Компьютерные технологии в математическом моделировании
Лекция 6.

Ассистент кафедры
математической физики
к.ф.-м.н.
Татьяна Евгеньевна Романенко

Содержание лекции

- Гравитационная задача N тел
- SymPy
- SciPy
- Применение к реальным задачам

Гравитационная задача N тел

- Математическая постановка задачи

$$\frac{dr_i}{dt} = v_i \ ,$$

$$\frac{dv_i}{dt} = \sum_{j \neq i}^{N} G m_j \frac{r_j - r_i}{\left| r_j - r_i \right|^3} \ ,$$

где m_j, r_i, v_i— масса, радиус-вектор и скорость i-го тела соответственно (i изменяется от 1 до N), G — гравитационная постоянная. Массы тел, а также положения и скорости в начальный момент времени считаются известными. Необходимо найти положения и скорости всех частиц в произвольный момент времени.

Гравитационная задача N тел

- Метод численного решения. Алгоритм Верле

$v_{n+1} \equiv v(t_n + \Delta t)$ и $x_{n+1} \equiv x(t_n + \Delta t)$, тогда

$$v_{n+1} = v_n + a_n \Delta t + O[(\Delta t)^2]$$

$$x_{n+1} = x_n + v_n \Delta t + \frac{1}{2} a_n (\Delta t)^2 + O[(\Delta t)^3]$$

$$x_{n-1} = x_n - v_n \Delta t + \frac{1}{2} a_n (\Delta t)$$

$$x_{n+1} + x_{n-1} = 2x_n + a_n (\Delta t) + O[(\Delta t)]$$

$$x_{n+1} = 2x_n - x_{n-1} + a_n (\Delta t)$$

Гравитационная задача N тел

- Метод численного решения. Алгоритм Верле

Аналогично вычитание разложений в ряд Тейлора для x_{n+1} и x_n дает

$$v_n = \frac{x_{n+1} - x_{n-1}}{2\Delta t}$$

При этом связанная с *алгоритмом Верле* глобальная погрешность имеет третий порядок для координаты и второй порядок для скорости.

Однако скорость не участвует в интегрировании уравнений движения.

Гравитационная задача N тел

- Метод численного решения. Алгоритм Верле

Менее известной, но математически эквивалентной версией алгоритма Верле является схема

$$x_{n+1} = x_n + v_n \Delta t + \frac{1}{2} a_n (\Delta t)$$

и

$$v_{n+1} = v_n + \frac{1}{2}(a_{n+1} + a_n)\Delta t$$

Данная схема называемая *скоростной* формой алгоритма Верле, является самостартующей и не приводит к накоплению погрешности округления.

SymPy: Решение уравнений

- Свободная система
- Интегрирована напрямую в язык (Sage)
- Может использоваться как библиотека

- Математические объекты представлены точно
- Возможность символьного упрощения
- Переменные объявляются с помощью symbols()

SymPy: Простые примеры

```
import math
a = math.sqrt(9)
3.0
b = math.sqrt(8)
2.8284271247461903
c = math.sqrt(2)
1.4142135623730951
d = b*c
4.000000000000001
```

```
import sympy
a = sympy.sqrt(9)
3
b = sympy.sqrt(8)
2*sqrt(2)
c = sympy.sqrt(2)
sqrt(2)
d = b*c
4
```

SymPy: Простые примеры

```
from sympy import symbols, expand, factor
x, y = symbols("x y")
expr = x + 2*y
x + 2*y
expr2 = expr - x
2*y
expr3 = x*expr
x*(x + 2*y)
expr4 = expand(expr3)
x**2 + 2*x*y
expr5 = factor(expr4)
x*(x + 2*y)
```

```
x +1 == 4
False
Eq(x+1, 4)
Eq(x+1, 4)
(x+1)**2 == x**2 + 2*x + 1
False
```

```
x = symbols('x')
expr = x + 1
x = 2
print(expr)
x + 1
print(expr.subs(x,2))
x + 1
```

SymPy: Базовые операции

Подстановка: подвыражение и значение

```
from sympy import symbols, cos
x, y, z = symbols("x y z")
expr1 = cos(x) + 1
expr2 = expr1.subs(x, y*z)
cos(y*z) + 1

expr3 = x**y
expr3 = expr3.subs(y, x**y)
x**(x**y)
expr3 = expr3.subs(y, x**x)
x**(x**(x**x))
```

```
from sympy import symbols, cos,
    sin, expand_trig
x, y, z = symbols("x y z")
expr = sin(2*x) + cos(2*y)
expr2 = expand_trig(expr)
2*sin(x)*cos(x) + 2*cos(y)**2 - 1

expr3 = expr.subs(sin(2*x),
    2*sin(x)*cos(x))
2*sin(x)*cos(x) + cos(2*y)
```

SymPy: Базовые операции

Подстановка: подвыражение и значение

```
from sympy import symbols, cos, sin
x, y, z = symbols("x y z")
expr = sin(2*x) + cos(2*y)
expr.subs(x, 0).subs(y,0)
1
expr.subs({x:0, y:0})
1
expr2 = x**2 + y**2 - z**2
expr2.subs([(x, 1), (y, 3), (z, 2)])
6
expr3 = x**4 - 4*x**3 + 4*x**2 - 2*x + 3
replacements = [(x**i, y**i) for i in range(5) if i % 2 == 0]
expr3.subs(replacements)
-4*x**3 - 2*x + y**4 + 4*y**2 + 3
```

SymPy: Строки <-> Expressions
sympify, evalf & lambdify

```
from sympy import *
str_expr = "x**2 + 3*x - 1/2"
expr = sympify(str_expr)
x**2 + 3*x - ½
expr.subs(x, 2)
19/2
expr.evalf(subs={x: 2})
9.5000000000000
```

```
from sympy import *
expr1 = sqrt(8)
expr1.evalf()
2.82842712474619
expr1.evalf(3)
2.83
expr2 = cos(1)**2 + sin(1)**2
expr3 = (expr2 - 1).evalf()
-0.e-124
expr3 = (expr2 - 1)
                    .evalf(chop=True)
0
```

SymPy: Строки <-> Expressions
sympify, evalf & lambdify

```
from sympy import lambdify, Symbol
import numpy as np
x = Symbol("x")
a = np.linspace(0, np.pi, 3)
expr = sin(x)
f = lambdify(x, expr, "numpy")
f(a)
[  0.00000000e+00   1.00000000e+00   1.22464680e-16]
def new_sin(x):
    return x
f = lambdify(x, expr, {"sin":new_sin})
f(a)        [ 0.          1.57079633  3.14159265]
f(0.1)      0.1
```

SymPy: Simplify

- Общее упрощение: simplify()
- Полиномиальные\рациональные функции
  - expand()
  - factor()
  - collect()
  - cancel()
  - apart()
- Тригонометрические функции
  - trigsimp()
  - expand_trig()
- Степенные функции
  - powsimp()
  - expand_power_exp ()
- Экспоненциальные функции
  - expand_log(), logcombine()
- Специальные функции

SymPy: Дифференцирование, интегрирование и пределы

```
from sympy import *
x = symbols("x")
expr = x*cos(x)
d_expr = diff(expr, x)
-x*sin(x) + cos(x)

expr2 = x**4
d3_expr2 = diff(expr2, x, x, x)
24*x
d3_expr2 = diff(expr2, x, 3)
24*x
```

SymPy: Дифференцирование, интегрирование и пределы

```
from sympy import *
x = symbols("x")
expr = -x*sin(x) + cos(x)
i_expr = integrate(expr, x)
x*cos(x)

expr2 = integrate(exp(-x), (x, 0, oo))
1

expr3 = limit(sin(x)/x, x, 0)
1
```

SymPy: Решение уравнений

```
from sympy import Symbol, solve
x = Symbol("x")
solve(x**2 - 1)
[-1, 1]

pos = Symbol("pos", positive=True)
solve(pos**2 - 1)
[1]

from sympy import sin, limit
solve(sin(x)/x)
[pi]

solve(sin(x)/x, check=False)
[0, pi]
```

```
eq = x**2*(1/x - z**2/x)
solve(eq, x)
[]

solve(eq, x, check=False)
[0]

limit(eq, x, 0, '-')
0

limit(eq, x, 0, '+')
0
```

SymPy: Решение уравнений

```
from sympy import root
eq = root(x**3 - 3*x**2, 3) + 1 - x
solve(eq)
[]

solve(eq, check=False)
[1/3]

from sympy import solve_poly_system
from sympy.abc import x, y[]
solve_poly_system([x*y – 2*y, 2*y**2 – x**2], x, y)

[(0, 0), (2, -sqrt(2)), (2, sqrt(2))]

eq1 = x*y - 2*y
eq2 = 2*y**2 - x**2
solve_poly_system([eq1, eq2], x, y)
```

SymPy: Решение уравнений

```
from sympy import Symbol, solve, lambdify
x = Symbol("x", positive=True)
z = Symbol("z")
eq = x**2 - z**2*x
res = solve(eq, x)

f = lambdify( z, res[0])
```

4

SymPy: Решение уравнений

```
from sympy import Symbol, solve, lambdify, , Matrix
k1 = Symbol("k1")
k2 = Symbol("k2")
k3 = Symbol("k3")
km1 = Symbol("km1")
km2 = Symbol("km2")
x = Symbol("x")
y = Symbol("y")
eq1 = k1*(1 - x - y) - km1*x - k3*x*y
eq2 = k2*(1 - x - y)**2 - km2*y**2 - k3*x*y

res = solve([eq1, eq2], x, k2)
```

```
[(-k1*(y - 1)/(k1 + k3*y + km1), y*(k1 + k3*y + km1)*(-k1*k3*y + k1*k3 +
    k1*km2*y + k3*km2*y**2 + km1*km2*y)/((y - 1)**2*(k3*y +
    km1)**2))]
```

SymPy: Решение уравнений

```
A = Matrix([eq1, eq2])
var_vector = Matrix([x, y])
jacA = A.jacobian(var_vector)
```

Matrix([[-k1 - k3*y - km1, -k1 - k3*x], [k2*(2*x + 2*y - 2) - k3*y, k2*(2*x + 2*y - 2) - k3*x - 2*km2*y]])

```
det_jacA = jacA.det()
```

k1*k3*x - k1*k3*y + 2*k1*km2*y + 2*k2*k3*x**2 - 2*k2*k3*x - 2*k2*k3*y**2 + 2*k2*k3*y - 2*k2*km1*x - 2*k2*km1*y + 2*k2*km1 + k3*km1*x + 2*k3*km2*y**2 + 2*km1*km2*y

```
trace_jacA = jacA.trace()
```

-k1 + k2*(2*x + 2*y - 2) - k3*x - k3*y - km1 - 2*km2*y

```
trace_func = lambdify( (x, y, k1, k2, k3, km1, km2) , trace_jacA)
```

SymPy: Решение ОДУ

```
from sympy import *
y = symbols("y", cls=Function)
f(x)
ode = Eq(y(x).diff(x, 2) - 2*y(x).diff(x) + y(x)- sin(x), y(x))
Eq(y(x) - sin(x) - 2*Derivative(y(x), x) + Derivative(y(x), x, x), y(x))
res = dsolve(ode, y(x))
Eq(y(x), C1 + C2*exp(2*x) - sin(x)/5 + 2*cos(x)/5)
```

SciPy: Решение ОДУ

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

def g(y, x):
    y1 = y[0]
    y2 = y[1]
    y3 = y[2]
    f1 = -0.04 * y1 + 10e+4*y2*y3
    f2 = 0.04*y1 - 10e+4*y2*y3 - 3 * 10e+
    f3 = 3 * 10e+7*y2*y2
    return [f1, f2, f3]
init = 1.0, 0.0, 0
# First integrate from 0 to 2
x = np.linspace(0,3,300)
sol=odeint(g, init, x)
```

```
plt.figure(1)
plt.grid(True)
plt.xlabel('x')
plt.ylabel('y1')
plt.plot(x, sol[:,0], color='b')
plt.show()
```