# Part 3: Diffusion Equations

*Solving Generalized Diffusion Equations with the Finite Difference Method*

Petar Bosnic

*University of South-Eastern Norway*

November 27, 2024

**Note**

This document was developed as of the PhD course *Numerical Solutions to Partial Differential Equations* under the guidance of Professors Svein Linge and Knut Vågsæther at the University of South-Eastern Norway (USN). Course material includes the textbooks:

- *Finite Difference Computing with PDEs: A Modern Software Approach* by Hans Petter Langtangen and Svein Linge (DOI: 10.1007/978-3-319-55456-3)
- *Riemann Solvers and Numerical Methods for Fluid Dynamics: A Practical Introduction* by E. F. Toro (ISBN: 978-3-540-25202-3 978-3-540-49834-6)

# Contents

# 1 1D Diffusion Solver Development

This section introduces the `unified_diffusion_solver` function, a solver for the one-dimensional diffusion equation with a source term. The function supports both Forward Euler (FE) and Backward Euler (BE) time-stepping methods, and scalar and vectorized implementations to enhance computational efficiency. Following this, the `theta_diffusion_solver` function is presented, implementing the unified theta rule for solving 1D diffusion equations. Finally, a mathematical analysis of schemes is provided to explain the origins of numerical errors and instabilities.

## 1.1 Mathematical Formulation

The diffusion equation with a source term is given by:

$$u_t = au_{xx} + f(x,t),$$

where:

- $u(x,t)$: The quantity being diffused (e.g., temperature).
- $a$: Diffusion coefficient, which controls the rate of diffusion.
- $f(x,t)$: Source term, representing external input to the system.

The domain is defined over $x \in [0, L]$, and the solver computes the solution $u(x,t)$ over the time interval $t \in [0, T]$.

**Code Implementation**

## 1.2 Function Parameters

The solver is defined as:

```python
def unified_diffusion_solver(I, f, a, L, dt, F, T, method='FE', version=
    'scalar', user_action=None):
```

The key parameters are:

- `I`: Initial condition function, $I(x) = u(x,0)$.
- `f`: Source term function, $f(x,t)$.
- `a`: Diffusion coefficient.
- `L`: Length of the spatial domain.
- `dt`: Time step size.
- `F`: Fourier number, defined as $F = \frac{a\Delta t}{\Delta x^2}$.
- `T`: Total simulation time.
- `method`: Time-stepping method (`'FE'` for Forward Euler, `'BE'` for Backward Euler).
- `version`: Implementation type (`'scalar'` or `'vectorized'`).
- `user_action`: Optional function for custom post-processing at each time step.

## 1.3 Spatial and Temporal Discretization

The solver creates mesh points in space and time using the specified spatial resolution ($\Delta x$) and time step size ($\Delta t$):

```
Nt = int(round(T/float(dt))) # Number of time steps
t = np.linspace(0, Nt*dt, Nt+1) # Time mesh
dx = np.sqrt(a*dt/F) # Compute spatial step from Fourier number
Nx = int(round(L/dx)) # Number of spatial points
x = np.linspace(0, L, Nx+1) # Spatial mesh
```

## 1.4 Time-Stepping Methods

The solver supports two methods for advancing the solution in time:

### 1.4.1 Forward Euler Method

The Forward Euler (FE) method is an explicit time-stepping scheme used to update the solution of the diffusion equation at each time step. The update formula for the discrete solution $u_i^n$ is derived from the diffusion equation:

$$u_t = au_{xx} + f(x,t).$$

Discretizing the time derivative $u_t$ using a forward difference and the second spatial derivative $u_{xx}$ using a central difference gives:

$$\frac{u_i^{n+1} - u_i^n}{\Delta t} = a\frac{u_{i-1}^n - 2u_i^n + u_{i+1}^n}{\Delta x^2} + f(x_i, t_n).$$

Rearranging for $u_i^{n+1}$:

$$u_i^{n+1} = u_i^n + F\left(u_{i-1}^n - 2u_i^n + u_{i+1}^n\right) + \Delta t\, f(x_i, t_n),$$

where $F$ is the Fourier number, defined as:

$$F = \frac{a\Delta t}{\Delta x^2}.$$

**Scalar Implementation** In the scalar implementation, the solution at each grid point $i$ is updated sequentially using a loop over the spatial domain:

```
for n in range(Nt): # Loop over time steps
    if version == 'scalar':
        for i in range(1, Nx): # Loop over spatial points, excluding
            boundaries
            u[i] = u_n[i] + F * (u_n[i - 1] - 2 * u_n[i] + u_n[i +
                1]) + dt * f(x[i], t[n])
```

- The loop over $n$ iterates through all time steps $t_n$, from $n = 0$ to $n = N_t$.
- For each time step, the loop over $i$ updates the solution $u[i]$ at all interior grid points ($i = 1$ to $i = N_x - 1$).

- The term $F\left(u_{i-1}^n - 2u_i^n + u_{i+1}^n\right)$ accounts for the diffusion effect.
- The source term $\Delta t\, f(x_i, t_n)$ incorporates external contributions at each spatial point $x_i$ and time $t_n$.

**Vectorized Implementation** In the vectorized implementation, the updates for all interior grid points are performed simultaneously using array slicing:

```
elif version == 'vectorized':
        u[1:Nx] = u_n[1:Nx] + F * (u_n[:-2] - 2 * u_n[1:-1] + u_n[2:]) +
            dt * f(x[1:Nx], t[n])
```

- The slicing notation `u[1:Nx]` selects all interior grid points.
- The terms `u_n[:-2]`, `u_n[1:-1]`, and `u_n[2:]` correspond to $u_{i-1}^n$, $u_i^n$, and $u_{i+1}^n$, respectively.
- This implementation eliminates the need for a loop over $i$, improving computational efficiency for large grids.

**Boundary Conditions** After updating the interior points, Dirichlet boundary conditions are applied to enforce $u[0] = u[N_x] = 0$, fixing the solution at the domain boundaries:

```
u[0] = u[Nx] = 0 # Dirichlet boundary conditions
```

**User-Defined Actions** If a `user_action` function is provided, it is executed at each time step to allow for custom post-processing or visualization:

```
if user_action is not None:
        user_action(u, x, t, n + 1)
```

**Updating for the Next Time Step** At the end of each time step, the arrays $u_n$ and $u$ are swapped, preparing the solver for the next iteration:

```
u_n, u = u, u_n
```

### 1.4.2 Backward Euler (Implicit Method)

The Backward Euler method, also known as an implicit method, is a time-stepping scheme designed to solve the diffusion equation with unconditional stability. This method evaluates the solution at the next time step $n + 1$, requiring the solution of a linear system at each time step.

**Discretization** The diffusion equation:

$$u_t = au_{xx} + f(x,t),$$

is discretized using the Backward Euler scheme. The time derivative $u_t$ is approximated using a backward difference in time and central difference in space:

$$\frac{u_i^n - u_i^{n-1}}{\Delta t} = a\frac{u_{i+1}^n - 2u_i^n + u_{i-1}^n}{\Delta x^2} + f(x_i, t^n).$$

Rearranging to isolate the terms for $u_i^n$, we obtain:

$$u_i^n - F\left(u_{i+1}^n - 2u_i^n + u_{i-1}^n\right) = u_i^{n-1} + \Delta t\, f(x_i, t^n),$$

where $F = \frac{a\Delta t}{\Delta x^2}$ is the Fourier number.

Distribute $F$:

$$u_i^n(1+2F) - Fu_{i+1}^n - Fu_{i-1}^n = u_i^{n-1} + \Delta t\, f(x_i, t^n).$$

Reorganize:

$$-Fu_{i-1}^n + (1+2F)u_i^n - Fu_{i+1}^n = u_i^{n-1} + \Delta t\, f(x_i, t^n).$$

This equation requires solving a linear system at each time step, which can be expressed in matrix form as:

$$A\mathbf{u}^n = \mathbf{b},$$

where $A$ is the coefficient matrix, $\mathbf{u}$ is solution vector and $\mathbf{b}$ is the right-hand side vector.

The matrix $A$ has the following structure:

$$A = \begin{bmatrix}
A_{0,0} & A_{0,1} & 0 & 0 & \cdots & 0 & \\
A_{1,0} & A_{1,1} & A_{1,2} & 0 & \cdots & 0 & \\
0 & A_{2,1} & A_{2,2} & A_{2,3} & \cdots & 0 & \\
\vdots & \vdots & \vdots & \ddots & \ddots & \vdots & \\
0 & \cdots & \cdots & A_{i,i-1} & A_{i,i} & A_{i,i+1} & \vdots \\
\vdots & \vdots & \vdots & \vdots & \ddots & \ddots & 0 \\
0 & 0 & \cdots & \cdots & A_{Nx-1,Nx-2} & A_{Nx-1,Nx-1} & A_{Nx-1,Nx} \\
0 & 0 & \cdots & \cdots & 0 & A_{Nx,Nx-1} & A_{Nx,Nx}
\end{bmatrix}$$

The non-zero elements of the matrix $A$ are defined as follows for internal points $i = 1, \ldots, N_x - 1$:

$$A_{i,i-1} = -F,$$

$$A_{i,i} = 1 + 2F,$$

$$A_{i,i+1} = -F.$$

The first and last equations correspond to the boundary conditions, where the solution is known. The matrix elements at the boundaries are:

$$A_{0,0} = 1,$$

$$A_{0,1} = 0,$$
$$A_{N_x,N_x-1} = 0,$$
$$A_{N_x,N_x} = 1.$$

$$A = \begin{bmatrix} 1 & 0 & 0 & \dots & 0 \\ 0 & 1+2F & -F & \dots & 0 \\ 0 & -F & 1+2F & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix},$$

and the right-hand side vector $\mathbf{b}$ is given by:

$$\mathbf{b} = \begin{bmatrix} 0 \\ u_1^{n-1} + \Delta t f(x_1, t^n) \\ u_2^{n-1} + \Delta t f(x_2, t^n) \\ \vdots \\ u_{N_x-1}^{n-1} + \Delta t f(x_{N_x-1}, t^n) \\ 0 \end{bmatrix}.$$

Here: - The boundary conditions are incorporated into $A$ and $\mathbf{b}$: - $A[0,0] = A[N_x, N_x] = 1$ ensures Dirichlet boundary conditions ($u_0^n = u_{N_x}^n = 0$). - The corresponding entries in $\mathbf{b}$ are zero to enforce these fixed boundary values. - $F = \frac{a\Delta t}{\Delta x^2}$ is the Fourier number, which determines the weighting of neighboring points in the spatial discretization.

At each time step, solving $A\mathbf{u}^n = \mathbf{b}$ provides the updated solution $\mathbf{u}^n$ for the system.

**Implementation in Code** The matrix $A$ is a key component of the Backward Euler method, representing the coefficients of the spatial terms in the implicit discretization. Its construction is implemented as follows:

```
# Construct dense matrix A
A = np.zeros((Nx + 1, Nx + 1)) # Initialize an (Nx + 1) x (Nx + 1)
    matrix with zeros
b = np.zeros(Nx + 1) # Initialize the right-hand side vector
```

- The matrix $A$ is initialized as a zero matrix of size $(N_x+1)\times(N_x+1)$, where $N_x+1$ corresponds to the number of spatial grid points.
- The vector $b$ is initialized with zeros. This vector will store the right-hand side values of the linear system for the current time step.

Next, the coefficients of $A$ are populated for the internal grid points (excluding boundary points):

```
# Populate A for internal points
for i in range(1, Nx):
    A[i, i - 1] = -F # Coefficient for u_{i-1}^n
    A[i, i + 1] = -F # Coefficient for u_{i+1}^n
    A[i, i] = 1 + 2 * F # Coefficient for u_i^n
```

Boundary conditions are implemented by setting the first and last diagonal entries of $A$ to 1, ensuring Dirichlet boundary conditions:

```
# Boundary conditions
A[0, 0] = A[Nx, Nx] = 1 # Dirichlet boundary conditions
```

- By setting $A[0,0] = 1$ and $A[N_x, N_x] = 1$, we ensure that the solution at the boundaries remains fixed (e.g., $u[0] = 0$ and $u[N_x] = 0$).

The right-hand side vector $\mathbf{b}$ is computed at each time step based on the previous solution $\mathbf{u}^n$ and the source term $f(x,t)$:

```
# Compute right-hand side b
for i in range(1, Nx):
      b[i] = u_n[i] + dt * f(x[i], t[n])
b[0] = b[Nx] = 0 # Dirichlet boundary conditions
```

- For internal points, $b[i]$ is updated with the value of $u_i^{n-1}$ (the solution at the previous time step) and the contribution from the source term $f(x, t^n)$.
- The boundary points $b[0]$ and $b[N_x]$ are set to zero to enforce Dirichlet boundary conditions.

Finally, the linear system $A\mathbf{u}^n = \mathbf{b}$ is solved at each time step to compute the updated solution:

```
# Solve the linear system
u[:] = np.linalg.solve(A, b) # dense matrix solver
```

- The numpy function `np.linalg.solve` is a method for solving linear systems of the form $A\mathbf{u} = \mathbf{b}$. It computes $\mathbf{u} = A^{-1}\mathbf{b}$.
- The solution array $u$ is updated with the computed values of $\mathbf{u}^n$, representing the numerical solution at the next time step. The use of slicing (`u[:]`) ensures that the array $u$ is updated in place, preserving memory efficiency.
- `np.linalg.solve`
  - The function is efficient for dense matrices like $A$, which are fully populated due to the second-order spatial derivative terms.
  - The matrix $A$ for the diffusion equation is sparse, as most of its elements are zero. While dense solvers like `np.linalg.solve` can handle small to moderately sized systems efficiently, they do not exploit the sparsity of $A$, which can lead to unnecessary computations.

**Vectorized Implementation Using Sparse Matrices** :

The **vectorized implementation** of the Backward Euler method leverages the efficiency of sparse matrices to solve the system of equations more computationally efficiently than using dense matrices. Sparse matrices are particularly beneficial for problems with tridiagonal structures, because they minimize memory usage and computation.

**Sparse Matrix Setup** In this implementation, the tridiagonal coefficient matrix $A$ is constructed using the `spdiags` function from the `scipy.sparse` module. This approach is more efficient than constructing a dense matrix because it directly creates a compressed sparse column (CSC) representation. The matrix $A$ for the Backward Euler method is defined as:

$$A = \begin{bmatrix} 1 & 0 & 0 & \dots & 0 \\ 0 & 1+2F & -F & \dots & 0 \\ 0 & -F & 1+2F & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix},$$

where:

- The main diagonal contains $1 + 2F$, representing the central coefficients in the spatial discretization.
- The lower diagonal contains $-F$, corresponding to $u_{i-1}^n$.
- The upper diagonal contains $-F$, corresponding to $u_{i+1}^n$.

The following code sets up the sparse matrix:

```
diagonal = np.ones(Nx + 1) * (1 + 2 * F)
lower = np.ones(Nx) * -F
upper = np.ones(Nx) * -F

diagonal[0] = diagonal[-1] = 1 # Dirichlet boundary conditions
lower[-1] = 0
upper[0] = 0

A = spdiags(
      [np.append(lower, 0), diagonal, np.append(0, upper)],
      [-1, 0, 1],
      Nx + 1,
      Nx + 1,
      ).tocsc()
```

**Right-Hand Side Vector** At each time step, the right-hand side vector $\mathbf{b}$ is constructed as:

$$\mathbf{b}_i = u_i^{n-1} + \Delta t \cdot f(x_i, t_{n+1}),$$

where:

- $u_i^{n-1}$ is the solution from the previous time step.
- $f(x_i, t_{n+1})$ is the source term evaluated at the next time step.

The code for constructing $\mathbf{b}$ is:

```
b = u_n + dt * f(x, t[n + 1])
b[0] = b[-1] = 0.0 # Dirichlet boundary conditions
```

**Solving the Linear System**

$$A\mathbf{u}^n = \mathbf{b},$$

where:

- $A$ is the sparse coefficient matrix constructed using the tridiagonal structure of the discretized equation.
- $\mathbf{b}$ is the right-hand side vector incorporating the solution from the previous time step and the source term.
- $\mathbf{u}^n$ is the solution vector.

The sparse system is solved using the `spsolve` function from the `scipy.sparse.linalg` module:

```
u[:] = spsolve(A, b)
```

**Explanation:**

- `spsolve(A, b)` directly solves the linear system $A\mathbf{u}^n = \mathbf{b}$ using efficient numerical methods designed for sparse matrices.
- The operator $u[:]$ ensures that the solution is stored in-place in the array $u$, avoiding the creation of additional arrays and saving memory.
- Sparse solvers like `spsolve` reduce computational cost and memory usage compared to dense solvers, as they operate only on non-zero entries in the matrix $A$.

**Perform any user-defined actions** :

```
if user_action is not None:
user_action(u, x, t, n + 1)
```

The optional `user_action` function allows for custom operations such as visualization, data storage, or error analysis at each time step.

**Update the solution arrays for the next iteration**:

```
u_n, u = u, u_n
```

This swaps the arrays $u_n$ (solution at the current time step) and $u$ (solution at the next time step) in-place, preparing for the next iteration.

**Complete Code for Time Stepping:**

```
for n in range(Nt):
    b = u_n + dt * f(x, t[n + 1])
    b[0] = b[-1] = 0.0 # Dirichlet boundary conditions

    u[:] = spsolve(A, b)

    if user_action is not None:
        user_action(u, x, t, n + 1)

    u_n, u = u, u_n
```

**Advantages of Sparse Matrices**

- Reduced memory usage compared to dense matrices, as only the non-zero elements are stored.
- Faster computation, especially for large grids, as operations are performed only on non-zero elements.

### 1.4.3 The General $\theta$-Rule

The $\theta$-rule provides a framework for solving time-dependent partial differential equations (PDEs) by blending explicit and implicit schemes. For a general PDE of the form:

$$\frac{\partial u}{\partial t} = G(u),$$

the $\theta$-rule is expressed as:

$$u_i^{n+1} - u_i^n = \Delta t \left[ \theta G(u_i^{n+1}) + (1-\theta) G(u_i^n) \right],$$

where:
- $\theta$ controls the explicitness or implicitness of the scheme:

- $\theta = 0$: Forward Euler (explicit).
- $\theta = 1$: Backward Euler (implicit).
- $\theta = 0.5$: Crank-Nicolson (semi-implicit, second-order accurate in time).

**Applying the $\theta$-Rule to the Diffusion Equation** :
Consider the 1D diffusion equation:

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

where $a$ is the diffusion coefficient and $f(x,t)$ is a source term. Using the $\theta$-rule, the discretized equation becomes:

$$\frac{u_i^{n+1} - u_i^n}{\Delta t} = \theta a \frac{u_{i+1}^{n+1} - 2u_i^{n+1} + u_{i-1}^{n+1}}{\Delta x^2} + (1-\theta) a \frac{u_{i+1}^n - 2u_i^n + u_{i-1}^n}{\Delta x^2} + \theta f_i^{n+1} + (1-\theta) f_i^n.$$

Rearranging to isolate $u_i^{n+1}$, the scheme becomes:

$$u_i^{n+1} = u_i^n + \Delta t \left[ \theta a \frac{u_{i+1}^{n+1} - 2u_i^{n+1} + u_{i-1}^{n+1}}{\Delta x^2} + \theta f_i^{n+1} \right] + \Delta t \left[ (1-\theta) a \frac{u_{i+1}^n - 2u_i^n + u_{i-1}^n}{\Delta x^2} + (1-\theta) f_i^n \right].$$

**Code Implementation: $\theta$-Rule Diffusion Solver** :

The implementation of the $\theta$-rule for solving the 1D diffusion equation combines explicit and implicit contributions to provide a stable and flexible numerical scheme. Below, we explain the major steps and considerations in the implementation.

**Initialization and Setup** :

The solver begins by setting up the spatial and temporal grids, ensuring consistency between $\Delta x$ and $\Delta t$. The Fourier number $F = \frac{a\Delta t}{\Delta x^2}$ is a critical parameter that governs the numerical stability of the scheme:

```
# Number of time steps, rounded to ensure compatibility with dt
Nt = int(round(T / float(dt)))

# Create an array of time points from 0 to T with Nt+1 points
t = np.linspace(0, Nt * dt, Nt + 1) # Mesh points in time

# Calculate the spatial step size dx using the Fourier number F and dt
dx = np.sqrt(a * dt / F)

# Calculate the number of spatial grid points based on the domain length L and
    dx
Nx = int(round(L / dx))          # Number of spatial grid points

# Create an array of spatial points from 0 to L with Nx+1 points
x = np.linspace(0, L, Nx + 1)    # Mesh points in space

# Adjust dx to match the actual spacing between points in the x array
dx = x[1] - x[0]

# Adjust dt to match the actual spacing between points in the t array
dt = t[1] - t[0]
```

**Initial Condition** :

The initial condition $u(x, t = 0)$ is set using a user-defined function $I(x)$. This establishes the starting state of the system:

```
# Initialize arrays to store the solution at the current and previous time
    steps
u = np.zeros(Nx + 1)  # Array for the current solution
u_n = np.zeros(Nx + 1) # Array for the previous solution

# Set the initial condition based on the user-defined function I(x)
for i in range(Nx + 1): # Loop through all spatial points
       u_n[i] = I(x[i])   # Assign the initial displacement at each spatial
            point
```

If a `user_action` is provided, it is applied to the initial solution to enable custom processing, such as visualization or logging.

**Matrix Assembly** :

The matrix $A$, representing the implicit terms in the $\theta$-rule, is constructed as a sparse tridiagonal matrix for computational efficiency:

```
# Construct the sparse matrix A for the theta-rule scheme
diagonal = np.ones(Nx + 1) * (1 + 2 * theta * F) # Main diagonal with coefficients
lower = np.ones(Nx) * (-theta * F)           # Lower diagonal with coefficients
upper = np.ones(Nx) * (-theta * F)           # Upper diagonal with coefficients

# Adjust the boundary conditions for Dirichlet boundaries
```

```
            diagonal[0] = diagonal[-1] = 1
            lower[-1] = 0
            upper[0] = 0

            # Construct the sparse matrix A in compressed sparse row (CSR) format for efficiency
            A = diags([lower, diagonal, upper], offsets=[-1, 0, 1], format="csr")
```

The matrix $A$ captures the implicit contributions of the scheme:

$$A_{i,i-1} = -\theta F, \quad A_{i,i} = 1 + 2\theta F, \quad A_{i,i+1} = -\theta F.$$

The boundary conditions are incorporated by adjusting the diagonal and neighboring entries to enforce Dirichlet conditions ($u_0 = u_N = 0$).

**Time-Stepping Loop** :
At each time step, the right-hand side vector $\mathbf{b}$ is computed:

```
            for n in range(Nt): # Loop over all time steps
                # Initialize the right-hand side vector b with the current solution
                b = u_n.copy()

                # Add the contributions from the explicit terms (1 - theta)F for the spatial
                    derivative
                b[1:Nx] += (1 - theta) * F * (u_n[0:Nx-1] - 2 * u_n[1:Nx] + u_n[2:Nx+1])

                # Add the contributions from the source term f(x, t) weighted by theta
                # (1 - theta) accounts for the explicit part and theta for the implicit part
                b += dt * ((1 - theta) * f(x, t[n]) + theta * f(x, t[n + 1]))

                # Apply Dirichlet boundary conditions by fixing boundary values to 0
                b[0] = b[-1] = 0
```

**Solving the Linear System** :

```
                    # Solve the linear system
                    u[:] = spsolve(A, b)
```

**Updating and Finalizing** :
After solving for $\mathbf{u}^n$, the solution is updated for the next iteration:

```
                    u_n[:] = u
```

The loop continues until the final time $T$ is reached. The function optionally applies `user_action` at each time step to facilitate visualization or additional computations.

**Output** :
The solver returns the final solution $\mathbf{u}$, the spatial grid $x$, the temporal grid $t$, and the total CPU time. This provides a comprehensive result for analysis or further processing:

```
                    return u, x, t, t1 - t0
```

## 1.5 GitHub Repository

Follow the link to the code repository: Diffusion PDE Solver and numerical examples)

## 1.6 Analysis of Diffusion Equation

### 1.6.1 Analyzing Fourier Components in Numerical Schemes

**Wave Representation of Solutions** The solution to the diffusion equation can be expressed as a superposition of wave components in the form:

$$u(x,t) = \sum_{k \in K} b_k e^{-\alpha k^2 t} e^{ikx},$$

where:

- $b_k$: Amplitude of the wave component with wavenumber $k$.
- $e^{ikx}$: Represents a spatial oscillation with wavenumber $k$.
- $e^{-\alpha k^2 t}$: Damping factor, responsible for the decay of wave amplitude over time.

**Initial Conditions and Fourier Representation** The initial condition $I(x)$ can also be represented as a Fourier series:

$$I(x) = \sum_{k \in K} b_k e^{ikx},$$

where the Fourier coefficients $b_k$ are calculated to best approximate $I(x)$. This involves:

- **Fourier Analysis:** Decomposing $I(x)$ into a sum of wave components.

**Numerical Schemes and Wave Components** For a single wave component of the form:

$$e^{-\alpha k^2 t} e^{ikx},$$

the numerical scheme approximates the damping factor $e^{-\alpha k^2 t}$ with an amplification factor $A$.

The exact amplification factor is given by:

$$A_e = e^{-\alpha k^2 \Delta t}.$$

However, the numerical amplification factor $A$ may deviate from $A_e$, leading to errors in wave behavior.

### 1.6.2 Analysis of Finite Difference Schemes for the Diffusion Equation

Finite difference schemes for the diffusion equation:

$$u_t = \alpha u_{xx}.$$

The focus is on the behavior of wave components, addressing stability, accuracy, and truncation error.

**Wave Components and Amplification Factor** Solutions to the diffusion equation are represented as:

$$u(x,t) = e^{-\alpha k^2 t} e^{ikx},$$

where:

- $e^{ikx}$: Spatial oscillation (wave).
- $e^{-\alpha k^2 t}$: Exponential decay of wave amplitude.

Numerical solutions take the form:

$$u_q^n = A^n e^{ikq\Delta x} = A^n e^{ikx},$$

where $A^n$ is the amplification factor representing wave decay after $n$ steps. Substituting this form into the scheme provides $A$.

**Stability** The exact amplification factor is:

$$A_e = e^{-\alpha k^2 \Delta t},$$

ensuring $|A_e| < 1$.

Numerical stability requires:

$$|A| \leq 1.$$

- $|A| > 1$: Unstable growth.
- $1 - |A| < 0$: Non-physical oscillations.

**Accuracy** Accuracy is determined by how well $A$ approximates $A_e$:

$$A \approx A_e.$$

Small deviations indicate high accuracy. A Taylor expansion of $A/A_e$:

$$\frac{A}{A_e} = 1 + (\text{Error terms}),$$

shows errors in terms of $\Delta x, \Delta t, k$, with:

- First-order accuracy: Errors scale with $\Delta t$.
- Second-order accuracy: Errors scale with $\Delta x^2$.

### 1.6.3 Analysis of the Forward Euler scheme

To analyze the stability and accuracy of the Forward Euler scheme, we substitute a wave component of the form:

$$u_q^n = A^n e^{ikq\Delta x},$$

into the discretized equation:

$$u_q^{n+1} = u_q^n + F\left(u_{q+1}^n - 2u_q^n + u_{q-1}^n\right),$$

where:

- $A^n$: Amplitude of the wave at time step $n$.
- $e^{ikq\Delta x}$: Complex exponential representing a wave with wavenumber $k$ and spatial position $q\Delta x$.
- $F = \frac{\alpha \Delta t}{\Delta x^2}$: Fourier number, a key parameter governing stability and accuracy.

**Derivation of Amplification Factor** Substituting $u_q^n = A^n e^{ikq\Delta x}$ into the scheme, the left-hand side becomes:

$$u_q^{n+1} = A^{n+1} e^{ikq\Delta x}.$$

The right-hand side is:

$$u_q^n + F\left(u_{q+1}^n - 2u_q^n + u_{q-1}^n\right).$$

Substituting $u_q^n = A^n e^{ikq\Delta x}$, we get:

$$u_q^n = A^n e^{ikq\Delta x},$$

$$u_{q+1}^n = A^n e^{ik(q+1)\Delta x} = A^n e^{ikq\Delta x} e^{ik\Delta x},$$

$$u_{q-1}^n = A^n e^{ik(q-1)\Delta x} = A^n e^{ikq\Delta x} e^{-ik\Delta x}.$$

Thus, the right-hand side becomes:

$$A^n e^{ikq\Delta x} + F\left(A^n e^{ikq\Delta x} e^{ik\Delta x} - 2A^n e^{ikq\Delta x} + A^n e^{ikq\Delta x} e^{-ik\Delta x}\right).$$

**Factoring Out Common Terms** Factor $A^n e^{ikq\Delta x}$ from all terms:

$$A^n e^{ikq\Delta x}\left[1 + F\left(e^{ik\Delta x} - 2 + e^{-ik\Delta x}\right)\right].$$

The left-hand side $u_q^{n+1}$ is:

$$A^{n+1} e^{ikq\Delta x}.$$

Equating both sides:

$$A^{n+1} e^{ikq\Delta x} = A^n e^{ikq\Delta x}\left[1 + F\left(e^{ik\Delta x} - 2 + e^{-ik\Delta x}\right)\right].$$

Cancel $e^{ikq\Delta x}$ from both sides:

$$A^{n+1} = A^n\left[1 + F\left(e^{ik\Delta x} - 2 + e^{-ik\Delta x}\right)\right].$$

Define the amplification factor $A = \frac{A^{n+1}}{A^n}$:

$$A = 1 + F\left(e^{ik\Delta x} - 2 + e^{-ik\Delta x}\right).$$

**Simplification of Exponentials** The term $e^{ik\Delta x} + e^{-ik\Delta x}$ simplifies using the Euler formula:

$$e^{ik\Delta x} + e^{-ik\Delta x} = 2\cos(k\Delta x).$$

Thus:

$$A = 1 + F\left(2\cos(k\Delta x) - 2\right).$$

Factor out $2F$:

$$A = 1 - 2F\left(1 - \cos(k\Delta x)\right).$$

Using the trigonometric identity $1 - \cos(x) = 2\sin^2\left(\frac{x}{2}\right)$:

$$A = 1 - 4F\sin^2\left(\frac{k\Delta x}{2}\right).$$

**Final Expression for $A$** The amplification factor is:

$$A = 1 - 4F\sin^2\left(\frac{k\Delta x}{2}\right).$$

The **numerical solution** for the Forward Euler scheme, incorporating the amplification factor $A$, is expressed as:

$$u_q^n = A^n e^{ikq\Delta x} = \left[1 - 4F\sin^2\left(\frac{k\Delta x}{2}\right)\right]^n e^{ikq\Delta x}.$$

**Stability Condition** For stability, we require $|A| \leq 1$. The critical case occurs when $\sin^2\left(\frac{k\Delta x}{2}\right) = 1$, leading to the condition:

$$4F \leq 1 \quad \Rightarrow \quad F \leq \frac{1}{4}.$$

Thus, the time step must satisfy:

$$\Delta t \leq \frac{\Delta x^2}{4\alpha}.$$

This detailed derivation highlights the role of Fourier analysis in understanding the behavior of the Forward Euler scheme and deriving its stability and accuracy properties.

**Accuracy** The exact solution decays exponentially, with an amplification factor given by:

$$A_e = e^{-4Fp^2},$$

where $F = \frac{\alpha\Delta t}{\Delta x^2}$ is the Fourier number and $p = \frac{k\Delta x}{2}$ is a scaled wavenumber. To analyze the accuracy of the Forward Euler scheme, we compare the numerical amplification factor $A$ with $A_e$ by deriving their ratio $\frac{A}{A_e}$.

**Step 1:** Expand $A_e$ Using Taylor Series
The exponential $A_e = e^{-4Fp^2}$ can be expanded using the Taylor series for small $F$:

$$A_e = 1 - 4Fp^2 + \frac{(4Fp^2)^2}{2!} - \frac{(4Fp^2)^3}{3!} + \cdots.$$

**Step 2:** Write $\frac{A}{A_e}$
The ratio $\frac{A}{A_e}$ is expressed as:

$$\frac{A}{A_e} = \frac{1 - 4F\sin^2(p)}{1 - 4Fp^2 + \frac{(4Fp^2)^2}{2!} - \frac{(4Fp^2)^3}{3!} + \cdots}.$$

**Step 3:** Approximate the Denominator
For small $F$, the denominator $A_e$ can be approximated using the **binomial expansion**:

$$\frac{1}{A_e} \approx 1 + 4Fp^2 + \frac{(4Fp^2)^2}{2} + \cdots .$$

**Step 4:** Expand the Ratio
Substituting $\frac{1}{A_e}$ into $\frac{A}{A_e}$, we obtain:

$$\frac{A}{A_e} = (1 - 4F\sin^2(p))\left(1 + 4Fp^2 + \frac{(4Fp^2)^2}{2} + \cdots\right).$$

Expanding the product and keeping terms up to second order in $F$, we have:

$$\frac{A}{A_e} = 1 - 4F\sin^2(p) + 4Fp^2 - (4F\sin^2(p))(4Fp^2) + \cdots .$$

Simplifying:

$$\frac{A}{A_e} = 1 - 4F\sin^2(p) + 4Fp^2 - 16F^2p^2\sin^2(p) + \cdots .$$

This shows the leading error terms in the ratio, revealing how the numerical scheme approximates the exact decay behavior. For small Fourier numbers $F$ and small $p$, the dominant error term is:

$$\mathcal{O}(F) = \mathcal{O}\left(\frac{\alpha\Delta t}{\Delta x^2}\right).$$

This analysis reveals that the Forward Euler scheme is first-order accurate in time, as the error scales with $\Delta t$, and second-order accurate in space, as the error scales with $\Delta x^2$.

### 1.6.4 Analysis of the Backward Euler Scheme

The Backward Euler scheme is an implicit numerical method for solving the diffusion equation, making it unconditionally stable. Unlike the Forward Euler scheme, which explicitly computes the next time step using current values, Backward Euler requires solving a system of equations for each time step. This stability makes it particularly suitable for diffusion problems.

The Backward Euler scheme checks if these wave components satisfy the discrete numerical scheme:

$$[D_t u = \alpha D_x D_x u]_q^n.$$

By substituting the wave form $u_q^n = A^n e^{ikq\Delta x}$ into the scheme, the amplification factor $A$, which describes the decay of the wave amplitude over time, is derived.

**Derivation of Amplification Factor** :

**1. Wave Component Assumption** :
Assume a wave component solution of the form:

$$u_i^n = A^n e^{ikx_i} = A^n e^{iki\Delta x}.$$

At time $n-1$:

$$u_i^{n-1} = A^{n-1} e^{ikx_i}.$$

For spatial points:

$$u_{i+1}^n = A^n e^{ik(i+1)\Delta x}, \quad u_{i-1}^n = A^n e^{ik(i-1)\Delta x}, \quad u_i^n = A^n e^{iki\Delta x}.$$

Substitute these into the Backward Euler equation:

$$-FA^n e^{ik(i-1)\Delta x} + (1+2F)A^n e^{iki\Delta x} - FA^n e^{ik(i+1)\Delta x} = A^{n-1} e^{iki\Delta x}.$$

Factor out $A^n e^{iki\Delta x}$:

$$A^n e^{iki\Delta x}\left[-Fe^{-ik\Delta x} + (1+2F) - Fe^{ik\Delta x}\right] = A^{n-1} e^{iki\Delta x}.$$

Cancel $e^{iki\Delta x}$:

$$A^n\left[-Fe^{-ik\Delta x} + (1+2F) - Fe^{ik\Delta x}\right] = A^{n-1}.$$

**2. Simplify the Exponential Terms** :
Use the identity:

$$e^{ik\Delta x} + e^{-ik\Delta x} = 2\cos(k\Delta x),$$

to rewrite:

$$A^n\left[(1+2F) - 2F\cos(k\Delta x)\right] = A^{n-1}.$$

Reorganize:

$$A = \frac{1}{1+2F(1-\cos(k\Delta x))}.$$

**3. Simplify Using Trigonometric Identities** :
Using:

$$1-\cos(k\Delta x) = 2\sin^2\left(\frac{k\Delta x}{2}\right),$$

the amplification factor becomes:

$$A = \frac{1}{1+4F\sin^2\left(\frac{k\Delta x}{2}\right)}.$$

**4. Final Result** :
The amplification factor for the Backward Euler scheme is:

$$A = \frac{1}{1+4F\sin^2\left(\frac{k\Delta x}{2}\right)}.$$

**Stability Analysis** :

From the expression for $A$, we see that $0 < A < 1$ for any $\Delta t > 0$ and any wavenumber $k$, since the denominator in $A$ is always $> 1$. This ensures that:

- All numerical wave components are stable, with amplitudes decaying over time.
- Since $A > 0$, the scheme avoids oscillations, ensuring monotonic decay of amplitudes.

**Numerical Solution** :

The numerical solution for a single wave component is:

$$u_q^n = A^n e^{ikq\Delta x} = \left[\frac{1}{1 + 4F\sin^2(p)}\right]^n e^{ikq\Delta x}$$

This represents a stable and non-oscillatory decay of the wave amplitude with time.

## 1.7 Analysis of Amplification Factor Accuracy for Numerical Schemes

This section analyzes the accuracy of amplification factors for described numerical schemes solving the diffusion equation, focusing on their behavior for different Fourier numbers ($F = \alpha\Delta t/\Delta x^2$).

### 1.7.1 Insights from Figure 1

- Figure 1 shows $A$ for different $F$ values from 0.01 to 20.
- At $F = 20$:
  - Long waves (low $p$) are weakly damped.
  - Short waves (high $p$) dominate, leading to oscillations.
- At $F = 2$:
  - Short waves are insufficiently damped, leaving residual noise.
- At $F = 0.01$:
  - All waves are effectively damped, ensuring a stable and accurate solution.

This analysis demonstrates the importance of carefully selecting $F$ to balance stability and accuracy, particularly when short-wave components are significant, as illustrated in Figure 1.

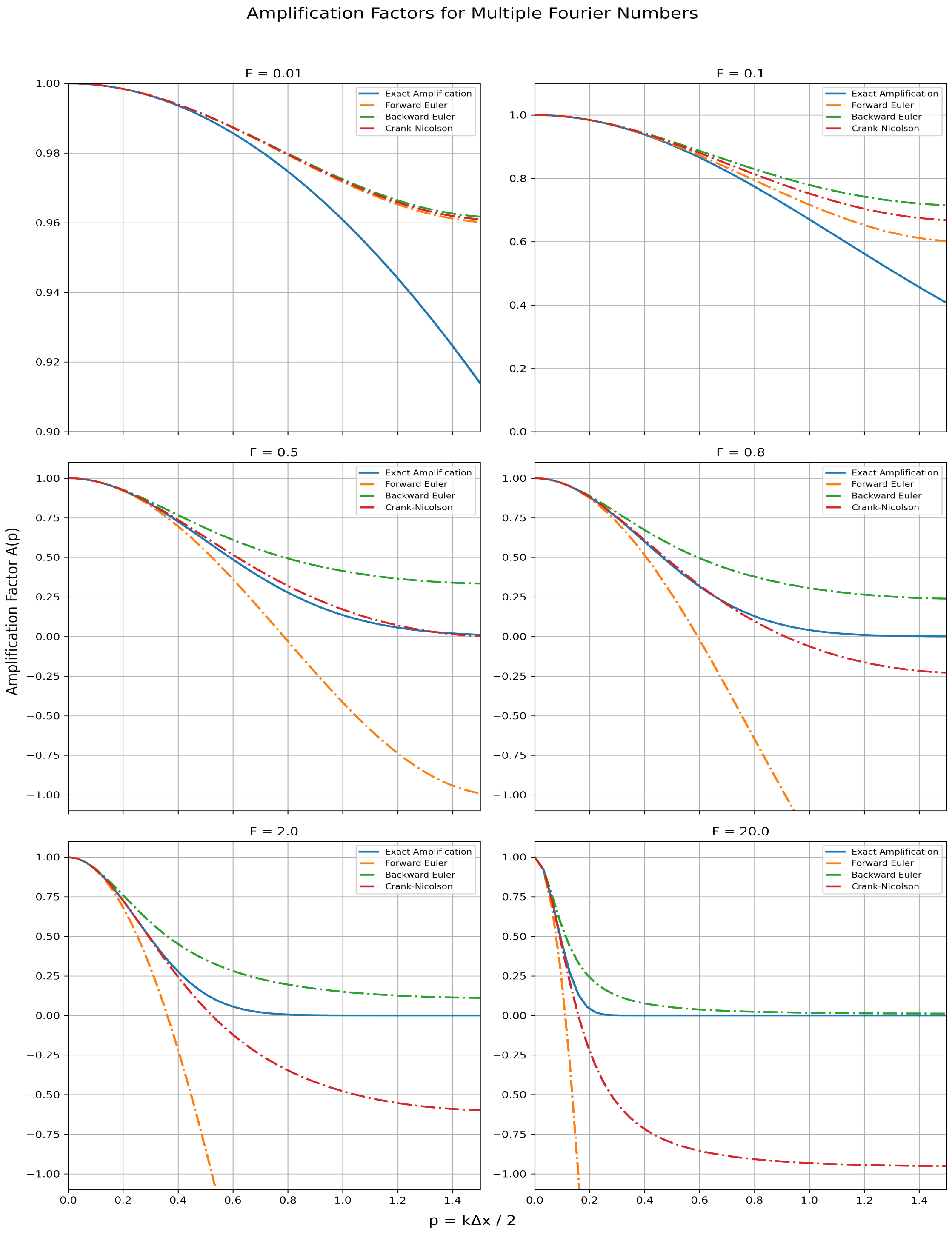

Figure 1: Amplification factors for different F and numerical schemes

# 2 2D Diffusion Solver Development

Mathematical formulation the diffusion equation in two spatial dimensions can be expressed as:

$$\frac{\partial u}{\partial t} = \alpha \left( \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right) + f(x, y),$$

in a rectangular domain $(x, y) \in (0, L_x) \times (0, L_y),\ t \in (0, T]$.

The boundary conditions are given as:

$$u = 0 \quad \text{on the boundary},$$

and the initial condition is specified as:

$$u(x, y, 0) = I(x, y).$$

## 2.1 The Unified $\theta$-Rule Discretization

To solve the two-dimensional diffusion equation numerically, a general $\theta$-rule for time discretization is implemented. For the spatial derivatives, standard second-order accurate finite differences are used.

$$[D_t u]^{n+\frac{1}{2}} = \theta \left[\alpha \left(D_x D_x u + D_y D_y u\right) + f\right]^{n+1} + (1-\theta) \left[\alpha \left(D_x D_x u + D_y D_y u\right) + f\right]^{n}.$$

Written explicitly, this becomes:

$$\frac{u_{i,j}^{n+1} - u_{i,j}^{n}}{\Delta t} = \theta\alpha \left( \frac{u_{i-1,j}^{n+1} - 2u_{i,j}^{n+1} + u_{i+1,j}^{n+1}}{\Delta x^2} + \frac{u_{i,j-1}^{n+1} - 2u_{i,j}^{n+1} + u_{i,j+1}^{n+1}}{\Delta y^2} \right) + \theta f_{i,j}^{n+1}$$

$$+(1-\theta)\alpha \left( \frac{u_{i-1,j}^{n} - 2u_{i,j}^{n} + u_{i+1,j}^{n}}{\Delta x^2} + \frac{u_{i,j-1}^{n} - 2u_{i,j}^{n} + u_{i,j+1}^{n}}{\Delta y^2} \right) + (1-\theta) f_{i,j}^{n}.$$

We rearrange to collect the unknowns on the left-hand side:

$$u_{i,j}^{n+1} - \theta F_x \left(u_{i-1,j}^{n+1} - 2u_{i,j}^{n+1} + u_{i+1,j}^{n+1}\right) - \theta F_y \left(u_{i,j-1}^{n+1} - 2u_{i,j}^{n+1} + u_{i,j+1}^{n+1}\right)$$

$$= (1-\theta) F_x \left(u_{i-1,j}^{n} - 2u_{i,j}^{n} + u_{i+1,j}^{n}\right) + (1-\theta) F_y \left(u_{i,j-1}^{n} - 2u_{i,j}^{n} + u_{i,j+1}^{n}\right) + \Delta t \theta f_{i,j}^{n+1} + \Delta t (1-\theta) f_{i,j}^{n} + u_{i,j}^{n},$$

where:

$$F_x = \frac{\alpha \Delta t}{\Delta x^2}, \quad F_y = \frac{\alpha \Delta t}{\Delta y^2},$$

are the Fourier numbers in the $x$- and $y$-directions, respectively.

## 2.2 Construction of the System Matrix

The equations for the two-dimensional diffusion problem are coupled at the new time level $n + 1$, requiring the solution of a system of linear equations $A\mathbf{c} = \mathbf{b}$, where $A$ is the coefficient matrix, $\mathbf{c}$ is the vector of unknowns, and $\mathbf{b}$ is the right-hand side.

**Boundary Conditions** For a mesh with $N_x = 3$ and $N_y = 2$, the boundary equations are:

$$u_{i,j}^{n+1} = 0, \quad \text{for } i = 0, 1, 2, 3 \text{ and } j = 0, 2.$$

This leaves two interior points: $(i, j) = (1, 1)$ and $(2, 1)$.

**Interior Equations** The equations at interior points are:

$$u_{i,j}^{n+1} - \theta F_x \left(u_{i-1,j}^{n+1} - 2u_{i,j}^{n+1} + u_{i+1,j}^{n+1}\right) - \theta F_y \left(u_{i,j-1}^{n+1} - 2u_{i,j}^{n+1} + u_{i,j+1}^{n+1}\right)$$

$$= (1-\theta)F_x \left(u_{i-1,j}^{n} - 2u_{i,j}^{n} + u_{i+1,j}^{n}\right) + (1-\theta)F_y \left(u_{i,j-1}^{n} - 2u_{i,j}^{n} + u_{i,j+1}^{n}\right) + \Delta t\theta f_{i,j}^{n+1} + \Delta t(1-\theta) f_{i,j}^{n} + u_{i,j}^{n}.$$

**Coefficient Matrix Structure** The entries of the coefficient matrix $A$ for interior points are:

$$A_{m(i,j),m(i,j)} = 1 + \theta(F_x + F_y), \quad A_{m(i,j),m(i-1,j)} = A_{m(i,j),m(i+1,j)} = -\theta F_x,$$

$$A_{m(i,j),m(i,j-1)} = A_{m(i,j),m(i,j+1)} = -\theta F_y,$$

with boundary entries:

$$A_{m(i,j),m(i,j)} = 1, \quad \text{for boundary points.}$$

The right-hand side vector $\mathbf{b}$ includes contributions from the previous time step and the source term:

$$b_{m(i,j)} = u_{i,j}^{n} + (1-\theta) \left[F_x(u_{i-1,j}^{n} - 2u_{i,j}^{n} + u_{i+1,j}^{n}) + F_y(u_{i,j-1}^{n} - 2u_{i,j}^{n} + u_{i,j+1}^{n})\right] + \Delta t\theta f_{i,j}^{n+1} + \Delta t(1-\theta) f_{i,j}^{n}.$$

**Sparsity of the Matrix** The resulting matrix $A$ is sparse, meaning that most of its elements are zero. The nonzero elements are arranged in a banded structure, confined to the main diagonal and a few adjacent diagonals. This banded sparsity significantly reduces the storage requirements and computational effort, as only the nonzero elements need to be considered during calculations. For a mesh with $N_x + 1 = N_y + 1 = N$, the fraction of nonzero elements is $\frac{5}{N^2}$, demonstrating the efficiency of sparse matrix methods for solving such systems.

## 2.3 Algorithm for the Coefficient Matrix

### General Approach

- Initialize $A$ as a zero matrix and $b$ as a zero vector.
- Loop through all grid points $(i, j)$, where:
  - $i = 0, \ldots, N_x$: Points in the $x$-direction.
  - $j = 0, \ldots, N_y$: Points in the $y$-direction.
- For each grid point $(i, j)$, compute the corresponding linear index $p = j(N_x + 1) + i$.
- Determine whether $(i, j)$ is a boundary or interior point and update $A$ and $b$ accordingly.

To construct the coefficient matrix $A$ and right-hand side vector $b$, the following algorithm is applied:

**Algorithm 1** Construction of Coefficient Matrix $A$ and Right-Hand Side $b$ for 2D Diffusion Equation

```
1: Initialize matrix $A$ and vector $b$ with zeros.
2: for $i = 0$ to $Nx$ do
3:     for $j = 0$ to $Ny$ do
4:         $p \leftarrow j \cdot (Nx + 1) + i$                          ▷ Compute the linear index
5:         if Point $(i, j)$ is on the boundary then
6:             $A_{p,p} \leftarrow 1$
7:             $b_p \leftarrow 0$
8:         else
9:                                                     ▷ Fill coefficients for interior points
10:            $A_{p,m(i-1,j)} \leftarrow F_x$
11:            $A_{p,m(i+1,j)} \leftarrow F_x$
12:            $A_{p,m(i,j-1)} \leftarrow F_y$
13:            $A_{p,m(i,j+1)} \leftarrow F_y$
14:            $A_{p,p} \leftarrow 1 + F_x + F_y$
15:            Compute $b_p$ based on the $\theta$-rule
16:        end if
17:    end for
18: end for
19: for $j = 0$ to $Ny$ do              ▷ Handle boundary lines and interior points separately
20:                                                     ▷ Boundary: $j = 0$ (bottom boundary)
21:    for $i = 0$ to $Nx$ do
22:        $p \leftarrow j \cdot (Nx + 1) + i$
23:        $A_{p,p} \leftarrow 1$
24:    end for
25:                                                     ▷ Interior lines: $1 \leq j < Ny$
26:    for $i = 1$ to $Nx - 1$ do
27:        $p \leftarrow j \cdot (Nx + 1) + i$
28:        Fill $A_{p,m(i-1,j)}$, $A_{p,m(i+1,j)}$, $A_{p,m(i,j-1)}$, $A_{p,m(i,j+1)}$, $A_{p,p}$, and $b_p$
29:    end for
30:                                                     ▷ Boundary: $j = Ny$ (top boundary)
31:    for $i = 0$ to $Nx$ do
32:        $p \leftarrow j \cdot (Nx + 1) + i$
33:        $A_{p,p} \leftarrow 1$
34:    end for
35: end for
36: for $i = 0$ to $Nx$ do                                  ▷ Handle right-hand side vector $b$
37:                                                     ▷ Boundary: $j = 0$ (bottom boundary)
38:    for $j = 0$ to $Ny$ do
39:        $p \leftarrow j \cdot (Nx + 1) + i$
40:        $b_p \leftarrow 0$
41:    end for
42:                                                     ▷ Interior points: $1 \leq i < Nx$
43:    for $j = 1$ to $Ny - 1$ do
44:        $p \leftarrow j \cdot (Nx + 1) + i$
45:        Compute $b_p$
46:    end for
47:                                                     ▷ Boundary: $j = Ny$ (top boundary)
48:    for $j = 0$ to $Ny$ do
49:        $p \leftarrow j \cdot (Nx + 1) + i$
50:        $b_p \leftarrow 0$
51:    end for
52: end for
```

## 2.4 Implementation in Python

The function `theta_diffusion_2D` solves the 2D diffusion equation using the theta-rule method. The algorithm discretizes the spatial and temporal domains, assembles a coefficient matrix and right-hand side vector, and iteratively solves the system of equations at each time step.

### 2.4.1 Function Signature and Parameters

```
def theta_diffusion_2D(
        I, f, alpha, Lx, Ly, Nx, Ny, dt, T, theta=0.5,
        U_0x=0, U_0y=0, U_Lx=0, U_Ly=0, user_action=None
        ):
```

The solver takes the following arguments:

- $I$: Initial condition function $I(x, y)$.
- $f$: Source term function $f(x, y, t)$.
- $\alpha$: Diffusion coefficient.
- $L_x, L_y$: Dimensions of the spatial domain.
- $N_x, N_y$: Number of spatial grid points in the $x$ and $y$ directions.
- $dt$: Time step size.
- $T$: Total simulation time.
- $\theta$: Weighting parameter controlling the scheme:
  - $\theta = 0$: Explicit (Forward Euler).
  - $\theta = 0.5$: Semi-Implicit (Crank-Nicolson).
  - $\theta = 1$: Fully implicit (Backward Euler).
- $U_{0x}, U_{0y}, U_{Lx}, U_{Ly}$: Boundary conditions.
- `user_action`: Optional function for visualization or output.

—

### 2.4.2 Setup and Initialization

```
import time
t0 = time.time() # Start measuring CPU time

x = np.linspace(0, Lx, Nx + 1) # Mesh points in x direction
y = np.linspace(0, Ly, Ny + 1) # Mesh points in y direction
dx = x[1] - x[0]
dy = y[1] - y[0]

Nt = int(round(T / dt)) # Number of time steps
t = np.linspace(0, Nt * dt, Nt + 1) # Time array
```

- Initializes the computational grid in space and time.
- $dx$ and $dy$ are the grid spacings in $x$ and $y$ directions:

$$dx = \frac{L_x}{N_x}, \quad dy = \frac{L_y}{N_y}.$$

- $Nt$ is the total number of time steps:

$$Nt = \frac{T}{dt}.$$

- $t$ is the time array containing all time steps.

—

### 2.4.3 Fourier Numbers and Source Term Handling

```
Fx = alpha * dt / dx**2
Fy = alpha * dt / dy**2

if f is None or f == 0:
    f = lambda x, y, t: 0
```

- $F_x$ and $F_y$ are the Fourier numbers in $x$ and $y$ directions:

$$F_x = \frac{\alpha \Delta t}{\Delta x^2}, \quad F_y = \frac{\alpha \Delta t}{\Delta y^2}.$$

- If no source term $f$ is provided, it defaults to zero everywhere.

—

### 2.4.4 Initializing Solution Arrays

```
u = np.zeros((Nx + 1, Ny + 1)) # Solution at the new time level
u_n = np.zeros((Nx + 1, Ny + 1)) # Solution at the previous time level

Ix = range(0, Nx + 1)
Iy = range(0, Ny + 1)
It = range(0, Nt + 1)
```

- $u$ and $u_n$ store the solution at the current and previous time steps, respectively.
- $Ix$, $Iy$, and $It$ define the index ranges for grid points in $x$, $y$, and $t$.

— —

### 2.4.5 Initial Condition Setup

```
# Load initial condition into u_n
for i in Ix:
    for j in Iy:
        u_n[i, j] = I(x[i], y[j])
```

—

### 2.4.6 Matrix and RHS Vector Initialization

```
N = (Nx + 1) * (Ny + 1) # Total number of unknowns
A = np.zeros((N, N)) # Coefficient matrix
b = np.zeros(N) # Right-hand side vector

m = lambda i, j: j * (Nx + 1) + i # Linear index for (i, j)
```

**Explanation:**

- Initializes the coefficient matrix $A$ and RHS vector $\mathbf{b}$ for the linear system $A\mathbf{c} = \mathbf{b}$.
- The function $m(i, j)$ maps a 2D grid point $(i, j)$ to a 1D index $p$.

—

### 2.4.7 Matrix Assembly

```
# Boundary conditions and internal points
for j in Iy:
    for i in Ix:
        p = m(i, j) # Linear index for the current point
        if j == 0 or j == Ny or i == 0 or i == Nx: # Boundary points
            A[p, p] = 1
        else: # Internal points
            A[p, m(i, j - 1)] = -theta * Fy # y-direction lower
            A[p, m(i - 1, j)] = -theta * Fx # x-direction lower
            A[p, p] = 1 + 2 * theta * (Fx + Fy) # Center
            A[p, m(i + 1, j)] = -theta * Fx # x-direction upper
            A[p, m(i, j + 1)] = -theta * Fy # y-direction upper
```

**Explanation:**

- Boundary points enforce $A_{p,p} = 1$.
- Interior points use the finite difference stencil, as detailed earlier.

—

### 2.4.8 Time-Stepping Loop

```
                # Time-stepping loop
                for n in It[:-1]:
                    # Compute the right-hand side
                    for j in Iy:
                        for i in Ix:
                            p = m(i, j)
                            if j == 0:
                                b[p] = U_0y(t[n + 1]) # Bottom boundary
                            elif j == Ny:
                                b[p] = U_Ly(t[n + 1]) # Top boundary
                            elif i == 0:
                                b[p] = U_0x(t[n + 1]) # Left boundary
                            elif i == Nx:
                                b[p] = U_Lx(t[n + 1]) # Right boundary
                            else:
                                b[p] = (
                                u_n[i, j]
                                + (1 - theta)
                                * (
                                Fx * (u_n[i + 1, j] - 2 * u_n[i, j] + u_n[i - 1, j
                                    ])
                                + Fy * (u_n[i, j + 1] - 2 * u_n[i, j] + u_n[i, j -
                                     1])
                                )
                                + theta * dt * f(i * dx, j * dy, (n + 1) * dt)
                                + (1 - theta) * dt * f(i * dx, j * dy, n * dt)
                                )

                            # Solve the linear system
                            c = scipy.linalg.solve(A, b)

                            # Fill u with the solution vector c
                            for i in Ix:
                                for j in Iy:
                                    u[i, j] = c[m(i, j)]

                            if user_action is not None:
                                user_action(u, x, xv, y, yv, t, n + 1)

                            # Update u_n for the next time step
                            u_n, u = u, u_n
```

**Linear System Solution:** The linear system $A\mathbf{c} = \mathbf{b}$ is solved using `scipy.linalg.solve`, where:

- $A$ is the coefficient matrix representing the finite difference scheme.
- $\mathbf{b}$ is the right-hand side vector computed in the previous step.
- $\mathbf{c}$ is the solution vector, which is reshaped into the 2D solution array $u[i, j]$.

**Solution Assignment:** The vector $\mathbf{c}$ is mapped back into the 2D array $u[i, j]$ using the inverse of the linear index $p = j \cdot (Nx + 1) + i$.

**User-Defined Actions:** If a `user_action` function is provided, it is called with the updated solution $u$ and relevant parameters. This can be used for visualization, saving data, or other purposes.

**Updating for the Next Time Step:** The values of $u_n$ are updated to the current solution $u$, preparing for the next iteration. This corresponds to the time-stepping rule:

$$u_n = u.$$

This concludes one iteration of the time-stepping loop, and the process repeats until the final time step is reached.

—

#### 2.4.9 Finalize and Return

```
t1 = time.time()
return t, t1 - t0
```

- The simulation time is measured by calculating the difference between `t1` (end time) and `t0` (start time).
- Returns the time array `t` and the total CPU time.

—

### 2.5 Verification Using the `quadratic_solvers_3` Function

This subsection demonstrates the usage of the function `quadratic_solvers_3`, which tests the solver for the 2D diffusion equation against an exact solution. The exact solution is quadratic in space and linear in time. The function verifies that the numerical solution matches the exact solution to within a specified tolerance.

```
def quadratic_solvers_3(theta, Nx, Ny):
    """Exact discrete solution of the scheme."""
    def u_exact(x, y, t):
        """Exact solution."""
        return 5 * t * x * (Lx - x) * y * (Ly - y)

    def I(x, y):
        """Initial condition."""
        return u_exact(x, y, 0)

    def f(x, y, t):
        """Source term."""
        return 5 * x * (Lx - x) * y * (Ly - y) + 10 * a * t * (y * (Ly -
            y) + x * (Lx - x))

    # Domain parameters
    Lx = 0.75
    Ly = 1.5
    a = 3.5
```

```
                                    dt = 0.5
                                    T = 2 # Total simulation time

                                    def assert_no_error(u, x, xv, y, yv, t, n):
                                            """Assert zero error at all mesh points."""
                                            xv, yv = np.meshgrid(x, y, indexing='ij')
                                            u_e = u_exact(xv, yv, t[n])
                                            diff = abs(u - u_e).max()
                                            tol = 1E-12
                                            msg = f"diff={diff:.6e}, step {n}, time={t[n]}"
                                            print(msg)
                                            assert diff < tol, msg

                                    # Test the solver with `theta_diffusion_2D`
                                    print(f"\nTesting theta={theta}, Nx={Nx}, Ny={Ny}")
                                            t, cpu = theta_diffusion_2D(
                                            I=I,
                                            f=f,
                                            alpha=a,
                                            Lx=Lx,
                                            Ly=Ly,
                                            Nx=Nx,
                                            Ny=Ny,
                                            T=T,
                                            dt=dt,
                                            theta=theta,
                                            U_0x=0,    # Explicit scalar boundary condition
                                            U_0y=0,    # Explicit scalar boundary condition
                                            U_Lx=0,    # Explicit scalar boundary condition
                                            U_Ly=0,    # Explicit scalar boundary condition
                                            user_action=assert_no_error,
                                            )

                                    print(f"Test completed for theta={theta}, Nx={Nx}, Ny={Ny}.")
                                    return t, cpu
```

### 2.5.1 Mathematical Representation and Explanation

The `quadratic_solvers_3` function verifies the numerical solution by comparing it to the exact solution at every time step. Here are the key components of the function:

**Exact Solution.** The exact solution is given by:

$$u(x, y, t) = 5t \cdot x(L_x - x) \cdot y(L_y - y),$$

where $L_x$ and $L_y$ define the domain dimensions. This solution is linear in time and quadratic in space.

**Source Term.** The source term $f(x, y, t)$ is derived by substituting the exact solution into the diffusion equation:

$$f(x, y, t) = 5x(L_x - x)y(L_y - y) + 10\alpha t \cdot [x(L_x - x) + y(L_y - y)] .$$

**Initial Condition.** The initial condition corresponds to the exact solution evaluated at $t = 0$:

$$I(x, y) = u(x, y, 0) = 0.$$

**Domain Parameters.** The simulation runs on a domain with:

$$L_x = 0.75, \qquad L_y = 1.5, \qquad \alpha = 3.5.$$

The total simulation time is $T = 2$, with a time step $\Delta t = 0.5$.

**Numerical Verification.** The function `assert_no_error` compares the numerical solution $u$ with the exact solution $u_e$ at every time step:

$$\text{diff} = \max |u - u_e|, \quad \text{tol} = 10^{-12}.$$

If the maximum error exceeds the tolerance, an exception is raised.

**Boundary Conditions.** Explicit scalar boundary conditions are applied:

$$U_{0x} = U_{0y} = U_{Lx} = U_{Ly} = 0.$$

### 2.5.2 Testing the `quadratic_solvers_3` Function

To ensure the robustness of the solver, the `test_quadratic_solvers_3` function runs multiple tests on the solver with varying grid sizes ($N_x$ and $N_y$) and theta values.

```python
def test_quadratic_solvers_3():
        """Test quadratic solution for various meshes and theta values."""
        for theta in [1, 0.5, 0]: # Backward Euler, Crank-Nicolson, Forward
            Euler
            for Nx in range(2, 6, 2):
                for Ny in range(2, 6, 2):
                print(f"\n*** Testing for {Nx}x{Ny} mesh with theta={
                    theta}")
                quadratic_solvers_3(theta, Nx, Ny)
```

**Description of the Test Function.** The `test_quadratic_solvers_3` function iterates through combinations of:

- **Theta values:** $\theta = 1$ (Backward Euler), $\theta = 0.5$ (Crank-Nicolson), and $\theta = 0$ (Forward Euler).
- **Grid sizes:** Mesh resolutions are varied with $N_x$ and $N_y$ taking values in the range $[2, 6]$ with a step of 2.

The `test_quadratic_solvers_3` function produces the following output when executed. Each block corresponds to a specific combination of mesh size $(N_x, N_y)$ and the $\theta$-value (1, 0.5, or 0). The outputs display the difference (`diff`) between the numerical solution and the exact solution at every time step.

```
*** Testing for 2x2 mesh with theta=1

Testing theta=1, Nx=2, Ny=2
diff=0.000000e+00, step 0, time=0.0
diff=1.297325e-16, step 1, time=0.5
diff=2.594651e-16, step 2, time=1.0
diff=4.989808e-16, step 3, time=1.5
diff=5.189301e-16, step 4, time=2.0
Test completed for theta=1, Nx=2, Ny=2.

*** Testing for 2x4 mesh with theta=1

Testing theta=1, Nx=2, Ny=4
diff=0.000000e+00, step 0, time=0.0
diff=2.359146e-16, step 1, time=0.5
diff=4.718293e-16, step 2, time=1.0
diff=1.104299e-15, step 3, time=1.5
diff=9.436586e-16, step 4, time=2.0
Test completed for theta=1, Nx=2, Ny=4.

...

*** Testing for 4x4 mesh with theta=0

Testing theta=0, Nx=4, Ny=4
diff=0.000000e+00, step 0, time=0.0
diff=0.000000e+00, step 1, time=0.5
diff=0.000000e+00, step 2, time=1.0
diff=0.000000e+00, step 3, time=1.5
diff=0.000000e+00, step 4, time=2.0
Test completed for theta=0, Nx=4, Ny=4.
PASSED
```

## Acknowledgments

I acknowledge the use of ChatGPT for writing assistance and code debugging in the preparation of this document.