Title:	Numerically Solving the Heat Equation
Author(s):	Garrett, Charles Kristopher
Intended for:	2017 LANL Parallel Computing Summer Research Internship lecture
Issued:	2017-06-20

LA-UR-17-24453

Numerically Solving the Heat Equation

Kris Garrett

June 2017

The Heat Equation

1D Equation

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

2D Equation

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

Boundary and Initial Conditions in 1D

$$u(a,t) = g_a(t)$$

$$u(b,t) = g_b(t)$$

$$u(x,0) = u_0(x)$$

Will concentrate on the 1D equation for this presentation

Discretization in Space/Time

$u(x,t)$ **Exact solution**

$u_i^n \approx u(x_i, t^n)$ **Approximate solution in x and t**

Common discretization of second derivative

$$\frac{\partial^2 u}{\partial x^2} = \frac{u(x-\Delta x) - 2u(x) + u(x+\Delta x)}{\Delta x^2} + O(\Delta x^2)$$

Means a quantity less than C Δx^2 where C is problem dependent but not discretization dependent

How to Get the Approximation

$$\frac{\partial^2 u}{\partial x^2} = \frac{u(x-\Delta x) - 2u(x) + u(x+\Delta x)}{\Delta x^2} + O(\Delta x^2)$$

Taylor Series

$$u(x+\Delta x) = u(x) + \Delta x u'(x) + \frac{\Delta x^2}{2} u''(x) + \frac{\Delta x^3}{6} u'''(x) + O(\Delta x^4)$$

$$u(x-\Delta x) = u(x) - \Delta x u'(x) + \frac{\Delta x^2}{2} u''(x) - \frac{\Delta x^3}{6} u'''(x) + O(\Delta x^4)$$

Add these, subtract 2u(x), and divide by Δx^2

Discretize in Time

Explicit Euler (1st Order)

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

Implicit Euler (1st Order)

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

Crank-Nicholson (2nd Order)

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

Discretize in Time

- **“Method of Lines” approach**
  - Discretized in all variables except time
  - Then discretize in time
- **Local truncation error for time integration: $O(\Delta t^{p+1})$**
  - Error from one time step
- **Global error for time integration: $O(\Delta t^{p})$**
  - Error after all time steps
  - Order reduces by 1 by accumulating all time step errors

Two Things Always Needed

- Stability
- Consistency

Lax Equivalence Theorem for Linear PDEs

$$\boxed{\text{Convergence = Stable + Consistent}}$$

What is Consistent?

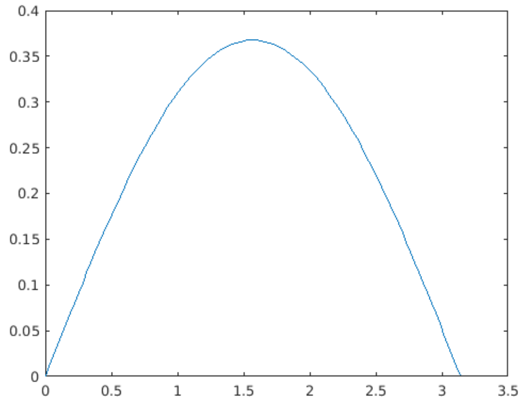
- Discretization converges to original PDE
- Example with implicit/explicit Euler steps

$$u_i^1 = u(x_i, t^1) + O(\Delta x^2) + O(\Delta t^2)$$

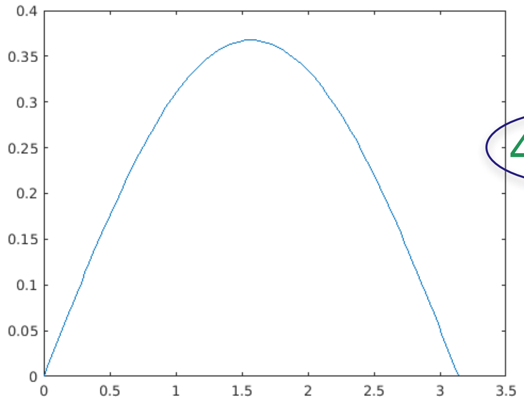
Importance of Stability

Initial Condition: $u_0(x) = \sin(x)$
Boundary Conditions: $u(0) = u(\pi) = 0$

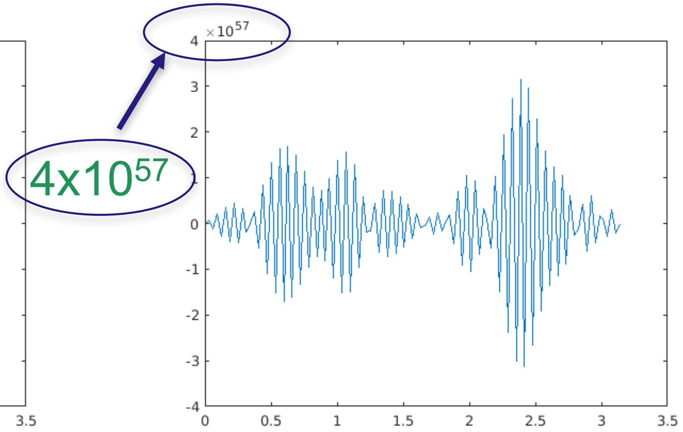
Implicit Euler
Large Δt



Explicit Euler
Small Δt



Explicit Euler
Large Δt



Importance of Stability

- **For stability**
  - Explicit time integration: always subject to time step restriction
    - Typically the time step restriction is based on the size of Δx
  - Implicit time integration:
    - Sometimes no time step restriction
    - Usually less restriction on time step
  - One technique to calculate time step restriction is Von Neumann Analysis

- **Remember: accuracy requires a time step restriction as well**

Von Neumann Analysis

- **Steps**
  - Apply Discrete Fourier Transform at time step n
  - Apply time step to one Fourier mode
  - See what conditions cause increase in size of Fourier mode

- **Stability requires all initial conditions are damped**
  - Keeps discretization and round off errors from increasing exponentially

Von Neumann Analysis

Step 1: Apply Discrete Fourier Transform at time step n

$$u_j^n = \sum_{k=0}^{N-1} \hat{u}_k^n e^{-i\underbrace{(2\pi j/N)}_{\theta_j}k}$$

Von Neumann Analysis

Step 2: Apply time step to one Fourier mode

$$\hat{u}_k^{n+1} e^{-i\theta_j k} = \hat{u}_k^n e^{-i\theta_j k} + \frac{\Delta t}{\Delta x^2} \left(\hat{u}_k^n e^{-i\theta_{j-1} k} - 2\hat{u}_k^n e^{-i\theta_j k} + \hat{u}_k^n e^{-i\theta_{j+1} k} \right)$$

… do some algebra …

$$\hat{u}_k^{n+1} = \underbrace{\left[1 + \frac{\Delta t}{\Delta x^2} \left(e^{i(2\pi/N)k} - 2 + e^{-i(2\pi/N)k} \right) \right]}_{\text{Amplification Factor}} \hat{u}_k^n$$

Von Neumann Analysis

Step 3: See what conditions cause increase in size of Fourier mode

Want value in brackets < 1
Maximum absolute value in brackets occurs for k = N/2

$$\left|1 - 4\frac{\Delta t}{\Delta x^2}\right| < 1$$

Leads to the stability requirement

$$\Delta t < \frac{1}{2}\Delta x^2$$

Implicit Methods

- **Von Neumann analysis for implicit Euler and Crank-Nicholson**
  - No time step restriction for stability
- **But you have to solve a linear system**
  - Takes more time to solve than explicit method

Implicit Methods

Linear System for implicit Euler or Crank-Nicholson

$$\begin{pmatrix} 1+2r & -r & & & \\ -r & 1+2r & -r & & \\ & \ddots & \ddots & \ddots & \\ & & -r & 1+2r & -r \\ & & & -r & 1+2r \end{pmatrix} u^{n+1} = b^n$$

Implicit Methods

- **Nice properties of linear system**
  - Strictly diagonally dominant
    - Gershgorin circle theorem implies system is positive definite
  - Symmetric system
    - Implies system is diagonalizable (basis of eigenvectors)
    - All eigenvalues are real

- **If Δt is approximately Δx, then r is very big**
  - As Δx goes to zero, system approaches weakly diagonally dominant
  - Harder for many iterative methods to converge

Solving Linear Systems

- **Dense linear algebra: O(N^3) flops**
  - Gaussian Elimination (Lapack, Scalapack)
    - Can exploit banded nature of linear system
  - Can exploit sparse nature of linear system (SuperLU)

- **Sparse linear algebra: O(I N^2) flops** ***I = Iterations***
  - Classical: Jacobi, Gauss-Seidel, SOR (Usually hand coded)
  - Krylov: CG, GMRES, etc (Petsc, Trilinos)
  - Multigrid, Algebraic Multigrid (Hypre, Petsc, Trilinos)
  - *Never stores the zeros of the matrix*

Dense Linear Algebra

- **A = LU**
  - L lower triangular, U upper triangular
  - Factorizing takes the most time: $O(N^3)$ flops
  - Solve Ax = b via LUx = b
  - Each triangular solve takes: $O(N^2)$ flops
  - Can reuse L and U for later solves
  - Really use A = PLU (Gaussian elimination with pivoting)
- **For symmetric, positive definite: use Cholesky factorization**
  - $A = L L^T$
  - No pivoting needed

Dense Linear Algebra

- **Uses BLAS (Basic Linear Algebra Subroutines)**
  - Highly optimized: MKL, ACML, cuBLAS, ATLAS, OpenBLAS
  - Implements for example:
    - Matrix matrix multiply
    - Matrix vector multiply
    - Dot product of vectors

Dense Linear Algebra

- **LAPACK and BLAS originally FORTRAN libraries**
  - CBLAS and LAPACKE for C interface
  - Can link to Fortran library from C/C++

- **LAPACK library variants**
  - ScaLAPACK – MPI version
  - MAGMA – GPU version
  - SuperLU – unsymmetric, sparse systems

Dense Linear Algebra

- **Other classical decompositions**
  - QR decomposition: A = QR
    - Q is an orthogonal matrix
    - R is an upper triangular matrix
    - Used for least squares problems
  - Eigendecomposition
    - $A = QDQ^T$ for symmetric problems
    - $A = QTQ^*$ for nonsymmetric problems
    - Q is orthogonal or hermitian
    - D is diagonal, real
    - T is triangular

Classical Sparse Linear Solvers

- **Write $A = N-M$. Solve $(N-M)x = b$**
  - Iterate $Nx^{k+1} = Mx^k + b$
  - Converges if and only if $\rho(N^{-1}M) < 1$ (all eigenvalue magnitudes < 1)
    - Jacobi: N is the diagonal of A
    - Gauss Seidel: N is the upper or lower triangular part of A

- **Easy to code but converges slowly**

Classical Sparse Linear Solvers

Linear System

$$(1+2r)x_i - rx_{i-1} - rx_{i+1} = b_i$$

Jacobi Method

$$(1+2r)x_i^{k+1} = rx_{i-1}^k + rx_{i+1}^k + b_i$$

Gauss Seidel Method

$$(1+2r)x_i^{k+1} - rx_{i-1}^{k+1} = rx_{i+1}^k + b_i$$

Krylov Linear Solvers

- **Gets the "best answer" from a Krylov subspace K^k(A,b) = {b, Ab, A^2b, …, A^{k-1}b}**
  - CG (Conjugate Gradient) used for symmetric, positive definite systems
    - Three vector recurrence relation
  - GMRES (Generalized Minimal Residual) used for nonsymmetric systems
    - Must hold all vectors in Krylov space
    - Actually use GMRES(m): restart after m steps to reduce memory required
    - Guaranteed convergence for positive definite systems
  - Note: other Krylov spaces are used for some Krylov solvers

Krylov Linear Solvers

```
% Matlab version of CG from Wikipedia
function [x] = conjgrad(A, b, x)
    r = b - A * x;
    p = r;
    rsold = r' * r;
    for i = 1:length(b)
        Ap = A * p;
        alpha = rsold / (p' * Ap);
        x = x + alpha * p;
        r = r - alpha * Ap;
        rsnew = r' * r;
        if sqrt(rsnew) < 1e-10
            break;
        end
        p = r + (rsnew / rsold) * p;
        rsold = rsnew;
    end
end
```

Notice only 3 extra vectors of memory required

Also need to perform dot products. Can hurt parallel performance

Multigrid Linear Solver

- **Uses the classical solvers**
  - These solvers converge quickly for certain discrete Fourier modes
  - When grid size changes, other Fourier modes converge quickly
  - Solves problem on grid sizes: h, 2h, 4h, 8h, etc.
  - Generally fastest solver for diffusion type equations

Iterative Solvers

- **Many iterative solvers need a preconditioner**
  - Ax = b
  - PAx = Pb (Left preconditioner)
  - PA should require less iterations
  - P should be easily invertible
  - There are also right and symmetric preconditioners
  - Very problem dependent

Other Important Linear Solver Topics

- **Norm of vector ||v||**

$$||v||_1 = |v_1| + |v_2| + \ldots + |v_n|$$

$$||v||_2 = \left(v_1^2 + v_2^2 + \ldots + v_n^2\right)^{1/2}$$

$$||v||_\infty = \max\{|v_1|, |v_2|, \ldots, |v_n|\}$$

- **Induced norm of matrix (max matrix stretches a vector)**

$$||A|| = \max_{||v||=1} ||Av||$$

Other Important Linear Solver Topics

- Condition number of a matrix

$$\kappa(A) = ||A^{-1}|| \cdot ||A||$$

- If you solve Ax = b+e, the relative error in solution compared to the relative error in RHS is

$$\frac{||A^{-1}e||}{||A^{-1}b||} \leq \kappa(A)\frac{||e||}{||b||}$$

- This is for *exact arithmetic*
- This shows the error in solution given error in data

Ensuring Correctness

- **You must run tests to ensure a correct answer and correct implementation**
- ***Verification Tests*: Make sure you are actually solving the heat equation**
  - Use known analytical solutions: $\sin(x)\, e^{-t}$
  - Method of manufactured solution: used when there is a known source.
    - Make up a solution and determine the source.
    - Put that source into your solver.

Ensuring Correctness

- ***Convergence Tests*: Make sure numerical implementation is correct**
  - It is common to code incorrectly and get first order convergence of a higher order method

- ***Unit Tests*: Test code in every file**
  - Very useful for large projects
  - When you find a bug, add a test to reproduce it
  - Makes pinpointing errors easier
  - Some projects require that every branch in code is tested

The End