Runge-Kutta Methods

Matheus Morroni

January 2020

1 Introduction

In the Euler method, we used information on the gradient or the derivative of y at the given time step to extrapolate the possible solution to the next point. The LTE (Lifting The Exponent) for the method is O(h^2), producing a first order numerical technique. Runge-Kutta methods are a group of methods which with good sense uses the information on the 'slope' at more than one point to extrapolate a possible solution to the next point.

But why the Euler method is not adequate to produce a high accuracy solve? There are several reasons that Euler's method is not recommended for this use, the most important points are: (i) the method is not very accurate when compared to other methods run at equivalent stepsize and (ii) neither is it very stable.

2 Maths Concepts

There are several ways to calculate the right-hand side f(x, y) that all agree to first order, but that different coefficients of higher-order error terms. Adding up the right combination of the coefficients, we could eliminate the error terms order by order. That is the basic idea utilised in the Runge-Kutta method.

Abramowitz and Stegun, and Gear, give several specific formulas that derive from the fundamental idea. By far the most often used is the fourth-order Runge-Kutta formula, which has a certain smoothness of organization about it:

$$\begin{aligned} k_1 &= hf(x_n, y_n) \\ k_2 &= hf(x_n + h/2, y_n + \tfrac{k_1}{2}) \\ k_3 &= hf(x_n + h/2, y_n + \tfrac{k_2}{2}) \\ k_4 &= hf(x_n + h, y_n + k3) \\ y_n + 1 &= y_n + \tfrac{k_1}{6} + \tfrac{k_2}{3} + \tfrac{k_3}{3} + \tfrac{k_4}{6} + O(h^5) \end{aligned}$$

3 Two Algorithms in Python

The first algorithm using lambda to calculate the solution.

```
def RK4(f):
    return lambda t, y, dt: (
            lambda dy1: (
            lambda dy2: (
            lambda dy3: (
            lambda dy4: (dy1 + 2*dy2 + 2*dy3 + dy4)/6
            )( dt * f( t + dt  , y + dy3   ) )
            )( dt * f( t + dt/2, y + dy2/2 ) )
            )( dt * f( t + dt/2, y + dy1/2 ) )
            )( dt * f( t       , y         ) )

def theory(t): return (t**2 + 4)**2 /16

from math import sqrt
dy = RK4(lambda t, y: t*sqrt(y))
```

```
t, y, dt = 0., 1., .1
while t <= 10:
    if abs(round(t) - t) < 1e-5:
        print("y(%2.1f)\t= %4.6f \t
        error: %4.6g" % ( t, y, abs(y - theory(t))))
    t, y = t + dt, y + dy( t, y, dt )
```

The output obtained with this algorithm is:

y(0.0) = 1.000000	error: 0
y(1.0) = 1.562500	error: 1.45722e-07
y(2.0) = 3.999999	error: 9.19479e-07
y(3.0) = 10.562497	error: 2.90956e-06
y(4.0) = 24.999994	error: 6.23491e-06
y(5.0) = 52.562489	error: 1.08197e-05
y(6.0) = 99.999983	error: 1.65946e-05
y(7.0) = 175.562476	error: 2.35177e-05
y(8.0) = 288.999968	error: 3.15652e-05
y(9.0) = 451.562459	error: 4.07232e-05
y(10.0) = 675.999949	error: 5.09833e-05

The second algorithm is an alternative form to calculate the solution without using lambda.

```
from math import sqrt

def rk4(f, x0, y0, x1, n):
    vx = [0] * (n + 1)
    vy = [0] * (n + 1)
    h = (x1 - x0) / float(n)
    vx[0] = x = x0
```

```
    vy[0] = y = y0
    for i in range(1, n + 1):
        k1 = h * f(x, y)
        k2 = h * f(x + 0.5 * h, y + 0.5 * k1)
        k3 = h * f(x + 0.5 * h, y + 0.5 * k2)
        k4 = h * f(x + h, y + k3)
        vx[i] = x = x0 + i * h
        vy[i] = y = y + (k1 + k2 + k2 + k3 + k3 + k4) / 6
    return vx, vy

def f(x, y):
    return x * sqrt(y)

vx, vy = rk4(f, 0, 1, 10, 100)
for x, y in list(zip(vx, vy))[::10]:
    print("%4.1f %10.5f %+12.4e" % (x, y, y - (4 + x * x)**2 / 16))
```

The output of this algorithm is:

y(0.0) = 1.00000	error: +0.0000e+00
y(1.0) = 1.56250	error: -1.4572e-07
y(2.0) = 4.00000	error: -9.1948e-07
y(3.0) = 10.56250	error: -2.9096e-06
y(4.0) = 24.99999	error: -6.2349e-06
y(5.0) = 52.56249	error: -1.0820e-05
y(6.0) = 99.99998	error: -1.6595e-05
y(7.0) = 175.56248	error: -2.3518e-05
y(8.0) = 288.99997	error: -3.1565e-05
y(9.0) = 451.56246	error: -4.0723e-05
y(10.0) = 675.99995	error: -5.0983e-05

4 References

- MIT Web Course Notes, Differential Equations Notes, Node 5
- Numerical Recipes in C The Art of Scientific Computing, Cambridge University Press, 1988-1992
- Runge-Kutta Method, rossettacode.org