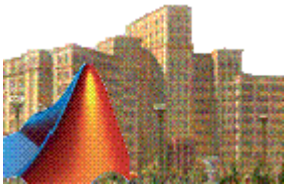


Доля П.Г.
Харьковский Национальный Университет
механико – математический факультет
кафедра геометрии им. А.В. Погорелова

# Использование MATLAB.
# Решение дифференциальных уравнений.

В предыдущих частях пособия были подробно рассмотрены основные элементы, необходимые для уверенного использования системы MatLab. В данной части мы приводим основные сведения о функциях MATLAB, предназначенных для решения дифференциальных уравнений и их систем.

## Оглавление

### Часть 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 части 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 будут описаны в других брошюрах.

## 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 dx/dt = -a·x(t)
z =C1*exp(-a*t)              % получаем x(t) = C1·e^(-at)
```

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 $\frac{dz}{dt} = 4z(t) + 3\sin t$

```
dsolve( 'Dz = 4*z + 3*sin(t)' )
ans = -3/17*cos(t)-12/17*sin(t)+exp(4*t)*C1
```

Поучаем $C_1e^{4t} - \frac{3}{17}\cos t - \frac{12}{17}\sin t$. 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'' + 4y' + 3y + 2 = 0
```

```
ans = exp(-3*t)*C2+exp(-t)*C1-2/3   % получаем
```
$C_1e^{-t} + C_2e^{-3t} - 2/3$

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 $y' = y \cdot ctg\, x + \dfrac{y^2}{\sin x}$. 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 e^{at}
```

**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

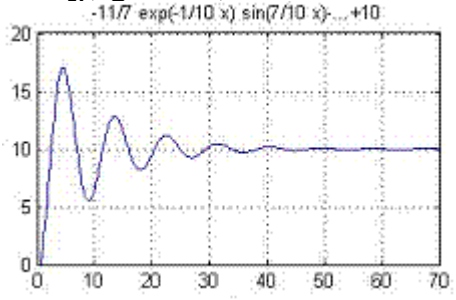
**q=dsolve('Du = v', 'Dv = -u')**
```
q =
    u: [1x1 sym]
    v: [1x1 sym]
```
**q.u**
```
ans = -C1*cos(t)+C2*sin(t)
```
**q.v**
```
ans = C1*sin(t)+C2*cos(t)
```

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;**

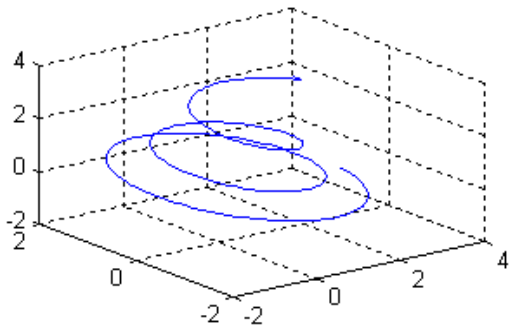
**z=diff(y,t)**
```
z = cos(t)-sin(t)
```
**ezplot(y,z,[0,2*pi])** `% строим фазовую траекторию = окружность`

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]);
```

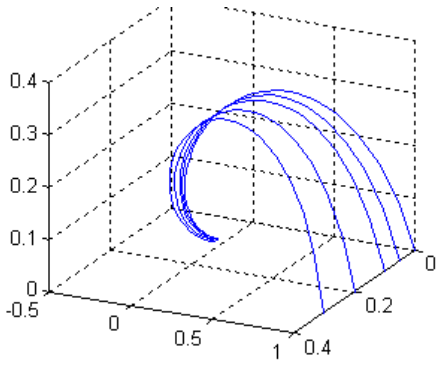



В следующем примере мы строим фазовые траектории системы дифференциальных уравнений

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

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

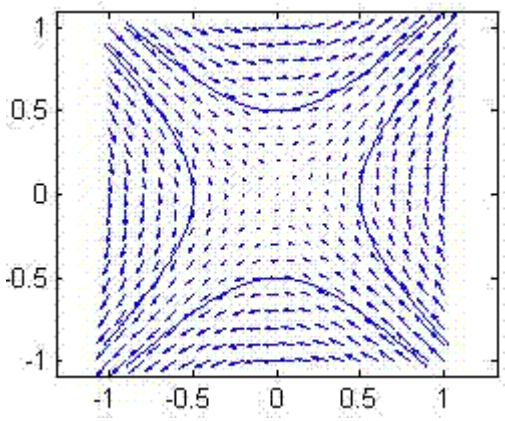
фазовой плоскости, а U,V – матрицы координат векторного поля в этих точках.

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

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

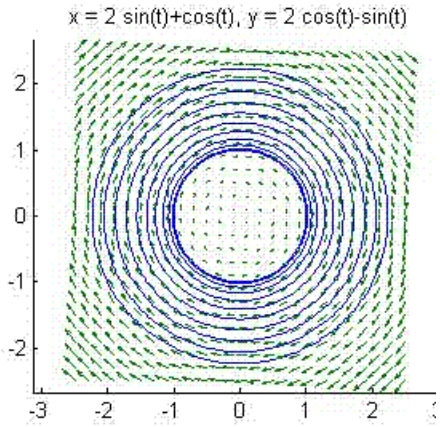
строит поле направлений этой системы и строит четыре фазовые траектории, проходящие через точки (0, 0.5), (0.5, 0), (-0.5,0), (0, -0.5).

```
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)
```

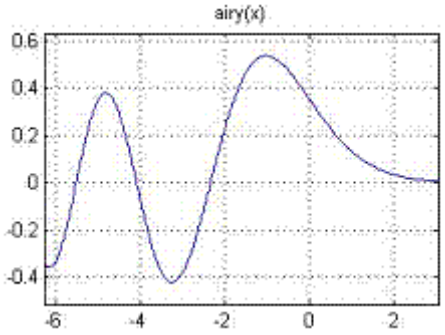
Однако следует иметь в виду, что обозначения функций Эйри в MatLab и MAPLE различается (функция `dsolve` наследует обозначения системы символьных вычислений MAPLE). Поэтому команда

**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')**

```
Цarning: 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 Использование функций ядра MAPLE для решения дифференциальных уравнений в MATLAB.

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`.

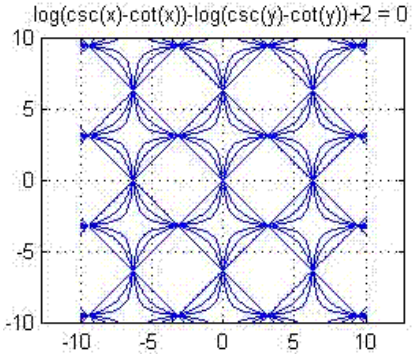
Создадим в пространстве имен ядра MAPLE переменную `sys`, которая будет содержать выражение системы двух дифференциальных уравнений
**maple 'sys:=diff(y(x),x)-2*z(x)-y(x)-x=0, diff(z(x),x)=y(x)'**
```
ans =
sys := diff(y(x),x)-2*z(x)-y(x)-x, diff(z(x),x) = y(x)
```
Тогда решить задачу Коши для этой системы 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

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

Задача сводится к дифференциальному уравнению

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

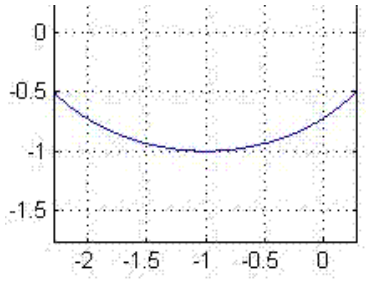
общее решение которого можно получить в параметрическом виде.

```
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 дифференциальных 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`

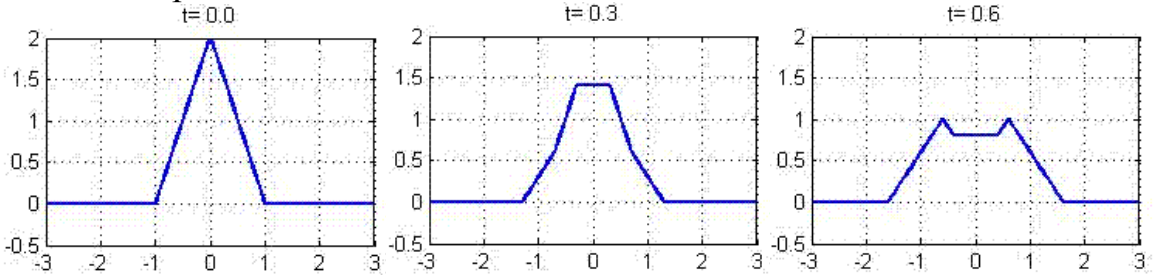
Функция `F1(x)`, используемая в решении, определяет профиль струны в начальный момент времени (`t=0`). Создадим функцию `F1(x)`

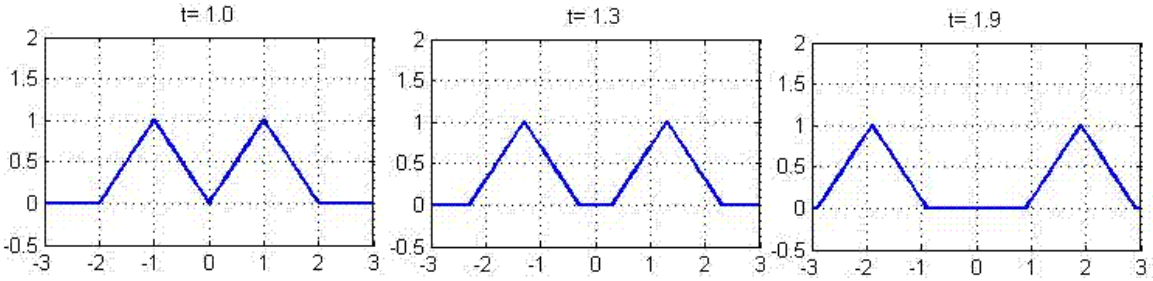
```
function F=F1(x)
F=0.5.*abs(x+1)-abs(x)+0.5.*abs(x-1);
```

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

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

и начальным условиям

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

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, ode113, ode15s, 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 `[t0, tend]` представляет диапазон изменения независимого переменного (начальное и конечное значение). `y0` – 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.

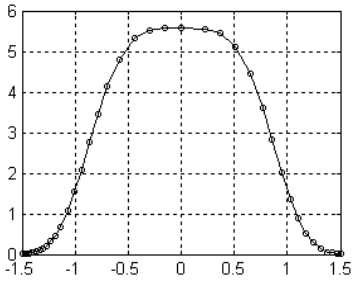
Пример 1. Рассмотрим задачу Коши $y' = -5t^3 \cdot y, \quad y(-1.5) = 0.01$.

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

$$\frac{d\,y_1}{dt} = y_2\,,\ \frac{d\,y_2}{d\,t} = -\frac{1}{5}y_2 - y_1,\ y_1(0) = 0,\ y_2(0) = 1$$

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]);**

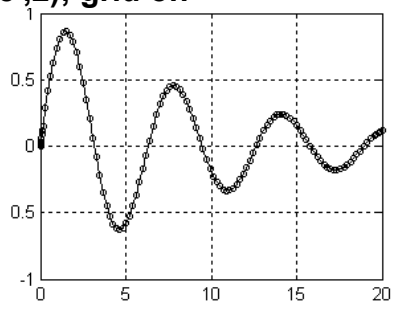
Здесь второй аргумент $[0, 20]$ функции `ode45` определяет диапазон изменения независимого аргумента, а третий $[0, 1]$ - начальные значения вектор – функции. Функция `ode45` возвращает решение в следующем виде

**[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

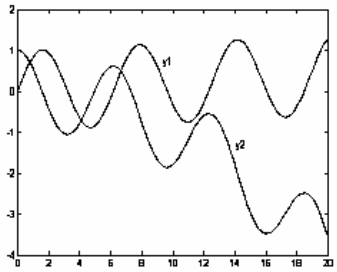
$$\begin{cases} y_1' = y_2 + K x^2 \\ y_2' = -y_1 \end{cases}, \quad \begin{pmatrix} y_1(0) \\ y_2(0) \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

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
```

□

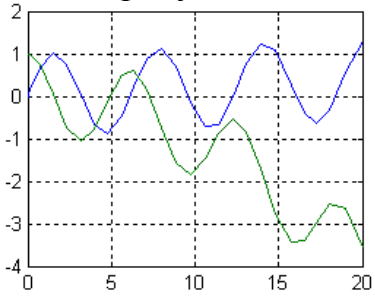
Интерфейс солверов допускает обращение к ним с одним выходным аргументом:

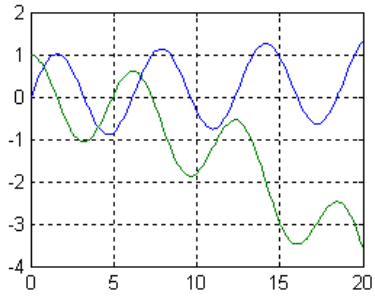
**sol = ode45(@odefn3,[0 ,20],[0,1]);**

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

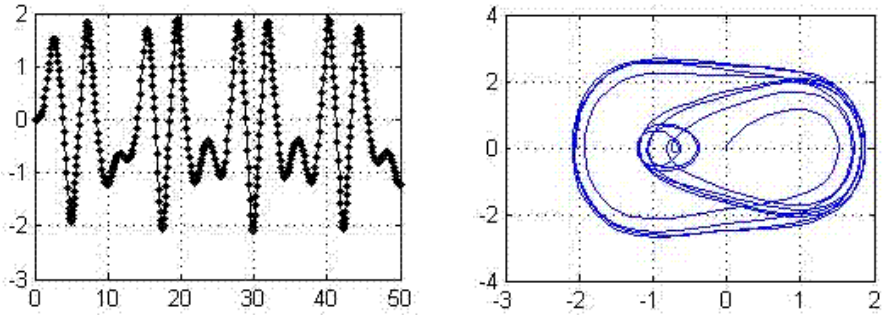
$$x'' + x^3 = \sin t\,,\ x(0) = 0, x'(0) = 0$$

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



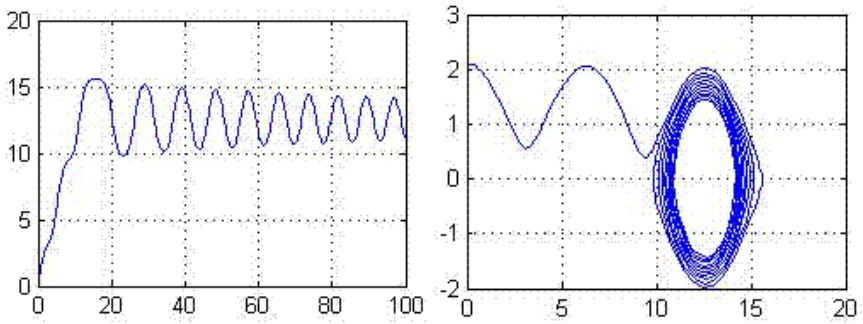
Пример 5. Исследуем решение задачи Коши для системы уравнений

$$x' = y(t),\ y' = -0.01\,y(t) - \sin x(t),\ x(0) = 0,\ y(0) = 2.1$$

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

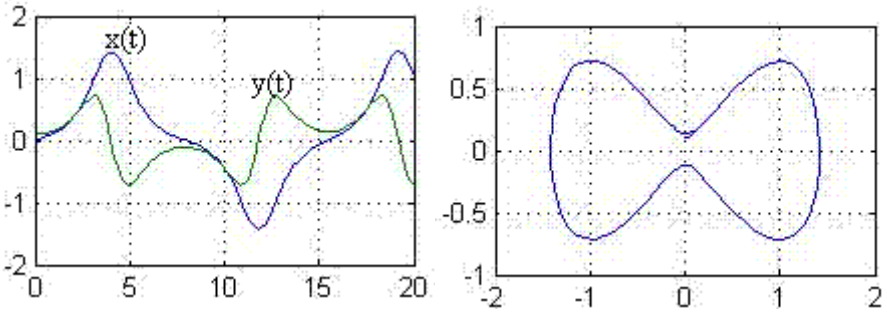
$$x' = y(t),\ y' = -x^3(t) + x(t),\ x(0) = 0,\ y(0) = 0.1$$

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.

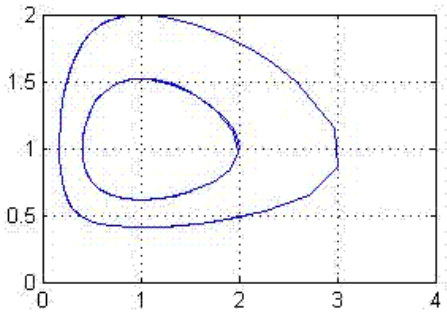



Пример 7. Рассмотрим двухвидовую модель «хищник – жертва», впервые построенную Вольтерра для объяснения колебаний рыбных уловов. Имеются два биологических вида, численностью в момент времени `t`, соответственно, `x(t)` и `y(t)`. Особи первого вида являются пищей для особей второго вида (хищников). Численности 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

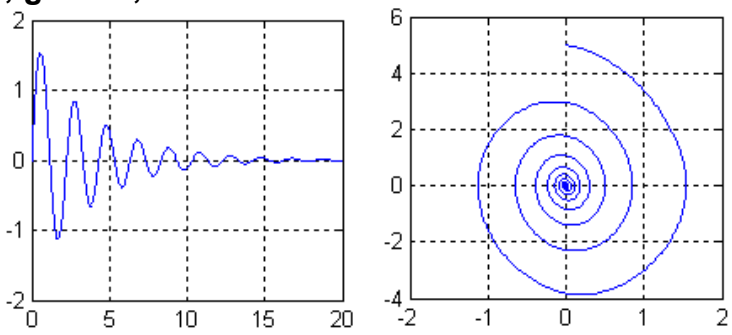
```
pend=@(t,y) [y(2); -0.5*y(2)-10*sin(y(1))];
```

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;
```



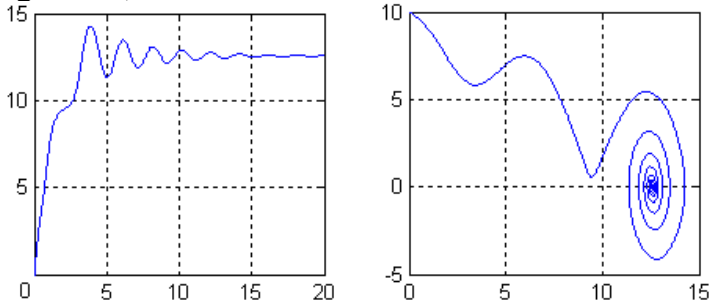

Как видно из левого графика максимальный угол отклонения маятника не превышают $\pi/2$ и колебания маятника затухают.

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;
```

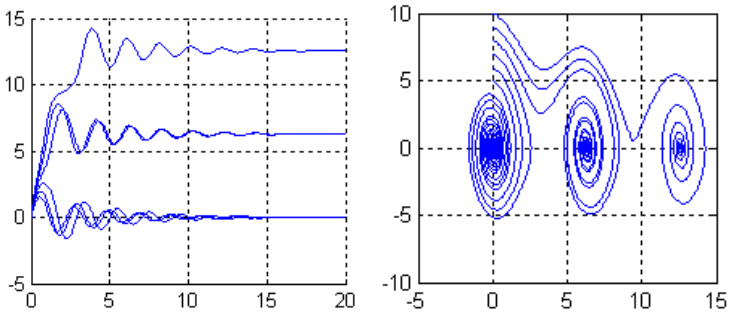



Максимальное значение угла составляет примерно 14 радиан. Маятник сделал два полных оборота вокруг точки закрепления (угол отклонения увеличился на $4\pi$), а затем колебания затухают в окрестности значения $4\pi$ радиан (для маятника угол поворота $4\pi$ представляет то же, что и 0 радиан, т.е. положение 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

$$z''' - x^2 z'' + x z' - z = 0$$

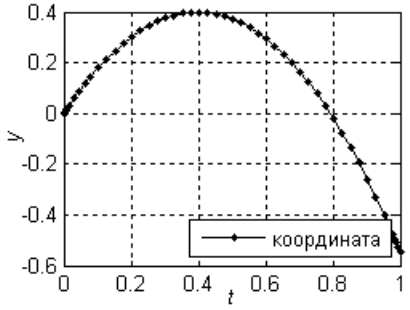
с начальными условиями $z(0)=0; z'(0)=2; z''(0)=-5$. Соответствующая задача Коши для системы ОДУ 1 – го порядка имеет вид

$$\begin{cases} u' = v \\ v' = w \\ w' = x^2 w - x v + u \end{cases}, \quad u(0)=0; v(0)=2; w(0)=-5$$

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

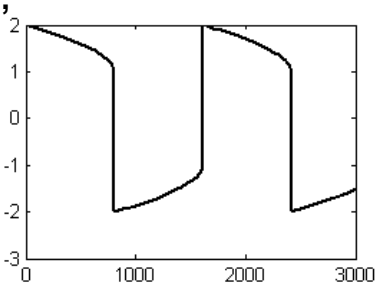
$$\begin{cases} y_1' = y_2 \\ y_2' = -y_1 + K \cdot (1 - y_1^2) \cdot y_2 \end{cases}, \quad \begin{pmatrix} y_1(0) \\ y_2(0) \end{pmatrix} = \begin{pmatrix} 2 \\ 0 \end{pmatrix}$$

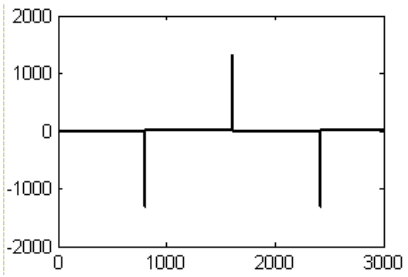
Создадим функцию правой части (полагаем `K=1000`)

```
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

$$\begin{cases} z_1' = z_2 \\ z_2' = -\dfrac{k\, z_1}{\left(z_1^2 + z_3^2\right)^{3/2}} \\ z_3' = z_4 \\ z_4' = -\dfrac{k\, z_3}{\left(z_1^2 + z_3^2\right)^{3/2}} \end{cases}$$

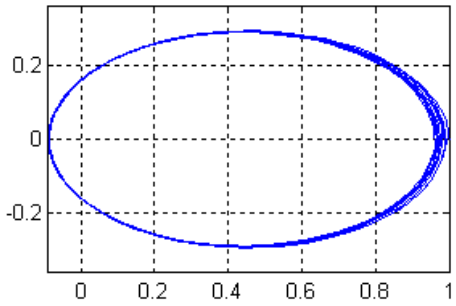
Для модельной задачи выберем *k=1* и зададим начальные условия

$$z_1(0) = 1, z_2(0) = 0, z_3(0) = 0, z_4(0) = 1$$

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 $x(t), y(t)$)

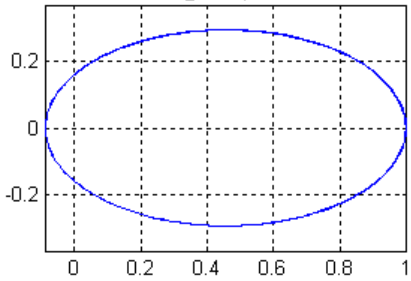



Для того, чтобы следить за движением точки по траектории, мы использовали функцию `comet`. Она позволяет получить анимированный график, на котором кружок, обозначающий точку, перемещается на плоскости, оставляя за собой след в виде линии – траектории движения. Следите за тем, чтобы окно с графиком было 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 $\varepsilon_i$ $i$-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.

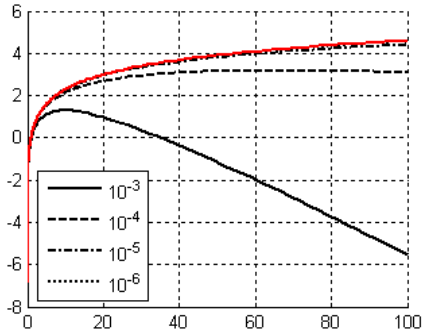
Пример 13. Решим дифференциальное уравнение $y'' = -\frac{1}{t^2}$ на отрезке $[a;100]$ с начальными условиями $y(a) = \ln(a)$, $y'(a) = 1/a$ при $a$=0.001. Его точное решение $y = \ln t$.

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` = $10^{-6}$.

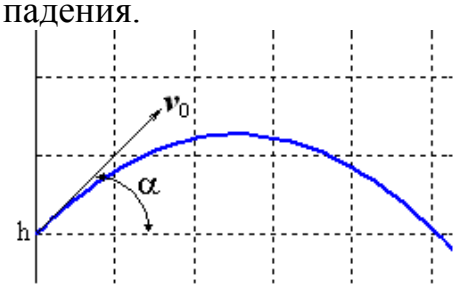



□

Пример 14. Решим задачу Коши, описывающую движение тела, брошенного с начальной скоростью $v_0$ под углом $\alpha$ к горизонту в предположении, что сопротивление воздуха пропорционально квадрату скорости. В векторной форме уравнение движения имеет вид

$$m\ddot{\mathbf{r}} = -\gamma \cdot \mathbf{v}\,|\,\mathbf{v}\,| - m\mathbf{g},$$

где $\mathbf{r}(t)$ радиус – вектор движущегося тела, $\mathbf{v} = \dot{\mathbf{r}}(t)$ – вектор скорости тела, $\gamma$ – 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`)

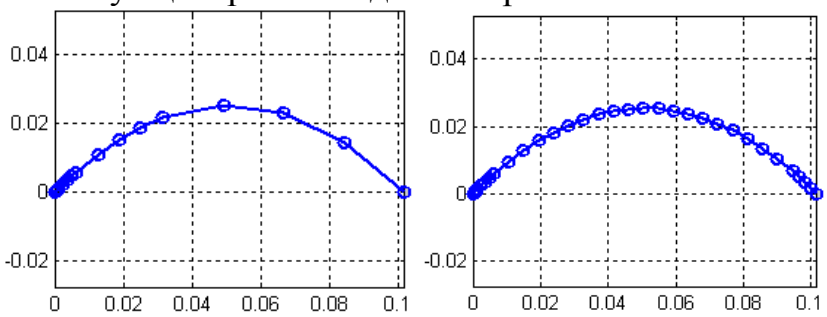
```
% движение тела брошенного под углом к горизонту
alph=pi/4;      % угол бросания тела
v0=1;           % начальная скорость
h=0;            % начальная высота
Y0=[0; v0.*cos(alph); h; v0.*sin(alph)];% вектор начальных условий
options = odeset('Events',@ex8events);  % сообщаем функцию обработчик событий
[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.



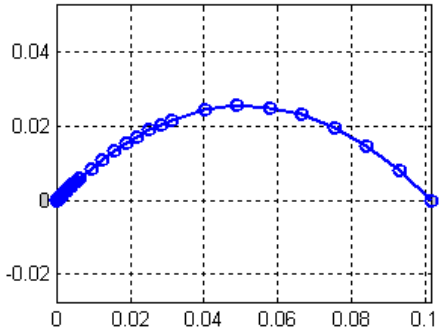

Однако уменьшение шага посредством параметра `MaxStep` увеличивает количество шагов, используемых численным методом, и увеличивает время решения системы ОДУ. Есть еще один параметр `Refine`, управляющий количеством точек решения, возвращаемых солвером на каждом интервале. Если `Refine` равен 1, то солвер 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 $\left[y(1), y(2), y(3), y(4)\right]$ 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. $\dot{y}_{\text{после отскока}}(t_k) = -0.9 \cdot \dot{y}_{\text{до отскока}}(t_k)$ 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

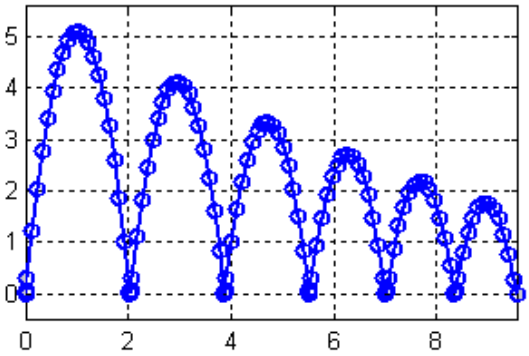
```
% сценария example15_1.m для моделирования движения одного прыжка мячика
h=0;       % начальная высота
vox=1;      % начальная скорость вдоль оси x
voy=10;     % начальная скорость вдоль оси y
Y0=[h; voy];    % вектор начальных условий
opts = odeset('Events',@events9,'MaxStep',0.1);  % задаем параметры
```

```
[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];
```

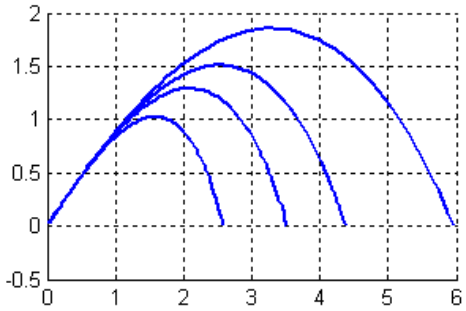
Если используется функция обработчик события параметра `Events`, то она также должна иметь этот дополнительный аргумент.

```
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

$$\begin{cases} y_1' = y_2 \\ y_2' = -y_2 - 5y_1 + \sin t \end{cases}, \qquad \begin{pmatrix} y_1(0) \\ y_2(0) \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

Создаем функцию правой части системы

```
function F=fun10(x,y)
F=[y(2);-y(2)-5.*y(1)+sin(x)];
```

Выполняем команды

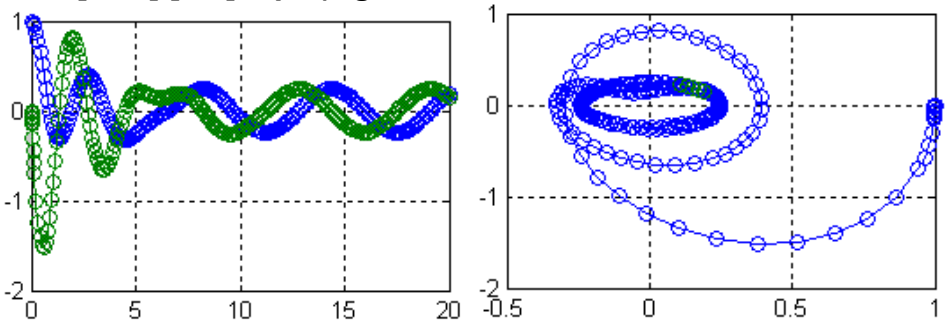
**opts = odeset('OutputFcn',@odeplot);**
**ode45(@fun10,[0 20],[1;0], opts); grid on;**

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)];
```

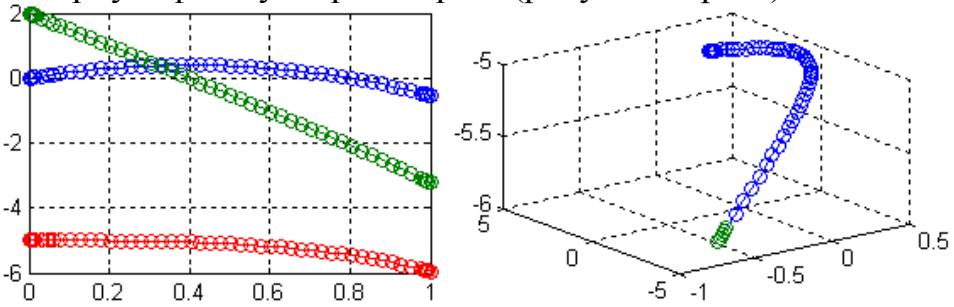
команда

**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

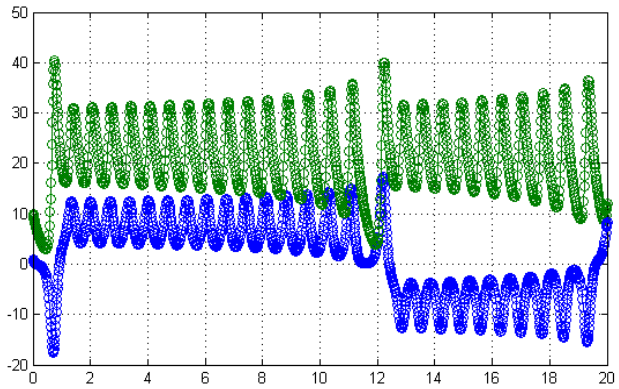
```
function f=lor(t,y,s,r,b)
% правая часть системы уравнений Лоренца
f=[s.*(y(2)-y(1));...
    -y(2)+(r-y(3)).*y(1);...
    -b.*y(3)+y(1).*y(2)];
```

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;**

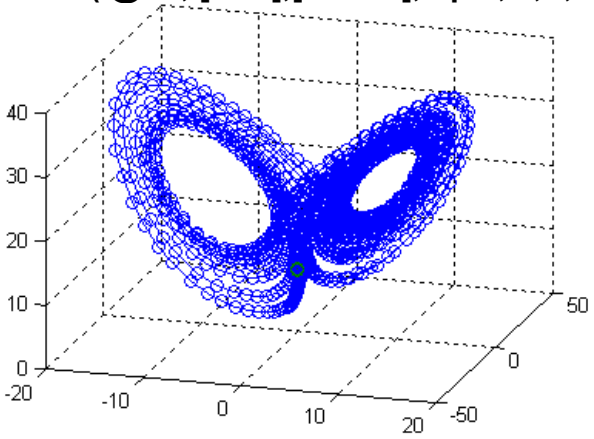


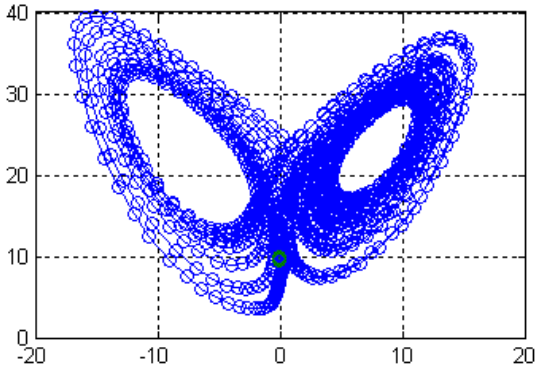

Для построения трехмерной фазовой траектории (следующий рисунок слева) выполним команды

**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

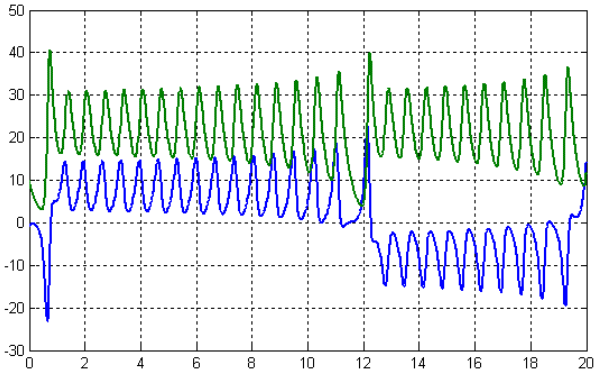
**opts=odeset('OutputSel',[1 3],'OutputFcn',@odephas2 );**
**ode45( @lor, [0 20],[1 -1 10], opts, s, r, b );**

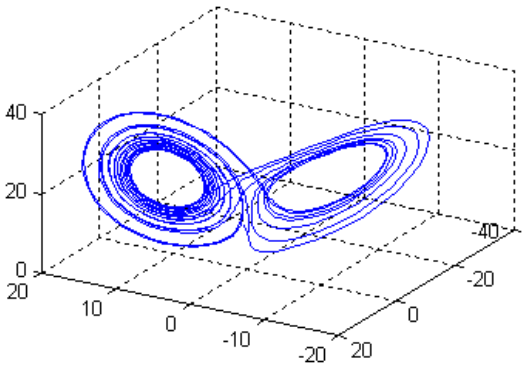



Если использовать возвращаемые значения, то предыдущие графики можно построить, например, следующим образом

```
[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

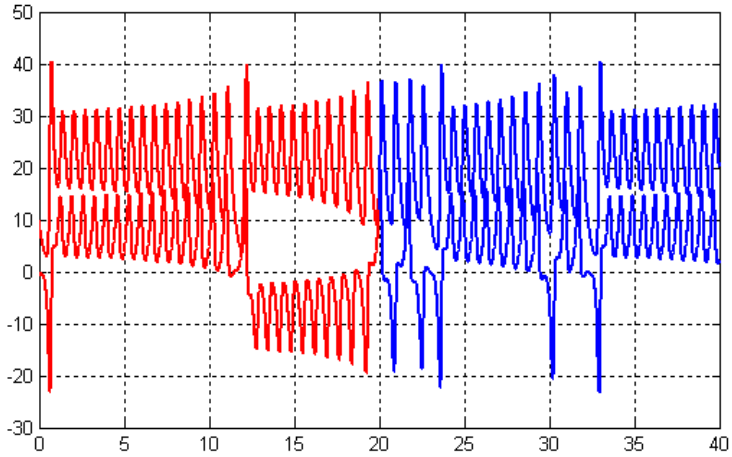
```
comet(Y(:,1), Y(:,3));
```

А движение по трехмерной фазовой траектории - командой

```
comet3(Y(:,1),Y(:,2), Y(:,3));
```

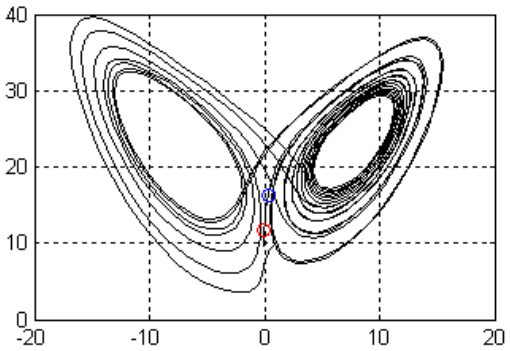
Если мы захотим продолжить расчет, начав решение системы в точке в которой закончился предыдущий расчет, то следует конечные значения решения первого расчета использовать как начальные значения для следующего.

```
[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

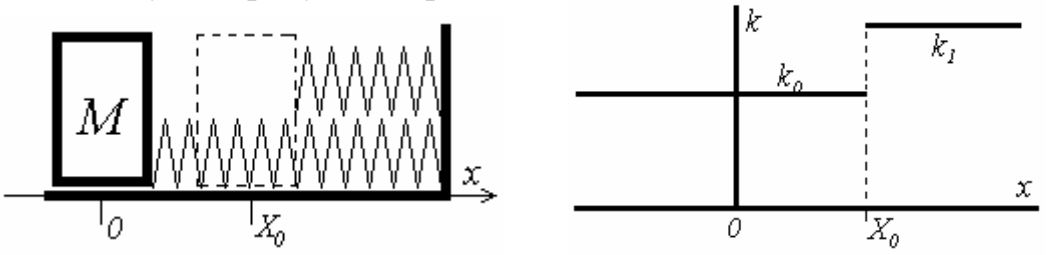
- `'init'` – передается солвером при первом вызове до начала интегрирования, параметр `t` является вектором из двух элементов – границ отрезка интегрирования, а `y` – является вектором начальных значений;
- `''` (пустая строка) передается после каждого шага интегрирования, параметр `t` является текущим 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.



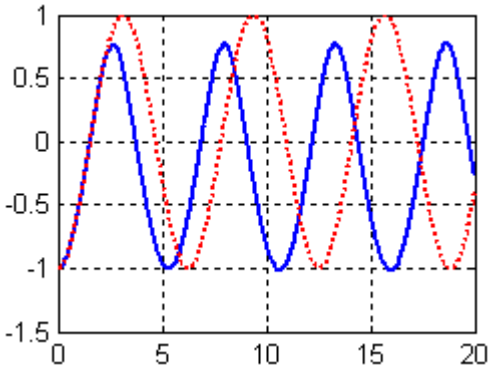

Для тела M выполняется уравнение движения $\ddot{x}(t) + k^2 x(t) = 0$. При движении вправо и достижении положения $X_0$ тело M присоединяется ко второй пружине и жесткость системы меняется. Когда тело движется влево, и проходит положение $X_0$ оно отрывается от второй пружины и жесткость уменьшается. 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 M 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`.

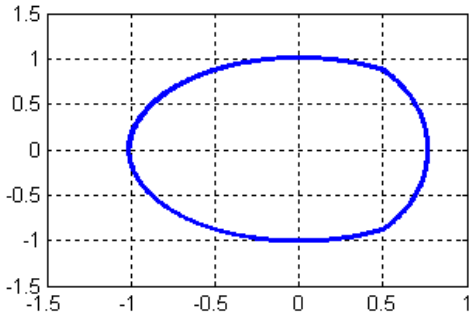



Для тренировки создадим функцию `bodymove2.m`, которая выполняет некоторый код перед началом интегрирования (выводит сообщение) и после его завершения (выводит сообщение и строит фазовую траекторию, если солвер `ode113` вызвать без возвращаемых значений)

```
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, ,[t0, tend], y0, 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.

## ЗАКЛЮЧИТЕЛЬНЫЕ ЗАМЕЧАНИЯ

В данной брошюре 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.