x' = y(t),\ y' = -x^3(t) + x(t),\ x(0) = 0,\ y(0) = 0.1$$

Весь код решения соберем в одну функцию

```
function odefn6
% решение системы уравнений
Y0=[0; 0.1];                % вектор начальных условий
[T,Y]=ode45(@fun,[0 16],Y0);   % решаем систему
plot(T,Y(:,1)); grid on;
pause;
plot(Y(:,1),Y(:,2));    % фазовая траектория
function F=fun(x,y)          % подфункция правой части системы
F=[y(2); -y(1)^3+y(1)];
```

На левом рисунке показаны графики функций `x(t)`, `y(t)`, а на правом – фазовая траектория.

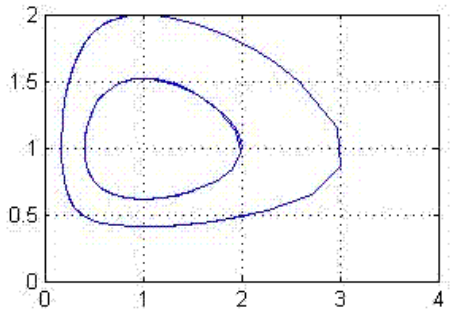


Пример 7. Рассмотрим двухвидовую модель «хищник – жертва», впервые построенную Вольтерра для объяснения колебаний рыбных уловов. Имеются два биологических вида, численностью в момент времени `t`, соответственно, `x(t)` и `y(t)`. Особи первого вида являются пищей для особей второго вида (хищников). Численности популяций в начальный момент времени известны. Требуется определить численность видов в произвольный момент времени. Математической моделью задачи является система дифференциальных уравнений Лотки – Вольтерра

$$\begin{cases} \dfrac{dx}{dt} = (a - b\,y)x \\ \dfrac{dy}{dt} = (-c + d\,x)y \end{cases}$$

где `a, b, c, d` – положительные константы. Проведем расчет численности популяций, если $a = 3, b = 3, c = 1, d = 1$, для двух вариантов начальных условий $x(0) = 2,\ y(0) = 1$ и $x(0) = 1,\ y(0) = 2$, для которых построим фазовые траектории. Весь код решения соберем в одну функцию

```
function odeLotVolt
% решение системы уравнений Лотки - Вольтерра
Y0=[2; 1];              % вектор начальных условий
[T,Y]=ode45(@fun,[0 7],Y0); % решаем систему
plot(Y(:,1),Y(:,2));    % фазовая траектория
grid on; hold on;
Y0=[1; 2];              % вектор начальных условий
[T,Y]=ode45(@fun,[0 7],Y0); % решаем систему
plot(Y(:,1),Y(:,2));    % фазовая траектория
function F=fun(x,y)        % подфункция правой части системы
F=[3*y(1).*(1-y(2)); y(2).*(y(1)-1)];
```



Из этого рисунка видно, что численность популяций меняется периодически.

Пример 8. Решим систему дифференциальных уравнений

$$\frac{dx}{dt} = y,\ \frac{dy}{dt} = -y - x\cdot(z-1) - x^3,\ \frac{dz}{dt} = x\cdot y - z$$

с начальными условиями $x(0)=1, y(0)=1, z(0)=0$ и построим ее фазовый портрет. Весь код решения соберем в одну функцию `odefn8.m`

```
function odefn8
% решение системы ОДУ
Y0=[1; 1; 0];                 % вектор начальных условий
[T,Y]=ode45(@fun,[0 25],Y0); % решаем систему
plot3(Y(:,1),Y(:,2),Y(:,3),'LineWidth',2);  % 3D фазовая траектория
grid on;
function F=fun(x,y)         % подфункция правой части системы
F=[y(2); 0.1*y(2)-y(1).*(y(3)-1)-y(1).^3;  y(1).*y(2)-0.1*y(3)];
```

Фазовая траектория показана на следующем рисунке

Пример 9. Исследуем поведения математического маятника. Пусть масса груза равна единице, а стержень, на котором подвешена масса, невесом. Тогда дифференциальное уравнение движения груза имеет вид

$$\varphi'' + k\varphi' + \omega^2 \sin\varphi = 0$$

где $\varphi(t)$ угол отклонения маятника от положения равновесия (нижнее положение), параметр k характеризует величину трения, $\omega^2 = g/l$ (g ускорение свободного падения, l – длина маятника). Для определения конкретного движения к уравнению движения надо добавить начальные условия $\varphi(0)=\varphi_0$, $\varphi'(0)=\varphi_0'$.

Преобразуем уравнение к системе ОДУ 1 – го порядка. Если обозначить $u \equiv \varphi,\ v \equiv \varphi'$, то получим

$$\begin{cases} u' = v \\ v' = -kv - \omega^2 \sin(u) \end{cases}, \quad u(0)=\varphi_0,\ v(0)=\varphi_0'$$

Выберем следующие значения параметров $k=0.5$, $\omega^2 = 10$ и начальные значения $\varphi_0 = 0, \varphi_0' = 5$.

Создаем функцию

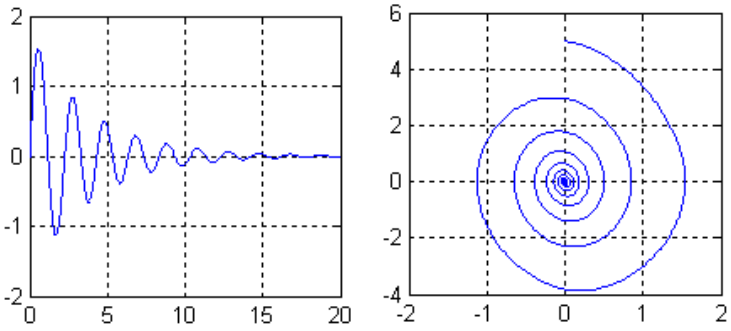
```
pend=@(t,y) [y(2); -0.5*y(2)-10*sin(y(1))];
```

Решаем систему и строим график (следующий рисунок слева)

```
[T,Y]=ode45(pend,[0:0.01:20],[0 5]);
plot(T,Y(:,1)); grid on;
```

Строим фазовую траекторию (рисунок справа)

```
plot(Y(:, 1),Y(:,2)); grid on;
```



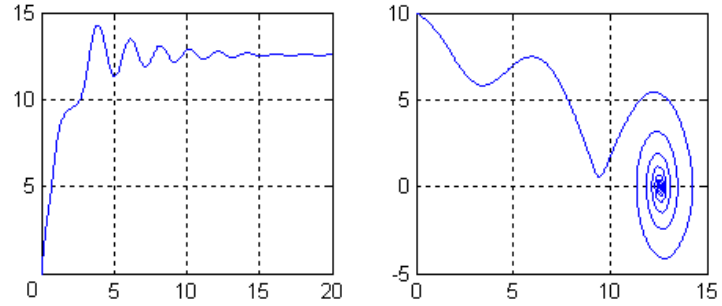
Как видно из левого графика максимальный угол отклонения маятника не превышают $\pi/2$ и колебания маятника затухают.

Увеличим начальную скорость до 10. Решаем задачу и строим график решения (следующий рисунок слева)

```
[T,Y]=ode45(pend,[0:0.01:20],[0 10]);
plot(T,Y(:,1)); grid on;
```

Строим фазовую траекторию (след. рис. справа)

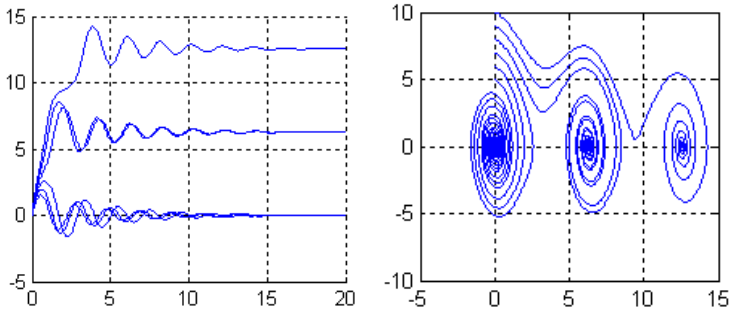
```
plot(Y(:, 1),Y(:,2)); grid on;
```



Максимальное значение угла составляет примерно 14 радиан. Маятник сделал два полных оборота вокруг точки закрепления (угол отклонения увеличился на 4π), а затем колебания затухают в окрестности значения 4π радиан (для маятника угол поворота 4π представляет то же, что и 0 радиан, т.е. положение равновесия).

Построим несколько графиков угла отклонения (след. рис. слева) и фазовых траекторий (след. рис. справа), задавая различную начальную скорость.

```
clf; hold on;    % графики угла отклонения
for v=5:10
    [T,Y]=ode45(pend,[0:0.01:20],[0 v]);
    plot(T,Y(:,1)); grid on;
end;
pause;          % ждет нажатия любой клавиши
clf; hold on;   % фазовые траектории
for v=5:10
    [T,Y]=ode45(pend,[0:0.01:20],[0 v]);
    plot(Y(:, 1),Y(:,2)); grid on;
end;
```



Как видим, начальная скорость при *v=5, 6, 7* недостаточна, чтобы маятник прошел верхнюю точку и сделал хотя бы один полный оборот. При начальной скорости *v= 8, 9* маятник совершает один полный оборот, а затем его колебания затухаю. При *v=10* маятник смог выполнить два полных оборота и только после этого его колебания стали затухать вокруг положения равновесия.

Пример 10. Решим ОДУ 3 – го порядка

$$z''' - x^2 z'' + x z' - z = 0$$

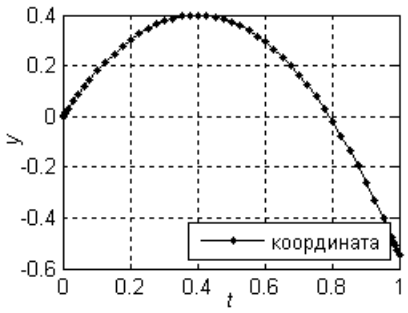
с начальными условиями $z(0)=0; z'(0)=2; z''(0)=-5$. Соответствующая задача Коши для системы ОДУ 1 – го порядка имеет вид

$$\begin{cases} u' = v \\ v' = w \\ w' = x^2 w - xv + u \end{cases}, \quad u(0)=0; v(0)=2; w(0)=-5$$

Весь процесс решения и визуализации реализуем в следующей m – функции без аргументов

```
function odeo3
% решение ОДУ 3 - го порядка
Y0=[0; 2; -5];                % вектор начальных условий
[T,Y]=ode45(@fun,[0 1],Y0); % решаем систему
plot(T,Y(:,1),'k.-');        % строим график решения
grid on;
xlabel('\itt');
ylabel('\ity');
legend('координата',4);
function F=fun(x,y)          % подфункция правой части системы
F=[y(2); y(3); x.^2.*y(3)-x.*y(2)+y(1)];
```

Вызывая эту функцию, получаем



Здесь функцию правой части системы мы оформили в виде подфункции.

Пример 11. В справочной системе MatLab приводится пример решения уравнения Ван-дер-Поля $z'' + z - K \cdot (1 - z^2) \cdot z' = 0$, решение которого обычными солверами дает неудовлетворительный результат. Запишем уравнение в виде системы

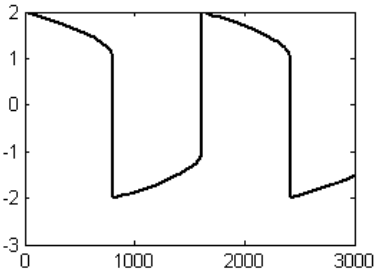
$$\begin{cases} y_1' = y_2 \\ y_2' = -y_1 + K \cdot (1 - y_1^2) \cdot y_2 \end{cases}, \quad \begin{pmatrix} y_1(0) \\ y_2(0) \end{pmatrix} = \begin{pmatrix} 2 \\ 0 \end{pmatrix}$$

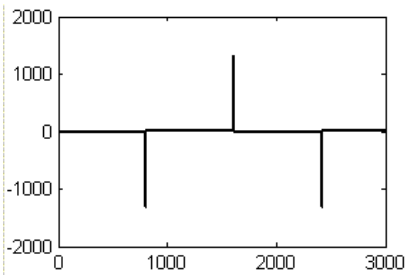
Создадим функцию правой части (полагаем `K=1000`)

```
function F=VanDerPol(x,y)
F=[y(2);-y(1)+1000*(1-y(1).^2).*y(2)];
```

Решаем систему с помощью солвера для жестких систем и строим график.

```
[X Y]=ode15s('VanDerPol',[0,3000],[2,0]);
plot(X,Y(:,1));
pause;
plot(X,Y(:,2));
```





Попытайтесь решить эту задачу с использованием солверов `ode23`, `ode45` или `ode113`. Если у вас не хватит терпения дождаться окончания расчета, то прервите вычисления комбинацией клавиш `Ctrl - Break`. Дело в том, что эта задача является примером так называемых жестких систем, для решения которых в MatLab имеются специальные солверы.

□

Если все попытки применения солверов `ode45`, `ode23`, `ode113` не приводят к успеху, то возможно, что решаемая система является жесткой. Для решения жестких систем подходит солвер `ode15s`, основанный на многошаговом методе Гира. Если требуется решить жесткую задачу с невысокой точностью, то хороший результат может дать солвер `ode23s`, реализующий одношаговый метод Розенброка второго порядка.

Итак, *при решении в MATLAB дифференциальных уравнений и систем с начальными условиями следует правильно выбирать солвер*.

Исходная задача может быть сама системой ОДУ. Но перед решением в MatLab ее следует преобразовать в систему ОДУ 1-го порядка.

Пример 12. Исследуем задачу о движении планеты вокруг Солнца под действием тяготения. Она записывается в виде системы ОДУ 2-го порядка

$$\ddot{x} = -\frac{k\,x}{\left(x^2 + y^2\right)^{3/2}}, \quad \ddot{y} = -\frac{k\,y}{\left(x^2 + y^2\right)^{3/2}},$$

где $x(t), y(t)$ - координаты движущейся планеты. Преобразуем исходную систему к системе ОДУ 1-го порядка. Обозначим $z_1 = x, z_2 = \dot{x}, z_3 = y, z_4 = \dot{y}$. Тогда

$$\begin{cases} z_1' = z_2 \\ z_2' = -\dfrac{k\, z_1}{\left(z_1^2 + z_3^2\right)^{3/2}} \\ z_3' = z_4 \\ z_4' = -\dfrac{k\, z_3}{\left(z_1^2 + z_3^2\right)^{3/2}} \end{cases}$$

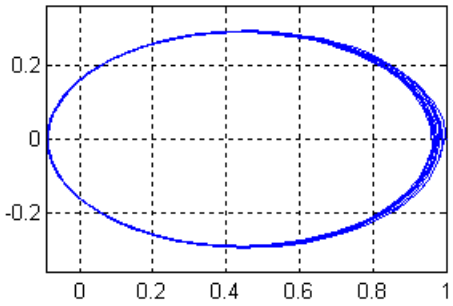
Для модельной задачи выберем *k=1* и зададим начальные условия

$$z_1(0)=1,\ z_2(0)=0,\ z_3(0)=0,\ z_4(0)=1$$

Решение удобно оформить в виде одной функции с подфункцией для вычисления правой части системы

```
function planet
% решение системы
Y0=[1; 0; 0; 0.4];            % вектор начальных условий
[T,Y]=ode45(@fun,[0 20],Y0); % решаем систему
plot(Y(:,1),Y(:,3));          % график траектории движения
grid on;
axis equal;
pause;
comet(Y(:,1),Y(:,3));  % анимация движения точки
function F=fun(x,y)    % подфункция правой части системы
F=[y(2);-y(1)/(y(1).^2+y(3).^2).^(3/2); ...
    y(4);-y(3)/(y(1).^2+y(3).^2).^(3/2)];
```

На следующем рисунке показана получаемая траектория движения планеты (кривая с параметрическим уравнением $x(t), y(t)$)

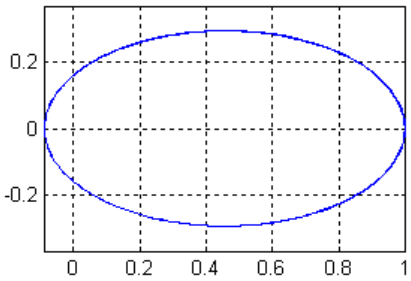


Для того, чтобы следить за движением точки по траектории, мы использовали функцию `comet`. Она позволяет получить анимированный график, на котором кружок, обозначающий точку, перемещается на плоскости, оставляя за собой след в виде линии – траектории движения. Следите за тем, чтобы окно с графиком было поверх остальных окон.

Известно, что траектория должна быть замкнутой кривой, а $x(t), y(t)$ должны быть периодическими функциями. Однако мы видим некоторое отклонение от замкнутости траектории. Это объясняется погрешностью вычислений. Повысить точность вычислений можно следующими командами

```
options=odeset('AbsTol',0.00000001, 'RelTol', 0.00001);
[T,Y]=ode45(@fun,[0 20],Y0, options); % решаем систему
```

(в коде функции `planet` замените строку с `ode45` на эти две строки)



Как видим, при повышении абсолютной и относительной точности вычислений траектория выглядит замкнутой.

□

Здесь мы вызвали солвер `ode45` с четырьмя аргументами. Формат вызова с четвертым аргументами для всех солверов одинаков

`[T, Y] = solver(odefun, ,[t0, tend], y0, opts)`

Он представляет структуру данных, управляющих ходом вычислительного процесса. Поля этой структуры следует заполнять заранее с помощью функции `odeset`.

`opts=odeset('name1',value1, 'name2', value2,...)`

Она создает новую структура `opts`, в которой свойства с указанными именами `name1, name2, …` принимают следующие за ними значения `value1,value2`. В формате

`opts=odeset(oldopts, 'name1',value1,'name2',value2,…)`

мы меняем в существующей структуре параметров `oldopts` соответствующие значения.

Можно управлять следующими параметрами солверов: точностью вычислений (параметры `RelTol, AbsTol, NormControl`), шагом интегрирования (параметры `InitialStep, MaxStep`), выходными данными (параметры `OutputFcn, OutputSel, Refine, Stats`), якобианом (параметры `Jacobian, JPattern, Vectorized`), матрицей масс и матрицей системы ОДУ (параметры `Mass, MStateDependence, MvPattern, MassSingular, InitialSlope`), событиями (параметр `Events`), два параметра только для `ode15s` (`MaxOrder, BDF`). С подробным предназначением параметров солверов можно познакомиться на странице справки функции `odeset`. Примеры использования основных параметров будут приведены ниже.

Функция `odeset` без параметров возвращает все имена свойств и их допустимые значения

odeset

```
          AbsTol: [ positive scalar or vector {1e-6} ]
          RelTol: [ positive scalar {1e-3} ]
     NormControl: [ on | {off} ]
       OutputFcn: [ function ]
       OutputSel: [ vector of integers ]
          Refine: [ positive integer ]
           Stats: [ on | {off} ]
     InitialStep: [ positive scalar ]
         MaxStep: [ positive scalar ]
             BDF: [ on | {off} ]
        MaxOrder: [ 1 | 2 | 3 | 4 | {5} ]
```

```
        Jacobian: [ matrix | function ]
        JPattern: [ sparse matrix ]
      Vectorized: [ on | {off} ]
            Mass: [ matrix | function ]
MStateDependence: [ none | weak | strong ]
       MvPattern: [ sparse matrix ]
    MassSingular: [ yes | no | {maybe} ]
    InitialSlope: [ vector ]
          Events: [ function ]
```

Чаще всего приходится управлять точностью вычислений. Есть два способа контроля точности в зависимости от значения параметра `NormControl` : по локальной погрешности ε_i i-ой компоненты вектора решений y_i (`NormControl=off`) и по евклидовой норме погрешности (`NormControl=on`). В первом случае точность считается достигнутой, если для каждой компоненты y_k вектора решения $\mathbf{y}=(y_1,...,y_n)^T$ системы на каждом шаге t_k выполняется условие $\varepsilon_i(t_k)<\max(RelTol\cdot|y_i(t_k)|, AbsTol(i))$. При этом каждая компонента вектора решений может иметь собственную абсолютную точность `AbsTol=[`a_1`, ..., `a_n`]`. Во втором случае (`NormControl=on`) точность считается достигнутой, если на каждом шаге вычислений t_k выполняется условие $\|\varepsilon\|\le\max(RelTol\cdot\|y\|, AbsTol)$, где евклидова норма $\|\ \|$ определяется формулой $\|y\|=\sqrt{\sum_{i=1}^{n}y_i^2}$ и абсолютная точность вычислений `AbsTol` должна быть скаляром. В MatLab приняты следующие значения этих параметров по умолчанию: `NormControl=off`, $AbsTol=10^{-6}$, $RelTol=10^{-3}$.

Шаг интегрирования солвера определяется двумя свойствами:

- `MaxStep` – задает максимальный шаг (по умолчанию десятая часть промежутка интегрирования);
- `InitialStep` – начальный шаг и, если он не задан, то выбирается солвером самостоятельно;

Рассмотрим пример, где установки точности вычислений по умолчанию требуют изменения.

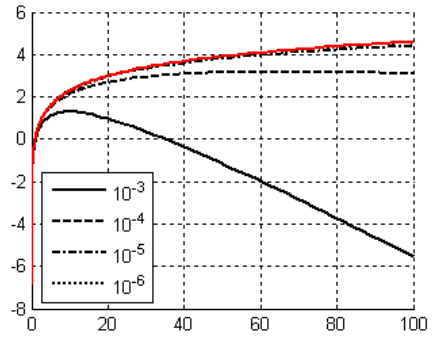
Пример 13. Решим дифференциальное уравнение $y''=-\frac{1}{t^2}$ на отрезке $[a;100]$ с начальными условиями $y(a)=\ln(a)$, $y'(a)=1/a$ при a=0.001. Его точное решение $y=\ln t$.

Приведем задачу к системе ОДУ первого порядка и создадим функцию `example13` для ее решения, в которой строим графики приближенного решения с относительной точностью 10^{-3}, 10^{-4}, 10^{-5}, 10^{-6} и график точного решения

```
function example13
a=0.001;
Y0=[log(a); 1/a];                    % вектор начальных условий
stl=str2mat('k-','k--','k-.','k:');
newplot; hold on; grid on;
for i=3:6
    rt=10^(-i);
    opts = odeset('RelTol', rt);     % задаем относительную точность
```

```
    [T,Y]=ode45(@ex7,[a 100],Y0,opts); % находим приближенное решение
    plot(T,Y(:,1),stl(i-2,:), 'LineWidth',2);
end
Z=log(T);
plot(T,Z,'r', 'LineWidth',2);           % добавляем кривую точного решения
legend('10^{-3}','10^{-4}','10^{-5}','10^{-6}','Location','SouthWest');
hold off;
% функция правой части системы
function F=ex7(x,y)
F=[y(2);-1./x.^2];
```

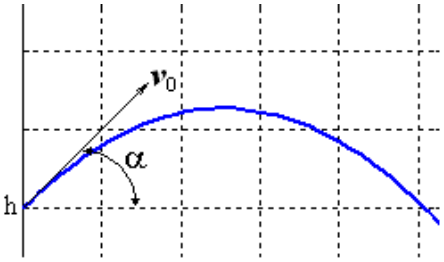
На следующем рисунке красной верхней линией показан график точного решения и графики приближенного решения при разной относительной точности. Как видим, относительной точности по умолчанию 10^{-3} для данной задачи явно недостаточно (сплошная черная нижняя кривая). Хорошее приближение к точному решению получилось только при `RelTol` = 10^{-6}.



Пример 14. Решим задачу Коши, описывающую движение тела, брошенного с начальной скоростью v_0 под углом α к горизонту в предположении, что сопротивление воздуха пропорционально квадрату скорости. В векторной форме уравнение движения имеет вид

$$m\ddot{\mathbf{r}} = -\gamma \cdot \mathbf{v}\left|\ \mathbf{v}\ \right| - m\mathbf{g},$$

где $\mathbf{r}(t)$ радиус – вектор движущегося тела, $\mathbf{v} = \dot{\mathbf{r}}(t)$ – вектор скорости тела, γ – коэффициент сопротивления, $m\mathbf{g}$ вектор силы тяжести тела массы m, g – ускорение свободного падения.



Особенность этой задачи состоит в том, что движение заканчивается в заранее неизвестный момент времени, когда тело падает на землю.

Если обозначить $k = \gamma / m$, то в координатной форме мы имеем систему уравнений

$$\ddot{x} = -k\,\dot{x}\sqrt{\dot{x}^2 + \dot{y}^2}$$

$$\ddot{y} = -k\,\dot{y}\sqrt{\dot{x}^2 + \dot{y}^2} - g$$

к которой следует добавить начальные условия: $x(0) = 0$, $y(0) = h$ (h начальная высота), $\dot{x}(0) = v_0 \cos\alpha$, $\dot{y}(0) = v_0 \sin\alpha$.

Положим $y(1)=x,\ y(2)=\dot{x},\ y(3)=y,\ y(4)=\dot{y}$. Тогда соответствующая система ОДУ 1 – го порядка примет вид

$$\begin{cases} y'(1)=y(2) \\ y'(2)=-k\,y(2)\sqrt{(y(2))^2+(y(4))^2} \\ y'(3)=y(4) \\ y'(4)=-k\,y(4)\sqrt{(y(2))^2+(y(4))^2}-g \end{cases}$$

Функция `bodyangle.m` для вычисления правой части системы ОДУ имеет вид

```
function F=bodyangle(x,y)
k=0.01;
g=9.81;
F=[y(2); -k.*y(2).*sqrt(y(2).^2+y(4).^2);...
   y(4); -k.*y(4).*sqrt(y(2).^2+y(4).^2)-g];
```

Сценарий решения `example14_1.m` нашей краевой задачи может иметь вид

```
% движение тела брошенного под углом к горизонту
alph=pi/4;      % угол бросания тела
v0=1;           % начальная скорость
h=0;            % начальная высота
tmax=0.2;      % интервал времени
Y0=[0; v0.*cos(alph); h; v0.*sin(alph)];    % вектор начальных условий
[T,Y]=ode45(@bodyangle,[0 tmax],Y0);        % приближенное решение
plot(Y(:,1),Y(:,3), 'LineWidth',2);         % график кривой x(t), y(t)
axis equal; grid on;
```

Однако, для определения длительности полета нам приходится подбирать значение `tmax`. Также приходится «на глазок» определять максимальную высоту и дальность полета. В `ode` солверах MalLab предусмотрена возможность определения моментов наступления событий, соответствующих некоторым особым значениям решения и реакция на них. Для этого используется специальная функция – обработчик события. В процессе решения MatLab выявляет события и вызывает пользовательский обработчик.

Формат функции обработчика события должен быть следующим:

`[value, isterminal, direction] = eventsfun(t, y)`

Функция должна возвращать три вектора `value, isterminal, direction` одинаковой длины, в которых *i* – я компонента соответствует обращению в ноль некоторого выражения, зависящего от `t` и компонентов `y(k)` вектор – функции решения системы ОДУ (аргументов `t,y` функции `events`). Компоненты этих векторов имеют следующий смысл:

- `value(i)` – выражение, составленное из аргумента `t` и компонент `y(k)` вектор – функции `y` решения системы, которое внутри солвера будет проверяться на обращение в ноль;
- `isterminal(i) = 1`, если интегрирование системы ОДУ следует остановить при выполнении условия `value(i)` = 0, или = 0, если останавливать вычисления не требуется;
- `direction(i) = 0`, если следует «отлавливать» все нули выражения `value(i)`, + 1 – если следует реагировать на нули при прохождении

которых выражение `value(i)` возрастает, и `–1` – если следует реагировать на нули при прохождении которых `value(i)` убывает.

Затем дескриптор этой функции следует передать солверу, указав его в структуре параметров `options` функции `odeset` (см. выше) в качестве значения параметра `Events`.

```
options=odeset('Events', eventsfun)
```

После этого солвер надо вызвать с 5 – ю выходными параметрами

```
[T,Y,TE,YE,IE] = solver(odefun, tspan, y0, options)
```

Смысл дополнительных параметров будет описан чуть ниже.

Для нашего примера функция `eventsfun` должна реагировать на событие обращения в ноль координаты *y* тела (компоненты решения `y(3)`) при ее убывании и обращения в ноль производной y' (компоненты решения `y(4)`). Из физических соображений ясно, что производная y' обращается в ноль только в одной точке – в самой верхней точке траектории тела, поэтому нам неважно убывает или возрастает y' в этой точке.

Создадим следующую функцию `ex8events.m` обработчик события

```
function [value,isterminal,direction] = ex8events(t,y)
% Определять момент времени, в который высота тела убывает и становится
% равной 0, и завершать вычисления, а также
% определить момент максимальной высоты (не останавливая вычислений),
% она достигается когда скорость по y равна 0
value = [y(3), y(4)]; % определять нули для компонент y(3) и y(4)
isterminal = [1,0];   % останавливать вычисления при y(3)=0
direction = [-1,0];   % функция y(3) убывает, для y(4) любое направление
```

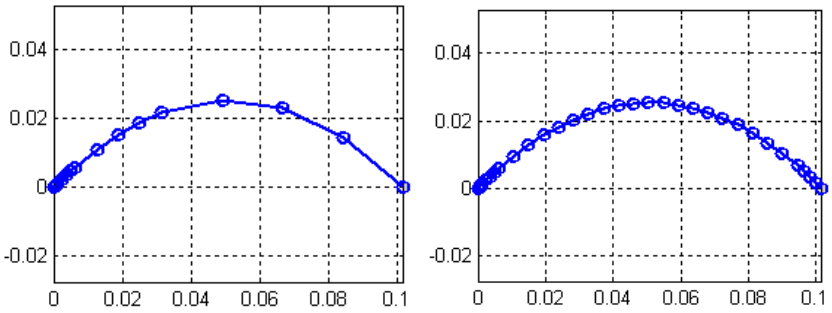
Посмотрим, как использовать такую функцию. Для этого скорректируем файл сценария следующим образом (сценарий `example14_2.m`)

```
% движение тела брошенного под углом к горизонту
alph=pi/4;      % угол бросания тела
v0=1;           % начальная скорость
h=0;            % начальная высота
Y0=[0; v0.*cos(alph); h; v0.*sin(alph)];% вектор начальных условий
options = odeset('Events',@ex8events);  % сообщаем функцию обработчик событий
[t,Y,te,ye,ie] = ode45(@bodyangle,[0 Inf],Y0,options); % приближенное решение
plot(Y(:,1),Y(:,3), '-bo', 'LineWidth',2);          % график кривой x(t),y(t)
axis equal; grid on;
```

На левом графике показана получаемая траектория. Вычисления закончились тогда, когда компонента решения `y(3)`, убывая, стала равной нулю. Видно, что в конце траектории шаг вычисления (интервал моментов времени) достаточно велик и не обеспечивает гладкость траектории. В таком случае можно использовать параметр `MaxStep`, который, как указано выше, задает максимальный шаг. Заменим строку сценария, содержащую вызов функции `odeset` следующей строкой

```
opts = odeset('Events',@ex8events,'MaxStep',0.025); % сообщаем функцию
                        % обработчик событий
```

На графике справа показана траектория, построенная по модифицированному сценарию. Гладкость кривой стала выше.



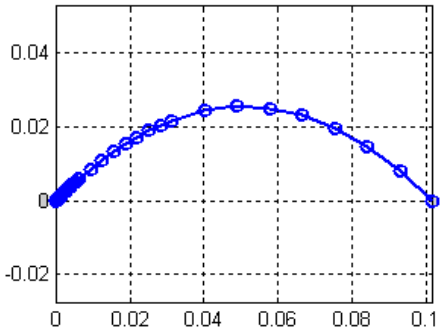
Однако уменьшение шага посредством параметра `MaxStep` увеличивает количество шагов, используемых численным методом, и увеличивает время решения системы ОДУ. Есть еще один параметр `Refine`, управляющий количеством точек решения, возвращаемых солвером на каждом интервале. Если `Refine` равен 1, то солвер вычисляет значение решения только в конечных точках интервалов времени, используемых численным алгоритмом. Если `Refine` равен `n > 1`, то солвер делит каждый временной отрезок на `n` подинтервалов и возвращает решение в каждой точке деления. При этом используются внутренние формулы для вычисления решения в точках временного отрезка, а не применяется алгоритм численного метода на мелком разбиении. Это экономит время вычисления. Все солверы, кроме `ode45` по умолчанию используют значение `Refine` равное 1, а `ode45` по умолчанию использует значение 4.

Параметр `Refine` не работает, когда вы явно задаете моменты времени `tspan`, в которые следует получить решение, т.е. когда вместо аргумента `tspan=[t0, tend]` задается вектор длины $length(tspan) > 2$.

Замените в сценарии `example14_2.m` строку, задающую параметры солвера, на следующую строку

```
opts = odeset('Events',@ex8events,'Refine',8); % сообщаем функцию
                    % обработчик событий
```

и выполните его.



Как видим, временные интервалы не стали равными, однако гладкость кривой повысилась.

Обратите также внимание на то, что мы использовали солвер с 5 – ю выходными аргументами. Если солверу в четвертом аргументе `options` передается параметр `Events`, то он (солвер) возвращает пять параметров, которые можно принять следующим образом

```
[T,Y,TE,YE,IE] = solver(odefun, tspan, y0, options)
```

где дополнительные возвращаемые параметры имеют следующий смысл:

– TE – вектор столбец моментов времени, в которые происходят события;
– YE – значения решения в эти моменты времени;
– IE – индексы k компонент выражений value, для которых произошли указанные события.

Так в нашем примере мы получаем

```
te % моменты наступления события
te =
    0.0721
    0.1441
ye % значения решения в моменты наступления событий
ye =
    0.0509    0.7067    0.0255         0
    0.1019    0.7063   -0.0000   -0.7068
ie % Индексы в векторе [y(3),y(4)], кот. возвращает Events функция
ie =
     2
     1
```

В нашем примере функция ex8events обработчик события Events возвращает вектор [y(3),y(4)]. Поэтому возвращаемый вектор ie=[2; 1] означает, что вначале наступило событие обращения в ноль функции y(4) (вторая компонента возвращаемого функцией ex8events вектора), а потом – y(3) (первая компонента). Эти события произошли в моменты времени te=[0.0721; 0.1441], а значения вектор – функции $[y(1), y(2), y(3), y(4)]$ решения в эти моменты находятся в матрице ye. В частности самая высокая точка траектории имеет координаты [0.0509 0.0255], длительность полета составила $t_{\max} = 0.1441$ (вторая компонента вектора te), а дальность полета $x_{\max} = 0.1019$ (первая компонента во второй строке матрицы ye). □

Если солвер вызывается с одним возвращаемым аргументом в формате

```
sol = solver(odefun, [t0, t1], y0, opts)
```

то структура sol будет иметь дополнительные поля sol.xe, sol.ye, и sol.ie, соответствующие возвращаемым параметрам TE, YE, IE, описанным выше. Например, если в нашем примере для решения использовать команду

```
sol = ode45(@bodyangle,[0 Inf],Y0,opts);  % приближенное решение
```

то будем иметь

```
sol.xe
ans =
    0.0721    0.1441
sol.ye
ans =
    0.0509    0.1019
    0.7067    0.7063
    0.0255   -0.0000
         0   -0.7068
sol.ie
ans =
     2     1
```

Продемонстрируем возможности использования параметра Events на примере решения задачи о прыгающем мячике.

Пример 15. Упругий мячик имеет начальное положение и скорость. Сопротивление воздуха пренебрежимо мало, но энергия движения расходуется при отскоке мяча от земли. Пусть при отскоке от земли вертикальная скорость мячика составляет 90% от вертикальной скорости в момент падения.

Уравнение движения (без учета сопротивления воздуха) имеет вид

$$m\ddot{\mathbf{r}} = -m\mathbf{g},$$

где $\mathbf{g}$ – вектор ускорения свободного падения, имеющий направление вертикально вниз. В покоординатной форме имеем $\ddot{x} = 0, \ddot{y} = -g$ и начальные условия $x(0) = x_0, y(0) = h, \dot{x}(0) = v_0^x, \dot{y}(0) = v_0^y$. Система распадается на два независимых уравнения, первое из которых имеет решение $x(t) = x_0 + v_0^x \cdot t$. Это указывает на то, что в горизонтальном направлении движение происходит с постоянной скоростью v_0^x. В нашем примере положим $x_0 = 0$. Второе уравнение также интегрируется, но мы хотим показать, как можно использовать параметр `Events`, и будем решать уравнение численно. В момент отскока t_k вертикальная скорость $\dot{y}$ меняет знак, т.е. $\dot{y}_{\text{непосредственно после отскока}}(t_k) = -0.9 \cdot \dot{y}_{\text{непосредственно до отскока}}(t_k)$ и становится положительной.

Функция правой части системы ОДУ 1 – го порядка имеет вид

```
function F=fun9(x,y)
g=9.81;
F=[y(2); -g];
```

Функция обработчик события имеет вид

```
function [value,isterminal,direction] = events9(t,y)
% Определять момент времени в который высота тела становится
% равной 0 при ее убывании и завершать вычисления
value = y(1);           % определять нулевую высоту
isterminal = 1;         % останавливать вычисления при y(1)=0
direction = -1;         % высота y(1) убывает
```

Сценарий для моделирования движения до первого приземления мячика на землю можно построить следующим образом

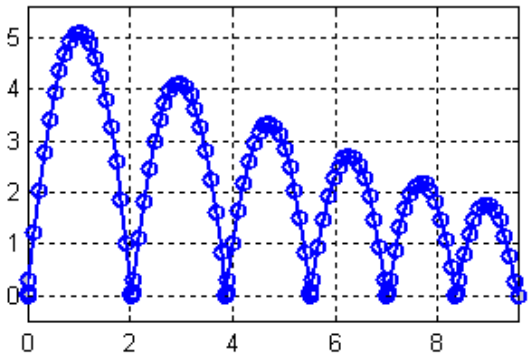
```
% сценария example15_1.m для моделирования движения одного прыжка мячика
h=0;       % начальная высота
vox=1;      % начальная скорость вдоль оси x
voy=10;     % начальная скорость вдоль оси y
Y0=[h; voy];    % вектор начальных условий
opts = odeset('Events',@events9,'MaxStep',0.1);  % задаем параметры
[t,Y,te,ye,ie] = ode45(@fun9,[0 Inf],Y0,opts);   % y координата мячика
X=vox.*t;                                        % x - координата мячика
plot(X,Y(:,1), '-bo', 'LineWidth',2);         % график кривой x(t), y(t)
axis equal; grid on;
```

Мы хотим смоделировать движение при многократном отскоке и приземлении мячика. Для этого исправим сценарий следующим образом

```
% сценария example15_2.m для моделирования движения прыгающего мячика
vox=1;      % начальная скорость вдоль оси x
voy=10;     % начальная скорость вдоль оси y
tstart=0;
tfinal=Inf;
Y0=[0; voy];    % вектор начальных условий
opts = odeset('Events',@events9,'MaxStep',0.1);  % задаем параметры
for i=1:6
    [t,Y,te,ye,ie]=ode23(@fun9,[tstart tfinal],Y0,opts); %y координата мячика
    X=vox.*t;
```

```
    plot(X,Y(:,1), '-bo', 'LineWidth',2);          % график кривой x(t), y(t)
    hold on;
    tstart=te;
    Y0=[0; 0.9.*abs(ye(2))];
end
axis equal; grid on;
hold off;
```

Полученный график траектории движения при векторе начальной скорости $\mathbf{v}(0)=(v_0^x, v_0^y)=(1,10)$ показан на следующем рисунке



Если мы хотим строить кривые при различных значениях начальной скорости, то вместо сценария удобно использовать функцию с аргументами, задающими начальные значения

```
function funex15(vox,voy)
% моделирование движения прыгающего мячика
% vox - начальная скорость вдоль оси x
% voy - начальная скорость вдоль оси y
opts = odeset('Events',@events9,'MaxStep',0.01);  % задаем параметры
tfin=[];    % единый вектор моментов времени для построения анимации
yfin=[];    % единый вектор координат y  для построения анимации
tstart=0;
tfinal=Inf;
Y0=[0; voy];                    % вектор начальных условий
for i=1:6
    [t,Y,te,ye,ie] = ode23(@fun9,[tstart tfinal],Y0,opts); % решение
    X=vox.*t;
    plot(X,Y(:,1),  'LineWidth',2);     % график участка кривой
    hold on;
    tstart=te;                  % начальный момент для следующего участка
    Y0=[0; 0.9.*abs(ye(2))];    % начальные условия для следующего участка
    tfin=[tfin t'];             % объединяем моменты времени
    yfin=[yfin Y(:,1)'];        % объединяем координаты y
end
axis equal; grid on; hold off;

pause;
X=vox.*tfin;
comet(X,yfin);                  % анимация движения мяча
```

Здесь мы еще дополнили код строками для построения анимации движения мячика с помощью функции `comet`. □

В такую функцию можно было бы включить код функций правой части системы `fun9(x,y)` и обработчика событий `events9(t,y)` в качестве подфункций. Тогда подфункции имели бы доступ ко всем локальным переменных основной функции, а значит, могли бы использовать их внутри себя в качестве дополнительных аргументов.

В старых версиях MatLab параметры правой части системы уравнений можно было передавать солверу дополнительными аргументами. Эта возможность осталась и в последних версиях. Если правая часть ОДУ зависит от параметров

```
y′= f(t, y, p1, ... , ps),
```

то их можно передавать солверу

```
[t, Y ]=solver(@fun,[t0, tend],y0,options, p1, p2,...)
```

При этом заголовок функции `fun`, вычисляющей правую часть, также должен содержать их в списке параметров

```
function dfdy=fun(t,y,p1,p2,...)
```

Пример 16. Вернемся к примеру 14, в котором мы решали задачу о движении тела, брошенного под углом к горизонту. Создадим функцию `bodyanglek.m` правой части системы, имеющую дополнительный параметр – коэффициент сопротивления среды *k*.

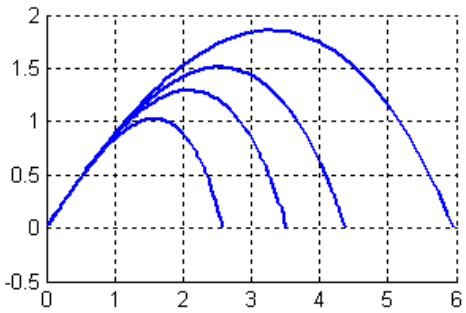
```
function F=bodyanglek(x,y,k)
g=9.81;
F=[y(2); -k.*y(2).*sqrt(y(2).^2+y(4).^2);...
   y(4); -k.*y(4).*sqrt(y(2).^2+y(4).^2)-g];
```

Если используется функция обработчик события параметра `Events`, то она также должна иметь этот дополнительный аргумент.

```
function [value,isterminal,direction] = ex8eventsk(t,y,k)
% Определять момент времени в который высота тела становится
% равной 0 при ее убывании и завершать вычисления, а также
% определить момент максимальной высоты (не останавливая вычислений),
% которая достигается когда скорость по y равна 0
value = [y(3), y(4)];      % определять нули для y(3) и y(4)
isterminal = [1,0];        % останавливать вычисления при y(3)=0
direction = [-1,0];        % функция y(3) убывает, для y(4) без разницы
```

При вызове солвера мы также должны передать ему дополнительный аргумент. Тогда сценарий решения `ex16_1.m` нашей краевой задачи будет иметь вид

```
% движение тела брошенного под углом к горизонту ex16_1.m
alph=pi/4;           % угол бросания тела
v0=10;               % начальная скорость
Y0=[0; v0.*cos(alph); 0; v0.*sin(alph)];    % вектор начальных условий
ak=[0.1  0.2  0.3  0.5];          % коэффициенты сопротивления среды
opts = odeset('Events',@ex8eventsk,'Refine',16);  % параметры
newplot; hold on;
for i=1:4
    [t,Y,te,ye,ie] = ode45(@bodyanglek,[0 Inf],Y0,opts,ak(i)); % решение
    plot(Y(:,1),Y(:,3), 'LineWidth',2);         % график траектории
end
grid on; hold off;
```



Однако, лучшим решением является использование анонимных функций. Вот пример сценария `exm16_2.m`, решающего ту же задачу, в котором мы каждый раз из функции `bodyanglek(x,y,k)` с тремя аргументами создаем функцию `ba=@(x,y) bodyanglek(x,y, ak(i))` правой части системы, содержащую только два аргумента.

```
% движение тела брошенного под углом к горизонту exm16_2.m
alph=pi/4;          % угол бросания тела
v0=10;              % начальная скорость
Y0=[0; v0.*cos(alph); 0; v0.*sin(alph)];    % вектор начальных условий
ak=[0.1  0.2  0.3  0.5];          % коэффициенты сопротивления среды
opts = odeset('Events',@ex8events,'Refine',16);  % параметры
newplot; hold on;
for i=1:4
    ba=@(x,y) bodyanglek(x,y, ak(i));
    [t,Y,te,ye,ie] = ode45(ba,[0 Inf],Y0,opts); % приближенное решение
    plot(Y(:,1),Y(:,3), 'LineWidth',2);         % график траектории
end
grid on; hold off;
```

При этом мы использовали прежний обработчик событий – функцию `ex8events(t,y)` без дополнительного параметра. □

Ранее мы отмечали, что вызов солверов без возвращаемого значения приводит к построению графика решения. Возможности вывода результата, предоставляемые солверами MATLAB, не исчерпываются только таким способом визуализации решения. Пользователь может выбрать альтернативное графическое представление результата или даже создавать свои функции для построения графиков. Для этого следует воспользоваться параметром `OutputFcn`. Его значение должно быть дескриптором функции (или строкой с ее именем), выполняющей требуемые операции. Имеется несколько стандартных функций:

- `odeplot` – построение графиков компонент решения;
- `odephas2` – построение фазовых траекторий для двумерных систем;
- `odephas3` – построение фазовых траекторий для трехмерных систем;
- `odeprint` – вывод числовой информации о решении.

По умолчанию параметр `OutputFcn` имеет значение дескриптора функции `odeplot`.

Фактически `OutputFcn` содержит имя функции, которая выполняется после каждого успешного шага интегрирования.

Пусть имеется задача

$$\begin{cases} y_1' = y_2 \\ y_2' = -y_2 - 5y_1 + \sin t \end{cases}, \quad \begin{pmatrix} y_1(0) \\ y_2(0) \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

Создаем функцию правой части системы

```
function F=fun10(x,y)
F=[y(2);-y(2)-5.*y(1)+sin(x)];
```

Выполняем команды

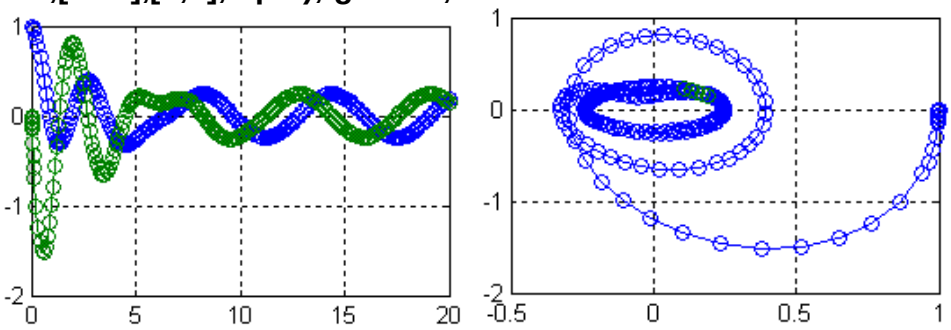
```
opts = odeset('OutputFcn',@odeplot);
ode45(@fun10,[0 20],[1;0], opts); grid on;
```

На следующем рисунке слева показан полученный график. Такой же график мы получаем при вызове солвера без возвращаемых параметров и без указания значения параметра `OutputFcn`.

ode45(@fun10,[0 20],[1;0]); grid on;

Следующие команды строят фазовую траекторию (график справа)

opts = odeset('OutputFcn',@odephas2);
ode45(@fun10,[0 20],[1;0], opts); grid on;



Для системы третьего порядка, рассмотренной в примере 10, с функцией правой части

```
function F=fun5(x,y)          % функция правой части системы
F=[y(2); y(3); x.^2.*y(3)-x.*y(2)+y(1)];
```

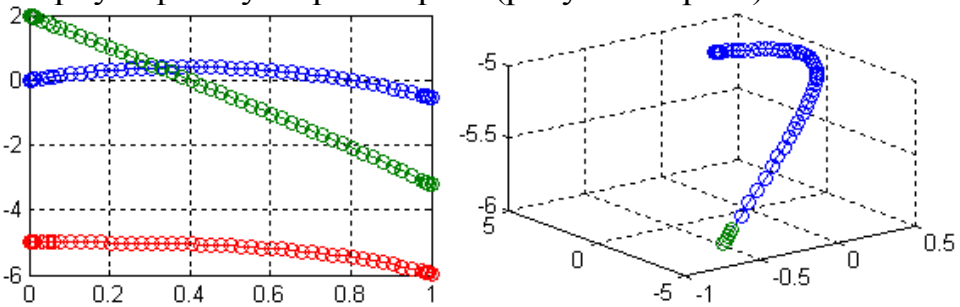
команда

ode45(@fun5,[0 1],[0; 2; -5]); grid on;

строит три графика. Они показаны на следующем рисунке слева. А команды

opts = odeset('OutputFcn',@odephas3);
ode45(@fun5,[0 1],[0; 2; -5] , opts); grid on;

строят трехмерную фазовую траекторию (рисунок справа)



Пример 17. Решим систему уравнений

$$\begin{cases} y_1' = s\cdot(y_2 - y_1) \\ y_2' = y_1\cdot(r - y_3) - y_2 \\ y_3' = y_1\cdot y_2 - b\cdot y_3 \end{cases},$$

где `s, r, b` некоторые параметры. Создадим функцию правой части системы

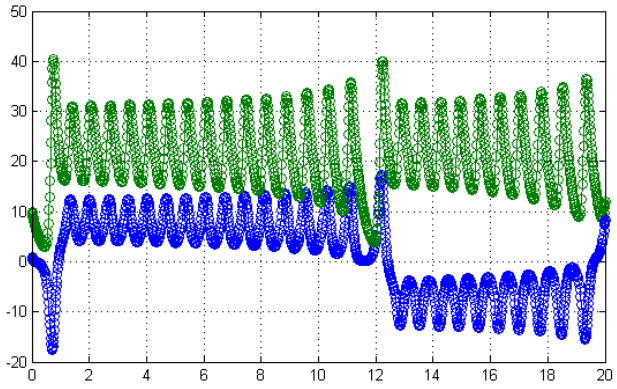
```
function f=lor(t,y,s,r,b)
% правая часть системы уравнений Лоренца
f=[s.*(y(2)-y(1));...
    -y(2)+(r-y(3)).*y(1);...
    -b.*y(3)+y(1).*y(2)];
```

В окне кода задаем значения параметров

s=10; r=25; b=3;

Чтобы вызов функции `ode45` без возвращаемых значений построил график двух компонент решения (а не трех) задаем параметр `OutputSel`, в котором указываем номера передаваемых компонент

```
opts = odeset('OutputSel',[2 3]);
ode45(@lor,[0 20],[ 1 -1 10], opts, s, r, b); grid on;
```

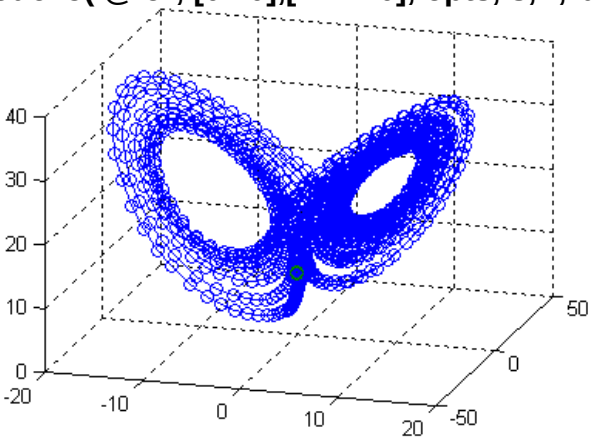


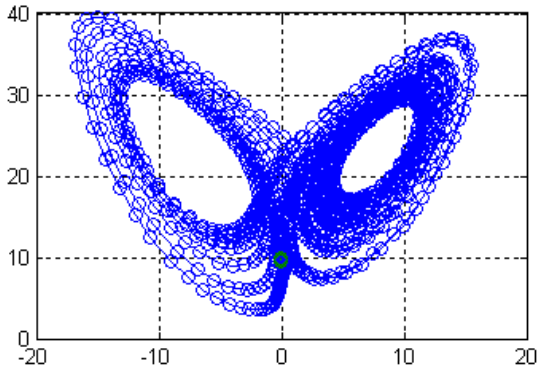
Для построения трехмерной фазовой траектории (следующий рисунок слева) выполним команды

```
opts = odeset( 'OutputFcn', @odephas3 );
ode45( @lor, [0 20], [1 -1 10], opts, s, r, b);
```

А для построения двумерной фазовой траектории компонент 1 и 3 (следующий рисунок справа) выполним команды

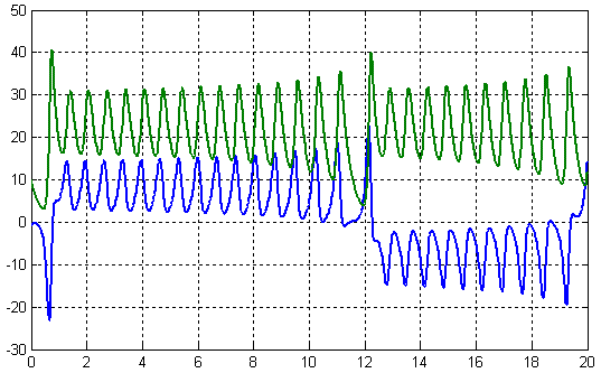
```
opts=odeset('OutputSel',[1 3],'OutputFcn',@odephas2 );
ode45( @lor, [0 20],[1 -1 10], opts, s, r, b );
```

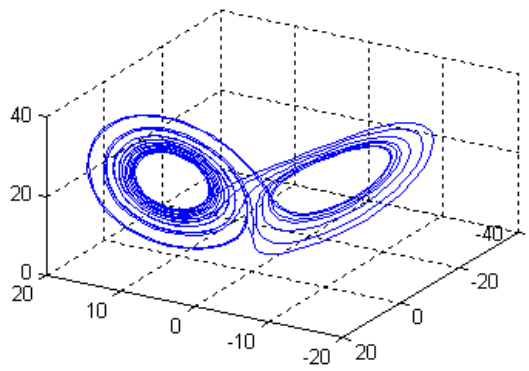




Если использовать возвращаемые значения, то предыдущие графики можно построить, например, следующим образом

```
[T,Y]=ode45(@lor,[0 20],[1 -1 10], [ ],s, r, b);
plot(T,Y( : , 2:3), 'LineWidth',2); grid on;      % График кривых 2 и 3
plot3(Y(:,1), Y(:,2), Y(:,3));  grid on;      % 3d фазовая траектория
```



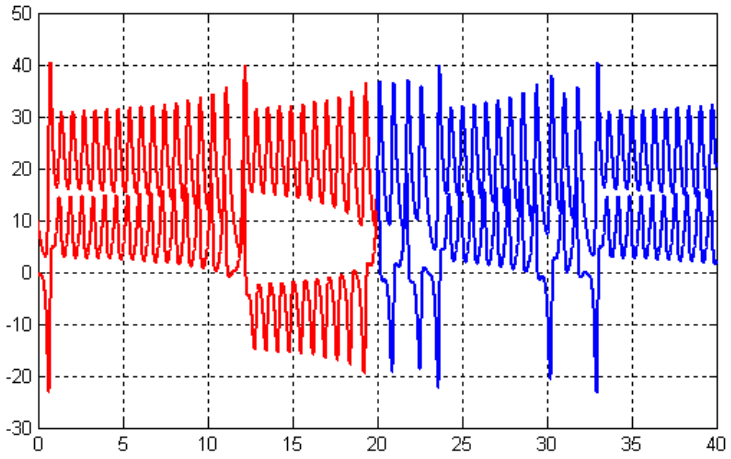


Движение точки по фазовой траектории компонент 1 и 3 можно выполнить командой

```
comet(Y(:,1), Y(:,3));
```

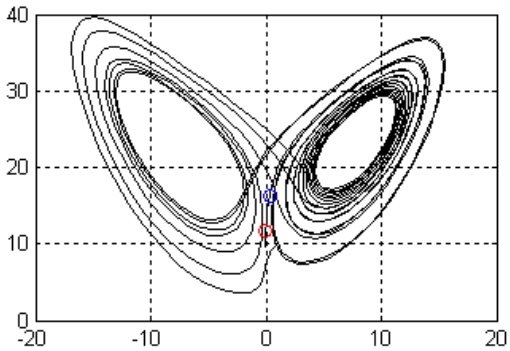
А движение по трехмерной фазовой траектории - командой
comet3(Y(:,1),Y(:,2), Y(:,3));
Если мы захотим продолжить расчет, начав решение системы в точке в которой закончился предыдущий расчет, то следует конечные значения решения первого расчета использовать как начальные значения для следующего.

```
[TY , Y]=ode45(@lor,[0 20],[1 -1 10], [ ],s, r, b);
Z0=Y(end, :)      % начальные значения второго расчета
[TZ , Z]=ode45(@lor,[20 40],Z0, [ ],s, r, b);
plot(TY,Y(:,2:3),'r', 'LineWidth',2); hold on;
plot(TZ,Z(:,2:3),'b', 'LineWidth',2); grid on;
hold off;
Z0 =
    8.2178   14.3566   11.9137
```



Чтобы посмотреть положение некоторых точек фазовой траектории, выполним команду

```
plot(Y(:,1), Y(:,3),'k',Y(end-10,1),Y(end-10,3),'or', Y(end-20,1),Y(end-20,3),'ob');
[Y(end-10,1),Y(end-10,3)]
ans =
    2.6544    9.0760
```



Все приведенные здесь команды удобно собрать в сценарий. □

Пользователь может создавать свои файл – функции для визуализации решения или обработки результатов каждого шага численного интегрирования. Для этого он должен задать свое значение параметра `OutputFcn`, т.е. присвоить ему значение дескриптора на свою функцию. Эта функция будет вызываться солвером перед первым шагом интегрирования, после каждого шага и в конце. В зависимости от возвращаемого ею значения солвер может останавливать вычисления либо продолжать их. Заголовок функции должен иметь вид

```
function status=outfun (t, y, flag)
```

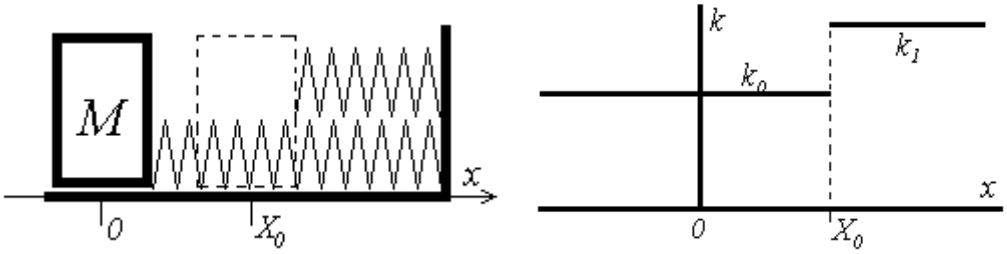
Входной аргумент `flag` является строковой переменной и может принимать одно из трех значений

- `'init'` – передается солвером при первом вызове до начала интегрирования, параметр `t` является вектором из двух элементов – границ отрезка интегрирования, а `y` – является вектором начальных значений;
- `''` (пустая строка) передается после каждого шага интегрирования, параметр `t` является текущим значением аргумента, `y` – вектором приближенных значений на текущем шаге; аргумент `t` может содержать несколько текущих значений, тогда `y` – матрица, каждый столбец которой содержит компоненты решения для соответствующего момента времени;
- `'done'` передается после завершения численного решения системы, `t` и `y` являются пустыми массивами.

Когда `flag` является пустой строкой, длина входного аргумента `t` определяется значением параметра `Refine`. По умолчанию оно равно 1 и все солверы (кроме `ode45`) на каждом шаге будут передавать функции `outfun` в параметре `t` только одно значение независимой переменной, а в параметре `y` – вектор значений решения в этот момент. Функция `outfun` должна возвращать в выходном аргументе `status` либо 0, либо 1. Если солвер обнаруживает, что функция `outfun` вернула 1, то процесс решения заканчивается, а если 0 – то продолжается.

Этой функции можно передавать значения не всех координат вектора решения `y`. Для указания вектора индексов компонент решения, которые следует передавать в `OutputFcn` , используется параметр `OutputSel`. Его значением должен быть вектор из целых чисел – номеров компонент вектора решения. По умолчанию передаются все компоненты.

Пример 18. Решим задачу о колебании массивного тела массы M на невесомых пружинах, одна из которых короче другой и короткая не привязана к телу (см. рисунок), трение отсутствует.



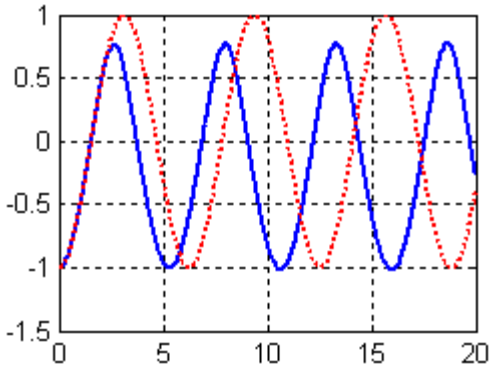
Для тела M выполняется уравнение движения $\ddot{x}(t)+k^2x(t)=0$. При движении вправо и достижении положения X_0 тело M присоединяется ко второй пружине и жесткость системы меняется. Когда тело движется влево, и проходит положение X_0 оно отрывается от второй пружины и жесткость уменьшается. Т.о. жесткость зависит от смещения груза M относительно положения равновесия, т.е. $k=k(x)$. При этом функция $k(x)$ является кусочно постоянной.

График функции $k(x)$ приведен на предыдущем рисунке справа. При решении этой задачи трудность состоит в том, что мы заранее не знаем в какие моменты времени тело М проходит положение X_0.

Пусть $k_0=1$, $k_1=1.5$ и начальные значения $x(0)= -1$, $x'(0)=v_0=0$. Решение оформим в виде функции `bodymove.m` без возвращаемых значений и без аргументов.

```
function bodymove
% движение тела с кусочно постоянной жесткостью пружины
X0=0.5;   % координата точки отрыва тела от пружины
Y0=[-1; 0];       % вектор начальных условий
k=1;
opts = odeset('OutputFcn',@koef,'MaxStep',0.01);  % сообщаем обработчик
[t,Y] = ode113(@fun,[0 20],Y0,opts); % приближенное решение
plot(t,Y(:,1), 'LineWidth',2);
grid on; hold on;
% график точного решения при k=1
plot(t,-cos(t), ':r','LineWidth',2);
hold off;

    % локальная функция с заданным вне ее коэффициентом жесткости k
    function F=fun(x,y)
        F=[y(2); -k.^2.*y(1)];
    end

    % функция контролер положения тела
    function status=koef(x,y, flag)
    % функция определяет, превысило ли значение первой компоненты
    % решения число X0 после каждого шага интегрирования
    % если да то меняем значение k, status=0 всегда
    if length(flag) == 0
        if y(1)>X0
            k=1.5;
        else
            k=1;
        end
        status = 0;
    end
    end
end
```

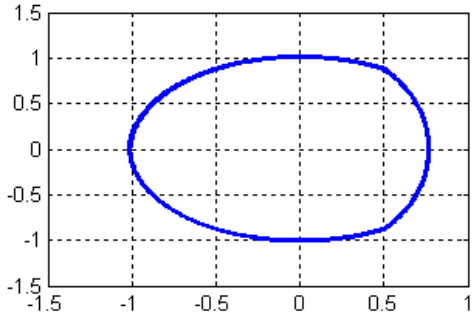
График решения показан на следующем рисунке сплошной линией. Пунктиром показано решение для случая неизменного `k=1`.



Для тренировки создадим функцию `bodymove2.m`, которая выполняет некоторый код перед началом интегрирования (выводит сообщение) и после его завершения (выводит сообщение и строит фазовую траекторию, если солвер `ode113` вызвать без возвращаемых значений)

```
function bodymove2
% движение тела с кусочно постоянной жесткостью пружины
% строит фазовую траекторию
X0=0.5;         % координата точки отрыва тела от пружины
Y0=[-1; 0];     % вектор начальных условий
k=1;
Z=[]; T=[];     % переменные для накопления значений

opts = odeset('OutputFcn',@koef,'MaxStep',0.01);  % сообщаем обработчик
ode113(@fun,[0 20],Y0,opts); % строим фазовую траекторию

    % локальная функция с заданным вне коэффициентом жесткости k
    function F=fun(x,y)
        F=[y(2); -k.^2.*y(1)];
    end

    % функция контролер положения тела
    function status=koef(x,y, flag)
    % функция определяет, превысило ли значение первой компоненты
    % решения число X0 после каждого шага интегрирования
    % если да то меняем значение k, status=0 всегда
    if length(flag) == 0
        if y(1)>X0
            k=1.5;
        else
            k=1;
        end
        T=[T; x];
        Z=[Z; [y(1) y(2)]];
        status = 0;
    elseif isequal(flag,'done')
        disp('Решение задачи завершено. Строим фазовую траекторию');
        plot(Z(:,1),Z(:,2), 'LineWidth',2);
        grid on;
    elseif isequal(flag,'init')
        disp('Начало интегрирования.');
    end
    end
end
```

Фазовая траектория показана на следующем рисунке.



Отметим некоторые особенности этого кода. Когда функция `koef` выполняется в конце интегрирования, солвер `ode113` еще не завершил своей работы и возвращаемые массивы `t` и `Y` (см. функцию `bodymove.m`) еще не могут быть использованы. Поэтому, чтобы построить фазовую траекторию, мы на участке кода после строки `if length(flag) == 0` наращиваем массивы `T` и `Z`, которые после завершения всех шагов интегрирования хранят значения векторов независимой переменной и компонент решения. Чтобы код функции `koef`, стоящий после сроки `elseif isequal(flag,'done')` выполнился, мы вызываем солвер без выходных параметров. В примере мы используем

солвер ode113, который на каждом шаге интегрирования передает функции koef одно значение t и два числа y(1) и y(2). Если, например, использовать солвер ode45, то потребуется модификация кода функции koef. □

Заметим, что функции, используемые в качестве значений параметра OutputFcn, близки по смыслу к функциям, используемым в качестве значений параметра Events. Однако, функции параметра Events не могут быть вызваны до начала или после завершения процесса интегрирования. Кроме того, функции параметра OutputFcn выполняются на каждом шаге интегрирования независимо от того, происходит какое – либо событие или нет.

В заключении приведем кратко информацию о некоторых других возможностях MatLab решения ОДУ.

После завершения счета солвер может вывести в командное окно итоговую информацию о вычислительной работе. Для этого в структуре управляющих параметров следует установить значение параметра ‘Stats’ в 'on’.

При решении системы дифференциальных уравнений солверы аппроксимируют матрицу Якоби правой части системы методом конечных разностей. Эффективность вычислений может быть повышена путем задания матрицы Якоби в явном виде. Для этого можно создать файл-функцию, которая возвращает матрицу Якоби для текущих значений независимой переменной и решения. Затем свойству Jacobian управляющей структуры необходимо присвоить дескриптор этой функции. Формат этой функции предопределен и с ним можно познакомиться по справочной системе.

Системы дифференциальных уравнений большой размерности могут иметь разреженную матрицу Якоби. В этом случае можно подсказать солверу, место расположения ее ненулевых элементов подлежащих аппроксимации. Для этих целей служит свойство JPattern, значением которого должна быть разреженная матрица с элементами 0 и 1.

До сих пор мы рассматривали задачу Коши для систем дифференциальных уравнений, разрешенных относительно производных

$$Y' = f(t, Y)$$

MATLAB позволяет решать системы дифференциальных уравнений, заданных в неявной форме

$$F(t, Y, Y') = 0$$

Для решения таких систем служит солвер ode15i. Важным частным случаем таких систем являются системы вида

$$M(t, Y)Y' = F(t, Y)$$

Матрица *M(t, Y)* называется матрицей масс системы и может быть как невырожденной, так и вырожденной. Системы с матрицей масс могут быть решены любым из солверов. При этом солвер ode23s может использовать только постоянную матрицу, а в случае вырожденных матриц необходимо прибегать к солверам ode15s и ode23t. Вырожденная матрица масс

соответствует системе с дифференциальными и алгебраическими уравнениями. Параметр Mass управляющей структуры как раз предназначен для этого случая. По умолчанию Mass это единичная матрица. Подробные сведения о настройках солверов для решения задач с матрицей масс приведены в справочной системе MATLAB. Управляющая структура позволяет задать кроме матрицы масс еще ряд опций с ней связанных, в частности расположение ее ненулевых элементов для задач большой размерности и информацию о вырожденности матрицы.

Если структура параметров options вами создана, то узнать текущие значения параметров можно командой

```
val = odeget(options,'nameopt')
```

В заключении приведем еще одну форму обращения к солверам MatLab

[T, Y, S] = solver(odefun, ,[t_0, t_{end}], y_0, options)

Здесь S является массивом из шести элементов, в которых накапливается статистика о ходе решения задачи: число успешных шагов; число неудачных шагов; количество вычислений правой части; количество вычислений матрицы Якоби; количество выполненных факторизаций; число решений линейной системы алгебраических уравнений. Например

```
ode45(@fun10,[0 20],[1;0]);
[T,Y,S]=ode45(@fun10,[0 20],[1;0]);
S
S =
    54
     2
   337
     0
     0
     0
```

Интересные простые примеры решения непростых ОДУ содержатся в файлах orbitode (ограниченная проблема трех тел), rigidode (уравнения Эйлера), vdpode (уравнение Ван дер Поля), ballode (прыгающий мячик) Чтобы посмотреть их код выполните команду type name, а чтобы выполнить этот код просто наберите имя name, где name имя одного из файлов. Упрощенные варианты некоторых из этих задач мы рассматривали в наших примерах.

Функции, связанные с солверами.

Есть две функции, предназначенные для совместного использования с солверами odextend и deval. Функция odextend используется для продолжения решения, полученного с помощью солверов, а deval – для получения значений решений в заданных точках.

Рассмотрим уравнение, решенное нами в примере 6. Функция правой части системы имеет вид

```
function F=funToEx(x,y)          % функция правой части системы
F=[y(2); -y(1)^3+y(1)];
```

Решим задачу на отрезке [0,8]

```
sol=ode45(@funToEx,[0 8],[0, 0.1]);   % решаем систему
```

Функция odeextend позволяет получить решение на более широком участке, например, на отрезке [0 16].

```
sol2=odextend(sol,@funToEx,16);
plot(sol2.x,sol2.y(1,:));
```

Первым аргументом odeextend является структура sol, которую возвращает солвер при решении задачи Коши. Вторым – дескриптор функции @funToEx правой части системы, третьим – конечное значение независимой переменной. Нет необходимости повторно передавать имя функции, вычисляющей правую часть системы ОДУ. Это имя хранится в структуре sol и вторым аргументом можно передавать пустой вектор. Кроме того, в структуре хранится имя солвера и его параметры. Поэтому задача решается с использованием того же самого солвера и тех же самых параметров.

```
sol=ode45(@funToEx,[0 8],[0, 0.1]);
solext=odextend(sol,[ ],16);
plot(solext.x,solext.y(1,:));
```

Допустимо использовать следующие форматы вызова

```
solext = odextend(sol, odefun, tfinal)
solext = odextend(sol, [], tfinal)
solext = odextend(sol, odefun, tfinal, yinit)
solext = odextend(sol, odefun, tfinal, [yinit, ypinit])
solext = odextend(sol, odefun, tfinal, yinit, options)
```

Аргумент tfinal указывает новое конечное значение независимого аргумента. Последнее значение решения sol.y(:,end), полученное солвером на первом этапе, является начальным значением для продолжения решения. Если вы хотите изменить это начальное значение или изменить параметры вычислительного процесса, то следует использовать другие форматы вызова функции odextend. Возвращаемая структура solext имеет те же поля, что и исходная структура sol и ее можно использовать также как sol.

Функция deval решение ОДУ, которое солвер возвращает в структуре sol, вычисляет в заданных точках. Формат ее вызова следующий

```
sxint = deval(sol,xint)
sxint = deval(xint,sol)
sxint = deval(sol,xint,idx)
sxint = deval(xint,sol,idx)
[sxint, spxint] = deval(...)
```

Первые два вызова эквивалентны. Численное решение задачи Коши или краевой задачи возвращается в структуре sol, которая является одним из аргументов функции deval. Другой аргумент xint является точкой или вектором – значениями независимого переменного, в которых вы желаете знать решение. Элементы вектора xint должны находиться на отрезке [sol.x(1),sol.x(end)]. Вектор sxint содержит значение решения для каждого элемента вектора xint.

Третий и четвертый вызовы функции deval позволяют в векторе idx передать номера тех компонент вектора решения, которые вы желаете

получить. Вызов функции deval с двумя возвращаемыми параметрами позволяет кроме значения вектора sxint решения задачи в указанных точках, получить значение производной решения spxint в этих точках, которые получаются путем полиномиальной интерполяции решения. Например, для решения sol последней задачи, можно получить значение решения в точках 1, 2, 3, 4 следующим образом

```
deval(sol,[1,2,3,4])
ans =
    0.1175    0.3582    0.8960    1.4182
    0.1540    0.3606    0.7006    0.0413
```

Каждый столбец содержит вектор значений решения в момент времени, указанный в соответствующем столбце вектора xint. Команда

```
[xv,pp]=deval(sol,[1,2,3,4])
xv =
    0.1175    0.3582    0.8960    1.4182
    0.1540    0.3606    0.7006    0.0413
pp =
    0.1540    0.3604    0.7010    0.0416
    0.1158    0.3122    0.1736   -1.4352
```

возвращает те же значения решения, но и их производные. Обратите внимание, что первая строка производных pp близка по значению ко второй строке вектора xv. Это потому, что в нашем примере вторая компонента решения является производной первой компоненты.

Замечания

В данной главе были приведены основные сведения о функциях MATLAB, предназначенных для решения задач Коши для обыкновенных дифференциальных уравнений и их систем. Рассмотрены функции символьного и численного решения таких задач. Попутно были рассмотрены возможности символьного решения ДУЧП. Однако этим возможности MATLAB не исчерпываются. Перечислим некоторые дополнительные возможности MATLAB для решения дифференциальных уравнений

- для решения краевых задач ОДУ имеется солвер bvp4c и несколько вспомогательных функций;
- для численного решения ДУЧП предназначены функции пакета расширения PDE TOOLBOX;
- имеются пакеты расширения (наборы функций и графические интерфейсы), предназначенные для исследования математических моделей, сводящихся к решению систем дифференциальных уравнений, возникающих в различных областях физики и техники. Одним из таких пакетов является SIMULINK.

Помимо этого, читатель, владеющий навыками программирования в MATLAB, может создавать свои собственные функции, реализующие любые «милые его сердцу» численные методы решения дифференциальных уравнений.

4. Решение дифференциальных уравнений в частных производных.

В предыдущих главах были подробно рассмотрены основные элементы, необходимые для уверенного использования MatLab. Однако в нем есть еще большое количество пакетов расширения, которые многократно увеличивают эффективность использования системы. Одним из таких пакетов является PDE TOOLBOX, предназначенный для численного решения дифференциальных уравнений в частных производных (ДУЧП) и их систем. В данной главе мы приводим основные сведения, которые помогут вам использовать возможности этого пакета, а также рассматриваем другие способы решения таких уравнений и их систем.

4.1 Введение в PDE Toolbox

Многие прикладные задачи сводятся к решению дифференциальных уравнений в частных производных (ДУЧП) и их систем. Одним из наиболее распространенных приближенных методов их решения является метод конечных элементов (МКЭ). Данная глава посвящена описанию возможностей пакета расширения Partial Differential Equations Toolbox (PDE Toolbox), предназначенного для решения краевых задач для дифференциальных уравнений в частных производных в двумерных областях методом конечных элементов.

Пакет состоит из набора функций, автоматизирующих реализацию МКЭ для решения различного типа ДУЧП $2-$го порядка и их систем: эллиптических, параболических и гиперболических. Кроме того, в состав пакета входит приложение `pdetool` с графическим интерфейсом пользователя, использование которого не требует глубокого понимания метода конечных элементов и упрощает доступ к набору функций пакета.

4.1.1 Графический интерфейс PDE Toolbox

Решение краевых задач для отдельных уравнений.

Рассмотрим пример решения задачи

$$\Delta u = 16 \cdot \left(x^2 + y^2\right), \quad u\big|_{\partial\Omega} = 1,$$

где область Ω представляет собой круг единичного радиуса с центром в начале координат. Выбор этой задачи объясняется тем, что ее точное решение известно $u = \left(x^2 + y^2\right)^2$, и мы сможем сравнить его с решением, полученным с помощью PDE TOOLBOX.

Наберите в командном окне MalLab команду

pdetool

Откроется окно программы PDE Toolbox

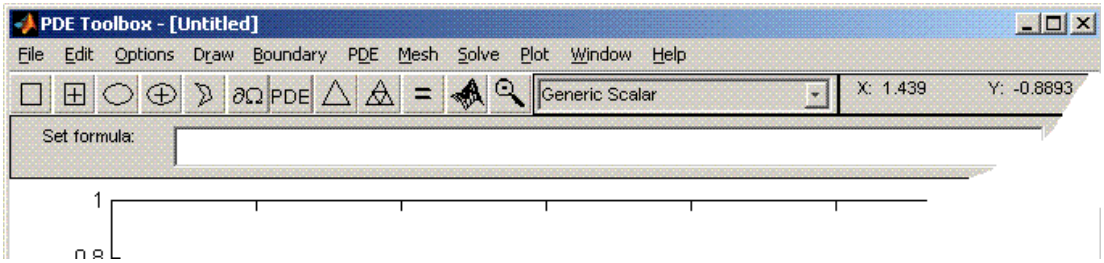


Рис. 1

Используя инструменты этого окна, мы можем ввести исходные данные, найти приближенное решение задачи и построить его график.

Ввод условий задачи осуществляется в четыре этапа:

1. **Задание двумерной области Ω.** Нажмите кнопку, на которой нарисован эллипс с плюсом внутри  , и при помощи мыши нарисуйте эллипс. Затем дважды щелкните мышью по появившейся области. Появится диалоговое окно с параметрами эллипса.

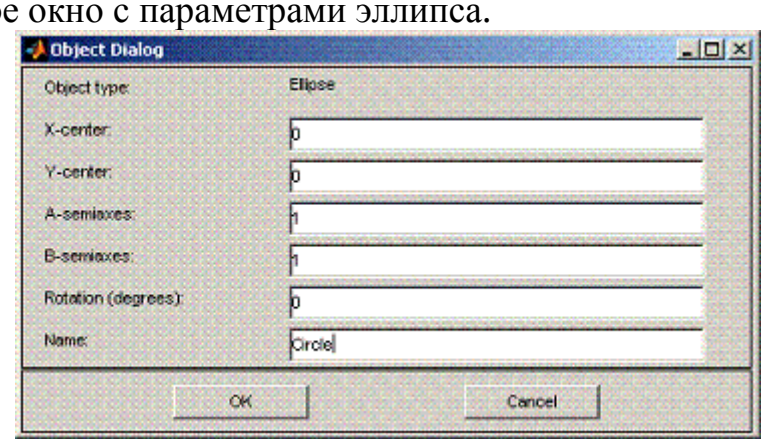


Рис. 2

Исправьте параметры так, чтобы получился единичный круг.

2. **Задание граничного условия**. Нажмите кнопку ∂Ω. Граница круга выделится красным. Это означает, что на ней заданы условия Дирихле. Для их уточнения, зайдите в меню `Boundary` и выберете пункт `Specify Boundary Conditions`. Перед вами появится окно задания граничных условий.

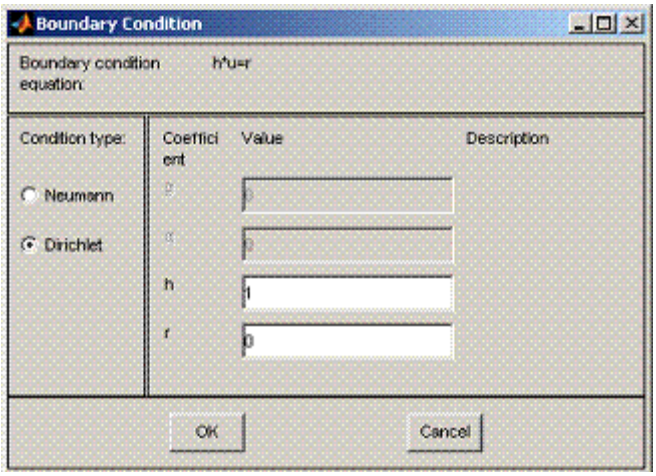


Рис. 3

В этом окне переключатель `Dirichlet`, указывает на задание граничных условий Дирихле. Исправьте в строке `r` значение 0 на 1. Тем самым, мы задаем граничное условие $u|_{\partial\Omega} = 1$.

3. **Задание конечно элементной сетки.** Триангуляция области может быть выполнена автоматически. Нажмите кнопку  . Это приводит к отображению используемой триангуляции области.

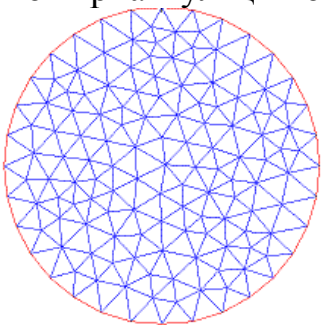


Рис. 4

Кнопка рядом  позволяет сгустить разбиение и, тем самым, повысить точность вычислений. Меню `Mesh` позволяет внести и другие изменения в конечноэлементную сетку. В нашем примере можно согласиться с разбиением области по умолчанию, и сетку не менять.

4. **Задание уравнения**. Зайдите в меню `PDE` и выберете пункт `PDE Specification`. Перед вами появится окно выбора вида ДУЧП. По умолчанию решается уравнение $-\Delta u = 10$. Исправьте в строке `f` значение 10 на $-16\cdot(x^2+y^2)$, т.е. введите это выражение, используя поэлементные знаки математических операций `-16*(x.^2+y.^2)`

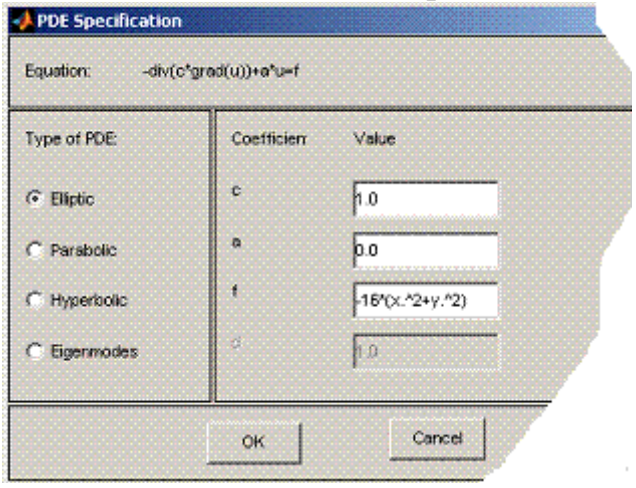
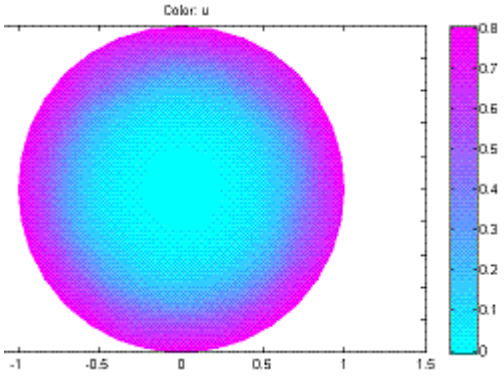


Рис. 5

Задав условия задачи, нажмите кнопку  для получения решения. В открытом окне вы получите цветной график приближенного решения, на котором цвет точек соответствует высоте (значению) решения *u(x,y)*. Палитра цветов отображается справа от рисунка.



Сравним полученное решение с точным $u=\left(x^2+y^2\right)^2$, построив в области их разность. Для этого нажмите кнопку, на которой изображен логотип MatLab  . Перед вами появится окно `Plot Selection`

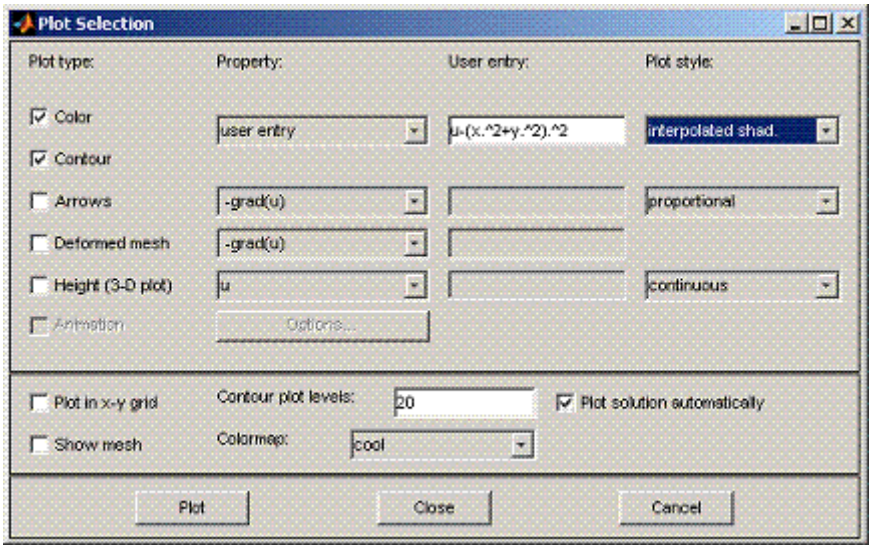
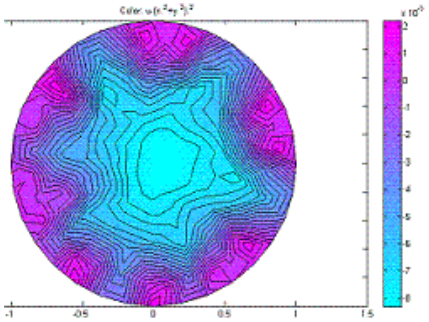


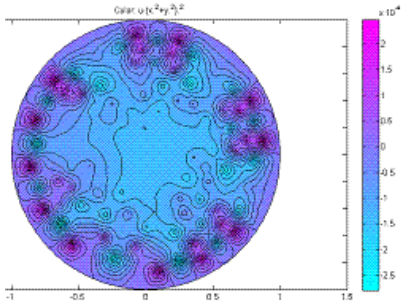
Рис. 6

Левая колонка этого окна содержит флаги, соответствующие способу визуализации результатов. Столбец полей `Property` состоит из раскрывающихся списков, предназначенных для выбора отображаемой функции. Зайдите в верхний список, соответствующий флагам `Color` и `Contour` и в списке выберете пункта `User Entry`. Тогда станет доступной соответствующая ячейка третьей колонки. Введите в нее выражение `u-(x.^2+y.^2).^2`. Установите также флажок `Contour`. Затем нажмите кнопку `Plot` для построения графика погрешности решения $u-\left(x^2+y^2\right)^2$.

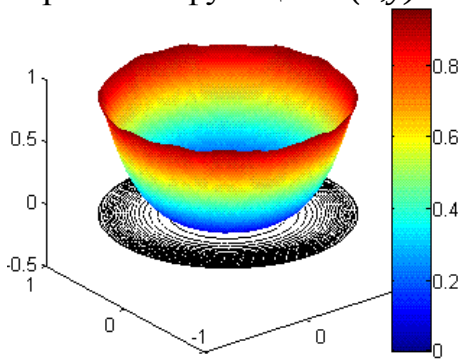


Из цветовой палитры графика вы можете определить значение максимальной погрешности (разности приближенного и точного решений). Кнопка 

позволяет уменьшить размеры треугольников разбиения области, а следовательно, повысить точность решения краевой задачи. Нажмите на нее, а затем решите задачу еще раз, нажав кнопку =. На следующем рисунке показана погрешность решения, которая, как мы видим, стала значительно меньше.



Вы можете также увидеть график решения в виде поверхности. Нажмите кнопку и установите в окне `Plot Selection` флаг `Height (3-D plot)`, а также в колонке полей `Property` напротив флага `Color` верните значение `u`. Затем нажмите кнопку `Plot`, которая создаст графическое окно `Figure 1` с графиком поверхности функции *u(x,y)*.



Пример 2. Решим задачу о стационарном распределении температуры в прямоугольной области с круговым отверстием в центре. Левая Γ_1 и правая Γ_3 границы прямоугольной области теплоизолированы. На верхней Γ_2, нижней Γ_4 и внутренней Γ_5 границе области поддерживается постоянная температура (разная на разных участках границы). Источников тепла нет.

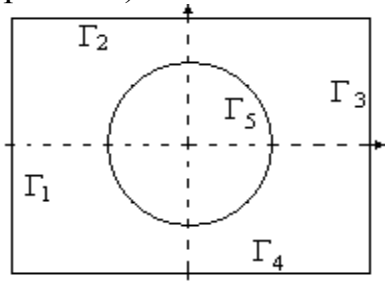


Рис. 7

Стационарное распределение температуры описывается дифференциальным уравнением

$$\frac{\partial^2 u}{\partial x^2}+\frac{\partial^2 u}{\partial y^2}=0 \quad \text{или} \quad div(grad\, u)=0$$

и граничными условиями $u\big|_{\tilde{A}_2} = u_2$, $u\big|_{\tilde{A}_4} = u_4$, $u\big|_{\tilde{A}_5} = u_5$, $\left.\frac{\partial u}{\partial x}\right|_{\tilde{A}_1} = 0$, $\left.\frac{\partial u}{\partial x}\right|_{\tilde{A}_3} = 0$.

Задание двумерной области Ω. Конструирование области, показанной на рисунке, состоит в создании прямоугольника и круга заданных размеров, и вычитание области круга из области прямоугольника.

Нарисуем мышью прямоугольник и круг. В окне свойств прямоугольника зададим координаты левого нижнего угла (-3, -2), а также ширину 6 и высоту 4 прямоугольника. Содержимое R1 поля name оставим без изменений. Аналогично в окне свойств эллипса введем координаты центра (0, 0) и одинаковые значения полуосей 1. Содержимое E1 поля name оставим без изменения.

В данный момент установлен режим рисования. Переход в него происходит при использовании инструментов рисования или выборе меню Draw – Draw Mode. Другие режимы задаются выбором пунктов меню Boundary – Boundary Mode (режим задания граничных условий), PDE – PDE Mode (режим выбора уравнения) и Mesh – Mesh Mode (режим построения сетки), или выбором подходящих инструментов и пунктов меню.

Следующий этап состоит в определении взаимосвязи между примитивами, образующими область. Связь между ними определяется в строке Set Formula. Знак плюс означает объединение объектов, а минус – вычитание. Нашей области соответствует формула R1 – C1.

Выбор уравнения и задание граничного условия. Меню Options – Application позволяет задать тип решаемой задачи. Его пункт Heat Transfer соответствует задаче о распределении тепла. Выберите этот пункт. Слева от названия появится флаг – среда pdetool теперь настроена на решение задачи теплопроводности. Использование раскрывающегося списка, размещенного на панели инструментов, приводит к аналогичному результату (на рис.1 в этом поле стоит выбор Generic Scalar). Теперь установим режим задания дифференциального уравнения, выбрав пункт PDE – PDE Mode. На экране отобразится область с отверстием. Выберите пункт PDE – PDE Specification... или выполните двойной щелчок по области. Появляется диалоговое окно PDE Specification (см. рис. 5, но с другими настройками). В верхней части окна на панели Equation содержится общий вид стационарного уравнения теплопроводности $-div(k \cdot grad\, T) = Q + h \cdot (Text - T)$, которое может быть решено в среде pdetool.

Значения коэффициентов устанавливаются в строках ввода, расположенных в средней части диалогового окна. Левая панель служит для выбора типа уравнения. Переключатель Elliptic соответствует задаче о стационарном распределении тепла, а Parabolic – нестационарному случаю. Установите переключатель Elliptic и задайте в строках ввода коэффициенты уравнения рассматриваемой задачи: *k=1, Q=0, h=0, Text=0* (физический смысл этих коэффициентов обсуждается в курсах математической

физики, а краткое пояснение находится справа от соответствующего поля). Нажмите ОК для сохранения проделанных изменений.

Переходим к заданию граничных условий. Выберите пункт меню `Boundary – Boundary Mode`. Среда pdetool перейдет в режим установки граничных условий, и в ее окне будут отображены только границы области. Обратите внимание, что прямоугольник имеет четыре границы, по числу сторон, и окружность также составлена из четырех дуг. Щелчком мыши сделайте текущей верхнюю границу прямоугольника и выберите в меню `Boundary` пункт `Specify Boundary Conditions...` (или выполните двойной щелчок по желаемому участку границы). Появится диалоговое окно `Boundary Condition`, предназначенное для задания граничного условия на выбранном участке границы (такое как на рис. 3). На верхней границе прямоугольника мы задаем температуру. Это граничное условие Дирихле. Убедитесь, что на панели `Condition type` включен переключатель `Dirichlet` и в полях значений параметров установите *h=1, r=100*. Сохраните значения, нажав ОК, и проделайте аналогичную операцию для нижней стороны прямоугольника, предварительно сделав ее текущей.

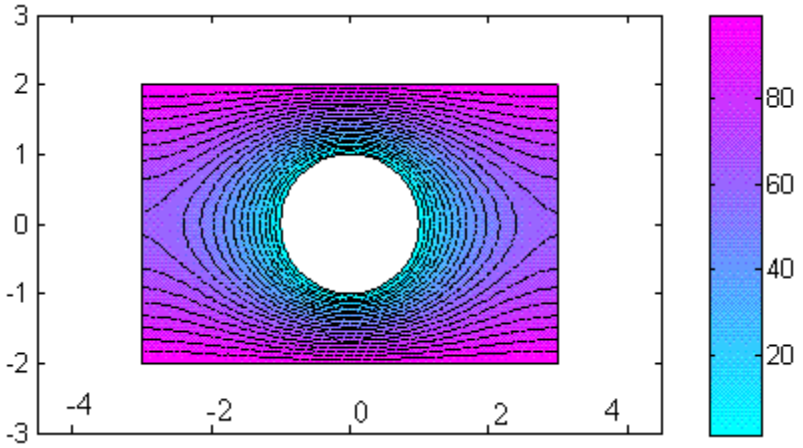
Граничные условия можно задавать по отдельности на каждой части границы, или объединить несколько частей и определить одинаковые граничные условия сразу для всей группы. Добавление части границы в группу производится щелчком мыши с одновременным удержанием клавиши Shift.

Установите нулевую температуру на границе отверстия, сгруппировав предварительно четыре части окружности. На правой и левой стороне прямоугольной области поток тепла равен нулю. Равенство нулю потока есть частный случай условия Неймана. Установите переключатель `Condition type` в положение `Neumann`. Для задания теплоизолированных границ нужно задать *g=0, q=0*. В режиме установки граничных условий границы с условиями Дирихле отображаются красным цветом, а синий цвет выделяет границы, на которых задано условие Неймана.

Задание разбиения области. Выполним триангуляцию области – покрытие области сеткой, состоящей из треугольников. Триангуляция и установка ее параметров производятся в меню `Mesh`. Перейдите в режим триангуляции, выбрав пункт `Mesh Mode`. Область разбивается на достаточно крупные треугольные элементы, причем считается, что граница области составлена из сторон некоторых элементов. Инициализация и включение режима триангуляции может быть произведена кнопкой с треугольником  . Для получения решения с приемлемой точностью начальной триангуляции недостаточно, следует уменьшить шаг разбиения области. Выберите пункт `Refine Mesh` или нажмите кнопку  с треугольником, разделенным на четыре части. Каждый выбор данного пункта приводит к уменьшению размеров треугольников. Учтите, что выбор слишком мелкой сетки может привести к значительным затратам времени на решение задачи, однако слишком грубая сетка приводит к большим погрешностям. Вернуться к начальной триангуляции

можно с помощью меню `Initialize Mesh` или кнопки с треугольником. Обратите также внимание на то, что уменьшение размеров ячеек сетки улучшает вид границ области.

Решение уравнения. Решение задачи производится выбором пункта `Solve – Solve PDE` или нажатием на кнопку со знаком равно =. Найденное распределение температуры отображается в окне среды pdetool контурным графиком с цветовой заливкой, рядом с которым расположен столбик с информацией о соответствии цвета значению температуры.



Изменение геометрии области, граничных условий, типа уравнения и его коэффициентов может быть выполнено даже после нахождения решения. Для этого следует перевести среду pdetool в соответствующий режим и произвести необходимые действия.

Сохранение работы производится в m-файле из пункта `Save as...` или `Save` меню `File`. Этот файл содержит функции PDE Toolbox, с помощью которых решается задача. Он содержит информацию о геометрии области, о типе и значениях коэффициентов уравнения и граничных условий, а также текущие установки среды. Впоследствии, загрузив этот файл, можно продолжить решение задачи.

Пример 3. Решим задачу о нестационарном распределении тепла в области, изображенной на рис. 7. На внешних границах прямоугольника поддерживается постоянная температура 1, а края отверстия подвергаются нагреву, температура изменяется одинаково во всех точках окружности линейно со временем. Внутри области нет источников тепла, в начальный момент времени температура равна нулю во всей области.

Задайте геометрию области и настройте среду pdetool на решение задачи теплопроводности. Перейдите к диалоговому окну `PDE Specification` и выберите параболический тип уравнения, установив переключатель `Parabolic`. Задайте следующие значения коэффициентов *rho=1, C=1, k=1, Q=0, h=0, Text=0*. На границах прямоугольника задайте единичную температуру. Для окружности в поле `r` введите формулу *t*, где переменная `t` используется для обозначения времени, и задайте *h=1*.

Задайте распределение температуры в начальный момент времени, и интервал времени, в который следует найти приближенное решение. Для этого выберите пункт `Solve - Parameters...` Появляется диалоговое окно `Solve Parameters` (рис. 8), вид которого соответствует типу решаемой задачи. Установите в строке `Time` вектор моментов времени 0:0.05:1, а в строке `u(t0)` задайте нулевое начальное распределение температуры, которое в общем случае может зависеть от *x* и *y*.

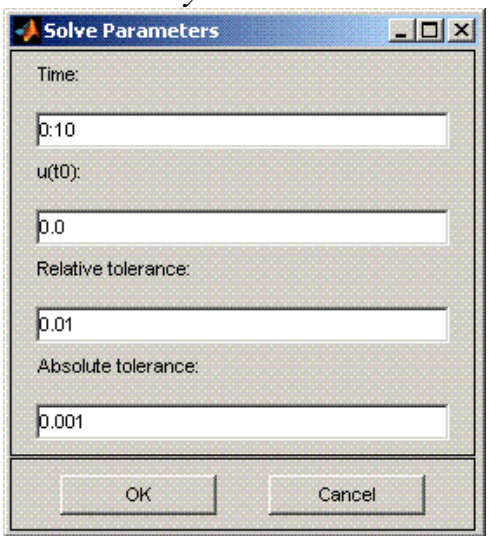


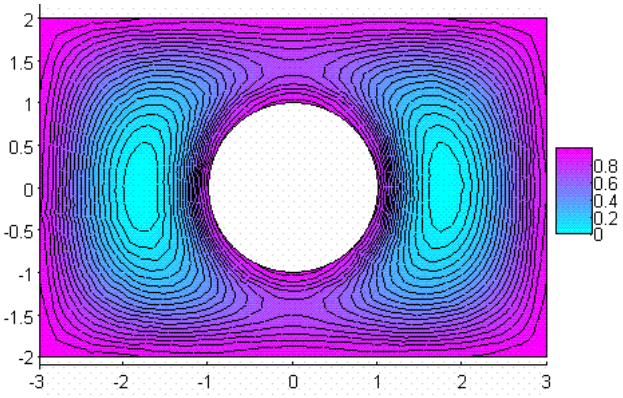
Рис. 8

Если потребуется найти решение не в равноотстоящие моменты времени, то в поле `u(t0)` можно задать вектор значений, например, [0 0.01 0.1 0.2 0.5 0.75 1] или можно использовать одну из функций MatLab генерирующую вектор.

Инициализируйте триангуляцию, уменьшите шаг сетки в несколько раз и решите задачу. Решение займет некоторое время, которое зависит от производительности компьютера. Процесс решения сопровождается выводом информации в командное окно.

```
53 successful steps
0 failed attempts
108 function evaluations
1 partial derivatives
13 LU decompositions
107 solutions of linear systems
```

В окне pdetool появляется распределение температуры в области в конечный момент времени



Для того чтобы проследить за динамикой процесса, следует установить в диалоговом окне `Plot Selection` флаг `Animation` и воспользоваться

кнопкой Options для определения параметров анимации. Откроется окно Animation Options, изображенное на следующем рисунке

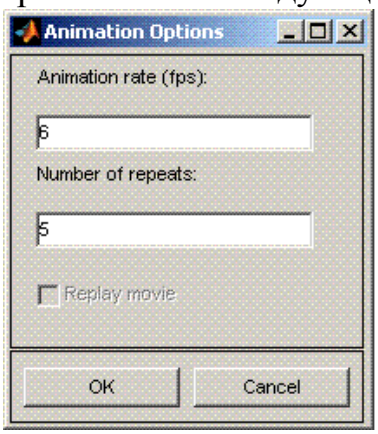


Рис. 9

Строка Animation rate (fps) служит для задания числа кадров в секунду (frames per second). Число повторов устанавливается в поле Number of repeats. Введите желаемые значения или оставьте те, что используются по умолчанию. Закройте окно кнопкой ОК и нажмите кнопку Plot в окне Plot Selection. Динамическое распределение температуры в области отображается в отдельном графическом окне.

Чтобы посмотреть отдельные кадры анимации их следует сохранить в массив. Для этого служит меню Plot – Export Movie. Открывается окно Export, в котором следует ввести имя массива, в элементах которого будут храниться кадры анимации. Например, введем имя MyMovie. Закроем окно кнопкой ОК, в окне Animation options установим число повторов равное единице и еще раз решим задачу. Во время решения все кадры будут сохранены в наш массив.

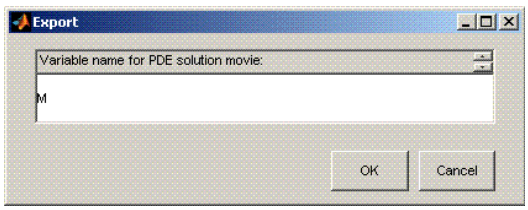


Рис. 10

Теперь мы можем преобразовать любой кадр, например четвертый, в массив данных командой

[X,Map] = frame2im(MyMovie(4));

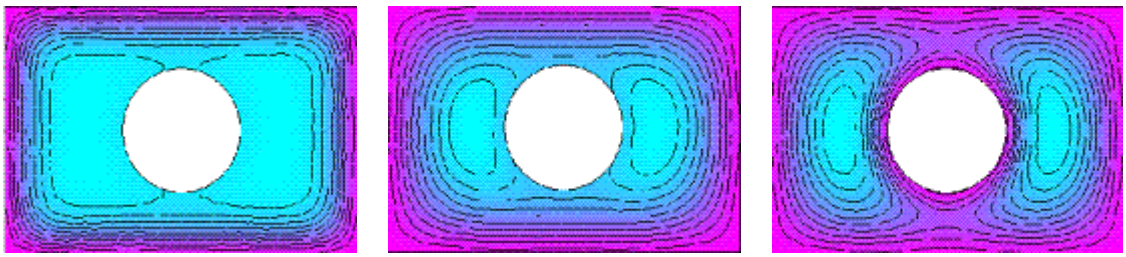
Функция **frame2im** преобразует единичный кадр анимации в числовой массив X и возвращает массив палитры цветов Map. Чтобы отобразить массив X в графическом окне мы можем выполнить команду

image(X)

Последовательно выполняя команды

[X,Map] = frame2im(MyMovie(5)); image(X)

с переменным номером кадров, вы можете построить (и сохранить в графический файл) любой кадр анимации. На следующем рисунке показано распределение температуры в моменты времени *t=0.1, t=0.45, t=1*.



Системы дифференциальных уравнений

Функции, входящие в состав PDE Toolbox, позволяют решить систему дифференциальных уравнений произвольной размерности. Среда pdetool оперирует только с системой второго порядка:

$$-\nabla\cdot(c_{11}\nabla u_1)-\nabla\cdot(c_{12}\nabla u_2)+a_{11}u_1+a_{12}u_2=f_1;$$
$$-\nabla\cdot(c_{21}\nabla u_1)-\nabla\cdot(c_{22}\nabla u_2)+a_{21}u_1+a_{22}u_2=f_2;$$

Ее решением является вектор-функция $(u_1(x,y),u_2(x,y))$. Для однозначного определения решения необходимо задать граничные условия. На различных частях границы можно задавать условия Дирихле

$$h_{11}u_1+h_{12}u_2=r_1$$
$$h_{21}u_1+h_{22}u_2=r_2$$

Неймана

$$\mathbf{n}\cdot\left(\begin{pmatrix}c_{11} & c_{12}\\ c_{21} & c_{22}\end{pmatrix}\nabla u\right)+\begin{pmatrix}q_{11} & q_{12}\\ q_{21} & q_{22}\end{pmatrix}\begin{pmatrix}u_1\\ u_2\end{pmatrix}=\begin{pmatrix}g_1\\ g_2\end{pmatrix}$$

или смешанные граничные условия более сложного вида (см. справку).

Выбор опции `Generic System` в подменю `Application` меню `Options` (или в раскрывающемся списке на панели инструментов) настраивает среду pdetool на решение подобной системы уравнений. Диалоговое окно `PDE Specification` позволяет задать коэффициенты системы уравнений (рис. 11).

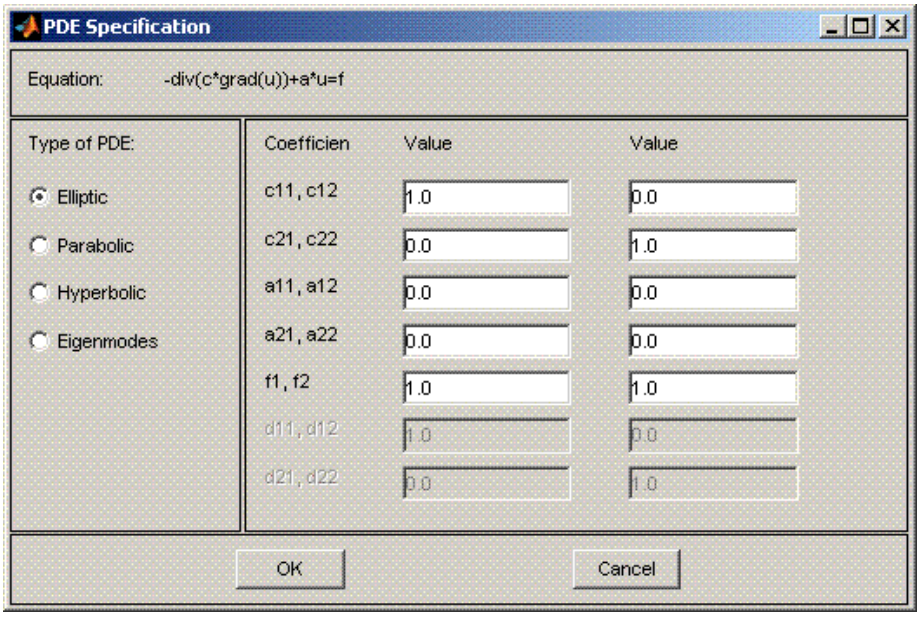


Рис. 11

В первом столбце `Value` этого окна сверху вниз задаются коэффициенты `c11, c21, a11, a21, f1`, а во втором – оставшиеся коэффициенты. Как видим, можно решать не только эллиптические (стационарные) системы, но и нестационарные (гиперболические и параболические), а также задачи на собственные значения.

Триангуляция двумерной области выполняется аналогично тому, как это делается для одного уравнения.

Пример 4. Решим систему дифференциальных уравнений в частных производных второго порядка

$$\frac{\partial^2 u}{\partial x^2}+\frac{\partial^2 u}{\partial y^2}=-q_0$$

$$\frac{\partial^2 v}{\partial x^2}+\frac{\partial^2 v}{\partial y^2}=-\frac{u}{D}$$

в единичном круге с граничными условиям $u|_{\partial\Omega}=0, v|_{\partial\Omega}=0$.

Вначале в среде pdetool выберем тип решаемой задачи – `Generic System` и откроем окно `PDE Specification` (рис. 11), в котором следует задать коэффициенты системы. В нашем случае для первого уравнения следует задать `c11=1, c12=0, a11=0, a12=0, f1=1`. Для второго уравнения задаем `c21=0, c22=1, a21=-1, a22=0, f2=0`. Эти значения соответствуют системе с q_0=1 и D=1.

Окно `Boundary Condition` (рис. 12) содержит переключатели `Neumann, Dirichlet` и `Mixed` для выбора одного из трех типов условий, перечисленных выше, и строки ввода для задания коэффициентов граничных условий. Нулевым значениям искомых функций на границе соответствуют условия Дирихле с параметрами **h11=1, h12=0, r1=0 и h21=0, h22=1, r2=0**.

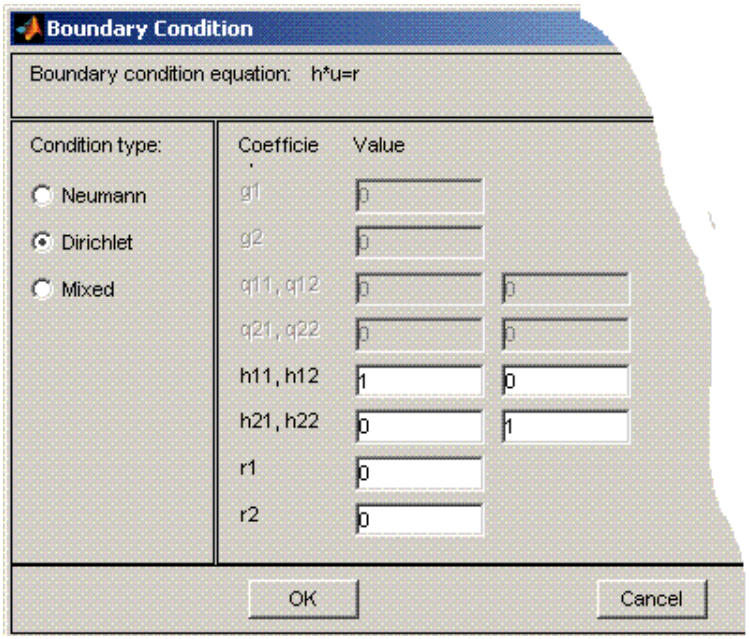
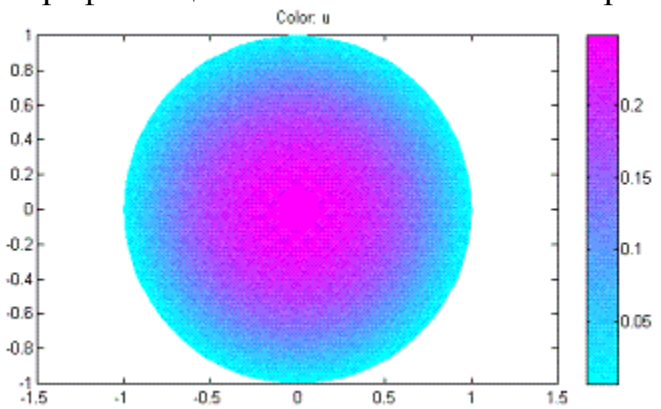
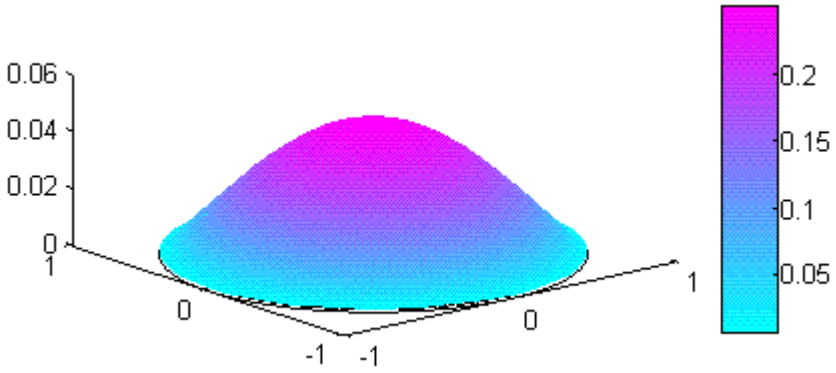


Рис. 12

Задаем, как обычно, триангуляцию области и решаем задачу. В окне pdetool получаем контурный график с цветовой заливкой и палитрой цветов справа.



По умолчанию, строится график первой функции (в нашем случае $u(x,y)$). Диалоговое окно `Plot Selection`, открываемое меню `Plot – Parameters`, предоставляют возможность визуализировать любую (первую или вторую) компоненты решения (они обозначены через **u** и **v**). Установите в этом окне флажок `Height (3-D plot)`, в поле напротив него в столбце `Property` выберите **v** и нажмите кнопку `Plot`. В графическом окне будет построен график функции $v(x,y)$.



Для нашей задачи известно точное решение. Оно имеет вид

$$u(x,y)=\frac{q_0}{4}\left(1-x^2-y^2\right)$$

$$v(x,y)=\frac{q_0}{64D}\left(x^4+2x^2y^2+y^4-4x^2-4y^2+3\right)=$$

$$=\frac{q_0}{64D}\left(x^2+y^2-1\right)\left(x^2+y^2-3\right)$$

Действительно, $u''_{xx}=-\frac{q_0}{2}, u''_{yy}=-\frac{q_0}{2}, u''_{xx}+u''_{yy}=-q_0$. Далее

$$v''_{xx}=\frac{q_0}{64D}\left(12x^2+4y^2-8\right),\quad v''_{yy}=\frac{q_0}{64D}\left(12y^2+4x^2-8\right)$$

и

$$v''_{xx}+v''_{yy}=\frac{q_0}{4D}\left(x^2+y^2-1\right)=-\frac{u}{D}.$$

Очевидно, что оба граничных условия удовлетворены.

Чтобы проверить полученное в pdetool приближенное решение создадим функцию, выполняющую вычисления функции *v(x,y)* по приведенной выше формуле

```
function w=CirclePlateSettlement(x,y)
% точное решение 1 ДУЧП примера 4
q0=1;
D=1;
w=q0/64/D*(x.^4 + 2.*x.^2.*y.^2+y.^4-4.*x.^2-4.*y.^2+3);
```

Теперь мы можем построить контурный график разности функции $v(x,y)$ и точного решения. Для этого откроем окно `Plot Selection` (рис. 13) и в первом поле столбца `Property` выберем `user entry`, а в первом поле столбца `user entry` введем формулу `v- CirclePlateSettlement(x,y)` и нажмем кнопку `Plot`.

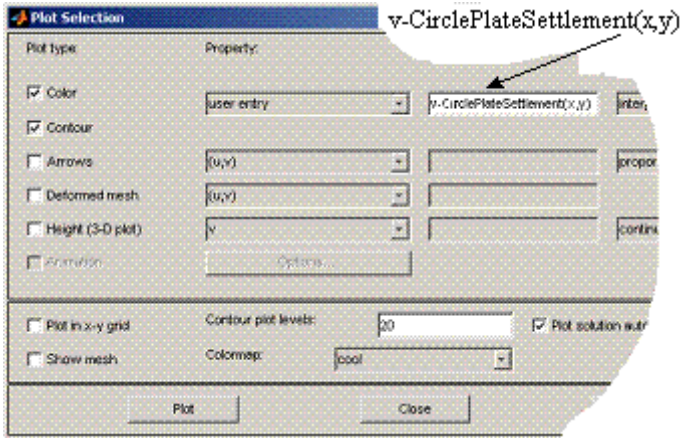
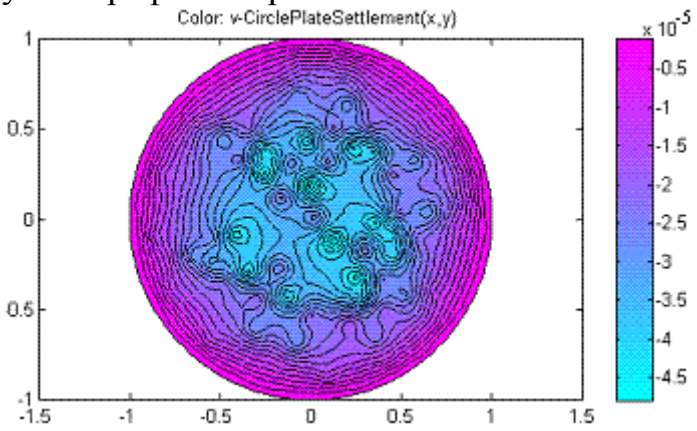


Рис. 13

В окне pdetool получим график погрешности



Числовой множитель 10^{-5} вверху палитры показывает, что погрешность весьма мала. Сохраним наше решение в файле `CirclePlatePde.m`

Нелинейные уравнения.

В среде `pdetool` можно решать нелинейные дифференциальные уравнения в частных производных

Пример 5. Минимальные поверхности. Решим ДУЧП следующего вида

$$-\nabla\cdot\left(\frac{1}{\sqrt{1+|\nabla u|^2}}\nabla u\right)=0$$

в единичном круге $\Omega=\{(x,y):\ x^2+y^2\le 1\}$ со значениями $u=x^2$ на границе $\partial\Omega$. Оно определяет форму минимальной поверхности, край которой находится на заданной высоте над границей круга.

1. Вначале укажем тип решаемого уравнения `Generic Scalar`.
2. Нарисуем единичный круг так, как мы это делали раньше.
3. Перейдем в режим задания граничных условий. Выполним команду меню `Edit - Select All` для выбора всех участков границы круга. Выполним двойной щелчок по контуру границы и зададим условие Дирихле в поле `r` в виде `x.^2`.
4. Уравнение в частных производных, описывающее минимальную поверхность, имеет вид

$$div\left(\frac{grad\,u}{\sqrt{1+(u'_x)^2+(u'_y)^2}}\right)=0$$

Откроем диалоговое окно `PDE Specification` и в поле **c** введем выражение `1./sqrt(1+ux.^2+uy.^2)`, а в полях `a` и `f` введем ноль. Проверьте, что установлен переключатель типа уравнения `Elliptic`.

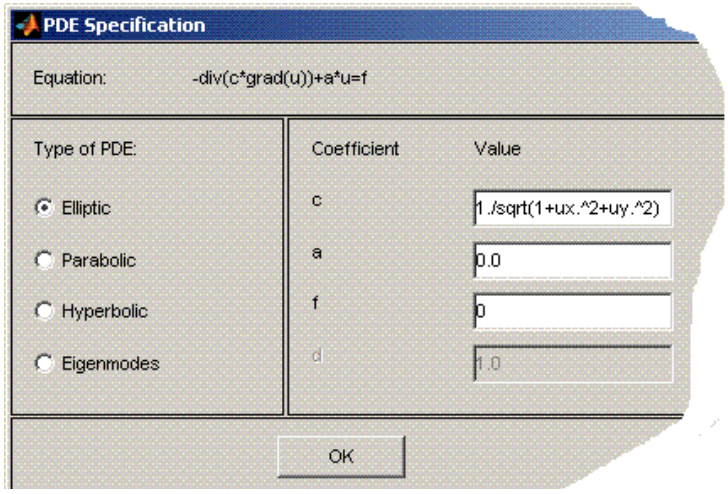


Рис. 14

5. На следующем шаге задайте триангуляцию
6. Перед тем как решать уравнение, выберите меню `Solve - Parameters` и в открывшемся окне установите флажок `Use nonlinear solver`,

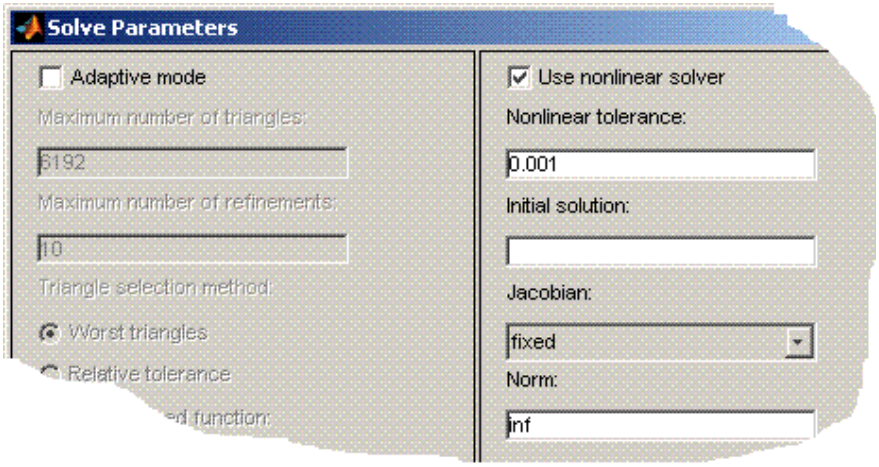
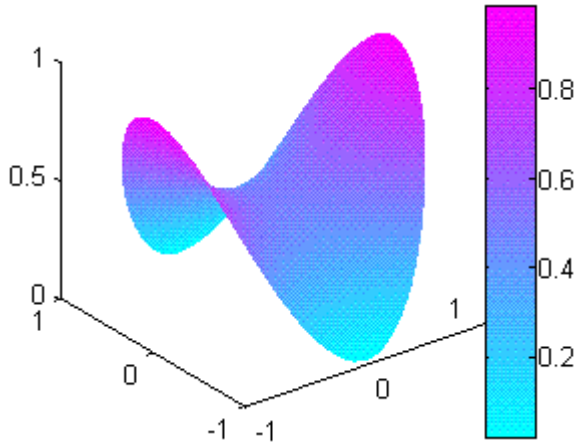


Рис. 15

а в поле `Nonlinear tolerance` введите 0.001.

7. Теперь решим уравнение. В диалоговом окне `Plot Selection`, установите флажок `Height (3-D plot)` и нажмите кнопку `Plot` для построения графика поверхности



Сохраним решение pdetool в файле `pdeminsurf.m`

4.1.2 Решение ДУЧП с помощью функций PDE Toolbox

Среда pdetool с графическим интерфейсом пользователя предназначена для облегчения доступа к ядру PDE Toolbox – набору функций, реализующих основные этапы решения задачи: задания и разбиения области, задание уравнения, граничных условий и визуализации результата.

Решение модельных задач

Пример 1. Использование функций PDE Toolbox рассмотрим на примере решения эллиптического уравнения

$$\frac{\partial^2 u}{\partial x^2}+\frac{\partial^2 u}{\partial y^2}=-20\cdot\sin\pi x\cdot\sin 2\pi y$$

в прямоугольнике с центром в начале координат, размерами 2×1 и нулевыми граничными условиями $u\big|_{\Gamma}=0$.

Последовательность решения задачи такова:

- задание геометрии области и сохранение ее в переменных рабочей среды;
- задание граничных условий и сохранение их в переменных рабочей среды;
- триангуляция области и сохранение результатов в переменных рабочей среды;
- присваивание некоторым переменным значений коэффициентов решаемого уравнения и создание строковых выражений функций, используемых в уравнении;
- вызов функции – решателя ДУЧП (для разных типов уравнений и систем они разные) с передачей ей аргументов, содержащих геометрию среды, граничные условия, информацию о триангуляции, а также значения коэффициентов и функций – членов уравнения; сохранение результата в переменную среды;
- вызов функции визуализации, аргументами которой являются переменные, содержащие информацию о триангуляции и вектор результата.

Разберем последовательно шаги решения задачи, описанные выше. При этом первые два шага пока выполним в среде pdetool с последующим экспортом информации в переменные рабочей среды.

Настроим среду pdetool на решение скалярного эллиптического уравнения (Рис. 16),

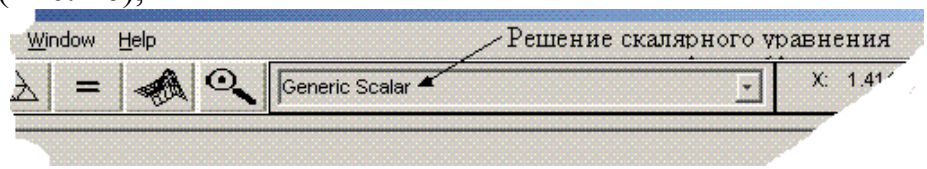


Рис. 16

установив переключатель `Elliptic` в окне `PDE Specification` (см. рис. 14).

Нарисуем в среде pdetool прямоугольник с центром в начале координат шириной два и высотой один и зададим нулевые граничные условия. Затем экспортируем информацию в переменные рабочей среды. Для этого выполним команду меню `Boundary` – `Export Decomposed Geometry, Boundary Cond's`.... Откроется диалоговое окно `Export`, в котором по умолчанию стоят имена `g` и `b` – первое для массива, хранящего информацию о геометрии границы, и второе – для хранения матрицы граничных условий.

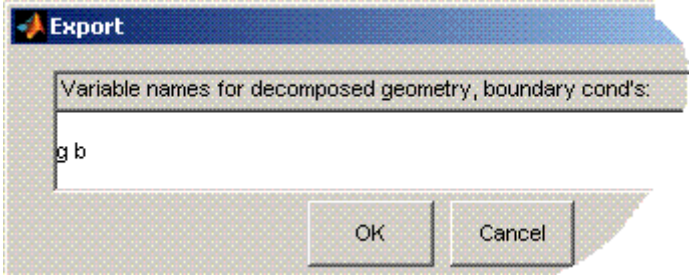


Рис. 17

Нажмем ОК, и в рабочей среде появятся двумерные массивы `g` и `b`. Посмотрим их содержимое

>>g

```
g =
    2.0000    2.0000    2.0000    2.0000
    1.0000   -1.0000   -1.0000    1.0000
   -1.0000   -1.0000    1.0000    1.0000
   -0.5000   -0.5000    0.5000    0.5000
   -0.5000    0.5000    0.5000   -0.5000
         0         0         0         0
    1.0000    1.0000    1.0000    1.0000
```

Двумерный массив g называется матрицей декомпозиционной геометрии (Decomposed Geometry matrix). Она хранит информацию о частях границы области, которые могут быть отрезком, частью дуги окружности или эллипса. Каждому столбцу матрицы соответствует участок границы.

Прямоугольник представляется четырьмя отрезками и поэтому в матрице 4 столбца. Двойка в первом элементе столбца означает, что столбец соответствует отрезку, второй и третий элементы содержат абсциссы начала и конца отрезка, а четвертый и пятый — ординаты. Функции пакета pdetool предполагают, что вся область разбита на некоторые непересекающиеся подобласти, имеющие собственные идентификаторы (номера). Идентификаторы подобластей, расположенных слева и справа от элемента границы (отрезка или дуги), записаны в шестом и седьмом элементе. В нашем примере есть только одна область – прямоугольник (идентификатор единица). Слева от отрезков, представляющих границу прямоугольника, областей нет – идентификатор ноль.

Двумерный массив b называется матрицей граничных условий. Каждый ее столбец соответствует части границы. В нашем случае область является прямоугольником с одинаковыми нулевыми граничными условиями Дирихле, и матрица состоит из четырех столбцов.

>>b

```
b =
     1     1     1     1
     1     1     1     1
     1     1     1     1
     1     1     1     1
     1     1     1     1
     1     1     1     1
    48    48    48    48
    48    48    48    48
    49    49    49    49
    48    48    48    48
```

Столбец условно делится на две части, верхняя часть столбца матрицы граничных условий содержит информацию о типе граничных условий и длинах формул. В нижней части записаны коэффициенты и формулы граничных условий, причем каждый символ строки с формулой хранится в последовательно идущих элементах, содержащих коды символов. В случае одного уравнения и граничного условия Дирихле коды символов формулы хранятся в элементах столбца, начиная с седьмого. Преобразуем их в символы с помощью функции char

char(b(7:end,:))

```
ans =
0000
0000
```

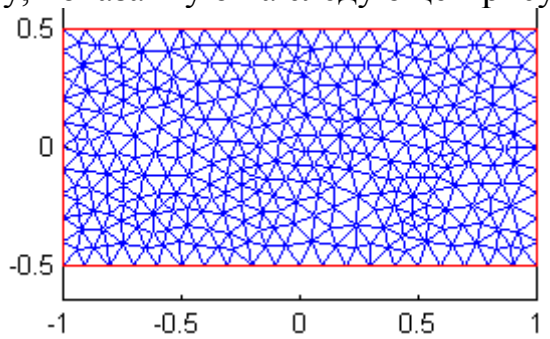
```
1111
0000
```

Первые два элемента зарезервированы под коэффициенты условия Неймана (они равны нулю, поскольку поставлено условие Дирихле). Третий элемент содержит значение `h` (в нашем случае единица), а остальные являются символами правой части выражения `h*u=r` (в нашем случае ноль).

Триангуляция области выполняется функцией **[p, e, t] = initmesh(g)**, которая разбивает плоскую область на треугольники. Ее входным аргументом является матрица декомпозиционной геометрии. Три выходных аргумента (матрицы) **p, e, t** содержат информацию о триангуляции (их структуру мы рассмотрим позже). Функция **initmesh** может содержать дополнительные аргументы, задаваемые парами: название свойства, значение. Например, свойство `Hmax` устанавливает максимально допустимое значение длины стороны треугольного элемента. Полученную сетку можно отобразить в отдельном окне при помощи функции **pdemesh**, входными аргументами которой являются вышеперечисленные массивы. Например, последовательность команд

```
>>[p, e, t] = initmesh(g, 'Hmax', 0.1);
>> pdemesh(p,e,t); axis equal
```

создает и выводит сетку, показанную на следующем рисунке



На следующем шаге мы создаем переменные, которые будут коэффициентами уравнения $-div(c \cdot grad(u)) = f$. Если бы мы определяли уравнение в среде pdetool, то открыли бы окно `PDE Specification`

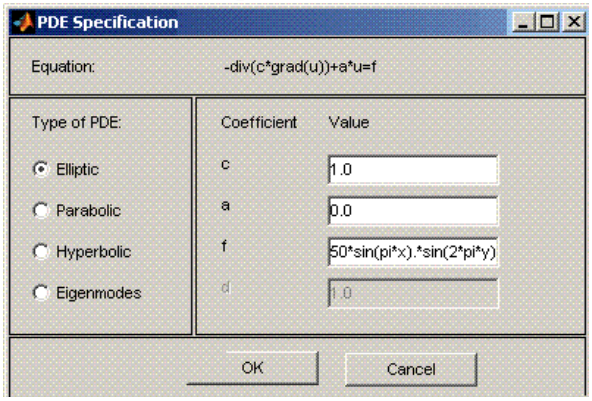


Рис. 18

в котором ввели бы значения коэффициентов `a` и `c` уравнения и выражение для функции `f` – правой части уравнения. Для работы с функциями пакета мы в рабочей среде создаем эти переменные, при этом выражение для функции создаем в строковом виде

>>a=0; % коэффициент уравнения
>>c=1; % коэффициент уравнения
>>f='20*sin(pi*x).*sin(2*pi*y)'; % Строка с формулой правой части уравнения

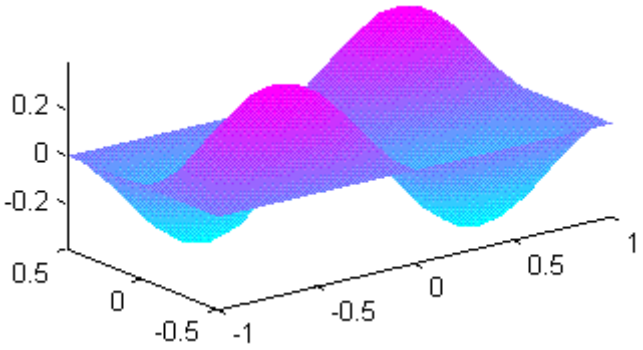
Теперь можно решать задачу. Для решения эллиптических уравнений используется функция **assempde**. Ей следует передать аргументы – матрицу граничных условий `b`, массивы `p, e, t,` содержащие информацию о триангуляции, и коэффициенты уравнения `c, a, f`.

>>u=assempde(b,p,e,t,c,a,f);

Функция **assempde** возвращает вектор решений – значения искомой функции в узлах сетки.

На последнем шаге можно построить график решения. Функция **pdesurf** рисует поверхность функции *u(x,y)*, используя информацию о триангуляции, хранящуюся в переменных `p` и `t` и вектор решения `u`.

>>pdesurf(p,t,u);



Решение приведенной задачи удобно организовать в форме сценария FuncPDE_1.m

```
% Сценарий решения первого ДУЧП функциями PDE Toolbox
% Предполагается, что матрица декомпозиционной геометрии g
% и матрица граничных условий b уже созданы.
% 1. Триангуляция с использованием массива декомпозиционной геометрии g
[p, e, t] = initmesh(g, 'Hmax', 0.1);
pdemesh(p,e,t);          % построение сетки для проверки
% 2. Задание коэффициентов уравнения
a=0;
c=1;
% 3. Определение строки с формулой правой части уравнения
f='5*pi^2*sin(pi*x).*sin(2*pi*y)';
% 4. Решение эллиптического уравнения
u=assempde(b,p,e,t,c,a,f);% используем матрицу граничных условий b
% 5. Построение графика поверхности решения
pdesurf(p,t,u);
axis equal;
```

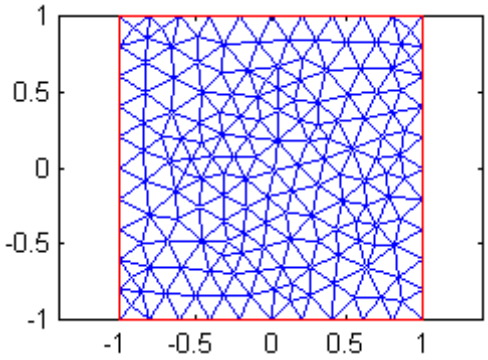
Меняя функцию правой части *f(x,y)*, мы можем для заданной формы области (прямоугольник размерами 2×1) и нулевых граничных условий $u\big|_{\Gamma}=0$ получать решения уравнения Пуассона. Если нужно изменить форму области или граничные условия, то следует изменить матрицу декомпозиционной геометрии `g` и матрицу граничных условий `b`. Их можно создавать, экспортируя из среды pdetool, или создавать вручную. Вместо этих матриц можно также использовать функции специальной структуры. О них мы поговорим позже.

Пример 2. Рассмотрим пример решения уравнения колебания квадратной мембраны, закрепленной на контуре. Эта задача отличается от предыдущей тем, что необходимо задать начальные условия и моменты времени в которые следует определять решение.

Нарисуем в среде pdetool квадрат с центром в начале координат со стороной 2 ($-1 \le x, y \le 1$) и зададим нулевые граничные условия $u|_{\Gamma} = 0$. Затем экспортируем информацию в переменные рабочей среды с помощью команды меню `Boundary – Export Decomposed Geometry, Boundary Cond's...`. Назовем матрицу декомпозиционной геометрии `gw` и матрицу граничных условий `bw`.

Выполним триангуляцию области и построим сетку командами

```
>>[p, e, t] = initmesh(gw, 'Hmax', 0.1);
>> pdemesh(p,e,t); axis equal
```



Матрица `p`, которую возвращает функция **initmesh**, содержит две строки – *x* и *y* координаты узлов сетки.

```
>>p
p =
Columns 1 through 8
 1.0000   -1.0000   -1.0000    1.0000    0.8000    0.6000    0.4000
-1.0000   -1.0000    1.0000    1.0000   -1.0000   -1.0000   -1.0000

...
Column 177
 -0.3128
 -0.0361
```

Для экономии места мы не приводим все элементы вектора `p`. Создадим вектора, содержащие *x* и *y* координаты узлов.

```
x=p(1,:)';
y=p(2,:)';
```

Они нам нужны для задания начальных условий в узлах сетки.

```
u0=sin(pi*x).*sin(pi*y);% начальное смещение точек мембраны
ut0= 0;       % начальная скорость точек мембраны
```

Теперь определим моменты времени, в которые будем находить решение

```
n=26;
tlist=linspace(0,5,26)  % вектор моментов времени
tlist =
  Columns 1 through 8
         0    0.2000    0.4000    0.6000   ...
```

Для решения гиперболического уравнения используется функция **hyperbolic** пакета PDE Toolbox. Ее аргументами являются:

– вектора u_0, u_{t0} начальных значений (начальных смещений и начальных скоростей) в узлах сетки;
– вектор t_{list} моментов времени, в которые определяется решение;
– матрица граничных условий bw;
– вектора **p, e, t,** возвращаемые функцией **initmesh**, содержащие информацию о триангуляции;
– коэффициенты c, a, f, d уравнения $d\frac{\partial^2 u}{\partial t^2} - div(c \cdot grad(u)) + a \cdot u = f$, те же, что мы вводили бы в окне PDE Specification.

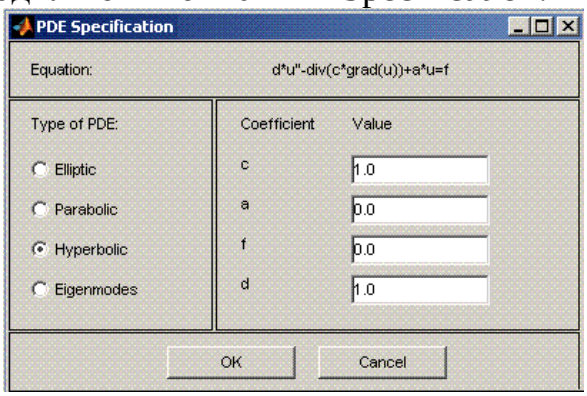


Рис. 19

В нашей задаче c=1, a=0, f=0, d=1. Выполним команду решения гиперболического уравнения

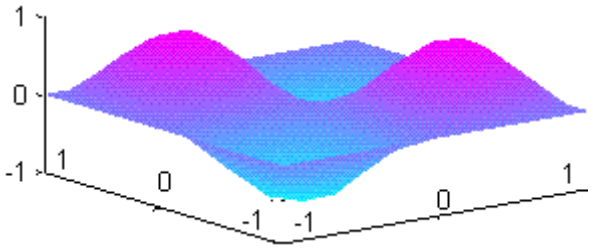
uw=hyperbolic(u0,ut0,tlist,bw,p,e,t,1,0,0,1);

Через некоторое время в командном окне появятся сообщения об успешном решении задачи

```
696 successful steps
70 failed attempts
1534 function evaluations
1 partial derivatives
180 LU decompositions
1533 solutions of linear systems
```

Функция **hyperbolic** возвращает решение в виде матрицы, каждый столбец которой представляет вектор смещения узлов сетки в некоторый момент времени. Например, для построения поверхности мембраны в 5-й момент времени (t_5=tlist(5)=0.8) можно использовать команду

pdesurf(p,t,uw(:,5));

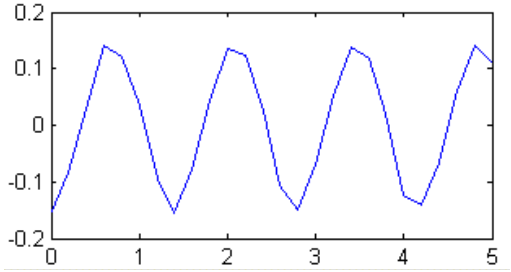


Наоборот, каждая строка матрицы uw представляет смещение некоторой точки (узла) в последовательные моменты времени. Например, 45 – й узел сетки имеет координаты

>> [x(45), y(45)]

```
ans =
    0.8710   -0.8741
```

График смещения точки во времени будет иметь вид
plot(tlist,uw(45,:));



Если бы сетка была более густой, а моменты времени, в которые мы определяем решение – короче, то предыдущий график получился бы более гладким. Функция **refinemesh** служит для уменьшения шага сетки. Первым входным аргументом является матрица декомпозиционной геометрии, затем передаются массивы **p, e, t** с информацией о триангуляции. В выходных аргументах возвращаются массивы, соответствующие измельченной сетке. По умолчанию сторона каждого элемента делится пополам, каждый треугольный элемент исходной сетки разбивается на четыре части. Команды, приведенные ниже, производят вышеописанное уменьшение шага сетки дважды

[p1,e1,t1]=refinemesh(gw,p,e,t);
[p2,e2,t2]=refinemesh(gw,p1,e1,t1);
pdemesh(p2,e2,t2); axis equal

Интересно, что 45 – й узел нового разбиения имеет те же координаты, что и 45 – й предыдущего разбиения

[p2(1,45),p2(2,45)]

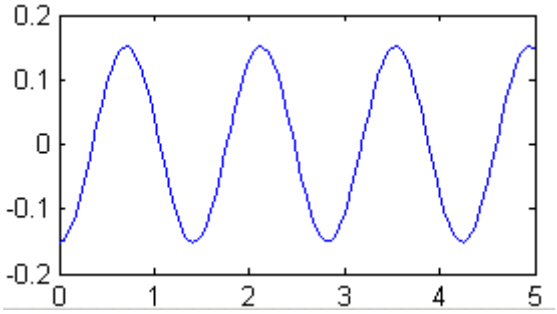
```
ans =
    0.8710   -0.8741
```

Уплотним моменты времени вектора `tlist`, пересчитаем вектора начальных условий u_0, u_{t0} для узлов нового разбиения и снова решим задачу (это займет больше времени, чем предыдущее решение)

n=101; tlist=linspace(0,5,n);
x=p2(1,:)'; y=p2(2,:)';
u0=sin(pi*x).*sin(pi*y); ut0= 0;
uw=hyperbolic(u0,ut0,tlist,bw,p2,e2,t2,1,0,0,1);

График колебаний во времени 45 – го узла новой сетки, имеющего координаты (0.8710, -0.8741), будет иметь более гладкий вид

plot(tlist,uw(45,:));



Используя матрицу решения `uw` можно построить анимацию процесса колебания мембраны. В следующем файле – сценарии создается массив кадров анимации M, который затем проигрывается функцией **movie** (используем

первый вектор моментов времени `tlist` и решение **uw=hyperbolic(u0,ut0,tlist,bw,p,e,t,1,0,0,1)**, полученное при первоначальном разбиении области)

```
% файл animw2.m
% сценарий создания анимации процесса колебания мембраны
% используется матрица uw решения в узлах сетки триангуляции,
% описываемой векторами p и t
umax=max(max(uw));      % максимальный элемент матрицы смещений
umin=min(min(uw));      % минимальный элемент матрицы смещений
newplot          % создаем новое графическое окно
for i=1:n,
    pdesurf(p,t,uw(:,i));      % рисуем кадр – график поверхности
    axis([-1 1 -1 1 umin umax]); % приводим кадр к общему масштабу
    M(i)=getframe;      % запоминаем кадр в массив
end
movie(M,4);           % проигрываем массив кадров 4 раза
```

Выполните команду

animw2

и в графическом окне будет показан процесс колебания мембраны. При этом массив кадров анимации `M` будет создан в рабочем пространстве. Чтобы еще раз посмотреть анимацию, можно, не запуская сценарий, выполнить команду

movie(M, кол-во повторений);

Заметим, что функция **max(A)**, где A – матрица, возвращает строку максимальных значений по столбцам A. Для того, чтобы найти максимальный элемент всей матрицы, мы в сценарии применяем дважды функцию **max**. Например

A=[1 7 3; -1 4 9; 5 0 3]

```
A =
     1     7     3
    -1     4     9
     5     0     3
```

max(A)

```
ans =
     5     7     9
```

max(max(A))

```
ans =
     9
```

Зададим начальное смещение мембраны в форме пирамиды и соберем все необходимые команды в один файл – сценарий `animw3.m`

```
% Сценарий решения уравнения колебаний мембраны функциями PDE Toolbox
% Предполагается, что матрица декомпозиционной геометрии gw
% и матрица граничных условий bw уже созданы в рабочем пространстве
%
% Триангуляция с использованием матрицы декомпозиционной геометрии gw
[p, e, t] = initmesh(gw, 'Hmax', 0.05);
% pdemesh(p,e,t); axis equal;
x=p(1,:)';
y=p(2,:)';
% начальное смещение точек мембраны в форме пирамиды
u0=(2-abs(x)-abs(y)-abs(abs(y)-abs(x)))/2;
ut0= 0;                 % начальная скорость точек мембраны 0
n=31; tlist=linspace(0,3,n);  % вектор моментов времени
uw=hyperbolic(u0,ut0,tlist,bw,p,e,t,1,0,0,1);   % решение уравнения
% анимация решения
umax=max(max(uw));      % максимальный элемент матрицы смещений
umin=min(min(uw));      % минимальный элемент матрицы смещений
newplot                 % создаем новое графическое окно
```

```
clear M;
for i=1:n,
    pdesurf(p,t,uw(:,i));           % рисуем кадр – график поверхности
    axis equal;
    axis([-1 1 -1 1 umin umax]);    % приводим кадр к общему масштабу
    M(i)=getframe;          % запоминаем кадр в массив
end
movie(M,4);                 % проигрываем массив кадров 4 раза
```

На следующем рисунке показана форма мембраны в различные моменты времени

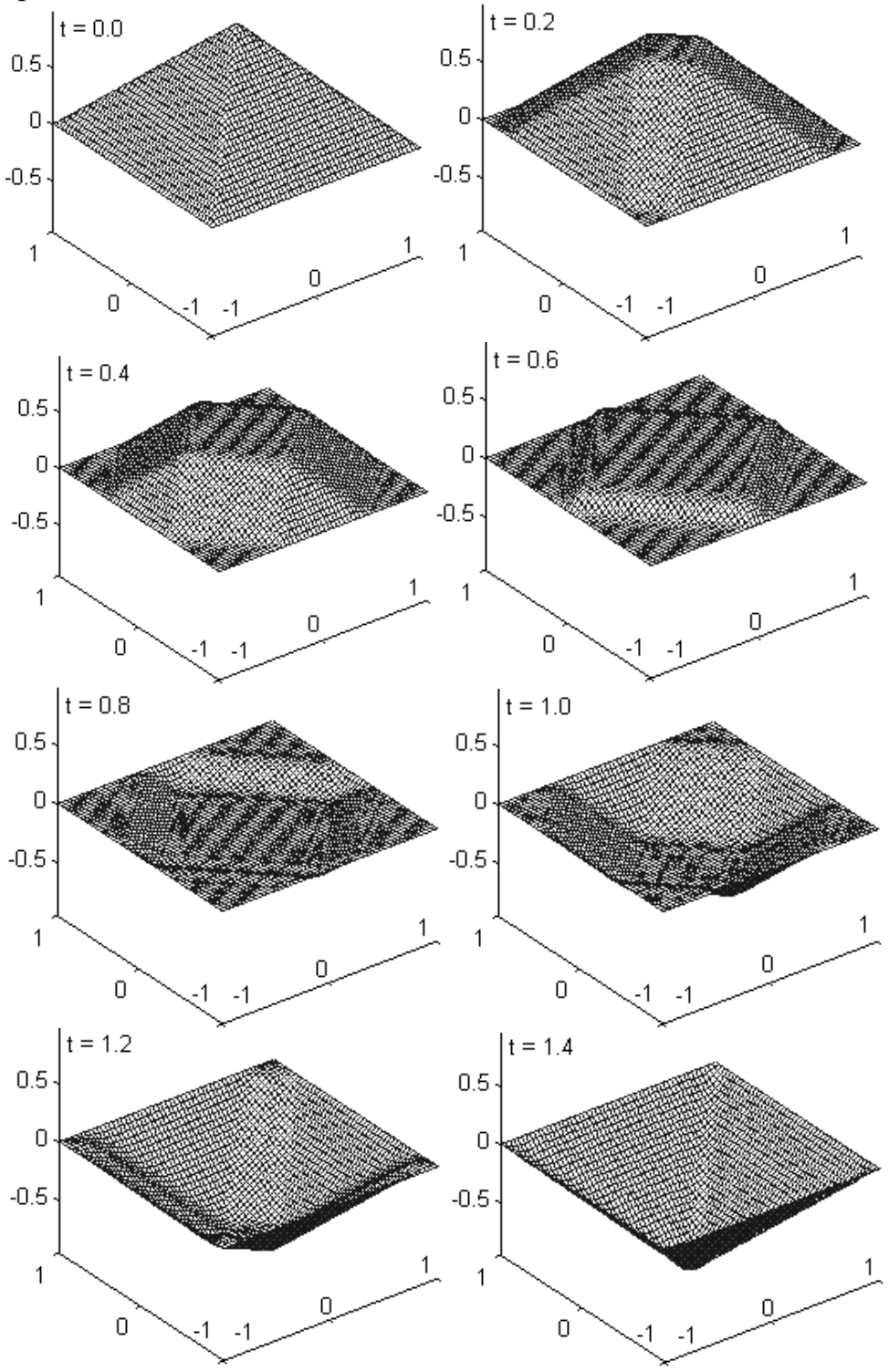


Рис. 20

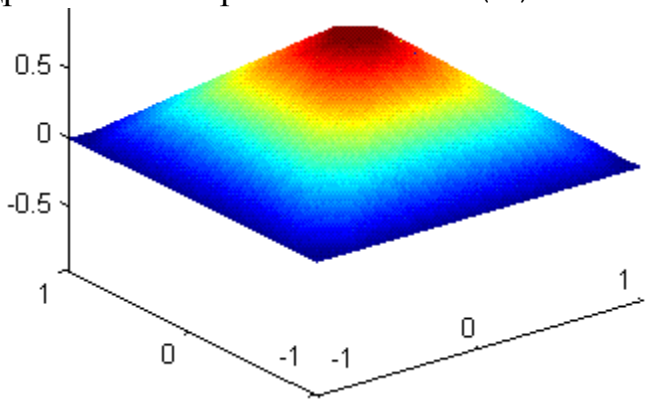
Для построения этих графиков мы создали сценарий `drawFrame1.m`, в котором переменная `myframe` содержит номер отображаемого кадра.

```
% сценарий рисование кадра с номером myframe
myframe=15;
pdeplot(p,[],t,'xydata',uw(:,myframe),'xystyle','off', ...
     'zdata',uw(:,myframe),'zstyle','continuous','colorbar','off',
     'mesh','on','xygrid','on');
axis equal;      % равенство длин засечек на осях
                 % задаем перед заданием пределов осей
axis([-1 1 -1 1 umin umax]);
colormap bone;
s=sprintf('t = %2.1f',tlist(myframe));
text(-1,0.9,0.9,s);       % вывод момента времени кадра
```

Функция **pdeplot(…)** является общей функцией, предназначенной для построения графиков решения, полученного с помощью функций PDE TOOLBOX. Функция **pdesurf(…)**, используемая нами ранее, является ее частным случаем. Используя ее, кадры анимации можно строить следующим образом

```
colormap 'default';
pdesurf(p,t,uw(:,2));
axis([-1 1 -1 1 umin umax]);
```

Здесь 2 – номер кадра в момент времени `tlist(2)=0.1`.



Опции функции **pdeplot(…)**, использованные нами в сценарии `drawFrame1.m`, предназначены для построения черно-белой каркасной поверхности.

Общие положения решения ДУЧП с использованием функций PDE Toolbox

Здесь мы более подробно разберем шаги решения краевых задач с использованием функций пакета, описанные выше.

Задание геометрии области.

Имеются два способа определения геометрии области. Первый заключается в параметрическом задании частей границ области в файл–функции, имеющей определенный формат. Второй, использованный нами в предыдущих примерах, состоит в задании области в среде pdetool, последующем экспорте информации в переменные рабочей среды и преобразовании в формат, понятный другим функциям пакета.

Нарисуем в среде pdetool прямоугольник с центром в начале координат шириной два и высотой один. Выберем меню `Draw – Export Geometry Description, Set Formula, Labels...` Появится диалоговое окно `Export`, изображенное на следующем рисунке.

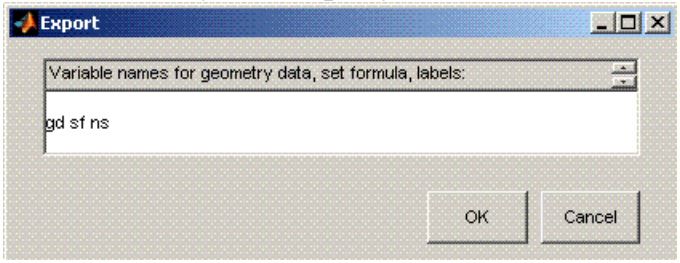


Рис. 21

Оно предлагает сохранить модель (конструктивную блочную геометрию – CSG модель) в глобальных переменных рабочей среды:

- матрицу геометрии области (Geometry Description matrix) в массиве **gd**;
- формулу связи геометрических примитивов в строке **sf**;
- соответствие между столбцами **gd** и названиями областей в **sf** в массиве **ns**.

Нажмите ОК и займитесь изучением содержимого экспортированных массивов.

Число столбцов матрицы геометрии области **gd** совпадает с числом геометрических примитивов, составляющих область. В рассматриваемом примере матрица состоит из одного столбца, соответствующего прямоугольнику.

```
>> gd
gd =
    3.0000    % тип примитива (3 = прямоугольник)
    4.0000    % число сторон многоугольника
    1.0000    % x координата 1-й вершины (правой нижней)
   -1.0000    % x координата 2-й вершины (левой нижней)
   -1.0000    % x координата 3-й вершины (левой верхней)
    1.0000    % x координата 4-й вершины (правой верхней)
   -0.5000    % y координата 1-й вершины (правой нижней)
   -0.5000    % y координата 2-й вершины (левой нижней)
    0.5000    % y координата 3-й вершины (левой верхней)
    0.5000    % y координата 4-й вершины (правой верхней)
```

Первый элемент столбца определяет тип геометрического примитива (всего допустимы четыре типа геометрических примитивов: круг, многоугольник, прямоугольник и эллипс). Первый элемент, равный трем, означает, что данный столбец соответствует прямоугольнику. Во второй строке указано количество сторон многоугольника (прямоугольник является частным случаем многоугольника, который имеет специальный формат хранения геометрии). В остальные элементы столбца записаны координаты вершин прямоугольника – вначале *x* координаты, начиная с правой нижней с обходом по часовой стрелке, потом *y* координаты.

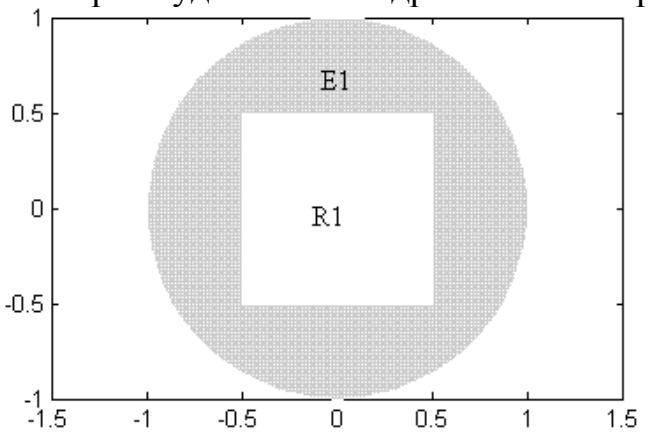
Структура столбцов для разных примитивов приведена в следующей таблице.

	Круг	Многоугольник	Прямоугольник	Эллипс
Номер типа	1	2	3	4
2-я строка	$X_{центра}$	Число сторон	Число сторон	$X_{центра}$
3-я строка	$Y_{центра}$	X_1	X_1	$Y_{центра}$
4-я строка	$R_{круга}$	Y_1	…	$X_{полуось}$
…		…	X_4	$Y_{полуось}$
		…	Y_1	угол поворота
		…	…	
		Y_n	Y_4	

Координаты вершин многоугольника записываются в столбец в порядке их создания. Другие CSG элементы имеют вид

```
>> sf      % формула области (имя единственной подобласти R1)
sf =
R1
>> ns
ns =
    82    % char(82)=R (код первого символа имени R)
    49    % char(49)=1 (код второго символа имени 1)
```

Для пояснения построим еще одну область – единичный круг с центром в начале координат, из которого удаляется квадрат меньшего размера.



Экспортируем, как описано выше, модель в переменные рабочей среды `gd, sf, ns` и изучим полученные матрицы.

```
>>gd
gd =
    4.0000    3.0000
         0    4.0000
         0    0.5000
    1.0000   -0.5000
    1.0000   -0.5000
         0    0.5000
         0   -0.5000
         0   -0.5000
         0    0.5000
         0    0.5000
```

Область сконструирована из двух примитивов и матрица `gd` состоит из двух столбцов. Первый соответствует эллипсу (номер типа 4), второй – прямоугольнику (номер типа 3). Номера типов записаны в первой строке каждого столбца.

В строковой переменной sf записана формула конструирования области – из области круга (эллипса) вычитается область квадрата.

```
>> sf
sf =
E1-R1
```

В массиве ns хранится соответствие между столбцами матрицы gd и названиями областей

```
>> ns
ns =
    69    82
    49    49
```

Столбцы матрицы ns хранят коды символов имен примитивов, из которых сконструирована область – первый столбец E1, второй – R1.

```
char(ns)
ans =
ER
11
```

Если переименовать области и эллипсу дать имя Ellipse, а квадрату – Rect, то после экспорта конструктивной блочной геометрии массив gd останется без изменений, строковая переменная sf будет равна

```
>> sf
sf =
Ellipse-Rect
```

Массив ns будет хранить коды символов имен примитивов

```
>> ns
ns =
    69    82
   108   101
   108    99
   105   116
   112    32
   115    32
   101    32
>> char(ns')
ans =
Ellipse
Rect
```

При этом, второй столбец, соответствующий имени Rect, дополнен пробелами (код символа 32) для выравнивания длин столбцов матрицы ns.

Функции, связанные с триангуляцией и заданием граничных условий, используют другой формат описания геометрии области – матрицу декомпозиционной геометрии (Decomposed Geometry matrix). Она хранит информацию о частях границы области, которые могут быть отрезком, частью дуги окружности или эллипса. Переход к декомпозиционной матрице производится при помощи функции **decsg**, входными аргументами которой являются переменные конструктивной блочной геометрии, в нашем случае gd, sf и ns, а выходным – матрица декомпозиционной геометрии. Каждый столбец этой матрицы соответствует элементу границы. В примерах 1 и 2 матрицу декомпозиционной геометрии мы получали одновременно с экспортом матрицы граничных условий.

Преобразуйте экспортированные величины (рассматриваемая область – прямоугольник, построенный выше) `gd`, `sf` и `ns`, вызвав функцию **decsg** из командной строки:

>> dl=decsg(gd,sf,ns)

```
dl =
    2.0000    2.0000    2.0000    2.0000
    1.0000   -1.0000   -1.0000    1.0000
   -1.0000   -1.0000    1.0000    1.0000
   -0.5000   -0.5000    0.5000    0.5000
   -0.5000    0.5000    0.5000   -0.5000
         0         0         0         0
    1.0000    1.0000    1.0000    1.0000
```

Прямоугольник представляется четырьмя отрезками. Двойка в первом элементе столбца означает, что столбец соответствует отрезку, второй и третий элементы содержат абсциссы начала и конца отрезка, а четвертый и пятый — ординаты. Преобразование в декомпозиционную геометрию предполагает, что вся область разбита на некоторые непересекающиеся подобласти, имеющие собственные идентификаторы (номера). Идентификаторы подобластей, расположенных слева и справа от элемента границы (отрезка или дуги), записаны в шестом и седьмом элементе. В нашем примере есть только одна область – прямоугольник (идентификатор единица). Слева от отрезков, представляющих границу прямоугольника, областей нет – идентификатор ноль.

В случае, когда столбец соответствует дуге эллипса или окружности, элементы столбца будут содержать информацию о центре и радиусе окружности или величине полуосей эллипса и угле поворота.

С помощью функции **pdegplot** можно построить контур области (границу) в отдельном графическом окне. Входным аргументом функции является матрица декомпозиционной геометрии (в нашем случае `dl`)

>>pdegplot(dl); axis equal

Функциям PDE TOOLBOX вместо матрицы декомпозиционной геометрии можно передавать имя файл – функции, вычисляющей координаты точек участков границы области по их параметрическим уравнениям. Из матрицы декомпозиционной геометрии такую функцию можно сгенерировать автоматически при помощи функции **wgeom**. Ее входными аргументом является матрица декомпозиционной геометрии и имя М-файла, в который следует записать файл-функцию. Если файл-функцию создать не удалось, то **wgeom** вернет минус единицу.

Создадим файл-функцию **recgeom**, которая описывает прямоугольную область, задаваемую матрицей `dl`

>>wgeom(dl, 'recgeom')

```
ans =
     3
```

В текущем каталоге MATLAB появится файл `recgeom.m`, структура которого приведена ниже

```
function [x,y]=recgeom(bs,s)
%RECGEOM    Gives geometry data for the recgeom PDE model.
%
```

```
%   NE=RECGEOM возвращает количество граничных сегментов
%
%   D=RECGEOM(BS) для граничного сегмента с номером BS возвращает матрицу
%   из одного столбца каждый элемент которого является
%   Row 1 contains the start parameter value.
%   Row 2 contains the end parameter value.
%   Row 3 contains the number of the left-hand regions.
%   Row 4 contains the number of the right-hand regions.
%
%   [X,Y]=RECGEOM(BS,S) gives coordinates of boundary points.
%   BS specifies the boundary segments and
%   S the corresponding parameter values.
%   BS may be a scalar.

nbs=4;

if nargin==0
  x=nbs; % number of boundary segments
  return
end

d=[
  0 0 0 0 % start parameter value
  1 1 1 1 % end parameter value
  0 0 0 0 % left hand region
  1 1 1 1 % right hand region
];

bs1=bs(:)';   % вектор строка или столбец bs становится строкой bs1

if find(bs1<1 | bs1>nbs)   % двойной знак || используется для скаляров
  error('Non-existent boundary segment number')
end

if nargin==1
  x=d(:,bs1);
  return
end

x=zeros(size(s));
y=zeros(size(s));
[m,n]=size(bs);
if m==1 & n==1,
  bs=bs*ones(size(s)); % expand bs
elseif m~=size(s,1) || n~=size(s,2),
  error('bs must be scalar or of same size as s');
end

if ~isempty(s),

% boundary segment 1
ii=find(bs==1);
if length(ii)
x(ii)=(-1-(1))*(s(ii)-d(1,1))/(d(2,1)-d(1,1))+(1);
y(ii)=(-0.5-(-0.5))*(s(ii)-d(1,1))/(d(2,1)-d(1,1))+(-0.5);
end

% boundary segment 2
ii=find(bs==2);
if length(ii)
x(ii)=(-1-(-1))*(s(ii)-d(1,2))/(d(2,2)-d(1,2))+(-1);
y(ii)=(0.5-(-0.5))*(s(ii)-d(1,2))/(d(2,2)-d(1,2))+(-0.5);
end
```

```
% boundary segment 3
ii=find(bs==3);
if length(ii)
x(ii)=(1-(-1))*(s(ii)-d(1,3))/(d(2,3)-d(1,3))+(-1);
y(ii)=(0.5-(0.5))*(s(ii)-d(1,3))/(d(2,3)-d(1,3))+(0.5);
end

% boundary segment 4
ii=find(bs==4);
if length(ii)
x(ii)=(1-(1))*(s(ii)-d(1,4))/(d(2,4)-d(1,4))+(1);
y(ii)=(-0.5-(0.5))*(s(ii)-d(1,4))/(d(2,4)-d(1,4))+(0.5);
end

end
```

Основное назначение файл - функции **[x,y]=recgeom(bs,s)** состоит в генерировании координат (`x,y`) точки участка границы с номером `bs` и параметром `s`. Фактически в коде этой функции записаны параметрические уравнения каждого участка границы. Чтобы вычислить координаты точки надо задать номер участка границы и значение ее параметра в параметрическом уравнении кривой.

Вызов **recgeom** без аргументов возвращает количество граничных отрезков. Вызов **recgeom(номер_участка_границы)** возвращает столбец, элементы которого являются начальным значением параметра в параметрическом уравнении участка границы, конечным значением параметра и номерами подобластей, расположенных слева и справа от указанного участка границы.

Локальная переменная `nbs` содержит число частей границы. Локальная матрица `d` содержит информацию о параметризации – число столбцов `d` равно числу участков границы, первый и второй элементы каждого столбца содержат начальное и конечное значения параметра (ноль и единица), в третьем и четвертом элементе записаны номера примитивов, расположенных по левую и правую сторону от участка границы (по направлению возрастания параметра). В блоках кода с комментарием `% boundary segment …` происходит поиск нужной части границы и вычисление координат (`x, y`) точки. Длины массивов `x, y` совпадают с длиной входного массива значений параметров `s`.

Область должна задаваться как минимум двумя граничными кривыми! Например, использование только одной пары уравнений $x = \cos t$, $y = \sin t$ для задания границы круга в теле функции декомпозиционной геометрии приведет к ошибке.

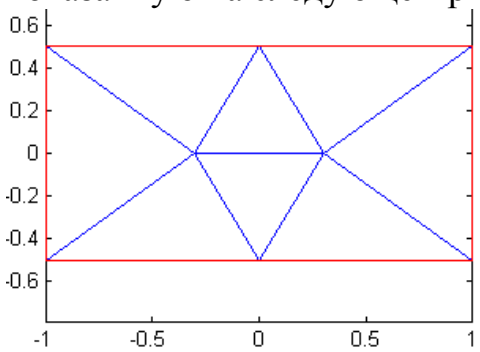
Умение создавать файл - функции с декомпозиционной геометрией области необходимо в том случае, когда область не может быть сконструирована только с использованием стандартных геометрических примитивов, таких как круг, эллипс, квадрат, прямоугольник или многоугольник. В своих файл – функциях вы можете использовать любые выражения и функции MatLab для вычисления координат точек границы.

Триангуляция

Функция **initmesh** разбивает плоскую область на треугольники. Ее входным аргументом является матрица декомпозиционной геометрии. Команда **[p, e, t] = initmesh(dl)** приводит к заполнению трех матриц **p, e, t**, которые содержат информацию о триангуляции. Функция **initmesh** может содержать дополнительные аргументы, задаваемые парами: название свойства, значение. Например, свойство `Hmax` устанавливает максимально допустимое значение длины стороны треугольного элемента. Полученную треугольную сетку можно отобразить в отдельном окне при помощи **pdemesh**, входными аргументами которой являются вышеперечисленные массивы. Например, последовательность команд

```
>>[p,e,t]=initmesh(dl,'Hmax',1.0);
>> pdemesh(p,e,t); axis equal
```

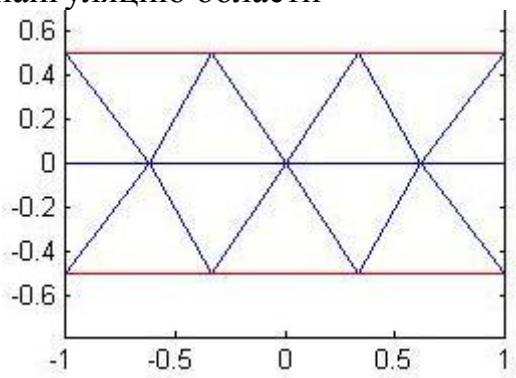
создает и выводит сетку, показанную на следующем рисунке



Вместо матрицы декомпозиционной геометрии аргументом функции **initmesh** может быть имя файл - функции с декомпозиционной геометрией области. Например, команды

```
[p,e,t]=initmesh('recgeom','Hmax',0.7);
pdemesh(p,e,t); axis equal
```

создают следующую триангуляцию области



Обычно значение параметра `Hmax` равно 1/10 некоторого характерного размера области (ширины, высоты, диаметра и т.д.).

Чтобы понять структуру массивов **p, e, t** перейдем в среду pdetool, нарисуем прямоугольник 2×1, из меню `Mesh - Parameters...`, вызовем диалоговое окно `Mesh Parameters` и в нем установим достаточно крупный размер стороны треугольного элемента (см. следующий рисунок)

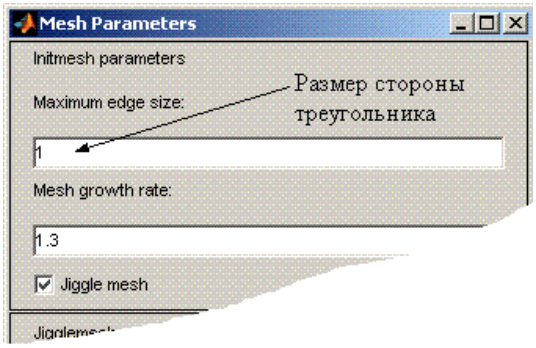


Рис. 22

Затем включим режимы нумерации узлов с помощью меню `Mesh - Show Node Labels` и нумерации элементов с помощью меню `Mesh - Show Triangle Labels` (рис. 23)

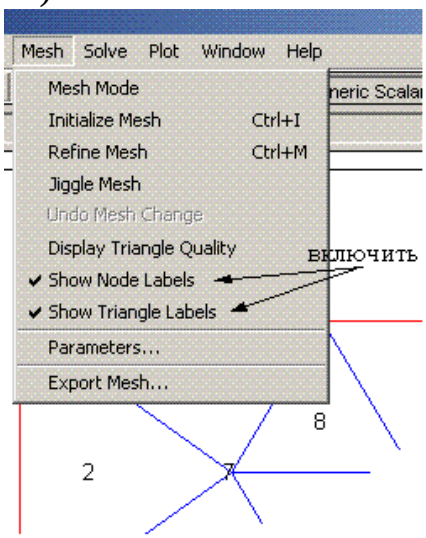


Рис. 23

Включим режим сетки (меню `Mesh - Mesh Mode`). В результате получим следующее изображение сетки (рис. 24)

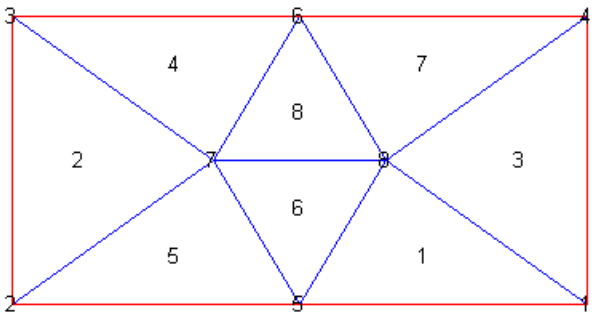


Рис. 24

Далее экспортируем сетку в глобальные массивы **p, e, t** при помощи пункта меню `Export Mesh...`,

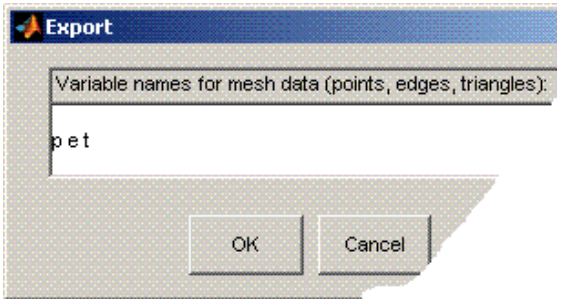


Рис. 25

Исследуем содержимое этих массивов в командном окне. В матрице **p** первая и вторая строки содержат *x* и *y* координаты узлов сетки

>> p

```
p =
    1.0000   -1.0000   -1.0000    1.0000         0         0   -0.3000    0.3000
   -0.5000   -0.5000    0.5000    0.5000   -0.5000    0.5000         0         0
```

Номер столбца матрицы **p** отвечает номеру узла сетки (номера узлов вы видите на рисунке 24).

Матрица **e** содержит информацию об одномерных элементах триангуляции, расположенных на границе.

>> e

```
e =
    1.0000    5.0000    2.0000    3.0000    6.0000    4.0000
    5.0000    2.0000    3.0000    6.0000    4.0000    1.0000
         0    0.5000         0         0    0.5000         0
    0.5000    1.0000    1.0000    0.5000    1.0000    1.0000
    1.0000    1.0000    2.0000    3.0000    3.0000    4.0000
         0         0         0         0         0         0
    1.0000    1.0000    1.0000    1.0000    1.0000    1.0000
```

Первая и вторая строки содержат номера начальной и конечной точек элементов (отрезков или дуг), принадлежащих границе области. Третья и четвертая строки содержат начальное и конечное значение параметра в параметрическом уравнении участка границы. Пятая строка содержит номер участка границы, которому принадлежит граничный элемент (номер столбца в матрице декомпозиционной геометрии `dl`). Шестая и седьмая строки содержат номера подобластей расположенных слева и справа от элемента.

В матрице **t** хранится информация о треугольных элементах разбиения области. Номер столбца матрицы t соответствует номеру треугольного элемента на рисунке 24.

>> t

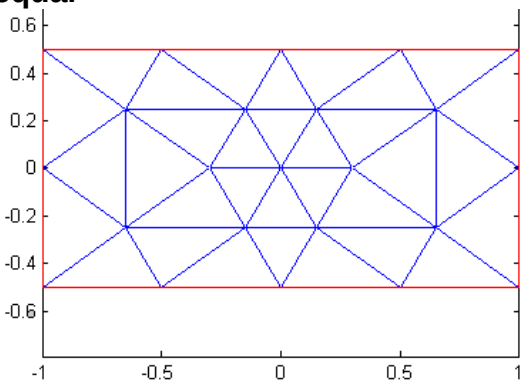
```
t =
     5     3     1     6     2     7     4     6
     1     2     4     3     5     5     6     7
     8     7     8     7     7     8     8     8
     1     1     1     1     1     1     1     1
```

Первые три элемента столбца содержат номера узлов треугольного элемента в направлении обхода против часовой стрелки, а четвертый – номер подобласти, которой принадлежит треугольный элемент.

Для уменьшения шага сетки используется функция **refinemesh.** Первым входным аргументом является матрица или функция декомпозиционной геометрии, далее передаются массивы с информацией о триангуляции. В выходных аргументах возвращаются массивы, соответствующие измельченной

сетке. По умолчанию сторона каждого элемента делится пополам, каждый треугольный элемент исходной сетки разбивается на четыре части. Команды, приведенные ниже, производят вышеописанное уменьшение шага сетки

[p1,e1,t1]=refinemesh(dl,p,e,t);
pdemesh(p1,e1,t1); axis equal



Функция **refinemesh** позволяет задать ряд дополнительных параметров, управляющих процедурой дробления сетки, в частности, делить только некоторые треугольные элементы или подобласти. Кроме того, можно выбрать другой алгоритм уменьшения шага сетки, который делит на две равные части наибольшую сторону конечного элемента. Первым входным аргументом может быть не только матрица декомпозиционной геометрии, но и имя файл - функции, описывающей декомпозиционную геометрию области.

Граничные условия и коэффициенты уравнения

Граничные условия в PDE Toolbox задаются двумя способами, либо при помощи матрицы граничных условий (Boundary Condition matrix), либо в файл - функции. Число столбцов матрицы граничных условий совпадает с числом столбцов в матрице декомпозиционной геометрии, т. е. каждый столбец отвечает участку границы области. На каждой части границы может быть задано условие одного из типов: Дирихле `h*u = r`, или условие Неймана $\langle \mathbf{n}, (c \cdot \nabla u) \rangle + qu = g$. Здесь (если `u` – скалярная функция, т.е. решается одно уравнение) `h, r, q, g` – постоянные или функции переменных *x*,*y*, параметр `c` может быть постоянной или матрицей `2 x 2`, **n** – вектор нормали к границе, $\langle\ \rangle$ - скалярное произведение векторов.

Разберем структуру матрицы граничных условий. Настроим среду pdetool на решение скалярного (см. рис. 26) эллиптического уравнения, установив переключатель `Elliptic` в окне `PDE Specification` (см. рис. 14).

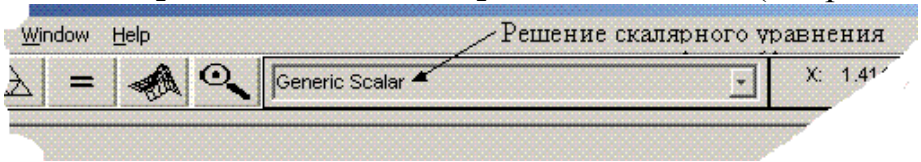


Рис. 26

Зададим на всех участках границы прямоугольника условие Дирихле, например x^2+y^2. Для этого в диалоговом окне `Boundary Condition` введем 1 в поле `h` и *x.^2 + y.^2* в поле `r`. Экспортируем в рабочую среду матрицу граничных условий, выбрав в меню `Boundary - Export Decomposed Geometry`,

Boundary Cond's… Появляется диалоговое окно Export, в котором по умолчанию стоят имена g и b – первое для массива декомпозиционной геометрии и второе – для хранения матрицы граничных условий. Матрицу декомпозиционной геометрии мы уже ранее строили с помощью функции **decsg**. Нажмем ОК и в рабочей среде появился двумерный массив b.

b

```
b =
     1     1     1     1
     1     1     1     1
     1     1     1     1
     1     1     1     1
     1     1     1     1
     9     9     9     9
    48    48    48    48
    48    48    48    48
    49    49    49    49
   120   120   120   120
    46    46    46    46
    94    94    94    94
    50    50    50    50
    43    43    43    43
   121   121   121   121
    46    46    46    46
    94    94    94    94
    50    50    50    50
```

Структура матрицы граничных условий приспособлена для хранения граничных условий в задачах, описываемых системой дифференциальных уравнений. Каждый столбец соответствует участку границы. В нашем случае область является прямоугольником с одинаковыми граничными условиями Дирихле, и матрица состоит из четырех столбцов.

Столбец условно делится на две части, верхняя часть столбца матрицы граничных условий содержит информацию о типе граничных условий и длинах формул. В нижней части записаны коэффициенты и формулы граничных условий. При этом символы строки с формулой хранятся в последовательно идущих элементах в виде своих кодов. В случае уравнения (а не системы) и граничного условия Дирихле формулы хранятся в элементах столбца, начиная с седьмого, в чем несложно убедиться, преобразовав элементы нижней части матрицы из кодов в символы

char(b(7:end,:)')

```
ans =
001x.^2+y.^2
001x.^2+y.^2
001x.^2+y.^2
001x.^2+y.^2
```

Здесь для наглядности представления формул мы транспонировали матрицу граничных условий. Первые два элемента зарезервированы под коэффициенты условия Неймана (они равны нулю, поскольку поставлено условие Дирихле). Третий элемент содержит значение h (равное единице), а остальные являются символами выражения x.^2+y.^2.

Матрица граничных условий имеет столько же столбцов, сколько матрица декомпозиционной геометрии – т.е. столько, сколько участков границы. Вид матрицы отражает структуру граничных условий – для каждого

участка свое граничное условие и, следовательно, свой столбец со своей структурой. Для скалярного уравнения на участке границы можно поставить граничное условие Неймана или условие Дирихле. Условие Неймана имеет вид $\frac{\partial u}{\partial \mathbf{n}} + qu = g$, а Дирихле – вид $h \cdot u = r$. И поэтому столбец матрицы должен содержать значения коэффициентов `q, g, h` и `r`.

При решении одного скалярного уравнения каждый столбец матрицы граничных условий будет содержать следующие элементы:

№ строки	Элемент столбца
1	Размерность системы уравнений = 1
2	Кол-во условий Дирихле (0 или 1)
3	n_q – к–во символов представляющих `q`
4	n_g – к –во символов представляющих `g`
5	n_h – к –во символов представляющих `h` (если условия Дирихле есть)
6	n_r – к –во символов представляющих `r` (если условия Дирихле есть)
…	Текст формулы `q` (коды символов)
…	Текст формулы `g` (коды символов)
…	Текст формулы `h` (коды символов) (если условия Дирихле есть)
…	Текст формулы `r` (коды символов) (если условия Дирихле есть)
…	Нули, если надо выровнять длину столбцов матрицы.

Например, для одного ДУЧП и граничного условия $\frac{\partial u}{\partial \mathbf{n}} = -x^2$ этот вектор – столбец будет иметь вид $[1\ 0\ 1\ 5\ '0'\ '-x.\hat{}2']$. Для граничного условия Дирихле $u|_{\partial\Omega} = x^2 - y^2$ вектор – столбец матрицы граничных условий, соответствующий участку границы на котором задано это условие, будет равен $[1\ 1\ 1\ 1\ 1\ 9\ '0'\ '0'\ '1'\ 'x.\hat{}2 - y.\hat{}2']$.

Символьные массивы (строки в одинарных кавычках) автоматически заменяются кодами символов. Однако лучше сразу заменить символьные массивы числовыми массивами, содержащими коды символов. Например, строку

s='x.^2-y.^2'

можно сразу заменить числовым вектором

c=double(s)

```
c =
   120    46    94    50    45   121    46    94    50
```

Для системы уравнений матрица граничных условий имеет более сложный вид, с которым вы можете познакомиться на странице справки функции **assemb.**

При вызове функций пакета PDE TOOLBOX вместо матрицы граничных условий можно использовать функцию граничных условий. Ее можно сгенерировать автоматически из матрицы граничных условий с помощью функции **wbound**. Ее аргументами является матрица граничных условий и имя генерируемой файл – функции граничных условий. Выполним команду

wbound(b,'myBound')

Функция **wbound** вернет -1, если файл граничных условий создать не удалось.

```
function [q,g,h,r]=myBound(p,e,u,time)
%MYBOUND   Boundary condition data.
%
bl=[
1 1 1 1
1 1 1 1
1 1 1 1
1 1 1 1
1 1 1 1
9 9 9 9
48 48 48 48
48 48 48 48
49 49 49 49
120 120 120 120
46 46 46 46
94 94 94 94
50 50 50 50
43 43 43 43
121 121 121 121
46 46 46 46
94 94 94 94
50 50 50 50
];
if any(size(u))
  [q,g,h,r]=pdeexpd(p, e, u, time, bl);
else
  [q,g,h,r]=pdeexpd(p, e, time, bl);
end
```

Как видим, созданная функция содержит локальный массив `bl` полностью совпадающий с матрицей граничных условий. Переменная `bl` затем передается недокументированной функции **pdeexpd(…)**. Заметим, что функция **any(A),** использованная в теле функции граничных условий возвращает логическую единицу (`true`), если в векторе есть хотя бы один ненулевой элемент.

Хотя описания функции **pdeexpd(…)** в справочной системе нет, вы можете посмотреть ее код, выполнив команду

type pdeexpd

Если вы вручную захотите создать свою функцию граничных условий, то в шаблон функции **myBound**, приведенный выше, вы должны будете вставить свою матрицу граничных условий. Заметим однако, что создание функции граничных условий в большинстве случаев не требуется и можно ограничится созданием матрицы граничных условий.

Солверы.

Приближенным решением граничной задачи является вектор значений в узлах сетки. Его создают специальные функции пакета PDE Toolbox называемые солверами (solver). Перед ее вызовом должна быть создана

матрица или файл–функция граничных условий `b`, заданы триангуляция в массивах `p, e, t` и заданы коэффициенты уравнения (разумеется, переменные могут называться по-другому). Если решается нестационарная задача, то необходимо создать функции, определяющие решение в начальный момент времени и вектор со значениями времени, в которые следует найти решение. Затем эти данные передаются функциям солверам. Они находят приближенное решение задачи в узлах сетки. Солверы реализованы в специальных файл - функциях – для каждого типа задачи свой солвер. В следующей таблице перечислены основные солверы

Имя функции	Описание типа решаемых задач
assempde	Эллиптические уравнения и системы
parabolic	Параболические уравнения и системы
hyperbolic	Гиперболические уравнения и системы
pdenonlin	Нелинейные стационарные уравнения
adaptmesh	Адаптивная генерация сетки и решение эллиптических уравнений и систем с заданной точностью
pdeeig	решение эллиптических задач на собственные значения

Интерфейс солвера **assempde** является наиболее универсальным. Она решает уравнение (или систему) вида $-div(c \cdot grad(u)) + a \cdot u = f$. Первым его входным аргументом является матрица или строка с именем файл – функции граничных условий. Далее указываются матрицы с информацией о триангуляции и коэффициенты уравнения `c, a, f`. Ее вызовы с одним выходным аргументом

```
u = assempde(b, p, e, t, c, a, f);        % b - матрица
u = assempde('bouncond', p, e, t, c, a, f);  % bouncond - функция
```

приводят к записи в вектор `u` значений решения в узлах сетки, координаты которых хранятся в `p`.

Входные аргументы солверов **parabolic** и **hyperbolic** такие же, как у **assempde**. Кроме того, задается вектор значений решения u_0 в начальный момент времени (в случае гиперболического уравнения еще и вектор u_{t0} с начальным значением производной решения по времени), вектор `tlist` моментов времени, в которые требуется найти решение, и коэффициент `d` уравнения при второй производной по времени. Эти солверы возвращают матрицу `u` с решением, каждый столбец которой есть вектор со значениями приближенного решения в узлах сетки в соответствующие моменты времени, указанные в `tlist`:

```
u = parabolic(u0, tlist, b, p, e, t, c, a, f, d)
u = hiperbolic(u0, ut0, tlist, b, p, e, t, c, a, f, d)
```

Об использовании других солверов вы можете узнать из справочной системы MatLab. Один из них **pdenonlin** мы неявно использовали, когда решали нелинейное уравнение при построении минимальной поверхности. В том

примере, перед тем как решать уравнение, мы выбрали меню `Solve – Parameters` и в открывшемся окне устанавливали флажок `Use nonlinear solver` (рис.15).

Визуализация результатов

Функции **pdegplot** и р**demesh** предназначены для графического отображения геометрии области и сетки, их использование описано выше. Основной функцией для визуализации решения является **pdeplot**, которая позволяет управлять видом получаемых графиков при помощи ряда дополнительных параметров. Функции **pdecont** и **pdesurf**, строящие линии уровня решения или трехмерный график, реализуют частные случаи обращения к **pdeplot.** Входными аргументами **pdecont** и **pdesurf** являются матрица `p` с координатами узлов, матрица `t` соответствия глобальной и локальной нумераций и вектор `u` со значениями решения в узлах. Четвертым дополнительным аргументом **pdecont** может являться вектор значений, которые требуется отобразить линиями уровня, или их число.

Первые три аргумента функции **pdeplot** являются массивами `p`, `e` и `t` с информацией о триангуляции. Затем парами задаются имя параметра (свойства) и его значение. В следующей таблице описаны некоторые из этих параметров

Имя параметра	Значение
`xydata`	Вектор со значениями решения в узлах для построения двумерного графика
`xystyle`	`off, flat, interp` (по умолчанию) – способы заливки контуров
`contour`	`on, off` (по умолчанию) – отображение или скрытие контурных линий
`zdata`	Вектор со значениями решения в узлах для построения трехмерного графика
`colormap`	`cool, hot, gray, bone`, ... – цветовые палитры
`mesh`	`on, off` (по умолчанию) – отображение конечно элементной сетки
`colorbar`	`off, on` (по умолчанию) – вывод шкалы соответствия цвета и значения
`title`	Строка с заголовком
`levels`	Число линий уровня или вектор со значениями решения, отображаемого линиями уровня

Использование одной из пар ‘`xydata`’ или ‘`zdata`’ обязательно, поскольку содержит вектор значений решения в узлах сетки.

Для иллюстрации использования описанных выше функций решим несколько задач в полукруге. Центр полукруга находится в начале координат, радиус единица, выбрана верхняя часть круга. Решим в этой области уравнение Пуассона, Лапласа, задачу о колебании полукруглой мембраны и найдем форму минимальной поверхности. Вначале создадим функцию декомпозиционной

геометрии области, при этом учтем, что область должна задаваться как минимум двумя граничными кривыми.

Функция декомпозиционной геометрии полукруга

```
function [x,y]=halfcirclegeom(bs,s)
% декомпозиционная функция полукруга

nbs=2;
if nargin==0,
  x=nbs; % number of boundary segments
  return
end

d=[
  0 0   % start parameter value
  1 1   % end parameter value
  1 1   % left hand region
  0 0   % right hand region
];

bs1=bs(:)';

if find(bs1<1 | bs1>nbs),
  error('Non-existent boundary segment number')
end

if nargin==1,
  x=d(:,bs1);
  return
end

x=zeros(size(s));
y=zeros(size(s));
[m,n]=size(bs);
if m==1 & n==1,
  bs=bs*ones(size(s)); % expand bs
elseif m~=size(s,1) | n~=size(s,2),
  error('bs must be scalar or of same size as s');
end

if ~isempty(s),
      % boundary segment 1
      ii=find(bs==1);
      if length(ii)
            x(ii)=cos(pi*s(ii));
            y(ii)=sin(pi*s(ii));
      end

      % boundary segment 2
      ii=find(bs==2);
      if length(ii)
            x(ii)=-1+2*s(ii);
            y(ii)=0;
      end
end
% ----------------- конец тела функции ----------------
```

Проверяем, что граница области создана правильно (следующий левый рисунок)

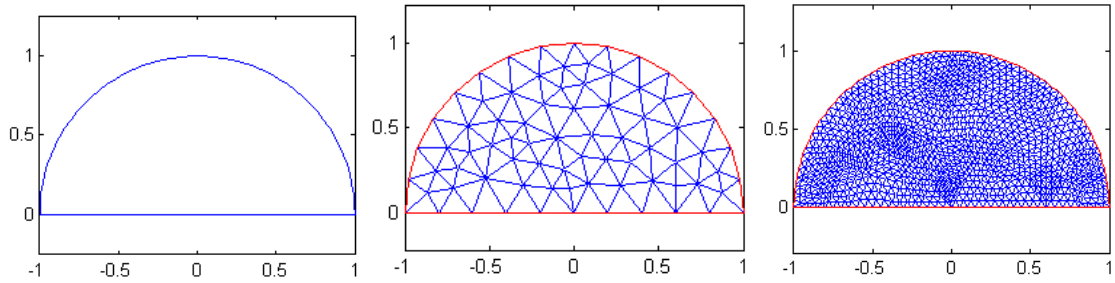
pdegplot('halfcirclegeom'), axis equal

Строим и рисуем сетку (средний рисунок)

```
[p, e, t] = initmesh('halfcirclegeom');
pdemesh(p,e,t); axis equal;
```

Сгустим и нарисуем сетку (правый рисунок)

```
[p,e,t]=refinemesh('halfcirclegeom',p,e,t);
[p,e,t]=refinemesh('halfcirclegeom',p,e,t);
pdemesh(p,e,t); axis equal;
```



Функцию **halfcirclegeom** и вектора **[p, e, t]** будем использовать для решения краевых задач в полукруге, приводимых в следующих примерах.

Пример 1. Решим уравнение Пуассона $\Delta u = -1$ с нулевыми граничными условиями $u|_{\partial\Omega} = 0$.

Для граничных условий Дирихле $h \cdot u = r$ на обеих участках границы (на диаметре и полуокружности) имеем `h=1, r=0`. Матрицу граничных условий можно построить командой

```
b=[1 1; 1 1; 1 1; 1 1; 1 1; 1 1; 48 48; 48 48; 49 49; 48 48]
```

```
b =
     1     1
     1     1
     1     1
     1     1
     1     1
     1     1
    48    48
    48    48
    49    49
    48    48
```

Здесь числа 48 и 49 являются кодами символов 0 и 1, представляющих выражения для `r` и `h`. Проверим

```
char(b(7:end,:)')
```

```
ans =
0010
0010
```

Эллиптическое уравнение $-div(c \cdot grad(u)) + a \cdot u = f$ принимает требуемую форму при `a=0, c=1, f=1`. Задаем коэффициенты и правую часть уравнения $-\Delta u = f$

```
a=0;    % коэффициент уравнения
c=1;    % коэффициент уравнения
f=1;    % Строка с формулой правой части уравнени
```

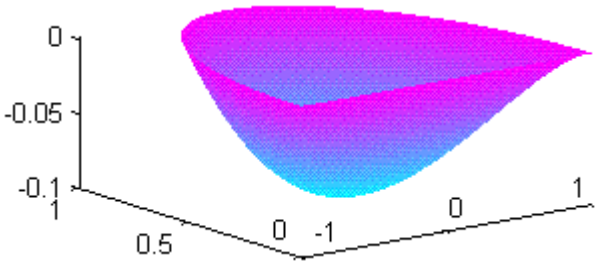
Для решения эллиптических уравнений используется функция **assempde**. Ей следует передать аргументы – матрицу граничных условий `b`, массивы

`p, e, t,` содержащие информацию о триангуляции, и коэффициенты уравнения `c, a, f`.

u=assempde(b,p,e,t,c,a,f);

Строим график поверхности решения `u(x,y)`

pdesurf(p,t,u);



Пример 2. Решим уравнение Лапласа $\Delta u = 0$ в полукруге с граничным условием $u\big|_{\partial\Omega} = |x|$.

Посмотрим какие коды имеют символы граничного условия $h \cdot u = r$, где `r=abs(x)`. Построим строку s, содержащей формулу, и преобразуем ее в числовой массив

s='abs(x)'

```
s =
abs(x)
```

double(s)

```
ans =
    97    98   115    40   120    41
```

Теперь можно создать матрицу граничных условий. На шестом месте столбцов матрицы граничных условий хранятся длины формул условия Дирихле. Количество символов в формуле `abs(x)` равно шести. Поэтому

b=[1 1; 1 1; 1 1; 1 1; 1 1; 6 6; 48 48; 48 48; 49 49;...
97 97; 98 98; 115 115; 40 40; 120 120; 41 41]

char(b')

```
ans =
001abs(x)
001abs(x)
```

Коды до 32 не представляют никакие символы, поэтому функция **char** отобразила их условными квадратами.

Матрицу граничных условий можно создать по – другому

b=[1 1; 1 1; 1 1; 1 1; 1 1; 6 6; '0' '0'; '0' '0'; '1' '1';'a', 'a'; 'b' 'b'; 's' 's'; '(' '('; 'x' 'x';')' ')']

char(b(7:end,:)')

```
ans =
001abs(x)
001abs(x)
```

Эллиптическое уравнение $-div(c \cdot grad(u)) + a \cdot u = f$ принимает требуемую форму при `a=0, c=1, f=0`. Задаем коэффициенты и правую часть уравнения $\Delta u = 0$

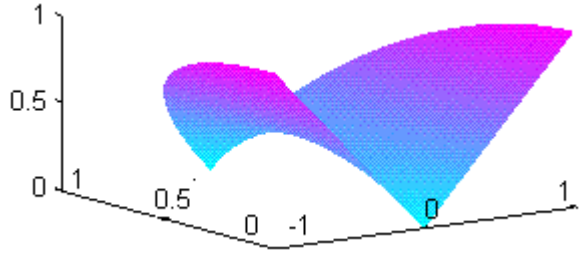
```
a=0;    % коэффициент уравнения
c=1;    % коэффициент уравнения
f = 0;     % Строка с формулой правой части уравнения
```

Находим решение

u=assempde(b,p,e,t,c,a,f);

Строим график поверхности решения `u(x,y)`

pdesurf(p,t,u);



Пример 3. Решим ДУЧП следующего вида

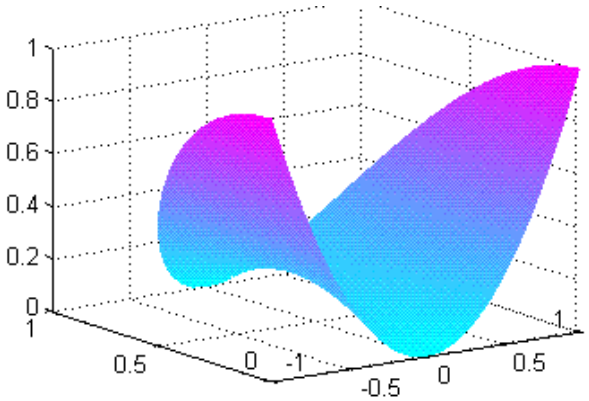
$$\nabla \cdot \left(\frac{1}{\sqrt{1+|\nabla u|^2}} \nabla u \right) = 0$$

в полукруге со значениями $u = x^2$ на границе $\partial\Omega$. Оно определяет форму минимальной поверхности, край которой находится на заданной высоте над контуром области.

Это нелинейное эллиптическое уравнение $-div(c \cdot grad(u)) + a \cdot u = f$ в котором коэффициент $c = \frac{1}{\sqrt{1+u_x'^2+u_y'^2}}$ является функцией производных u_x, u_y решения.

Для решения задачи создадим сценарий `t2.m`, использующий выше построенную функцию `halfcirclegeom` декомпозиционной геометрии полукруга.

```
% сценарий построения минимальной поверхности над полукругом
[p, e, t] = initmesh('halfcirclegeom');
[p, e, t]=refinemesh('halfcirclegeom',p,e,t);
% pdemesh(p,e,t); axis equal;

b=[1 1; 1 1; 1 1; 1 1; 1 1; 4 4; '0' '0'; '0' '0'; '1' '1';...
    'x', 'x'; '.' '.'; '^' '^'; '2' '2'];  % матрица граничных условий
a=0;       % коэффициент уравнения
f=0;       % Строка с формулой правой части уравнения
c='1./sqrt(1+ux.^2+uy.^2)';     % коэффициент уравнения
[u,res]=pdenonlin(b,p,e,t,c,a,f);
pdesurf(p,t,u); grid on;
```



Заметим, что для решения задачи мы выбрали солвер **pdenonlin**, который используется для решения нелинейных уравнений.

Пример 4. Решим ДУЧП минимальной поверхности, приведенное в предыдущем примере, в полукруге со значениями $1-|x|$ над прямолинейным участком границы и x^2-1 – над криволинейным. Для его решения создадим сценарий t3.m

```
% сценарий построения минимальной поверхности над полукругом
[p, e, t] = initmesh('halfcirclegeom');
[p,e,t]=refinemesh('halfcirclegeom',p,e,t);
pdemesh(p,e,t); axis equal;
b1=[1 1 1 1 1 6 48 48 49 120 46 94 50  45 49  0  0;
    1 1 1 1 1 8 48 48 49  49 45 97 98 115 40 120 41];
b=b1';

a=0;          % коэффициент уравнения
f=0;          % Строка с формулой правой части уравнения
c='1./sqrt(1+ux.^2+uy.^2)';    % коэффициент уравнения
[u,res]=pdenonlin(b,p,e,t,c,a,f,'Tol',0.01,'MaxIter',50);
pdesurf(p,t,u); grid on;


figure; axis([-1 1 0 1 -1 1]);
pdeplot(p,[],t,'xydata',u(:),'xystyle','off', ...
     'zdata',u(:),'zstyle','continuous','colorbar','off',...
     'mesh','on','xygrid','on');
colormap bone; grid on;
```

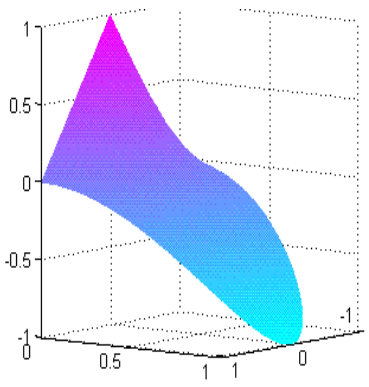
В конце первого столбца матрицы граничных условий мы поставили два нуля, чтобы выровнять длины столбцов.

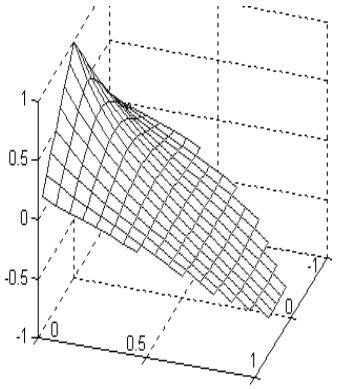
char(b(7:end,:))'

```
ans =
001x.^2-1
0011-abs(x)
```

Обратите также внимание, что при вызове функции **pdenonlin** мы использовали опции, задающие точность вычислений (уменьшаем точность `'Tol',0.01`) и максимальное количество итераций (увеличиваем максимальное количество итераций `'MaxIter',50`). Без задания этих опций функция **pdenonlin** не справлялась со своей задачей и выводила сообщения об ошибках.

Сценарий завершается командами построения цветного и каркасного графиков минимальной поверхности.





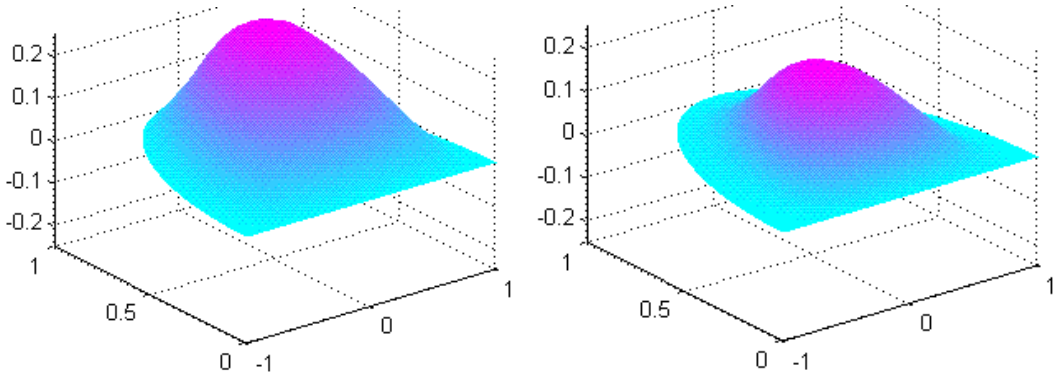
Пример 5. Решим задачу о колебании полукруглой мембраны закрепленной по контуру. Она состоит в решении уравнения $\frac{\partial^2 u}{\partial t^2} = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}$ с граничными условиями $u\big|_{\partial\Omega} = 0$.

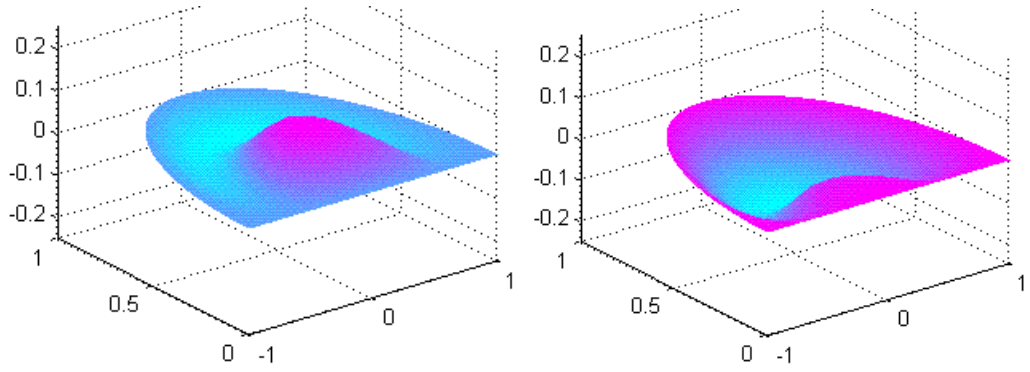
Общий вид гиперболического уравнения, решаемого пакетом PDE TOOLBOX, имеет вид $d\frac{\partial^2 u}{\partial t^2} - div(c\,grad(u)) + a\,u = f$. Чтобы оно имело требуемый вид, функции **hyperbolic** мы должны передать коэффициенты `d=1, c=1, a=0, f=0`.

Для решения задачи создадим сценарий `waveHalfCircle.m`, использующий выше построенную функцию `halfcirclegeom` декомпозиционной геометрии полукруга.

```
% сценарий построения поверхности колеблющейся мембраны над полукругом
[p, e, t] = initmesh('halfcirclegeom');
[p,e,t]=refinemesh('halfcirclegeom',p,e,t);
pdemesh(p,e,t); axis equal;

b1=[1 1 1 1 1 1 48 48 49 48 ;
    1 1 1 1 1 1 48 48 49  48 ];
b=b1';           % матрица граничных условий

x=p(1,:)';
y=p(2,:)';

u0=(1-x.^2-y.^2).*y;     % начальное смещение точек мембраны
ut0= 0;                  % начальная скорость точек мембраны

n=21;
tlist=linspace(0,2,n);  % вектор моментов времени

u=hyperbolic(u0,ut0,tlist,b,p,e,t,1,0,0,1);
% строим кадр
pdesurf(p,t,u(:,3)); axis([-1 1 0 1 -0.25 0.25]);grid on;
```

В последней строке сценария мы строим поверхность мембраны `u(:,3)` в третий момент времени `tlist(3) = 0.2`. На следующем графике показаны формы мембраны в моменты времени `tlist([3, 4,5, 6])=[0.2 0.3 0.4 0.5]`.





4.1.3 ДУЧП с одной пространственной переменной

В MATLAB численное решение краевых задач для уравнений в частных производных с одной временной `t` и одной пространственной переменной x выполняется функцией **pdepe(…)**. Она "умеет решать" параболическое дифференциальное уравнение следующего вида

$$c(x,t,u,u_x)\cdot u_t = x^{-m}\frac{\partial}{\partial x}\left(x^m \cdot b(x,t,u,u_x)\right) + s(x,t,u,u_x)$$

с граничными условиями

$$p(x_l,t,u) + q(x_l,t)\cdot b(x_l,t,u,u_x) = 0$$
$$p(x_r,t,u) + q(x_r,t)\cdot b(x_r,t,u,u_x) = 0$$

где x_l и x_r представляют левую и правую точку отрезка, и функция `u` вычисляется также в этих точках. Начальное условие должно иметь вид $u(0,x) = f(x)$. Отметим, что функция `b`, стоящая в уравнении и граничных условиях одна и та же. Функция **pdepe(…)** также "умеет решать" системы таких уравнений.

Функция вызывается командой

sol = pdepe(m, @pdefun, @icfun, @bcfun, xmesh, tspan, options),

где

- `m` параметр симметрии задачи: `m=0` – плоская симметрия, `m=1` – цилиндрическая симметрия, `m=2` – сферическая симметрия;
- `@pdefun` – дескриптор функции, определяющей коэффициенты `c, b, s` уравнения; заголовок функции следует оформлять следующим образом: `function [c,b,s] = pdefun(x,t,u,DuDx);`
- `@icfun` – дескриптор функции, определяющей начальное условие `f(x)`; заголовок функции следует оформлять следующим образом: `function u0 = icfun(x);`
- `@bcfun` – дескриптор функции, определяющей коэффициенты граничного условия; заголовок функции следует оформлять следующим образом:
  `function [pL,qL,pR,qR]=bcfun(xL,uL,xR,uR,t);`
- `xmesh` – вектор [x_1, x_2, …, x_n], определяющий точки, в которых будет вычисляться значения решения `u(t,x)` ($x_1<x_2<\ldots<x_n$);

– tspan – вектор $[t_0, t_1, \ldots, t_n]$ определяющий моменты времени, в которые будет вычисляться решение u(t,x) ($t_0<t_1<\ldots<t_n$, моментов времени должно быть три или более).
– options – необязательные аргументы, которые могут задавать точность определения решения, количество итераций и т.д.

Функция возвращает многомерный массив $u(t_j, x_k, i)$, представляющий аппроксимацию i – ой компоненты вектора решения системы уравнений $[u_1(t,x), u_2(t,x), \ldots, u_n(t,x)]$ в точках x_k=xmesh(k) в моменты времени t_j=tspan(j). В случае одного уравнения этот массив представляет матрицу значений u(j, k, 1) одной функции – решения $u_1(t,x)$. Строками этой матрицы являются значения решения в некоторый момент времени. Для вычисления значения решения и ее производных в точках, не включенных в вектор xmesh можно использовать функцию **pdeval**.

Опциями функции **pdepe** могут быть некоторые из опций, используемых при решении обыкновенных дифференциальных уравнений, – RelTol, NarmControl, InitialStep и MaxStep. Их задание выполняется функцией **odeset**. Значения этих опций по умолчанию удовлетворительно для большинства случаев.

Использование функции **pdepe** рассмотрим на примерах.

Пример 1. Решим на отрезке $[0, 1]$ нестационарное одномерное уравнение теплопроводности $\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2}$ с граничными условиями $u(t,0)=0$, $u'_x(t,1)=0$ и начальным условием $u(0,x)=0.5-\left|0.5-x\right|$.

В нашем случае m=0 и коэффициенты в уравнении общего вида должны быть равны

$$c(x,t,u,u_x)=1,\ b(x,t,u,u_x)=u_x,\ s(x,t,u,u_x)=0.$$

Коэффициенты граничных условий для нашего примера имеют вид

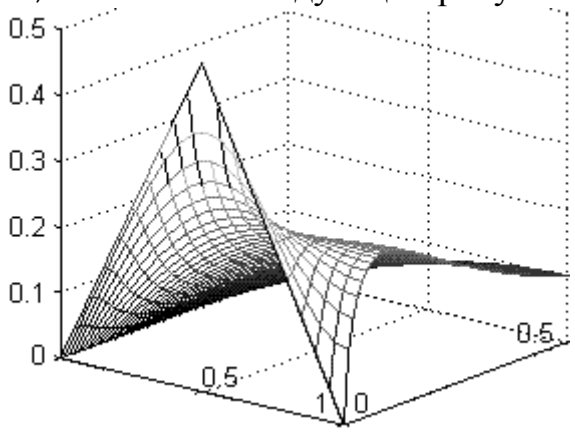
$$p(0,t,u)=u,\ q(0,t)=0,\ p(1,t,u)=0,\ q(1,t)=1.$$

Перед вызовом функции **pdepe** следует создать функции, вычисляющие коэффициенты уравнения и граничных условий. Их нельзя помещать в файл – сценарий, поэтому для решения задачи удобно создавать m – функцию pdex1.m (без аргументов), в конец которой можно поместить код подфункций для вычисления требуемых коэффициентов.

```
function pdex1
% Решение одномерного уравнения теплопроводности с нулевой
% температурой на  левом краю и условием теплоизолированности на правом.

m = 0;                      % плоская задача
x = linspace(0,1,21);       % вектор точек
t = linspace(0,0.5,51);     % вектор моментов времени

sol = pdepe(m,@koefpde,@initpde,@boundpde,x,t); % Решаем ДУЧП
u = sol(:,:,1);    % скалярное уравнение => функция решения одна
```

```
% график поверхности решения
figure;
[X,T] = meshgrid(x,t);
mesh(X,T,u);
colormap Gray;
title('Численное решение на 20 точках.')
xlabel('Расстояние x');
ylabel('Время t');

% график решения в заданный момент времени
figure;
numfr=21;           % номер момента времени
plot(x,u(numfr,:)); % график решения в этот момент времени
s1=sprintf('Температура при t = %0.2g',t(numfr));
s2=sprintf('u(x,%0.2g)',t(numfr));
title(s1);
xlabel('Расстояние x');
ylabel(s2);

% простейшая анимация
fig = plot(x,u(1,:),'erase','xor');
for k=2:length(t)
    set(fig,'xdata',x,'ydata',u(k,:));
    pause(.1);
end

% ---------------------------------------------------------------
% функции для вычисления коэффициентов и начального условия
% ---------------------------------------------------------------
function [c,f,s] = koefpde(x,t,u,DuDx)
% коэффициенты уравнения
c = 1;
f = DuDx;
s = 0;
% ---------------------------------------------------------------
function u0 = initpde(x)
% начальное значение
u0 = 0.5-abs(x-0.5);
% ---------------------------------------------------------------
function [pl,ql,pr,qr] = boundpde(xl,ul,xr,ur,t)
% коэффициенты граничных условий
pl = ul;
ql = 0;
pr = 0;
qr = 1;
```

График функции `u(t,x)` показан на следующем рисунке.



Пример 2. Рассмотрим пример решения нелинейной параболической системы ДУЧП с одной пространственной переменной

$$\frac{\partial u_1}{\partial t} = \frac{\partial^2 u_1}{\partial x^2} + u_1 \cdot (1 - u_1 - u_2)$$
$$\frac{\partial u_2}{\partial t} = \frac{\partial^2 u_2}{\partial x^2} + u_2 \cdot (1 - u_1 - u_2)$$

с граничными условиями

$$u'_{1x}(t,0) = 0;\ u_1(t,1) = 1$$
$$u_2(t,0) = 0;\ u'_{2x}(t,1) = 0$$

и начальными условиями

$$u_1(0,x) = x^2\ ;\ u_2(0,x) = x(x-2)$$

Общий вид системы двух параболических уравнений следующий

$$c_1(x,t,u,u_x) \cdot u'_{1t} = x^{-m} \frac{\partial}{\partial x}\left(x^m \cdot b_1(x,t,u,u_x)\right) + s_1(x,t,u,u_x)$$
$$c_2(x,t,u,u_x) \cdot u'_{2t} = x^{-m} \frac{\partial}{\partial x}\left(x^m \cdot b_2(x,t,u,u_x)\right) + s_2(x,t,u,u_x)$$

Для нашего примера `m=0` и коэффициенты будут равны

$$c = \begin{pmatrix} c_1 \\ c_2 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \end{pmatrix};\ b = \begin{pmatrix} b_1 \\ b_2 \end{pmatrix} = \begin{pmatrix} u'_{1x} \\ u'_{2x} \end{pmatrix};\ s = \begin{pmatrix} s_1 \\ s_2 \end{pmatrix} = \begin{pmatrix} u_1(1-u_1-u_2) \\ u_2(1-u_1-u_2) \end{pmatrix};$$

которые мы будем создавать в теле функции `eqnsys.m`

```
function [c,b,s]=eqnsys(x,t,u,DuDx)
% Файл создания коэффициентов системы двух PDE одной временной
% и одной пространственной переменной
c=[1; 1];
b=[1; 1].*DuDx;
s=[u(1).*(1-u(1)-u(2)); u(2).*(1-u(1)-u(2))];
```

Общий вид граничных условий для системы двух параболических уравнений следующий

$$p_1(x_l,t,u) + q_1(x_l,t) \cdot b_1(x_l,t,u,u'_x) = 0$$
$$p_1(x_r,t,u) + q_1(x_r,t) \cdot b_1(x_r,t,u,u'_x) = 0$$
$$p_2(x_l,t,u) + q_2(x_l,t) \cdot b_2(x_l,t,u,u'_x) = 0$$
$$p_2(x_r,t,u) + q_2(x_r,t) \cdot b_2(x_r,t,u,u'_x) = 0$$

Для нашего примера имеем

$$p(0,t,u) = \begin{pmatrix} p_1 \\ p_2 \end{pmatrix} = \begin{pmatrix} 0 \\ u_2 \end{pmatrix};\quad q(0,t) = \begin{pmatrix} q_1 \\ q_2 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix};$$
$$p(1,t,u) = \begin{pmatrix} p_1 \\ p_2 \end{pmatrix} = \begin{pmatrix} u_1 - 1 \\ 0 \end{pmatrix};\quad q(1,t) = \begin{pmatrix} q_1 \\ q_2 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix};$$

Их мы будем задавать функцией `boundsys.m`

```
function [pl,ql,pr,qr]=boundsys(xl,ul,xr,ur,t)
% функция создания коэффициентов граничного условия PDE системы
pl=[0;ul(2)];
ql=[1;0];
```

```
pr=[ur(1)-1;0];
qr=[0;1];
```

Для начальных условий $u_1(0,x)=x^2$; $u_2(0,x)=x(x-2)$ создадим функцию `initsys.m`

```
function value=initsys(x)
% начальное условие для PDE системы
value=[x^2;x.*(x-2)];
```

Теперь создадим сценарий `solvesys.m`, который будет решать систему и строить анимацию решения, т.е. строить график пары функций $u_1(x,t)$, $u_2(x,t)$ в различные моменты времени.

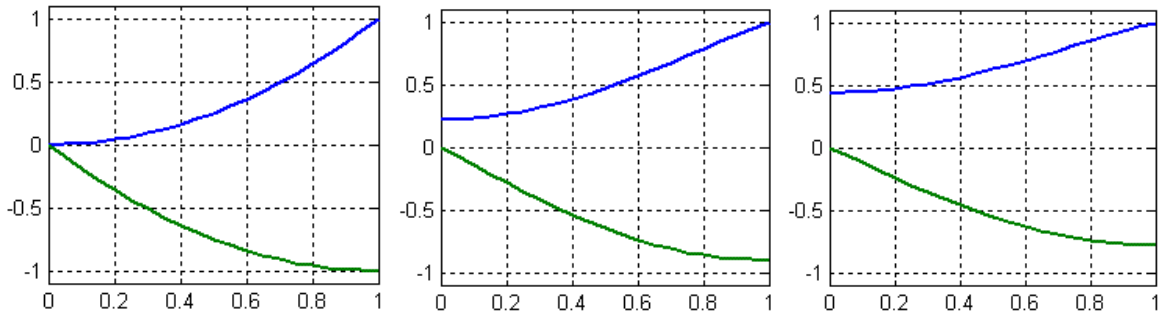
```
% сценарий решения PDE системы
m=0;
n=21;
x=linspace(0,1,n);
t=linspace(0,1,n);
sol=pdepe(m,@eqnsys,@initsys,@boundsys,x,t);
u1=sol(:,:,1);
u2=sol(:,:,2);

newplot;                              % создаем новое графическое окно
clear M;
for i=1:n,
    plot(x,u1(i,:),x,u2(i,:),'LineWidth',2);      % рисуем кадр
    axis([0 1 -1.1 1.1]);    % приводим кадр к общему масштабу
    grid on;
    title('u1(x,t), u2(x,t)');
    M(i)=getframe;                    % запоминаем кадр в массив
end
movie(M,4);                           % проигрываем массив кадров 4 раза
```

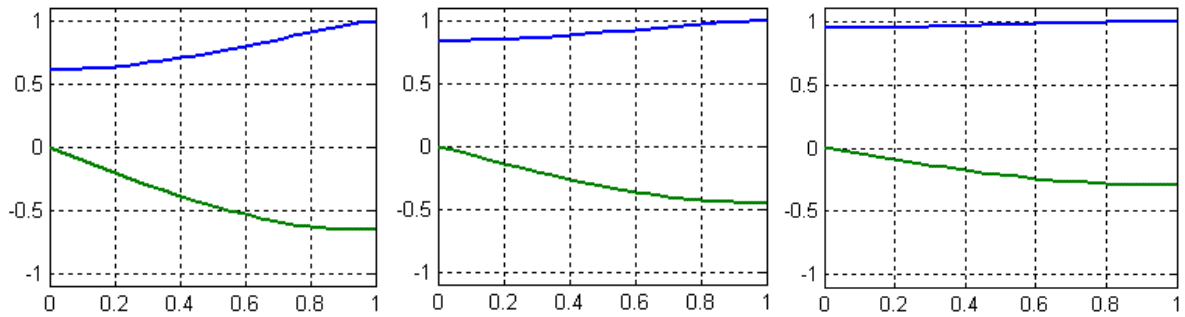
Для построения кадров по-отдельности (чтобы их можно было сохранить в отдельные графические файлы) создадим следующую функцию

```
function framesys2(nf,x,u1,u2)
% построение кадра решения PDE системы
% nf - номер момента времени (кадра)
plot(x,u1(nf,:),x,u2(nf,:),'LineWidth',2);
axis([0 1 -1.1 1.1]);
grid on;
```

Первый аргумент – номер кадра, который мы хотим построить, остальные – переменные, созданные в сценарии. Их необходимо либо передавать аргументами в тело нашей функции, либо объявлять как глобальные. Мы выбрали первый способ.

На следующем рисунке приведены графики функций u_1, u_2 в моменты времени `t = 0, 0.1, 0.2, 0.3, 0.5, 0.7.`





Заключительные замечания к главе.

В разделе 4.1 мы привели основные сведения, необходимые для решения некоторых ДУЧП. Однако пакет PDE TOOLBOX достаточно сложен и его возможности шире тех примеров, которые рассмотрены здесь. Перечислим некоторые из задач, решение которых возможно средствами пакета

- стационарные и нестационарные задачи теплопроводности;
- задачи диффузии;
- электростатические задачи;
- двумерные задачи теории упругости;
- задачи дифракции;
- распространение акустических и электромагнитных волн;
- определение собственных колебаний конструкций;
- ламинарное течение жидкости и газа;
- задачи теории пластин и оболочек.

Для решения той или иной прикладной задачи мы должны привести уравнения, описывающие явление, к виду, используемому одним из солверов пакета. Кроме того, читатель, владеющий навыками программирования в MATLAB, может легко расширить приведенный список, создавая собственные функции для решения задач из других областей техники и технологии.