Ch. 7 - Initial and Boundary Value Problems of Differential Equations

Section 7.1 - Initial Value Problems: Euler, Runge-Kutta and Adams Methods

Start with a basic/essential problem:

Compute approx. sol'n of ordinary differential equation (**ODE**) with given initial value:

$$\frac{dy}{dt} = f(t, y) \; , \qquad y(0) = y_0$$

You may have seen the simple, classice approach: **Euler's method**

- "stepping" or "marching" method
- computes sequence of values at discrete "future" times based on existing "history":

$$y_n = y(t_n); \; t_n = t_0 + n\Delta t = t_0 + nh$$

- Independent vs dependent variables; discretize which kind?

Discretize **independent** variable

- Convert continuous domain to discrete domain
- "Step" along discrete values of independent variable
- Compute approximate values of dependent variable
- Basic plan:

1. Approximate the derivative with a simple forward difference estimator

$$\frac{dy}{dt} = f(t, y)\ , \qquad y(0) = y_0 \rightarrow \frac{y_{n+1} - y_n}{h} \approx f(t_n, y_n)$$

2. Knowing y_n and t_n, compute y_{n+1}:

$$y_{n+1} = y_n + hf(t_n, y_n)$$

Euler's method:

- Computes the rate of change at the beginning of a time step (from t_n to t_{n+1})
- Ignores the change in that rate of change over short time steps.
- Makes assumption that is, of course, not completely valid so truncation error is incurred.

$\exists$ numerous generalizations of Euler's method of the form

$$y_{n+1} = y_n + h\phi$$

- ϕ is some function of known values
- ϕ is chosen to reduce the error incurred during a timestep.
- Example
  - Euler formula can be used to estimate future values (beyond start of the step)
  - Estimated derivatives at those times can be used to account for (and adjust for) how the rate of change varies across the timestep.

Heun's method:

1. Compute the right-hand side (RHS) at initial time (left end of the timestep).
2. Compute Euler estimate of y_{t+h}
3. Use Euler formula to estimate derivative at final time (right end of the timestep)
4. Use the average of the derivative estimates from left and right ends

Euler: $y_{n+1} = y_n + hf(t_n, y_n)$

Apply Euler formula to estimate value at end of step: $y_{euler} = y(t) + hf(t, y(t))$

Improve rate estimate by averaging estimated rates at ends of interval:

$$y(t+h) = y(t) + \frac{h}{2}\big(f(t, y(t)) + f(t+h, y_{euler})\big); \qquad \text{Eq. (7.1.13a)}$$

Modified Euler-Cauchy:

1. Compute derivate (RHS) at initial time (left side of interval)
2. Use that derivative value to compute "Euler" estimate of derivative at mid-interval.
3. Use mid-interval value to estimate rate of change over the interval.

$$rate_{left} = f(t, y(t))$$
$$y_{mid} = y(t) + \frac{h}{2} rate_{left}$$
$$rate_{mid} = f(t + \frac{h}{2}, y_{mid})$$
$$y_{RK2}(t+h) = y(t) + h(rate_{mid})$$

Put the pieces together:

$$y_{RK2}(t+h) = y(t) + hf\big(t + \frac{h}{2}, y(t) + \frac{h}{2}\ f(t, y(t))\big); \qquad \text{Eq. 7.1.13b}$$

Why is modified Euler-Cauchy given subscript "RK2"?

Runge-Kutta (RK) Methods

- Family of numerical ODE solvers that aim to reduce truncation error.
- Aim to reduce truncation error by including rate estimates at additional points in the timestep.
- Increase the order of truncation error by making the neglected terms involve higher powers of h
- Euler's method and modified Euler-Cauchy method are first 2 methods in the R-K family.

A very commonly used member of the family is **Fourth-Order Runge-Kutta (RK4)**:

1. Compute initial rate estimate: $f_1 = f(t_n, y_n)$
2. Use initial rate estimate to estimate midstep values: $f_2 = f(t_n + \frac{h}{2}, y_n + \frac{h}{2} f_1)$
3. Use the midstep estimate to compute improved midstep estimate
$$f_3 = f(t_n + \frac{h}{2}, y_n + \frac{h}{2} f_2)$$
4. Use improved midstep estimate to compute right-side estimate
$$f_4 = f(t_n + h, y_n + h f_3)$$
5. Compute weighted sum of contributions with coefficients chosen to cancel terms in the Taylor series.
Achieve local truncation error $\sim O(h^5) \implies$ global error $\sim O(h^4)$:
$$y_{n+1} = y_n + \frac{h}{6}[f_1 + 2f_2 + 2f_3 + f_4]$$

We will return for a closer look at error analysis, but for now pause to consider what is nice about Runge-Kutta methods:

Do you need previous values of y (before y_n) to compute y_{n+1}?

No, so given $y_0, f(t,y)$ you have what you need to compute $y_{n+1} \iff$ **self-starting**

Computing the RK4 estimate involves computing everything we need for the lower order RK2 estimate. What could we do with such information?

Estimate local error $\implies$ possible to implement automated adaptive stepsize control.

Adams method: Multi-stepping techniques

What we were really doing above is approximating the fundamental theorem of calculus (that a function is the integral of its derivative):

$$y_{n+1} = y_n + \int_{t_n}^{t_n+h} f(t,y)dt$$

For Runge-Kutta, we estimated the interval based on information at t_n.

Adams methods take an alternative approach:

Use previous values to construct a locally valid polynomial approximant $p(t,y)$ and integrate the polynomial:

$$y_{n+1} = y_n + \int_{t_n}^{t_n+h} p(t,y)dt$$

Adams-Bashforth methods

Construct the polynomial based on the current point and n previous points.

The first member of the family with $n = 0$ uses an order 0 polynomial:

$$p_1(t) = \text{constant} = f(t_n, y_n)$$

leading to

$$y_{n+1} = y_n + \int_{t_n}^{t_n+h} f(t_n, y_n)\, dt = y_n + hf(t_n, y_n)$$

which should look familiar (because it is Euler's method).

Aside on notation: The convention followed in the book is that the subscript on the polynomial indicates the number of points used, so the subscript is one greater than the degree of the polynomial.

Start to get something new when using 2 points, y_{n-1}, y_n, and degree 1 polynomial:

$$p_2 = f_{n-1} + \frac{f_n - f_{n-1}}{h}(t - t_{n-1})$$

$$y_{n+1} = y_n + \int_{t_n}^{t_n+h} p(t,y)dt$$

$$y_{n+1} = y_n + \int_{t_n}^{t_n+h} \left(f_{n-1} + \frac{f_n - f_{n-1}}{h}(t - t_{n-1})\right)dt$$

$$y_{n+1} = y_n + \frac{h}{2}\left[3f(t_n, y_n) - f(t_{n-1}, y_{n-1})\right]; \qquad \text{Eq. } (7.1.23)$$

This is a *two-step method*:

Need y_{n-1} and y_n to compute y_{n+1}, so NOT self-starting.

Consider initial state: given y_0 **NOT** enough info to compute y_1

Need **bootstrapping** technique to get started.

Let's look at implementations of these basic methods:

Euler's method

Let's start with a familiar initial value problem:

$$\frac{dy}{dt} = -y\,; \qquad y(0) = 1$$

```
import numpy as np
# define "right-hand side" function giving rate of change
def rhs(y,t):
    """
    compute right-hand side function specifying rate of change
    Args:
        t: float value of independent variable
        y: float value of dependent variable
    Returns:
        rate: float value for rate of change of y
    """
    rate = -y
    return rate
```

Now implement and test a function to take a single step:

```python
def euler_step(f,y,t0,t1):
    """
    compute next value for Euler's method ODE solver
    Args:
        f: name of right-hand side function that gives rate of change of y
        y: float value of dependent variable
        t0: float initial value of independent variable
        t1: float final value of independent variable

    Returns:
        y_new: float estimated value of y(t1)
    """
    f0 = f(y,t0)
    h = (t1-t0)
    y_new = y + h*f0
    return y_new

euler_step(rhs,1,0,.1)
```

```
0.9
```

Construct and test a multistep Euler solver that can be used for longer intervals:

```python
def euler_solve(f,y0,t):
    """
    Euler ODE solver
    Args:
        f: name of right-hand side function that gives rate of change of y
        y0: float initial value of dependent variable
        t: numpy array of float values of independent variable
    Returns:
        y: numpy array of float values of dependent variable
    """
    n = t.size
    y = [y0] #create solution as a list (of 1D numpy arrays) to which additional array values can be appended
    for i in range(n-1):
        y.append(euler_step(f,y[i],t[i],t[i+1])) #append new value to the list
    return np.array(y) #return the solution converted to a numpy array
steps = 4
t = np.linspace(0,1,steps+1)
y = euler_solve(rhs,1,t)
print("t values: ", t)
print("y values: ", y)
```

```
t values: [0. 0.25 0.5 0.75 1. ]
y values: [1. 0.75 0.5625 0.421875 0.31640625]
```

Compute values of exact solution $y(t) = exp(-t)$ for comparison:

```
n = t.size
y_exact = np.zeros(n)
for i in range(n):
    y_exact[i] = np.e**(-t[i])

error = (y_exact - y)
rel_error = (y_exact - y)/y_exact
print("e_abs: ", error)
print("e_rel: ", rel_error)
```

```
e_abs: [0. 0.02880078 0.04403066 0.05049155 0.05147319]
e_rel: [0. 0.03698094 0.07259429 0.10689062 0.13991864]
```

Now that we are ready to look at plots, let's shift over to the notebook version...