Numerical Methods for Differential Equations

Wang Zhi

University of Science and Technology of China

Date: January 10, 2023

Abstract

As we all know, solving differential equations analytically can be a very hard and sometimes even impossible task, this prompts research in solving equations numerically, and there has been great progress in this field since last century with the aid of computers. These methods are all relatively easy to implement but powerful enough to be practiced in science and engineering, some of which we shall discuss their schemes and relevant examples in this overview, such as Euler, Crank-Nicolson, Runge Kutta method, and the use of Python ODE solver.

1 Finite Difference Method

Basic Idea: use the finite difference to replace derivatives and turn differential equations into algebraic equations which can be solved by induction or by matrix operation, accuracy needs to be discussed here.

1.1 some basic schemes

The finite difference always starts by specifying a grid in which we evaluate: $0 \leq t \leq T, t_j = j\Delta t, x_j = x(t_j), j = 0, \ldots, J$, we deal with the explicit first-order ODE in this section and approximate the first-order derivative in different ways:

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

forward Euler: $\frac{x_{i+1}-x_i}{\Delta t} = f(t_i, x_i)$

Refined Euler: $\frac{x_{i+1}-x_i}{\Delta t} = \frac{1}{2}(f(t_i, x_i) + f(t_{i+1}, x_i + f(t_i, x_i)\Delta t))$

Central difference: $\frac{x_{i+1}-x_{i-1}}{2\Delta t} = f(t_i, x_i)$ by Taylor's expansion:

$$x_{i+1} - x_i = x_i^{'}\Delta t + \frac{1}{2}x_i^{''}(\Delta t)^2 + o((\Delta t)^2) \tag{1}$$

$$x_{i-1} - x_i = -x_i^{'}\Delta t + \frac{1}{2}x_i^{''}(\Delta t)^2 + o((\Delta t)^2) \tag{2}$$

(1) shows that forward Euler has error $o(\Delta t)$, and (1)-(2) shows that central difference has error $o((\Delta t)^2)$. In fact, the refined Euler is the second order Runge Kutta formula, which also has error $o((\Delta t)^2)$.

1.2 Crank-Nicolson and Von Neumann stability analysis

This method is particularly useful in solving diffusion equation

$$\frac{\partial u}{\partial t} = D\frac{\partial^2 u}{\partial x^2} + f(t,x)$$

the grid we use is $a \le x \le b, x_n = a + n\Delta x, n = 0, 1, \cdots, N, 0 \le t \le T, t_j = j\Delta t, j = 0, 1, \cdots, J$ and $u_j^n = u(t_j, x_n)$. and the basic idea is to average two time points when discretizing the space, the Crank-Nicolson scheme as follows:

$$\frac{u_{j+1}^n - u_j^n}{\Delta t} = D\frac{[(u_j^{n+1} - 2u_j^n + u_j^{n-1}) + (u_{j+1}^{n+1} - 2u_{j+1}^n + u_{j+1}^{n-1})]}{2(\Delta x)^2} + f(t_j, x_n)$$

the intuition behind the scheme is to use more nearby grid points, with more information it is likely to be more accurate and stable. The above equation is implicit in the sense that in order to solve u_{j+1}^n, we need u_{j+1}^{n+1} and u_{j+1}^{n-1}, thus by reorganizing we can get a matrix equation.

Let $\alpha = \frac{D\Delta t}{2(\Delta x)^2}$ and $u_i = (u_i^0, u_i^1, \cdots, u_i^N)^T \in \mathbb{R}^{N\times 1}$, a time slice, without considering the $f(t,x)$ term:

$$-\alpha u_{i+1}^{n+1} + (1+2\alpha)u_{i+1}^n - \alpha u_{i+1}^{n-1} = \alpha u_i^{n+1} + (1-2\alpha)u_i^n + \alpha u_i^{n-1}, n = 1, 2, \cdots, N-1$$

and if take into account the boundary condition such as homogeneous Neumann boundary condition $\frac{\partial u}{\partial x}(t,a) = \frac{\partial u}{\partial x}(t,b) = 0$, we can get the final matrix equation. Introduce ghost points u_j^{-1}, u_j^{N+1} and use Euler at the boundary, $\frac{u_j^0 - u_j^{-1}}{\Delta t} = \frac{u_j^{N+1} - u_j^N}{\Delta t} = 0$, i.e $u_j^{-1} = u_j^{N+1} = 0$ which allows the above identity to hold for $n = 0, N$.

$$\begin{bmatrix} 1+\alpha & -\alpha & 0 & 0 & \cdots & 0 & 0 \\ -\alpha & 1+2\alpha & -\alpha & 0 & \cdots & 0 & 0 \\ 0 & \ddots & \ddots & \ddots & & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & -\alpha & 1+2\alpha & -\alpha \\ 0 & 0 & 0 & \cdots & 0 & -\alpha & 1+\alpha \end{bmatrix} u_{i+1} = \begin{bmatrix} 1-\alpha & \alpha & 0 & 0 & \cdots & 0 & 0 \\ \alpha & 1-2\alpha & \alpha & 0 & \cdots & 0 & 0 \\ 0 & \ddots & \ddots & \ddots & & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & \alpha & 1-2\alpha & \alpha \\ 0 & 0 & 0 & \cdots & 0 & \alpha & 1-\alpha \end{bmatrix} u_i$$

we need only to compute the inverse of the matrix on the left-hand side once and store it, since it's a tridiagonal matrix, it is actually not very expensive in computing.

In numerical analysis, von Neumann stability analysis (also known as Fourier stability analysis) is a procedure used to check the stability of finite difference schemes as applied to linear partial differential equations. The analysis is based on the Fourier decomposition of numerical error and was developed at Los Alamos National Laboratory after having been briefly described in a 1947 article by British researchers Crank and Nicolson.

Here we illustrate the method by presenting a few examples.

Example 1.1. Consider the 1-dimensional heat equation

$$\frac{\partial u}{\partial t} = D\frac{\partial^2 u}{\partial x^2}$$

and we analyze the stability of the Forward-time Central-space scheme(FTCS) as follows:

$$u_{j+1}^n = u_j^n + r(u_j^{n+1} - 2u_j^n + u_j^{n-1}) \tag{3}$$

where $r = \frac{D\Delta t}{(\Delta x)^2}$ then we denote the difference between the numerical solution obtained by the computer by finite precision arithmetic and the exact solution to (3) by ϵ_j^n since they all satisfy equation (3), we know that ϵ_j^n also satisfy the equation. Given periodic boundary condition, we can expand $\epsilon(t,x)$ as a finite Fourier sum $\epsilon(t,x) = \sum_{i=-M}^{M} E_i(t)e^{ik_i x}$, and we care only about the behavior of $E_i(t)$. Plug $E(t)e^{ikx}$ into the equation and if $\frac{E(t+\Delta t)}{E(t)} \leq 1$, then the scheme is stable. An alternative way is to plug in $\xi^j e^{ikn\Delta x}$ and see whether $|\xi| \leq 1$ or not.

For the above example: $\frac{E(t+\Delta t)}{E(t)} = 1 - 4rsin^2(k\Delta x)$, and thus is stable when $r = \frac{D\Delta t}{(\Delta x)^2} \leq \frac{1}{2}$, this is called conditionally stable, the time step size needs to be smaller than a certain threshold in order to get a stable scheme, thus needs more computation when implementing it.

Example 1.2 (Crank-Nicolson is unconditionally stable)**.** plug in $\xi^j e^{ikn\Delta x}$, and simplify it to show that it's unconditionally stable.

$$\xi = -1 + \frac{2}{4\alpha sin^2(\frac{k\Delta x}{2}) + 1}, |\xi| \leq 1$$

then we consider a slightly more general diffusion equation:

$$\frac{\partial u}{\partial t} = D\frac{\partial^2 u}{\partial x^2} + \frac{\partial u}{\partial x}$$

we note that for the right-hand side, we can choose different schemes for the two terms respectively, such as one Crank-Nicolson and one forward Euler:

$$\frac{u_{j+1}^n - u_j^n}{\Delta t} = D\frac{[(u_j^{n+1} - 2u_j^n + u_j^{n-1}) + (u_{j+1}^{n+1} - 2u_{j+1}^n + u_{j+1}^{n-1})]}{2(\Delta x)^2} + \frac{u_j^{n+1} - u_j^n}{\Delta x}$$

or both Crank-Nicolson:

$$\frac{u_{j+1}^n - u_j^n}{\Delta t} = D\frac{[(u_j^{n+1} - 2u_j^n + u_j^{n-1}) + (u_{j+1}^{n+1} - 2u_{j+1}^n + u_{j+1}^{n-1})]}{2(\Delta x)^2} + \frac{(u_j^{n+1} - u_j^n) + (u_{j+1}^{n+1} - u_{j+1}^n)}{2\Delta x}$$

for the second one, plug in $\xi^j e^{ikn\Delta x}$, and simplify it to get

$$\xi = \frac{1 - 4\alpha sin^2(\frac{k}{2}) + 2\beta i sin(k)}{1 + 4\alpha sin^2(\frac{k}{2}) - 2\beta i sin(k)} = \frac{1-w}{1+w}$$

thus $|\xi| < 1$ for all k, shows that Crank-Nicolson is unconditionally stable. Thus we are free to choose whatever time step size and always end up with a stable approximation using Crank-Nicolson.

2 Runge Kutta Method

In order to obtain a more accurate approximation, it is tempting to use higher-order derivatives, since we have the exact Taylor's expansion: $x_{i+1} - x_i = x_i^{'}\Delta t + \frac{1}{2}x_i^{''}(\Delta t)^2 + \frac{1}{6}x_i^{(3)}(\Delta t)^3 + \cdots$, the higher order we know, the more accurate. However, we only have access to the first-order derivative in a first-order equation, which prompts us to use the weighted average of first-order derivatives as an alternative, we deal with the

explicit first-order ODE in this section:

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

2.1 derivation of 2nd order formula

Let $h = \Delta t$, we present the goal of obtaining a second-order formula:

$$k_1 = hf(t_n, x_n)$$

$$k_2 = hf(t_n + \alpha h, x_n + \beta k_1)$$

$$x_{n+1} = x_n + ak_1 + bk_2 + O(h^2)$$

and the derivation:

$$\begin{aligned} x_{n+1} - x_n &= hf(t_n, x_n) + \frac{1}{2}f^{'}(t_n, x_n)h^2 + O(h^2) \\ &= hf(t_n, x_n) + \frac{1}{2}h^2(\frac{\partial f}{\partial t}(t_n, x_n) + \frac{\partial f}{\partial x}(t_n, x_n)f(t_n, x_n)) + O(h^2) \end{aligned}$$

$$\begin{aligned} k_2 &= h(f(t_n, x_n) + \alpha h\frac{\partial f}{\partial t}(t_n, x_n) + \beta k_1\frac{\partial f}{\partial x}(t_n, x_n)) + O(h^2) \\ &= hf(t_n, x_n) + \alpha h\frac{\partial f}{\partial t}(t_n, x_n) + \beta h^2 f(t_n, x_n)\frac{\partial f}{\partial x}(t_n, x_n) + O(h^2) \end{aligned}$$

$$x_{n+1} = x_n + ahf(t_n, x_n) + bhf(t_n, x_n) + b\alpha h^2\frac{\partial f}{\partial t}(t_n, x_n) + b\beta h^2 f(t_n, x_n)\frac{\partial f}{\partial x}(t_n, x_n) + O(h^2)$$

Compare the coefficients:

$$\begin{cases} a + b = 1 \\ b\alpha = \frac{1}{2} \\ b\beta = \frac{1}{2} \end{cases}$$

by choosing $a = b = \frac{1}{2}, \alpha = \beta = 1$, we obtain RK2 formula:

$$x_{n+1} = x_n + \frac{1}{2}\Delta tf(t_n, x_n) + \frac{1}{2}\Delta tf(t_{n+1}, x_n + \Delta tf(t_n, x_n)) \tag{4}$$

One can emulate the process above to obtain higher-order RK formulas, considering both computing efficiency and accuracy, the RK4 formula is most widely used in industry and scientific research.

2.2 adaptive Runge Kutta method

In practice, you may have thousands of equations to solve, you can afford neither high error nor extremely small time step size because of computing efficiency, thus it is crucial to have an algorithm that computes the step size for you and changes the size to control the error at each evaluation. An adaptive ode solver automatically finds the best integration step-size at each time step. The Dormand-Prince method, which is implemented in Python's ODE solver. This solver requires six function evaluations per time step, and saves

computational time by constructing both fourth- and fifth-order methods using the same function evaluations. The user only have to specify error tolerance $\epsilon > 0$. Let x_{n+1}, X_{n+1} be the predictions made by 4'th and 5'th methods respectively, define $e = |x_{n+1} - X_{n+1}| = O((\Delta t)^5)$, given the default step Δt

$$\frac{e}{\epsilon} = \frac{(\Delta t)^5}{(\tau)^5}$$

i.e

$$\tau = (\frac{\epsilon}{e})^{\frac{1}{5}} \Delta t$$

but in practice we use instead

$$\tau = 0.9(\frac{\epsilon}{e})^{\frac{1}{5}} \Delta t$$

to avoid hovering too close to ϵ and rejecting, thus wasting, too much computations, here 0.9 is called a default safety factor. here τt is the step we should have used.

(1) $\tau > \Delta t$, this means we should accept X_{n+1} and increase the step.

(2) $\tau \leq \Delta t$, this is bad news since we have to reject it and use a smaller time step τ to redo the calculation.

This is similar to some algorithms used in deep learning, such as learning rate decay or moment method, in that they all function as means to adjust the parameters to the optimal at each step automatically.

3 Examples

In the first two examples, we demonstrate the usage of the Python ODE solver, and the implementation of Crank-Nicolson in the last one.

Example 3.1 (Python ODE solver)**.** Nowadays, we don't have to write a discretization algorithm from scratch because we can use the Python auto ODE solver, here the classic example of Lorenz attractor and spring with resistance is investigated.

```
import matplotlib.pyplot as plt
import numpy as np
from scipy.integrate import solve_ivp
from functools import partial
plt.style.use('seaborn-poster')

def my_lorenz(t, S, sigma, rho, beta):
    dS = np.array([sigma*(S[1]-S[0]),S[0]*(rho-S[2])-S[1],S[0]*S[1]-beta*S[2]])
    return dS
s = np.array([1, 2, 3])
dS = my_lorenz(0, s, 10, 28, 8/3)
print(dS)

def my_lorenz_solver(t_span, s0,t_eval, sigma, rho, beta):
    F = partial(my_lorenz,sigma=sigma,rho =rho, beta = beta)
```

```
    sol = solve_ivp(F,t_span,s0,t_eval= t_eval)
    T = sol.t
    X = sol.y[0]
    Y = sol.y[1]
    Z = sol.y[2]
    return [T,X,Y,Z]

t_eval = np.arange(0,50,0.01)
sigma = 10
rho = 28
beta = 8/3
t0 = 0
tf = 50
s0 = np.array([0, 1, 1.05])
t_span=[t0,tf]

[T, X, Y, Z] = my_lorenz_solver([t0, tf],s0,t_eval, sigma, rho, beta)
from mpl_toolkits import mplot3d
fig = plt.figure(figsize = (10,10))
ax = plt.axes(projection='3d')
ax.grid()
ax.plot3D(X, Y, Z)
# Set axes label
ax.set_xlabel('x', labelpad=20)
ax.set_ylabel('y', labelpad=20)
ax.set_zlabel('z', labelpad=20)
plt.show()
```

and the result is shown below: For the example of spring with resistance, the code and graphs are in the attachment.

Example 3.2 (Logistic equation)**.** We have introduced quite a few schemes so far, it is time to compare their performances, here we present the logistic equation as an example for which we shall use four different schemes and compare them with the exact solution.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import solve_ivp
from functools import partial

# we use the logistic equation to illustrate the diffenrence between these methods
r = 0.7
K = 20
p_0 = np.array([30])
```

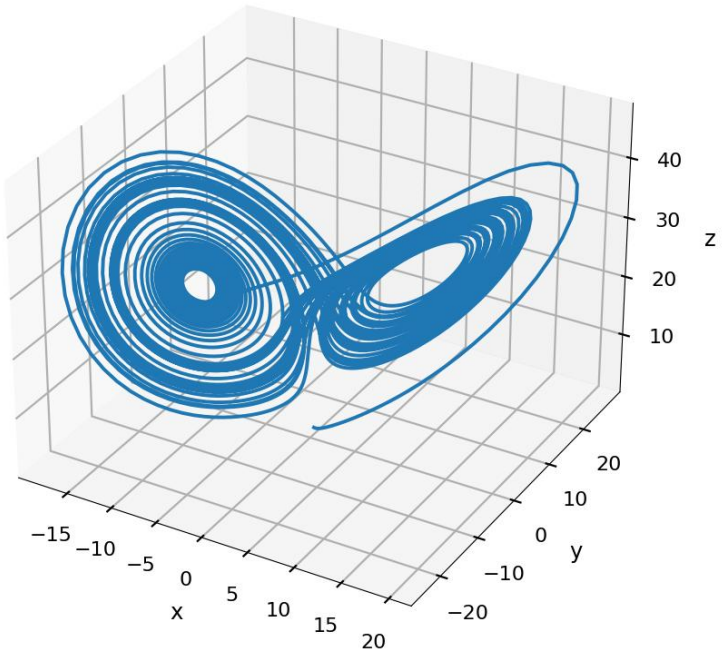

Figure 1: Lorenz attractor

```
t_s = 0
t_f = 10
t_span = [t_s,t_f]
step = 0.5
t_eval = np.arange(t_s,t_f,step)
sample_points = int((t_f-t_s)/step)
t = np.linspace(t_s,t_f,sample_points)

def logistic(t,P,r,K):
    dP = r*P*(1-P/K)
    return dP
def python_solver(t_span,p_0,t_eval,r,K):
    F = partial(logistic,r=r,K=K)
    sol = solve_ivp(F,t_span,p_0,t_eval= t_eval)
    return sol

def basic_Euler(sample_points,p_0,step,r,K):
    F = partial(logistic,r=r,K=K)
    X = [p_0]
    for i in range(sample_points-1):
        x = X[i]+F(0,X[i])*step
        X.append(x)
    return X

def refined_Euler(sample_points,p_0,step,r,K):
    F = partial(logistic,r=r,K=K)
    X = [p_0]
    for i in range(sample_points-1):
        x = X[i]+F(0,X[i])*step
        y = X[i]+0.5*F(0,X[i])*step+0.5*F(0,x)*step
        X.append(y)
    return X

def RK4(sample_points,p_0,step,r,K):
    F = partial(logistic,r=r,K=K)
    X = [p_0]
    for i in range(sample_points-1):
        k_1 = F(0,X[i])*step
        k_2 = F(0,X[i]+0.5*k_1)*step
        k_3 = F(0,X[i]+0.5*k_2)*step
        k_4 = F(0,X[i]+0.5*k_3)*step
        x = X[i] + 1/6*(k_1+2*k_2+2*k_3+k_4)
        X.append(x)
    return X
```

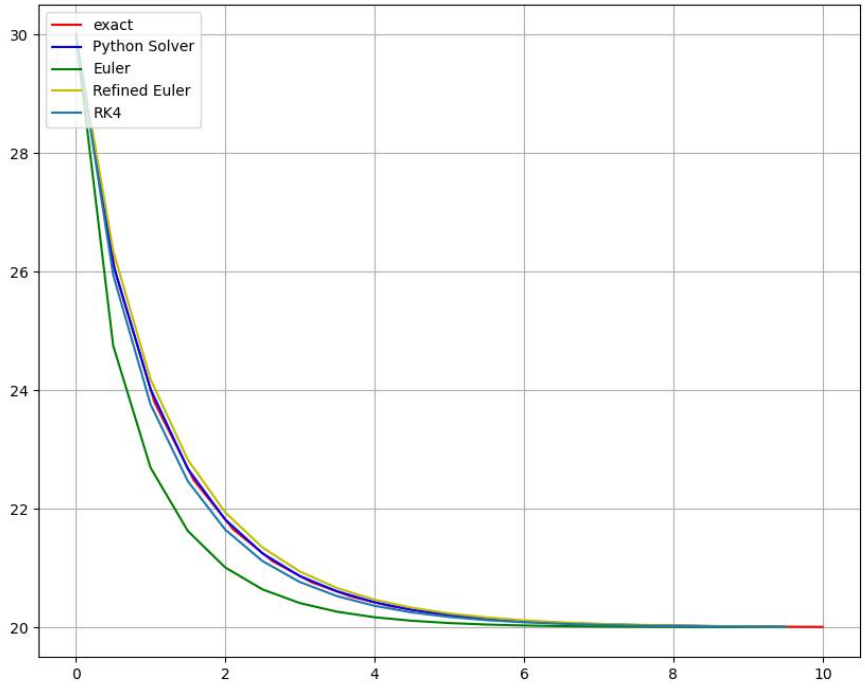

Figure 2: methods comparison

```
sol = python_solver(t_span,p_0,t_eval,r,K)
X_e = basic_Euler(sample_points,p_0,step,r,K)
X_re =refined_Euler(sample_points,p_0,step,r,K)
X_rk =RK4(sample_points,p_0,step,r,K)

plt.figure(figsize = (10, 8))
plt.plot(t,K*p_0*np.exp(r*t)/(K+ p_0*(np.exp(r*t)-1)),'r',label='exact')
plt.plot(sol.t,sol.y[0],'b',label = 'Python Solver')
plt.plot(sol.t,X_e,'g', label = 'Euler')
plt.plot(sol.t,X_re,'y', label = 'Refined Euler')
plt.plot(sol.t,X_rk, label = 'RK4')
plt.grid()
plt.legend(loc=2)
filename = "methods comparison"
plt.savefig(filename,format="jpg")

plt.show()
```

here we can see that Euler is most inaccurate as expected, due to rather a simple function, the performances

of RK4 and refined Euler are similar, and the ODE solver is the most accurate because it employs RK4 by default and can adjust its step size automatically. We can also observe that the Euler method curve starts with high error and builds up along with iteration, whereas other methods are better at adjusting themselves.

Example 3.3 (Crank-Nicolson)**.** Here I present the implementation of Crank-Nicolson on a simple 1-D heat equation, the graphs are in the attachment. I also use the finite sum of the solution's Fourier series to approximate it.

$$\begin{cases} \frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} & 0 \leq x \leq 1, t > 0 \\ u(0,x) = \begin{cases} 2x & 0 \leq x \leq \frac{1}{2} \\ 2-2x & \frac{1}{2} \leq x \leq 1 \end{cases} \\ u(t,0) = u(t,1) = 0 \end{cases}$$

4 References

1. Qingkai Kong, Timmy Siauw, Alexandre Bayen. *Python Programming and Numerical Methods, A Guide for Engineers and Scientists*
2. J. Crank, P. Nicolson, *A Practical Method for Numerical Evaluation of Solutions of Partial Differential Equations of The Heat Conduction Type*.
3. Jeffrey R. Chasnov, *Numerical Methods for Engineers*, The Hong Kong University of Science and Technology.
4. Wikipedia, Von Neumann stability analysis.