Diffusion Equation with Finite Difference Method
TF4062

Iwan Prasetyo
Fadjar Fathurrahman

1 Initial-boundary value problem for 1d diffusion

We consider 1d diffusion (or heat) equation:

$$\frac{\partial u}{\partial t} = \alpha \frac{\partial^2 u}{\partial t^2} + f(x,t) \tag{1}$$

Initial condition:

$$u(x,0) = I(x), \qquad x \in [0, L] \tag{2}$$

Boundary condition:

$$u(0,t) = 0, \qquad u(L,t) = 0, \qquad t > 0 \tag{3}$$

Spatial grid:

$$x_i = (i-1)\Delta x, \qquad i = 1, \ldots, N_x \tag{4}$$

Temporal grid:

$$t_n = (n-1)\Delta t, \qquad n = 1, \ldots, N_t \tag{5}$$

2 Forward Euler scheme

In forward Euler scheme, forward difference to approximate time derivative and second order central difference for spatial derivative are used to discretize the PDE (1):

$$\frac{u_i^{n+1} - u_i^n}{\Delta t} = \alpha \frac{u_{i+1}^n - 2u_i^n + u_{i-1}^n}{\Delta x^2} + f_i^n. \tag{6}$$

By using the following definition of *mesh Fourier number*:

$$F = \alpha \frac{\Delta t}{\Delta x^2}, \tag{7}$$

we can rearrange the equation (6) to

$$u_i^{n+1} = u_i + F\left(u_{i+1}^n - 2u_i^n + u_{i+1}^n\right) + f_i^n \Delta t. \tag{8}$$

Because the RHS of the equation (8) is known, it can be used used to advance the solution u_i^n directly for a given initial and boundary conditions. I can be shown that this scheme is conditionally stable. For a stable solution the following condition must be satisfied:

$$F \leq \frac{1}{2} \tag{9}$$

3 Backward Euler scheme

In backward Euler scheme, forward difference to approximate time derivative and second order central difference for spatial derivative are used to discretize the PDE (1):

$$\frac{u_i^n - u_i^{n-1}}{\Delta t} = \alpha \frac{u_{i+1}^n - 2u_i^n + u_{i-1}^n}{\Delta x^2} + f_i^n \tag{10}$$

which can be rearranged to

$$-Fu_{i-1}^n + (1+2F)u_i^n - Fu_{i+1}^n = u_i^{n-1} + f_i^n \tag{11}$$

for $i = 1, 2, \ldots, N_x$. We cannot write u_i^n directly in terms of known quantities. We have to solve a system of linear equations to find u_i^n. This linear system can be written as:

$$\mathbf{A}\mathbf{u} = \mathbf{b} \tag{12}$$

The matrix $\mathbf{A}$ has the following tridiagonal structure:

$$\begin{bmatrix}
A_{1,1} & A_{1,2} & 0 & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\
A_{1,2} & A_{2,2} & A_{2,3} & \cdots & \cdots & \cdots & & & \vdots \\
0 & A_{3,2} & A_{3,3} & A_{3,4} & \cdots & \cdots & & & \vdots \\
\vdots & \ddots & & \ddots & & 0 & & & \vdots \\
\vdots & & & \ddots & \ddots & \ddots & \ddots & & \vdots \\
\vdots & & & 0 & A_{i,j-1} & A_{i,j} & A_{i,j+1} & \ddots & \vdots \\
\vdots & & & & \ddots & \ddots & \ddots & \ddots & 0 \\
\vdots & & & & & \ddots & \ddots & \ddots & A_{N_x-1,N_x} \\
0 & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & A_{N_x,N_x-1} & A_{N_x,N_x}
\end{bmatrix} \tag{13}$$

where the matrix elements for inner points ($i = 2, 3, \ldots, N_x - 1$ are:

$$\begin{aligned}
A_{i,i-1} &= -F \\
A_{i,i} &= 1 + 2F \\
A_{i,i+1} &= -F
\end{aligned}$$

For boundary points, due to the boundary conditions defined in (3) we have

$$\begin{aligned}
A_{1,1} &= 1 \\
A_{1,2} &= 0 \\
A_{N_x-1,N_x-1} &= 0 \\
A_{N_x,N_x} & 1
\end{aligned}$$

For RHS, the elements of column vector $\mathbf{b}$ are $b_1 = 0$ and $b_{N_x} = 0$ and

$$b_i = u_i^{n-1} + f_i^{n-1}\Delta t, \qquad i = 2, \ldots, N_x - 1 \tag{14}$$

Because we have to solve a system of linear equations backward Euler scheme is categorized as an implicit scheme. It can be shown that this scheme is unconditionally stable.

4 Crank-Nicolson (CN) method

In the Crank-Nicolson method we require the PDE to be satisfied at the spatial mesh point x_i but midway between the points in the time mesh ($t_{n+\frac{1}{2}}$):

$$\frac{\partial}{\partial t} u_i^{n+\frac{1}{2}} = \alpha \frac{\partial^2}{\partial x^2} u_i^{n+\frac{1}{2}} + f_i^{n+\frac{1}{2}} \tag{15}$$

Using centered difference in space and time:

$$\frac{u_i^{n+1} - u_i^n}{\Delta t} = \frac{1}{\Delta x^2}\left(u_{i+1}^{n+\frac{1}{2}} - 2u_i^{n+\frac{1}{2}} + u_{i-1}^{n+\frac{1}{2}}\right) + f_i^{n+\frac{1}{2}} \tag{16}$$

$u_i^{n+\frac{1}{2}}$ is not the quantity that we want to calculate so we must approximate it. We can approximate it by an average between the value at t_n and t_{n+1}:

$$u_i^{n+\frac{1}{2}} \approx \frac{1}{2}(u_i^n + u_i^{n+1}) \tag{17}$$

We also can use the same approximation for $f_i^{n+\frac{1}{2}}$:

$$f_i^{n+\frac{1}{2}} \approx \frac{1}{2}(f_i^n + f_i^{n+1}) \tag{18}$$

Substituting these approximations we obtain:

$$u_i^{n+1} - \frac{1}{2}F\left(u_{i-1}^{n+1} - 2u_i^{n+1} + u_{i+1}^{n+1}\right) = u_i^n + \frac{1}{2}F\left(u_{i-1}^n - 2u_i^n + u_{i+1}^n\right) + \frac{1}{2}f_i^{n+1} + \frac{1}{2}f_i^n \tag{19}$$

We notice that the equation (19) has similar structure as the one we obtained for backward Euler method:

$$\mathbf{A}\mathbf{u} = \mathbf{b} \tag{20}$$

The element of the matrix $\mathbf{A}$ are:

$$\begin{aligned} A_{i,i-1} &= -\frac{1}{2}F \\ A_{i,i} &= 1 + F \\ A_{i,i+1} &= -\frac{1}{2}F \end{aligned}$$

for internal points $i = 2, \ldots, N_x - 1$. For boundary points we have:

$$\begin{aligned} A_{1,1} &= 1 \\ A_{1,2} &= 0 \\ A_{N_x,N_x-1} &= 0 \\ A_{N_x,N_x} &= 1 \end{aligned}$$

For the right-hand side vector $\mathbf{b}$ we have $b_1 = 0$ and $b_{N_x} = 0$ and

$$b_i = u_i^n + \frac{1}{2}F\left(u_{i+1}^n - 2u_i^n + u_{i-1}^n\right) + \frac{1}{2}\left(f_i^{n+1} + f_i^n\right)\Delta t \tag{21}$$

for internal points $i = 2, \ldots, N_x - 1$.

Because we have to solve a system of linear equations, Crank-Nicolson scheme is also categorized as an implicit scheme. It can be shown that this scheme is unconditionally stable.

5 Implementation

In this section we provide simple implementations of the following schemes:

- forward Euler (`diffusion_1d_explicit`),
- backward Euler (`diffusion_1d_implicit`),
- backward Euler (`diffusion_1d_CN`)

The following arguments are used:

- `L`: coordinate of the rightmost point. The leftmost point is taken to be 0.
- `T`: the final time when the solution must be computed.
- `Nx` and `Nt`: number of points in spatial and temporal grid, respectively.
- `α`: the coefficient α in the diffusion equation, taken to be a constant.
- `u0x`: a function describing initial condition $I(x)$.
- `bx0` and `bxL`: two functions describing boundary conditions at $x = 0$ and $x = L$, respectively. In general these functions may take time as an argument, however for our present case they simply return a number (zero).
- `f`: a function describing the source term. It takes spatial coordinate and time as the arguments.

```
function diffusion_1d_explicit(
    L::Float64, Nx::Int64, T::Float64, Nt::Int64,
    α::Float64, u0x, bx0, bxL, f
)

    Δx = L/(Nx-1)
    x = collect(range(0.0, stop=L, length=Nx))

    Δt = T/(Nt-1)
    t = collect(range(0.0, stop=T, length=Nt))

    u = zeros(Float64, Nx, Nt)

    for i in 1:Nx
        u[i,1] = u0x(x[i])
    end

    for k in 1:Nt
        u[1,k] = bx0(t[k])
        u[Nx,k] = bxL(t[k])
    end

    F = α*Δt/Δx^2

    if F >= 0.5
        @printf("diffusion_1d_explicit:\n")
        @printf("WARNING: F is greater than 0.5: %f\n", F)
        @printf("WARNING: The solution is not guaranteed to be stable !!\n")
    else
        @printf("diffusion_1d_explicit:\n")
        @printf("INFO: F = %f >= 0.5\n", F)
        @printf("INFO: The solution should be stable\n")
    end

    for n in 1:Nt-1
        for i in 2:Nx-1
            u[i,n+1] = F*( u[i+1,n] + u[i-1,n] ) + (1 - 2*F)*u[i,n] + f(x[i],
             ↪  t[n])*Δt
        end
    end

    return u, x, t
end
```

```julia
function diffusion_1d_implicit(
    L::Float64, Nx::Int64, T::Float64, Nt::Int64,
    α::Float64, u0x, bx0, bxL, f
)
    Δx = L/(Nx-1)
    x = collect(range(0.0, stop=L, length=Nx))

    Δt = T/(Nt-1)
    t = collect(range(0.0, stop=T, length=Nt))

    u = zeros(Float64, Nx, Nt)

    for i in 1:Nx
        u[i,1] = u0x(x[i])
    end

    for k in 1:Nt
        u[1,k] = bx0(t[k])
        u[Nx,k] = bxL(t[k])
    end

    F = α*Δt/Δx^2

    A = zeros(Float64, Nx, Nx)
    b = zeros(Float64, Nx)
    for i in 2:Nx-1
        A[i,i] = 1 + 2*F
        A[i,i-1] = -F
        A[i,i+1] = -F
    end
    A[1,1] = 1.0
    A[Nx,Nx] = 1.0

    for n in 2:Nt
        for i in 2:Nx-1
            b[i] = u[i,n-1] + f(x[i],t[n])*Δt
        end
        b[1] = 0.0
        b[Nx] = 0.0
        u[:,n] = A\b    # Solve the linear equations
    end
    return u, x, t

end
```

```julia
function diffusion_1d_CN(
    L::Float64, Nx::Int64, T::Float64, Nt::Int64,
    α::Float64, u0x, bx0, bxL, f
)

    Δx = L/(Nx-1)
    x = collect(range(0.0, stop=L, length=Nx))

    Δt = T/(Nt-1)
    t = collect(range(0.0, stop=T, length=Nt))

    u = zeros(Float64, Nx, Nt)

    for i in 1:Nx
        u[i,1] = u0x(x[i])
    end
```

```
    for k in 1:Nt
        u[1,k] = bx0(t[k])
        u[Nx,k] = bxL(t[k])
    end

    F = α*Δt/Δx^2

    A = zeros(Float64, Nx, Nx)
    b = zeros(Float64, Nx)
    for i in 2:Nx-1
        A[i,i] = 1 + F
        A[i,i-1] = -0.5*F
        A[i,i+1] = -0.5*F
    end
    A[1,1] = 1.0
    A[Nx,Nx] = 1.0

    for n in 1:Nt-1
        for i in 2:Nx-1
            b[i] = u[i,n] + 0.5*F*( u[i-1,n] - 2*u[i,n] + u[i+1,n] ) +
                   0.5*( f(x[i],t[n]) + f(x[i],t[n+1]) )*Δt
        end
        b[1] = 0.0
        b[Nx] = 0.0
        u[:,n+1] = A\b  # Solve the linear equations
    end
    return u, x, t

end
```

6 Verification

```
using Printf

import PyPlot
const plt = PyPlot
plt.rc("text", usetex=true)

include("diffusion_1d_explicit.jl")
include("diffusion_1d_implicit.jl")
include("diffusion_1d_CN.jl")

const L =  1.0
const α = 1.0

function analytic_solution(x, t)
    return 5*t*x*(L - x)
end

function source_term(x, t)
    return 10*α*t + 5*x*(L - x)
end

function initial_cond(x)
    return analytic_solution(x, 0.0)
end

function bx0(t)
    return 0.0
end
```

```julia
function bxL(t)
    return 0.0
end

function main()

    T = 0.1
    Nx = 21
    F = 0.5
    dx = L/(Nx-1)
    Δt = F*dx^2/α
    Nt = round(Int64,T/Δt) + 1

    # Please change accordingly (or use loop)
    u, x, t = diffusion_1d_explicit(L, Nx, T, Nt, α, initial_cond, bx0, bxL,
     ↪  source_term)

    u_e = analytic_solution.(x, t[end])
    diff_u = maximum(abs.(u_e - u[:,end]))
    println("diff_u = ", diff_u)

end

main()
```

7 Example 1

Only import parts are included. The remaining is similar to the verification program.

```julia
function initial_temp(x)
    return sin(π*x)
end

function bx0( t )
    return 0.0
end

function bxf( t )
    return 0.0
end

function source_term(x, t)
    return 0.0
end

function analytic_solution(x, t)
    return sin(π*x) * exp(-π^2 * t)
end

function main()
    α = 1.0
    L = 1.0
    T = 0.2
    Nx = 25
    Nt = 400

    u, x, t = diffusion_1d_explicit( L, Nx, T, Nt, α, initial_temp, bx0, bxf,
     ↪  source_term )

    u_a = analytic_solution.(x, t[end])
```

```
    u_n = u[:,end]
    rmse = sqrt( sum((u_a - u_n).^2)/Nx )
    mean_abs_diff = sum( abs.(u_a - u_n) )/Nx
    @printf("RMS error             = %15.10e\n", rmse)
    @printf("Means abs diff error = %15.10e\n", mean_abs_diff)
end

main()
```