# Numerical Methods for Temporal Flow Theory

## 1. Discretization Framework

### 1.1 Space-Time Grid

```

Grid Structure:

x\_i = iΔx, i = 0,1,...,Nx

t\_n = nΔt, n = 0,1,...,Nt

Field Variables:

W\_i^n = W(x\_i, t\_n)

ρ\_i^n = ρ(x\_i, t\_n)

Φ\_i^n = Φ(x\_i, t\_n)

Stability Condition:

CFL = |W\_max|Δt/Δx ≤ 1

```

### 1.2 Finite Difference Schemes

```

Temporal Derivatives:

∂W/∂t ≈ (W\_i^{n+1} - W\_i^n)/Δt [Forward Euler]

∂W/∂t ≈ (W\_i^{n+1} - W\_i^{n-1})/2Δt [Central]

Spatial Derivatives:

∂W/∂x ≈ (W\_{i+1}^n - W\_{i-1}^n)/2Δx

∂²W/∂x² ≈ (W\_{i+1}^n - 2W\_i^n + W\_{i-1}^n)/Δx²

```

## 2. Advanced Numerical Schemes

### 2.1 Runge-Kutta Methods

```

4th Order RK:

k1 = Δt·f(W^n)

k2 = Δt·f(W^n + k1/2)

k3 = Δt·f(W^n + k2/2)

k4 = Δt·f(W^n + k3)

W^{n+1} = W^n + (k1 + 2k2 + 2k3 + k4)/6

Error Estimate:

E\_RK4 ∝ O(Δt⁴)

```

### 2.2 Spectral Methods

```

Fourier Transform:

W(x,t) = ∑\_k W\_k(t)exp(ikx)

Evolution Equations:

dW\_k/dt = -ikc\_kW\_k - νk²W\_k

Implementation:

1. FFT spatial derivatives

2. Time-step in k-space

3. Inverse FFT for physical space

```

## 3. Adaptive Mesh Refinement

### 3.1 Grid Refinement Criteria

```

Refinement Indicator:

R\_i = |∇²W\_i|/|W\_i|

Refinement Rule:

if R\_i > R\_threshold:

Δx\_local = Δx/2

Δt\_local = Δt/2

Grid Structure:

Level\_l: {Δx\_l = Δx\_0/2^l}

```

### 3.2 Interface Handling

```

Interpolation:

W\_fine = I(W\_coarse)

Where I = cubic interpolation operator

Conservation:

∑W\_fine = W\_coarse

```

## 4. Parallelization Strategy

### 4.1 Domain Decomposition

```

Spatial Division:

Process\_p: [i\_start(p) : i\_end(p)]

Ghost Cells:

n\_ghost = order/2

x\_ghost = [i\_start-n\_ghost : i\_start-1]

Communication:

MPI\_SendRecv for boundary data

```

### 4.2 Load Balancing

```

Work Distribution:

W\_p = N\_cells(p)/N\_total

Balance Criterion:

max(W\_p)/min(W\_p) ≤ 1.2

Dynamic Adjustment:

if imbalance > 20%:

redistribute\_domains()

```

## 5. Numerical Solvers

### 5.1 Elliptic Solver

```

Multigrid Method:

1. Restriction: I\_h^{2h}

2. Smoothing: Gauss-Seidel

3. Prolongation: I\_{2h}^h

V-cycle:

error = ||Au - f||/||f|| < ε

```

### 5.2 Hyperbolic Solver

```

MUSCL Scheme:

F\_{i+1/2} = F(W\_L, W\_R)

Where:

W\_L = W\_i + ½φ(r)(W\_i - W\_{i-1})

W\_R = W\_{i+1} - ½φ(r)(W\_{i+2} - W\_{i+1})

```

## 6. Stability Analysis

### 6.1 Von Neumann Analysis

```

Amplification Factor:

g(k) = W^{n+1}/W^n = 1 - 2r(1-cos(kΔx))

Stability Condition:

|g(k)| ≤ 1 ∀k

Where:

r = νΔt/Δx²

```

### 6.2 Energy Methods

```

Discrete Energy:

E^n = ∑\_i |W\_i^n|²Δx

Conservation:

E^{n+1} ≤ (1 + CΔt)E^n

```

## 7. Error Control

### 7.1 Error Estimation

```

Local Truncation Error:

τ\_i^n = ||W\_i^n - W(x\_i,t\_n)||

Richardson Extrapolation:

W\_exact ≈ W\_h + (W\_h - W\_{2h})/3

```

### 7.2 Adaptive Time Stepping

```

Step Size Control:

Δt\_new = Δt \* min(2.0, max(0.5, (ε/E)^{1/4}))

Where:

E = estimated error

ε = tolerance

```

## 8. Boundary Treatments

### 8.1 Physical Boundaries

```

Dirichlet:

W\_boundary = W\_specified

Neumann:

(∂W/∂n)\_boundary = q\_specified

Radiation:

(∂W/∂t + c∂W/∂n)\_boundary = 0

```

### 8.2 Numerical Boundaries

```

Non-Reflecting:

∂W/∂t + c∂W/∂n + W/2R = 0

Perfectly Matched Layer:

∂W/∂t = ... + σ(x)W

```

## 9. Data Analysis

### 9.1 Diagnostics

```

Conservation Check:

δE = |E^{n+1} - E^n|/|E^n| < ε\_energy

Divergence Check:

div\_error = ||∇·W||\_∞ < ε\_div

```

### 9.2 Visualization

```

Field Plotting:

contour(x, t, W)

quiver(x, y, W\_x, W\_y)

Spectral Analysis:

plot(k, |FFT(W)|²)

```

## 10. Implementation Code

### 10.1 Core Algorithm

```python

def temporal\_flow\_solver(W\_init, t\_final, Δt, Δx):

# Initialize

W = W\_init.copy()

t = 0

while t < t\_final:

# RK4 Step

k1 = compute\_rhs(W)

k2 = compute\_rhs(W + 0.5\*Δt\*k1)

k3 = compute\_rhs(W + 0.5\*Δt\*k2)

k4 = compute\_rhs(W + Δt\*k3)

# Update

W\_new = W + (Δt/6)\*(k1 + 2\*k2 + 2\*k3 + k4)

# Error Check

if error\_estimate(W\_new, W) > tolerance:

Δt = adjust\_timestep(Δt)

continue

W = W\_new

t += Δt

return W

```

### 10.2 Parallel Implementation

```python

def parallel\_solver(local\_grid):

# MPI Setup

comm = MPI.COMM\_WORLD

rank = comm.Get\_rank()

size = comm.Get\_size()

while not converged:

# Exchange boundary data

send\_boundaries(comm, rank, size)

# Solve on local domain

local\_solution = temporal\_flow\_solver(local\_grid)

# Global reduction for convergence check

error = comm.allreduce(local\_error, op=MPI.MAX)

return local\_solution

```