Taylor Series Method for Solving Ordinary Differential Equations

Ian Smith

January 2019

Abstract

The procedure described here is a simple and accurate numerical approach for evolving a system of nonlinear ordinary differential equations (ODEs). Time varying quantities are represented using arbitrary degree Taylor Series in the coordinates of interest, with the derivative terms in the series generated via recurrence relationships. Horner's method is then used to calculate the coordinate values at the next timestep.

1 Introduction

Starting with an ordinary differential equation (ODE), or system of such equations:

$$\dot{x} = f(x) \tag{1}$$

We can represent the time variation of the solution x as a Taylor Series in the time domain. Solving the equation in this context means that from the given initial condition(s), x, we want to evolve the system state to a later time. The interval between those initial condition(s) and that later time is called the time step, h. So, in the time domain we have:

$$\begin{aligned} x(h) &= x + \frac{x'h}{1!} + \frac{x''h^2}{2!} + \frac{x'''h^3}{3!} + ... + \frac{x^k h^k}{k!} + ... \\ &= \sum_{k=0}^{\infty} \frac{x^k h^k}{k!} \end{aligned} \tag{2}$$

Rather than using the x derivatives x^k (note superscript!) directly, and carrying around the factorial terms, we define a "jet" of (subscripted) Taylor Coefficients (TCs) as follows:

$$x_k = \frac{x^k}{k!} \tag{3}$$

so that we now have the simpler (polynomial) form:

$$x(h) = \sum_{k=0}^{\infty} x_k h^k \tag{4}$$

Differentiating this with respect to time (h) we can represent the LHS of the ODE:

$$\dot{x}(h) = \sum_{k=1}^{\infty} k x_k h^{k-1} = \sum_{k=0}^{\infty} (k+1) x_{k+1} h^k \tag{5}$$

but we can also say, as in (4), that:

$$\dot{x}(h) = \sum_{k=0}^{\infty} \dot{x}_k h^k \tag{6}$$

equating the polynomials (5) and (6) by powers of h we get for the k^{th} term:

$$\dot{x}_k = (k+1) x_{k+1} \tag{7}$$

or

$$x_{k+1} = \frac{\dot{x}_k}{k+1} \tag{8}$$

This is a key relationship between the "derivative" of the ODE, and the next term in the taylor series.

2 Procedure

We will use the letter F to represent the ODE function in (1) when expressed in terms of the TCs.

As can be seen from (2) and (3), $x_0 = x^0$ is just x, the initial condition(s), and $\dot{x}_0$ is just the differential equation F itself. We can evaluate it and divide by $k+1$ to get x_1. Now we need to use the differential equations with x_1 to get $\dot{x}_1$, and divide by $k+1$ again to get x_2, and so on. Explicitly, the two processes are:

$$\begin{aligned} \dot{x}_k &= F(x_k) \\ x_{k+1} &= \frac{\dot{x}_k}{k+1} \end{aligned} \tag{9}$$

Which would typically be combined into a single step in code:

$$x_{k+1} = \frac{F(x_k)}{k+1} \tag{10}$$

In this way we build up the derivative "jet" term by term. Obtaining $F(x_k)$ from $f(t)$ is where the recurrence relations (where necessary) come in.

Once we have enough terms, we use Horner's method to sum the polynomial.

3 Examples

3.1 Exponential Decay

Time domain, $\dot{x} = f(x)$:

$$\dot{x} = -ax \tag{11}$$

Taylor coefficients, $\dot{x}_k = F(x_k)/k + 1$:

$$x_{k+1} = \frac{-ax_k}{k+1} \tag{12}$$

In this simple case (multiplication by a scalar) we do not need to use any recurrence formula, as x_{k+1} depends only on x_k. Addition and subtraction of TCs are similar in this regard.

Where multiplication and division of TCs and functions with TC arguments are concerned, the situation is slightly more complex.

3.2 Code example

```
#!/usr/bin/env python3
from sys import argv
n = int(argv[3])  # integrator controls
h = float(argv[4])
steps = int(argv[5])
x = float(argv[6])  # coordinate
a = float(argv[7])  # parameter
cx = [0.0 for _ in range(n + 1)]  # jets
for step in range(1, steps):
    print("{:.9e} {:.5e}".format(x, step * h))
    cx[0] = x
    for k in range(n):  # build up the jets using the recurrence relations an
        cx[k + 1] = a * x / (k + 1)
    x = cx[n]
    for i in range(n - 1, -1, -1):  # Horner's method
        x = x * h + cx[i]
```

3.3 Lorenz Equations

Time domain, $\dot{x} = f(x)$:

$$\begin{aligned} \dot{x} &= \sigma(y - x) \\ \dot{y} &= \rho x - xz - y \\ \dot{z} &= xy - \beta z \end{aligned} \tag{13}$$

Taylor coefficients, $\dot{x}_k = F(x_k)/k + 1$:

$$\begin{aligned} x_{k+1} &= \frac{\sigma(y_k - x_k)}{k+1} \\ y_{k+1} &= \frac{\rho x_k - (x.z)_k - y_k}{k+1} \\ z_{k+1} &= \frac{(x.y)_k - \beta z_k}{k+1} \end{aligned} \tag{14}$$

In this case we need to use the recurrence formula for the two products $(x.z)_k$ and $(x.y)_k$, e.g:

$$(x.y)_k = \sum_{j=0}^{\infty} x_k y_{k-j} \tag{15}$$

In general, an x_{k+1} depends on all the lower degree terms x_0 to x_k, which is why we need to store the jets for the duration of the time step.

3.4 Code example

```
#!/usr/bin/env python3
from sys import argv
n = int(argv[3])  # integrator controls
h = float(argv[4])
steps = int(argv[5])
x = float(argv[6])  # coordinates
y = float(argv[7])
z = float(argv[8])
s = float(argv[9])  # parameters
r = float(argv[10])
b = float(argv[11]) / float(argv[12])
cx = [0.0 for _ in range(n + 1)]  # jets
cy = [0.0 for _ in range(n + 1)]
cz = [0.0 for _ in range(n + 1)]
for step in range(1, steps):
    print("{:.9e} {:.9e} {:.9e} {:.5e}".format(x, y, z, step * h))
    cx[0] = x
    cy[0] = y
    cz[0] = z
    for k in range(n):  # build up the jets using the recurrence relations an
        cx[k + 1] = s * (cy[k] - cx[k]) / (k + 1)
        cy[k + 1] = (r * cx[k] - sum(cx[j] * cz[k - j] for j in range(k + 1))
        cz[k + 1] = (sum(cx[j] * cy[k - j] for j in range(k + 1)) - b * cz[k]
    x = cx[n]
```

```
y = cy[n]
z = cz[n]
for i in range(n - 1, -1, -1):  # Horner's method
    x = x * h + cx[i]
    y = y * h + cy[i]
    z = z * h + cz[i]
```

3.5 Hamiltonian Equations of Motion

In the Time domain:

$$\begin{aligned} \dot{q} &= \frac{\partial H}{\partial p} = C_1 \\ \dot{p} &= -\frac{\partial H}{\partial q} = -C_2 \end{aligned} \tag{16}$$

where C_1 and C_2 are constants at a given time step. In this case the Taylor coefficients are trivial:

$$\begin{aligned} q_{k+1} &= \frac{C_1}{k+1} \\ p_{k+1} &= \frac{-C_2}{k+1} \end{aligned} \tag{17}$$

The key idea here is to generate the partial differentials of the Hamiltonian using Automatic Differentiation, in this case using the technique of Dual Numbers (forward mode AD).

In this way we can simulate Hamiltonian systems without relying on numerical differences for differentiation by time or configuration variables.

It also removes any need to perform symbolic differentiation with respect to configuration variables, so that it can readily be used with any Hamiltonian supported by the operations defined in the Dual Numbers code.