Computational Physics Lab

Ordinary Differential Equations & Initial Value Problems

Prof. Paul Eugenio
Department of Physics
Florida State University

Apr 2, 2019

peugenio@fsu.edu
Ordinary Differential Equations

READ Discussions in

Chapter 8 Sections 1-4
Recall from Exercise 7

Radioactive Decays

\[ \frac{dN(t)}{dt} = -\frac{N(t)}{\tau} \]

we used the forward difference and found a solution:

\[ N(t+h) \approx N(t) - h \frac{N(t)}{\tau} = N(t) + hf(N,t) \]

\[ x(t+h) = x(t) + hf(x,t) + O(h^2) \]

Euler's Method for solving differential equations
Initial Value Problem

To determine a solution we need its initial or boundary conditions

\[
N(t) = N(t_0) e^\left(-\frac{\Delta t}{\tau}\right)
\]

exact

```
def f(N, t):
    tau = 2.0
    return -N/tau

h, tMin, tMax = 0.1, 0, 15

# Set initial conditions: at t=0, N(t) = 100%
time = [tMin]
N = [1]

for i, t in enumerate(np.arange(tMin, tMax, h)):
    time += [t]
    N += [N[i] + h * f(N[i], t)]

plt.plot(time, N, "*"
plt.show()
```

<table>
<thead>
<tr>
<th>t</th>
<th>N</th>
<th>N(t)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.01</td>
<td>0.95</td>
<td>0.951229</td>
</tr>
<tr>
<td>0.02</td>
<td>0.9025</td>
<td>0.904837</td>
</tr>
<tr>
<td>0.03</td>
<td>0.857375</td>
<td>0.860708</td>
</tr>
<tr>
<td>0.04</td>
<td>0.814506</td>
<td>0.818731</td>
</tr>
<tr>
<td>0.05</td>
<td>0.773781</td>
<td>0.778801</td>
</tr>
<tr>
<td>0.06</td>
<td>0.735092</td>
<td>0.740818</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Simultaneous Differential Equations

Two 1st-Order Differential Equations

\[
\frac{dx}{dt} = f_x(x, y, t) \quad \frac{dy}{dt} = f_y(x, y, t)
\]

Using vectorized notation

\[
\frac{dr}{dt} = f(r, t)
\]

where \( r = \begin{bmatrix} x \\ y \end{bmatrix} \)

and \( f(r, t) = \begin{bmatrix} f_x(r, t) \\ f_y(r, t) \end{bmatrix} \)

Solution using Euler's method

\[
r(t+h) = r(t) + hf(r, t)
\]

\[
x(t+h) = x(t) + hf_x(x, y, t) \quad y(t+h) = y(t) + hf_y(x, y, t)
\]
N\textsuperscript{th}-order Differential Equations

Problems Involving N\textsuperscript{th}-order Ordinary Differential Equations Can Always be Reduced to the Study of a set of 1\textsuperscript{st}-Order Differential Equations

N\textsuperscript{th}-order ODE Transformed to N 1\textsuperscript{st}-order ODEs

Example:

\[
\frac{d^2 y}{dt^2} + f(t) \frac{dy}{dt} = g(t)
\]

\[
\frac{dy}{dt} = v(t)
\]

\[
\frac{dv}{dt} = g(t) - f(t) v(t)
\]
Differential Equations

\[
y(t+h) = y(t) + hv(t) \quad \quad v(t+h) = v(t) + h(g(t) - f(t)v(t))
\]

for i, t in enumerate(np.arange(h, tMax, h)):
    tPoints += [t]
    yPoints += [yPoints[i] + h * fy(yPoints[i], vPoints[i], t)]
    vPoints += [vPoints[i] + h * fv(yPoints[i], xPoints[t], t)]

Be aware that order matters
ODE & Diff. Eq. Errors

Forward difference

\[ x(t+h) = x(t) + h x'(t) + \frac{h^2}{2} x''(t) + ... \]

Uses info at 1 point “t”

Central difference

\[ x(t+h) = x(t) + h x'(x(t+h/2), t+h/2) + O(h^3) \]

Uses info at 2 points to improve accuracy

O(h^2) truncation error
ODE & Diff. Eq. Errors

Forward difference

\[ x(t+h) = x(t) + h x'(t) + \frac{h^2}{2} x''(t) + \ldots \]

Uses info at 1 point “t”

O(h^2)

Central difference

\[ x(t+h) = x(t) + h \left[ x'(x(t+h/2), t+h/2) \right] + O(h^3) \]

Uses info at 2 points to improve accuracy

General Runge-Kutta Method

\[ x(t+h) = x(t) + h \left( \sum_{i}^{N} c_i x'(x_i, t_i) \right) + O(h^{N+1}) \]

Use info at many points to further increase accuracy
Fourth-Order Runge-Kutta

\[ \frac{dx}{dt} = f(x, t) \]

\[ x(t+h) = x(t) + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) \]

\[ k_1 = hf(x, t) \]
\[ k_2 = hf(x + \frac{1}{2}k_1, t + \frac{1}{2}h) \]
\[ k_3 = hf(x + \frac{1}{2}k_2, t + \frac{1}{2}h) \]
\[ k_4 = hf(x + k_3, t + h) \]

The Most Commonly Used Method!

O(h^5)

truncation error
Fourth-Order Runge-Kutta using vectorized notation

\[ \frac{d \mathbf{r}}{dt} = \mathbf{f}(\mathbf{r}, t) \]

\[ \mathbf{r}(t+h) = \mathbf{r}(t) + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) \]

\[ k_1 = hf(\mathbf{r}, t) \]
\[ k_2 = hf(\mathbf{r} + \frac{1}{2}k_1, t + \frac{1}{2}h) \]
\[ k_3 = hf(\mathbf{r} + \frac{1}{2}k_2, t + \frac{1}{2}h) \]
\[ k_4 = hf(\mathbf{r} + k_3, t + h) \]
Implementing Runge Kutta

Because of Python's Dynamic Typing and Vectorization, one function works for any dimension!

```python
def rung_kutta4(f, r, t, h):
    """ 4th order Runge-Kutta method """
    k1 = h*f(r, t)
    k2 = h*f(r+0.5*k1, t+0.5*h)
    k3 = h*f(r+0.5*k2, t+0.5*h)
    k4 = h*f(r+k3, t+h)
    return (k1 + 2*k2 + 2*k3 + k4)/6

def fN(N, t):
    """ 1 dimensional function """
    tau = 2.0
    return -N/tau

N, NPoints = 1, []
for t in tPoints:
    NPoints += [N]
    N += runge_kutta4(fN, N, t, h)
plt.plot(tPoints, NPoints)

def f2D(r, t):
    """ 2 dimensional function """
    g, L = 9.81, 0.1
    theta, omega = r[0], r[1]
    fTheta = omega
    fOmega = -g/L * theta
    return np.array([fTheta, fOmega], float)

r = np.array([10*np.pi/180, 0], float)
for t in tPoints:
    thetaPoints += [r[0]]
    omegaPoints += [r[1]]
    r += runge_kutta4(f2D, r, t, h)
plt.plot(tPoints, thetaPoints)
```
The van der Pol oscillator

The van der Pol oscillator appears in electronic circuits and in laser physics

\[ \frac{d^2 x}{dt^2} - \mu (1 - x^2) \frac{dx}{dt} + \omega^2 x = 0 \]

Solve with initial conditions \( x = 1 \) and \( \frac{dx}{dt} = 0 \).
The van der Pol oscillator

The van der Pol oscillator appears in electronic circuits and in laser physics

\[
\frac{d^2 x}{dt^2} - \mu (1 - x^2) \frac{dx}{dt} + \omega^2 x = 0
\]

Solve with initial conditions \( x = 1 \) and \( \frac{dx}{dt} = 0 \).

\[
f(r, t) = \begin{bmatrix}
\frac{dx}{dt} &=& v \\
\frac{dv}{dt} &=& -\omega^2 x + \mu (1 - x^2) v
\end{bmatrix}
\]
The van der Pol oscillator

The van der Pol oscillator appears in electronic circuits and in laser physics

\[
\frac{d^2 x}{dt^2} - \mu (1 - x^2) \frac{dx}{dt} + \omega^2 x = 0
\]

Solve with initial conditions \( x = 1 \) and \( dx/dt = 0 \).

\[
f(r, t) = \begin{cases}
\frac{dx}{dt} &= v \\
\frac{dv}{dt} &= -\omega^2 x + \mu (1 - x^2) v
\end{cases}
\]

def f_vanderPol(r, t):
    """ vectorized function for the van der Pol oscillator """
    mu, omega = 3.0, 1.0           # constants
    x = r[0]
    v = r[1]
    fx = v
    fv = -omega**2 * x + mu * (1 - x**2) * v
    return np.array([fx, fv], float)
The van der Pol oscillator

The van der Pol oscillator appears in electronic circuits and in laser physics

# main

# initialization
N = 1000  # Number of steps
Tmin, Tmax = 0.0, 20.0  # time range
tStep = (Tmax – Tmin) / N  # time step

# define time values
Time = np.arange(Tmin, Tmax, tStep)

# creating lists for plotting
xPoints, vPoints = [], []

# set initial conditions
x0, v0 = 1.0, 0.0
r = np.array([x0, v0], float)

# find numerical solution to Dif. Eq.
for t in time:
xPoints += [r[0]]
vPoints += [r[1]]
r += runge_kutta4(f_vanderPol, r, t, tStep)

# Now generate plots
...

The van der Pol oscillator

See Python source code: vanderpol.py
Poincare' sections from Exercise 9

phase offset: 0°

click on plot to see animated plot