常微分方程的解析解和数值解

`符号解`以代数符号与运算的形式完整地写出解的代数式，强调**解的代数性**

数值解在给定方程的**一些初始条件下**不按式子来，而是**算出一个数值**，这个数值可以不需要完完本本的精确，满足一定的要求即可

举一个简单的例子，解一个与圆有关的方程得到的符号解为 $x = 0.32\pi$，求一个数值就仅需在 π 的限制下，得出 $x = 1.0048$ 。

实际科学与工程中，大多数微分方程求不出符号解，只能在`不同取值条件`下求一个数值解

> - 如何编写算法使精度提高？
> - 数值解会随着初始条件变化而变化，如何变化？函数值又与自变量之间怎么变化？
> - 这涉及到微分方程数值解的演化问题和灵敏度分析问题

符号解：`sympy`

使用`sympy.dsolve`解决微分方程符号解的一种良好方式，而对于有初始值的微分方程问题，在求出其通解形式后通过解方程组的方法得到参数。通过声明符号变量求得最优解。

例一：

$$y'' + 2y' + y = x^2$$

```python
from sympy as sp
# 符号函数
y = sp.symbols("y", cls=sp.Function)
x = sp.symbols("x")
# diff(x, 2) 代表 y 对 x 求两次
eq = sp.Eq(y(x).diff(x, 2) + 2 * y(x).diff(x, 1) + y(x), x * x)
sp.dsolve(eq, y(x))
```

$$\$y(x) = x^2 - 4x + (C_1 + C_2x)e^{-x} + 6\$$$

例二：

解方程组：

$$\begin{cases} \dfrac{dx_1}{dt} = 2x_1 - 3x_2 + 3x_3,\ x_1(0) = 1 \\ \dfrac{dx_2}{dt} = 4x_1 - 5x_2 + 3x_3,\ x_2(0) = 2 \\ \dfrac{dx_3}{dt} = 4x_1 - 4x_2 + 2x_3,\ x_3(0) = 3 \end{cases}$$

解法一（使用`dsolve`）：

```
t = sp.symbols("t")
x1, x2, x3 = sp.symbols("x1, x2, x3", cls=sp.Function)
# 函数，所以以 x1(0) 形式
eq1 = sp.Eq(x1(t).diff(t), 2 * x1(t) - 3 * x2(t) + 3 * x3(t))
eq2 = sp.Eq(x2(t).diff(t), 4 * x1(t) - 5 * x2(t) + 3 * x3(t))
eq3 = sp.Eq(x3(t).diff(t), 4 * x1(t) - 4 * nx2(t) + 2 * x3(t))
# 初始条件以字典的方式传入
con = {x1(0): 1, x2(0): 2, x3(0): 3}
s = sp.dsolve([eq1, eq2, eq3], ics=con)
s
```

```
[Eq(x1(t), 2*exp(2*t) - exp(-t)),
 Eq(x2(t), 2*exp(2*t) - exp(-t) + exp(-2*t)),
 Eq(x3(t), 2*exp(2*t) + exp(-2*t))]
```

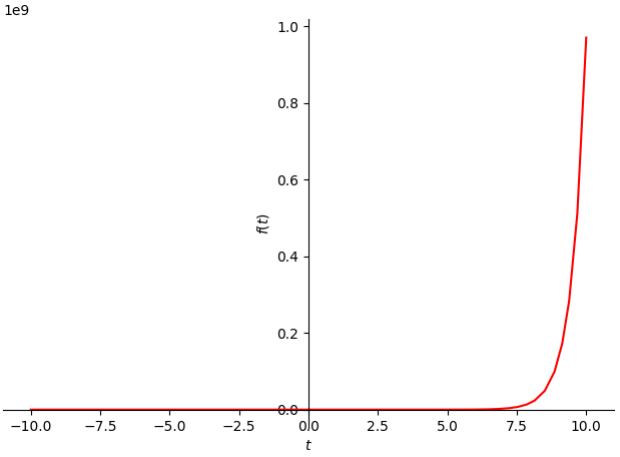
解法二（使用矩阵）：

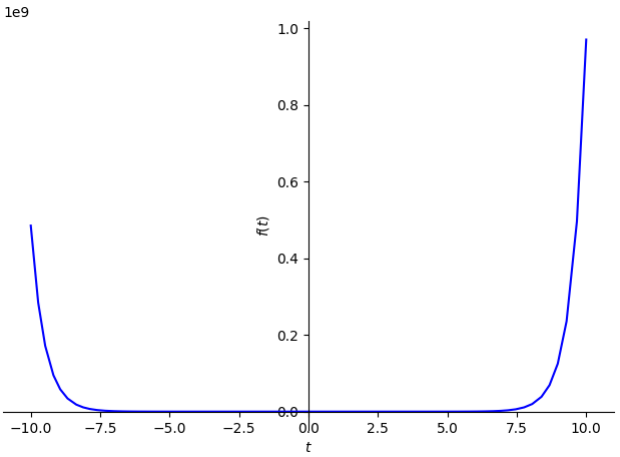
```
# 一维向量
x = sp.Matrix([x1(t), x2(t), x3(t)])
A = sp.Matrix([[2, -3, 3], [4, -5, 3], [4, -4, 2]])
# 注意，这里不能使用 sp.Eq() 的形式
eq = x.diff(t) - A * x
s = sp.dsolve(eq, ics={x1(0): 1, x2(0): 2, x3(0): 3})
s
```

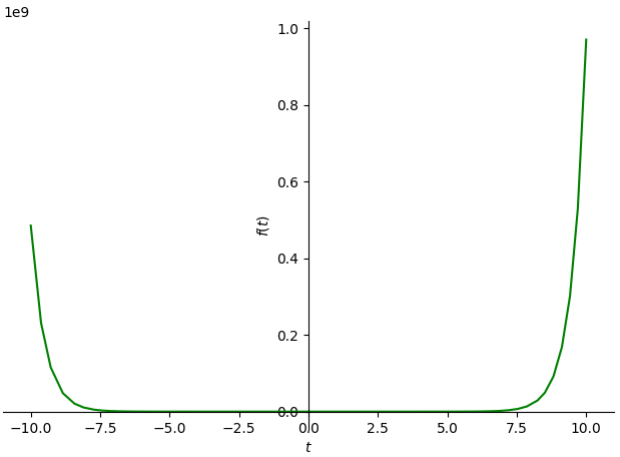
```
[Eq(x1(t), 2*exp(2*t) - exp(-t)),
 Eq(x2(t), 2*exp(2*t) - exp(-t) + exp(-2*t)),
 Eq(x3(t), 2*exp(2*t) + exp(-2*t))]
```

而 `sympy` 中的绘图代码：

```
from sympy.plotting import plot

t = sp.Symbol("t")
plot(2 * sp.exp(2 * t) - sp.exp(-t), line_color="red")
plot(2 * sp.exp(2 * t) - sp.exp(-t) + sp.exp(-2 * t), line_color="blue")
plot(2 * sp.exp(2 * t) + sp.exp(-2 * t), line_color="green")
```







数值解：`scipy`

Python 求微分方程的数值解需要应用 `scipy.odeint` 方法，内部 `ode` 使用了欧拉法和龙格库塔法

`scipy.integrate` 下面有两个函数可以支持:

`odeint()` 函数需要至少三个变量，第一个是微分方程函数，第二个是微分方程初值，第兰个是微分的自变量。

使用 `odeint`，老式，但实在。

调用 `solve_ivp` 来生成微分方程的数值解，包含的参数包括：系统模型函数，时间 `t` 的范围，状态初值，以及参数的真体值。

使用 `solve_ivp`，支持方法更多，但没那么稳定

例三：

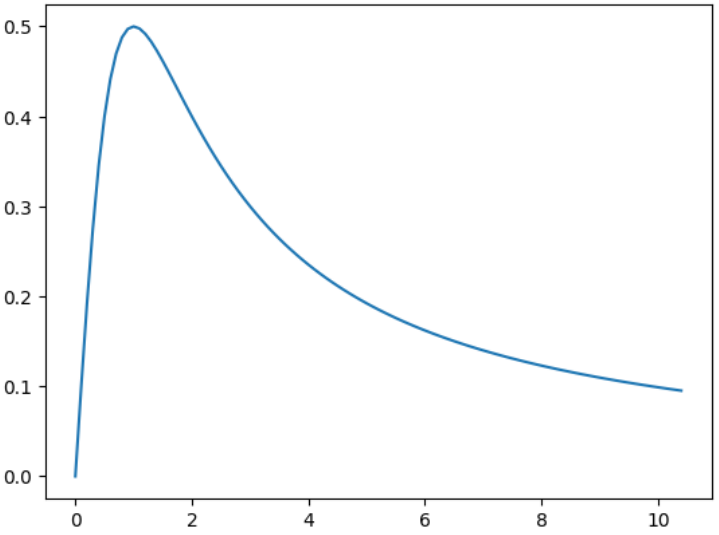
解方程：

$$y' = \frac{1}{x^2+1} - 2y^2,\ y(0) = 0$$

```python
from scipy.integrate import odeint
import matplotlib.pyplot as plt
import numpy as np

dy = lambda y, x: 1 / (1 + x**2) - 2 * y**2
# 从 0 开始，每次增加 0.1，加到 10.5 为止（取不到 10.5）
x = np.arange(0, 10.5, 0.1)
# 微分方程 dy，y 的首项（y(0) 等于多少），自变量列表
sol = odeint(dy, 0, x)
print("x = {}\ny = {}".format(x, sol.T))
plt.plot(x, sol)
plt.show()
```

```
x = [ 0.   0.1  0.2  0.3  0.4  0.5  0.6  0.7  0.8  0.9
 1.   1.1  1.2  1.3
  1.4  1.5  1.6  1.7  1.8  1.9  2.   2.1  2.2  2.3  2.4
 2.5  2.6  2.7
  2.8  2.9  3.   3.1  3.2  3.3  3.4  3.5  3.6  3.7  3.8
 3.9  4.   4.1
  4.2  4.3  4.4  4.5  4.6  4.7  4.8  4.9  5.   5.1  5.2
 5.3  5.4  5.5
  5.6  5.7  5.8  5.9  6.   6.1  6.2  6.3  6.4  6.5  6.6
 6.7  6.8  6.9
  7.   7.1  7.2  7.3  7.4  7.5  7.6  7.7  7.8  7.9  8.
8.1  8.2  8.3
  8.4  8.5  8.6  8.7  8.8  8.9  9.   9.1  9.2  9.3  9.4
 9.5  9.6  9.7
  9.8  9.9 10.  10.1 10.2 10.3 10.4]
y = [[0.         0.09900996 0.19230775 0.2752294
 0.34482762 0.40000004
  0.44117651 0.46979872 0.48780495 0.49723764 0.50000005
0.49773759
  0.49180331 0.48327139 0.47297298 0.46153847 0.44943821
0.437018
  0.42452831 0.41214752 0.40000002 0.38817007 0.37671235
0.3656598
  0.35502961 0.34482761 0.33505157 0.32569362 0.31674209
0.30818279
  0.30000001 0.2921772  0.28469751 0.27754416 0.27070064
0.26415095
```

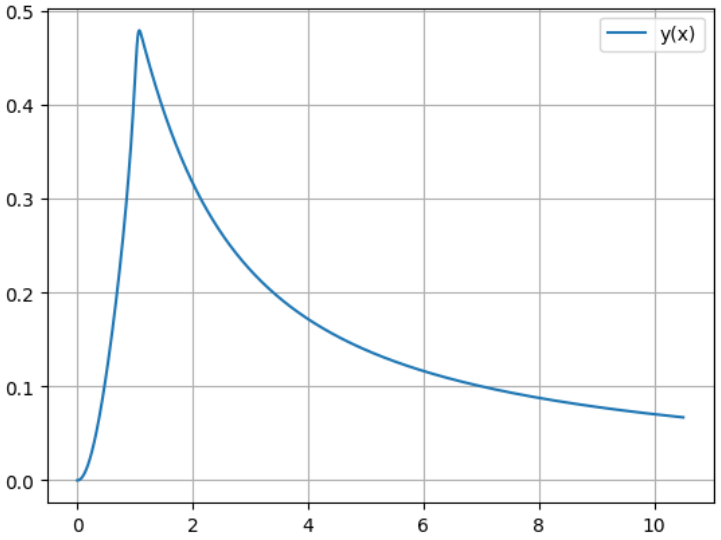
```
15    0.25787966 0.25187202 0.24611399 0.24059223 0.23529412
    0.23020775
16    0.22532189 0.22062596 0.21611002 0.2117647  0.20758123
    0.20355132
17    0.19966722 0.19592163 0.19230769 0.18881895 0.18544936
    0.18219319
18    0.17904509 0.176      0.17305315 0.17020006 0.16743649
    0.16475845
19    0.16216216 0.15964407 0.15720081 0.15482919 0.15252621
    0.15028901
20    0.1481149  0.1460013  0.1439458  0.1419461  0.13999999
    0.13810542
21    0.1362604  0.13446306 0.13271162 0.13100436 0.12933968
    0.12771603
22    0.12613195 0.12458602 0.12307692 0.12160336 0.12016412
    0.11875804
23    0.11738401 0.11604095 0.11472785 0.11344373 0.11218765
    0.11095873
24    0.10975609 0.10857892 0.10742643 0.10629786 0.10519247
    0.10410958
25    0.10304851 0.10200862 0.10098928 0.09998989 0.09900989
    0.09804873
26    0.09710586 0.09618078 0.09527299]]
27
```



使用差分法求解：

```
1  X = 10.5
2  x0 = 0
```

```
y0 = 0
dx = 0.001

x = x0
y = y0

x_list = [x0]
y_list = [y0]

while x < X:
    # 先求一个近似的 y_(n+1) 以便于后面的计算求出更精确的
    y_ = y + 1 / (x**2 + 1) - 2 * y**2
    x_ = x + dx
    y = 0.5 * (1 / (x_**2 + 1) - 2 * y_**2 + 1 / (x**2 + 1) - 2 * y**2) + y
    x = x_
    y_list.append(y)
    x_list.append(x)

plt.plot(x_list, y_list, label="y(x)")
plt.legend()
plt.grid("on")
plt.show()
```

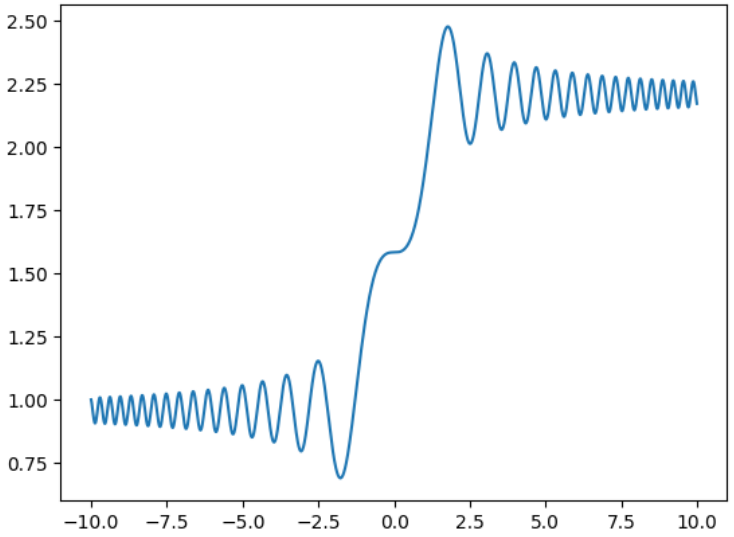


可以看到精度较上面较差。

例四：

$$\begin{cases} \dfrac{dy}{dt} = \sin t^2 \\ y(-10) = 1 \end{cases}$$

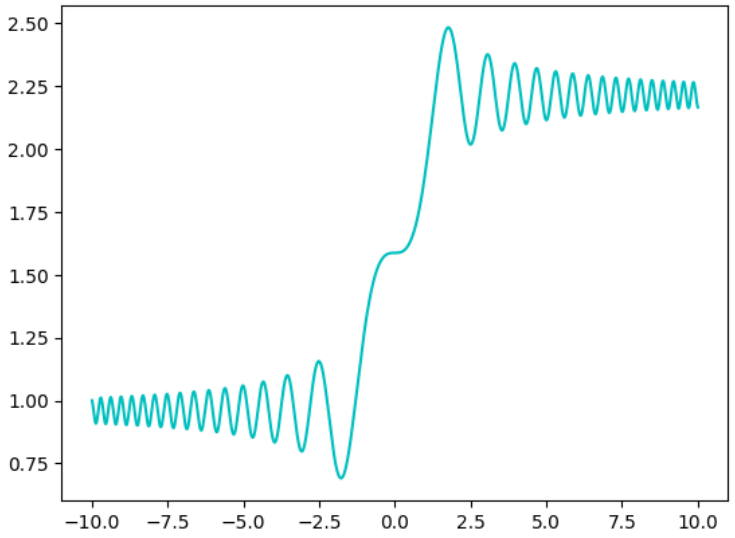
```python
# 注意，这里的式子无论使用 lambda 还是正常定义函数的方式
# 均要传入两个变量（下面为 y 和 t ）
dy_dt = lambda y, t: np.sin(t**2)
t = np.arange(-10, 10, 0.01)
y = odeint(dy_dt, 1, t)
plt.plot(t, y)
```



$$\begin{cases} \left[\dfrac{dy}{dt} \approx \dfrac{y_{n+1} - y_n}{\Delta t}\right] \\ \left[y_{n+1} = y_n + \dfrac{\Delta t}{2}(f(t_n, y_n) + f(t_{n+1}, y_{n+1}))\right] \end{cases}$$

```python
# 差分法求解
dt = 0.01
t0 = -10.0
y0 = 1
T = 10.0

t_list = [t0]
y_list = [y0]

t = t0
y = y0

while t < T:
```

```
    t += dt
    y = y + dt * np.sin(t**2)
    t_list.append(t)
    y_list.append(y)

plt.plot(t_list, y_list, 'c', label="T")
plt.show()
```



例五：

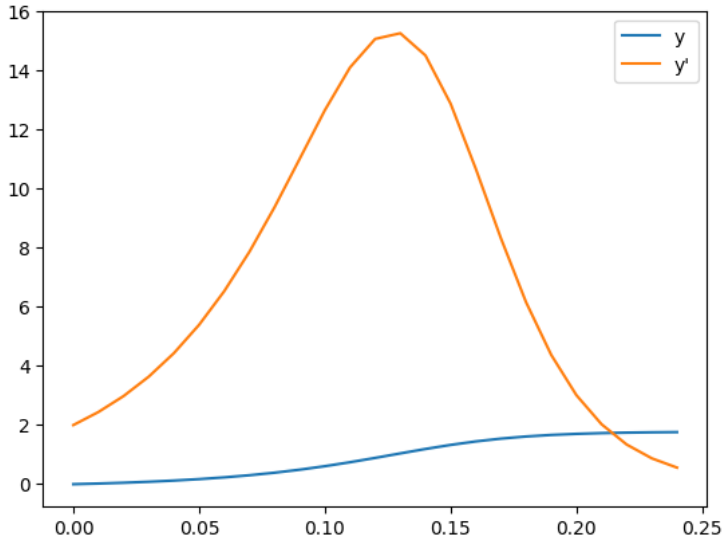
高阶微分方程，必须做变量替换，化为一阶微分方程组，再用 `odeint` 求数值解：

$$\begin{cases} y'' - 20(1-y^2)y' + y = 0 \\ y(0) = 0,\ y'(0) = 2 \end{cases}$$

做高阶时，输入为数组 $[y,\ y']$ 去求导 $[y',\ y'']$

```
# 定义一个函数，表示待求解的二阶常微分方程
def fvdp(y, t):
    dy1 = y[1]  # 第一个方程表示y'，即y的一阶导数
    dy2 = 20 * (1 - y[0] ** 2) * y[1] - y[0]  # 第二个方程表示y''，即y的二阶导数
    return [dy1, dy2]  # 返回y'和y''组成的数组


# 定义一个函数，用于求解二阶常微分方程并绘制结果
def solve_second_order_ode():
    x = np.arange(0, 0.25, 0.01)  # 创建一个数组，表示自变量x的取值范围
```

```
    y0 = [0.0, 2.0]  # 初值条件，表示y(0)=0.0，y'(0)=2.0
    y = odeint(fvdp, y0, x)  # 使用odeint函数求解二阶常微分方程

    (y1,) = plt.plot(x, y[:, 0], label="y")  # 绘制y关于x的图像
    (y1_1,) = plt.plot(x, y[:, 1], label="y'")  # 绘制y'关于x的图像
    plt.legend(handles=[y1, y1_1])  # 显示图例

solve_second_order_ode()  # 调用函数进行求解和绘图
```



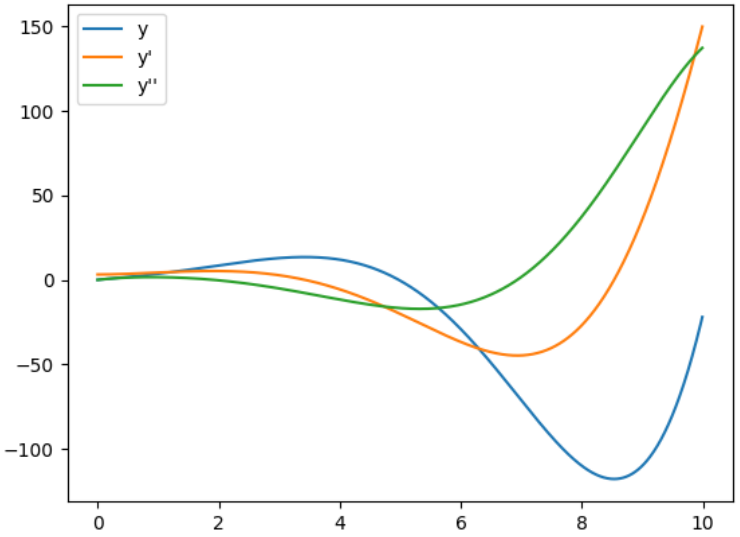

```
x0 = 0
X = 0.25
dx = 0.01
y0 = 0
yp0 = 2

x_list = [x0]
y_list = [y0]
yp_list = [yp0]

x = x0
y = y0
yp = yp0

while x < X:
    y_ = y + dx * (yp)
    yp_ = yp + dx * (20 * (1 - y**2) * yp - y)
    x += dx
```

```
    y = y_
    yp = yp_
    x_list.append(x)
    y_list.append(y)
    yp_list.append(yp)

plt.plot(x_list, y_list, label="y")
plt.plot(x_list, yp_list, label="y'")
plt.legend()
plt.grid("on")
```

例六：

$$\begin{cases} y''' + y'' - y' + y = \cos t \\ y(0) = 0,\ y'(0) = \pi,\ y''(0) = 0 \end{cases}$$

```
def fvdp(y, t):
    dy1 = y[1]
    dy2 = y[2]
    dy3 = np.cos(t) - y[2] + y[1] - y[0]
    return [dy1, dy2, dy3]


def solution():
    x = np.arange(0, 10, 0.01)
    y0 = [0, np.pi, 0]
    y = odeint(fvdp, y0, x)
    (y1,) = plt.plot(x, y[:, 0], label="y")
    (y1_1,) = plt.plot(x, y[:, 1], label="y'")
    (y1_2,) = plt.plot(x, y[:, 2], label="y''")
    plt.legend(handles=[y1, y1_1, y1_2])


solution()
```

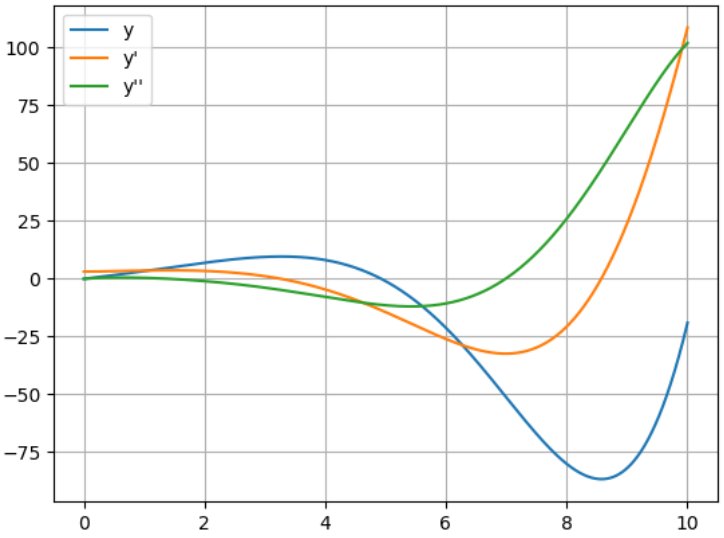


总结：使用 `odeint` 解多元方程组时，首先判断是几阶微分方程，低阶的导数使用变量代替，如 `dy1 = y[1]`、`dy2 = y[2]` 等，而对应阶数，如 `dy3` 就用低阶的变量表示（因为给定初值条件中，没有 `y[3]` 这个，而方程中需要用的）

使用差分法：

```
x0 = 0
X = 10
dx = 0.01
y0 = 0
yp0 = np.pi
ypp0 = 0

x_list = [x0]
y_list = [y0]
yp_list = [yp0]
ypp_list = [ypp0]

x = x0
y = y0
yp = yp0
ypp = ypp0

while x < X:
    y_ = y + dx * yp
    yp_ = yp + dx * ypp
    ypp_ = ypp + dx * (yp - ypp - y - np.cos(x))
    x += dx
    y = y_
    yp = yp_
```

```
    ypp = ypp_
    x_list.append(x)
    y_list.append(y)
    yp_list.append(yp)
    ypp_list.append(ypp)

plt.plot(x_list, y_list, label="y")
plt.plot(x_list, yp_list, label="y'")
plt.plot(x_list, ypp_list, label="y''")
plt.legend()
plt.grid("on")
plt.show()
```



可以看出准确率较高。

```
n = 1000
dx = 0.01

x = np.zeros(n)
y = np.zeros(n)
yp = np.zeros(n)
ypp = np.zeros(n)

x[0] = 0
y[0] = 0
yp[0] = np.pi
ypp[0] = 0

for i in range(n - 1):
    x[i + 1] = 10 * i / n
```

```
    y[i + 1] = y[i] + dx * yp[i]
    yp[i + 1] = yp[i] + dx * ypp[i]
    ypp[i + 1] = ypp[i] + dx * (np.cos(x[i]) - y[i] + yp[i] - ypp[i])

plt.plot(x, y, label="y")
plt.plot(x, yp, label="y'")
plt.plot(x, ypp, label="y''")
plt.legend()
plt.grid(True)
plt.show()
```

使用 `solve_ivp` 求解:

```
import numpy as np
from scipy.integrate import odeint, solve_ivp
import matplotlib.pyplot as plt


def fvdp(t, y):
    dy1 = y[1]
    dy2 = y[2]
    dy3 = np.cos(t) - y[2] + y[1] - y[0]
    return [dy1, dy2, dy3]


def solution():
    x = np.linspace(0, 6, 1000)
    y0 = [0, np.pi, 0]
    tspan = (0.0, 6.0)
    # 由于传参顺序不一致，所以使用 lambda 表达式来调换顺序
    y = odeint(lambda y, t: fvdp(t, y), y0, x)
    y_ = solve_ivp(fvdp, t_span=tspan, y0=y0, t_eval=x)

    plt.subplot(211)
    (l1,) = plt.plot(x, y[:, 0], label="y0")
    (l2,) = plt.plot(x, y[:, 1], label="y1")
    (l3,) = plt.plot(x, y[:, 2], label="y2")
    plt.legend(handles=[l1, l2, l3])

    plt.subplot(212)
```

```
    (l4,) = plt.plot(y_.t, y_.y[0, :], label="y0 - solve_ivp")
    (l5,) = plt.plot(y_.t, y_.y[1, :], label="y1 - solve_ivp")
    (l6,) = plt.plot(y_.t, y_.y[2, :], label="y2 - solve_ivp")
    plt.legend(handles=[l4, l5, l6])

    plt.show()


solution()
```

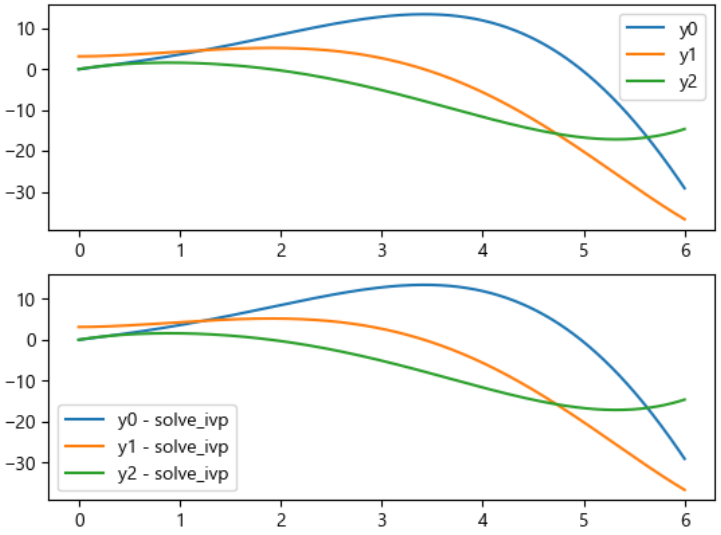
注意：

- `odeint` 传参数时，有 `func`、`y0`、`t` 这几个基本的参数，而它要求的 `func` 函数是 先 `y` 后 `t`，即先因变量，后自变量；

  返回值中的 `y` 每一列代表一个元素，一行代表某个时间点

- `solve_ivp` 传参数时，有 `func`、`y0`、`t_span`、`t_eval`，`func` 的传参顺序与 `odeint` 中的不同，先 `t` 后 `y`，即先自变量，后因变量 `t_span` 代表自变量的范围，用元组表示；`t_eval` 则是离散的自变量，便于求解，

  `solve_ivp` 会返回一个对象，`obj.y` 代表因变量的取值，`obj.t` 代表自变量的取值，其中 `obj` 为相应的对象；`y` 是每一行为相应的元素，每一列代表对应的时间点（`t` 同理）。



使用 `Python` 求解微分方程组：

1. 基于基本原理自己写相关函数
2. 利用 Python 的 ode 求解包，熟悉相关的输入输出，完成数值求解

3. 无论是常微分方程组还是偏微分方程组，使用的都是同一套思路，就是**差分代替微分**

例 7：

$$\begin{cases} x'(t) = -x^3 - y \\ y'(t) = -y^3 + x \end{cases}$$

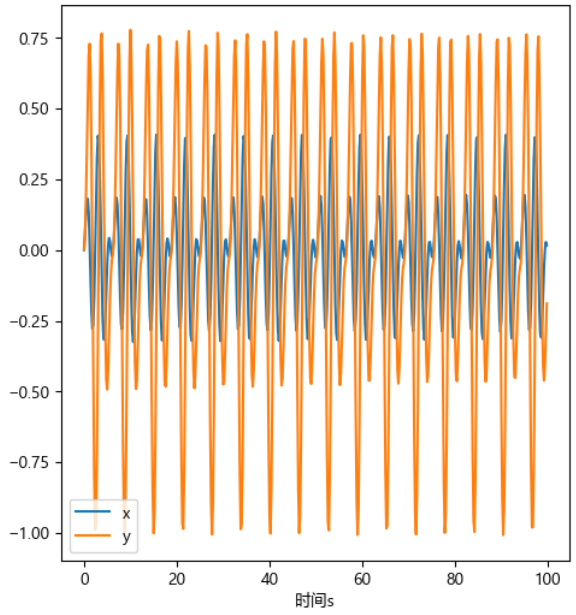
```python
plt.rcParams["font.sans-serif"] = ["Microsoft YaHei"]


def fun(t, w):
    x = w[0]
    y = w[1]
    return [-(x**3) - y, -(y**3) + x]


y0 = [1, 0.5]

yy = solve_ivp(fun, (0, 100), y0, method="RK45", t_eval=np.arange(0, 100, 0.2))
t = yy.t
data = yy.y
plt.plot(t, data[0, :])
plt.plot(t, data[1, :])
plt.xlabel("时间s")
plt.show()
```

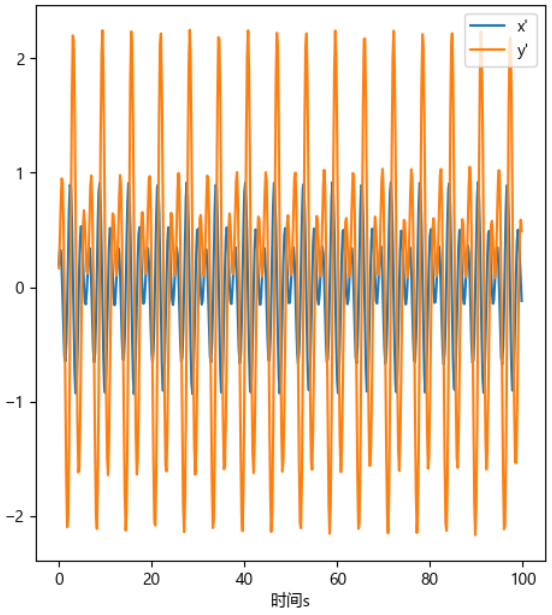
例 8：

$$\begin{cases} x''(t) + y'(t) + 3x(t) = \cos(2t) \\ y''(t) - 4x'(t) + 3y(t) = \sin(2t) \end{cases}, \ x''(0) = \frac{1}{5}, \ y''(0) = \frac{6}{5}, \ x'(0) = y'(0) = 0$$

```python
def func(t, w):
    x = w[0]
    y = w[1]
    dx = w[2]
    dy = w[3]
    # 求导以后 [x, y, dx, dy] 变为 [dx, dy, d2x, d2y]
```

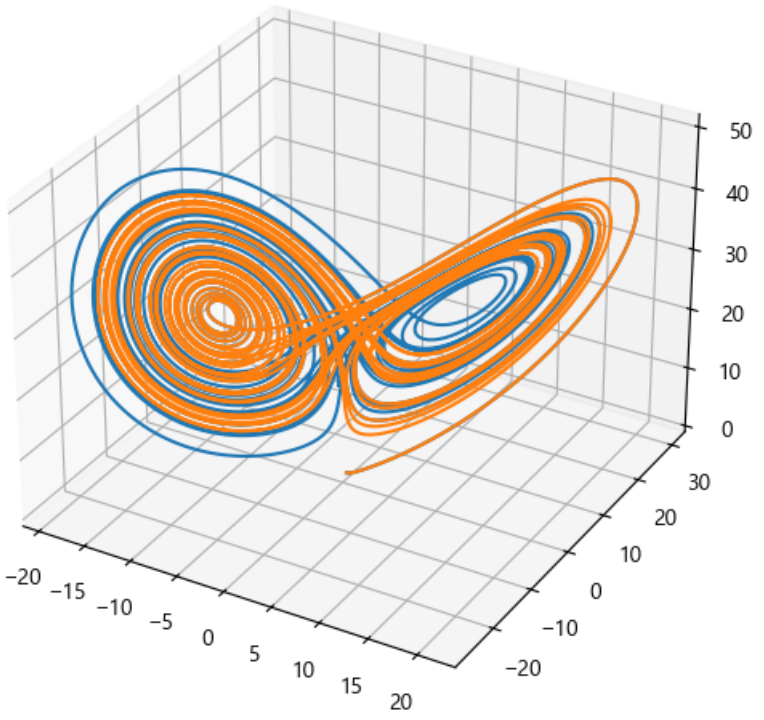
```
    # d2x 为 w[2], d2y 为 w[5]
    return [dx, dy, -dy - 3 * x + np.cos(2 * t), 4 * dx - 3 * y + np.sin(2 * t)]


y0 = [0, 0, 1 / 5, 1 / 6]
yy = solve_ivp(func, (0, 100), y0, method="RK45", t_eval=np.arange(0, 100, 0.2))
t = yy.t
data = yy.y
plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
plt.plot(t, data[0, :])
plt.plot(t, data[1, :])
plt.legend(["x", "y"])
plt.xlabel("时间s")
plt.subplot(1, 2, 2)
plt.plot(t, data[2, :])
plt.plot(t, data[3, :])
plt.legend(["x'", "y'"])
plt.xlabel("时间s")
plt.show()
```





解洛伦茨系统（蝴蝶效应）

$$\begin{cases} \dfrac{dx}{dt} = p(y-x) \\ \dfrac{dy}{dt} = x(r-z) \\ \dfrac{dz}{dt} = xy - bz \end{cases}$$




```python
import numpy as np
from scipy.integrate import odeint
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt


def dmove(Point, t, sets):
    p, r, b = sets
    x, y, z = Point
    return np.array([p * (y - x), x * (r - z), x * y - b * z])


t = np.arange(0, 30, 0.001)
P1 = odeint(dmove, (0.0, 1.0, 0.0), t, args=([10.0, 28.0, 3.0],))
```

```
P2 = odeint(dmove, (0.0, 1.01, 0.0), t, args=([10.0, 28.0, 3.0],))

fig = plt.figure()
ax = Axes3D(fig,auto_add_to_figure=False)
fig.add_axes(ax)
ax.plot(P1[:, 0], P1[:, 1], P1[:, 2])
ax.plot(P2[:, 0], P2[:, 1], P2[:, 2])
plt.show()
```

总结

设计的`func`函数，其返回值要是对应参数列表的各项导数所组成的列表。

如二阶微分方程中，

```
def fvdp(y, t):
    dy1 = y[1]  # 第一个方程表示y'，即y的一阶导数
    dy2 = 20 * (1 - y[0] ** 2) * y[1] - y[0]  # 第二个方程表示y''，即y的二阶导数
    return [dy1, dy2]  # 返回y'和y''组成的数组
```

`dy1`为参数`y[0]`的导数（`y[0]`即为`y`），而`dy2`为`y[1]`（即`y'`）的导数。

类似的，更高阶的：

```
def fvdp(y, t):
    dy1 = y[1]
    dy2 = y[2]
    dy3 = np.cos(t) - y[2] + y[1] - y[0]
    return [dy1, dy2, dy3]
```

`dy1`是更高阶`y`的导数，以此类推。

多元方程的时候：

```
def fun(t, w):
    x = w[0]
    y = w[1]
    return [-(x**3) - y, -(y**3) + x]
```

看着方程：

$$\begin{cases} x'(t) = -x^3 - y \\ y'(t) = -y^3 + x \end{cases}$$

所以，`odeint`或者是`solve_ivp`都是求解列表中每一个参数的原函数，而且就一阶。如一阶导变回零阶。