Finite Difference Methods for Partial Differential Equations

James Cass

January 11, 2019

1 Introduction

This repository contains software for solving second-order linear partial differential equations using finite difference methods. The Jupyter notebooks "Examples.ipynb" and "Solutions.ipynb" present the code that goes along with this report, and will be referred to throughout as 'the examples' and 'the solutions'.

2 Finite Difference Methods

Consider the following partial differential equation problem.

$$\frac{\partial u}{\partial t} = \kappa \frac{\partial^2 u}{\partial x^2} \tag{1}$$
$$u(0,t) = 0 \tag{2}$$
$$u(L,t) = 0 \tag{3}$$
$$u(x,0) = \sin(\pi x/L) \tag{4}$$

This is the one dimensional heat equation describing a rod with an initial temperature distribution and a fixed temperature at both ends.

We can discretize the PDE by writing the equation for a set of points i in space and points j in time,

$$\frac{\partial u_{i,j}}{\partial t} = \kappa \frac{\partial^2 u_{i,j}}{\partial x^2}. \tag{5}$$

If the space between these instants of time j is small, then the derivative $\partial u_{i,j}/\partial t$ will be approximately equal to the differences

$$\text{forward difference} \quad \frac{u_{i,j+1} - u_{i,j}}{\Delta t} \tag{6}$$
$$\text{backward difference} \quad \frac{u_{i,j} - u_{i,j-1}}{\Delta t} \tag{7}$$
$$\text{central difference} \quad \frac{u_{i,j+1} - u_{i,j-1}}{2\Delta t} \tag{8}$$

with the approximation becoming more accurate as $\Delta t \to 0$. For the space derivative, we will use a second central difference approximation,

$$\frac{\partial^2 u}{\partial x^2}(x_i, t_i) = \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{\Delta x^2}. \tag{9}$$

Plugging the three types of difference for the time derivative into the heat equation and rearranging so that all terms at time j are on one side and terms at time $j+1$ on the other leads to the following schemes in matrix form:

$$
\begin{aligned}
&\text{Forward Euler} && \mathbf{u}^{(j)} = A_{\mathrm{FE}}\mathbf{u}^{(j+1)} && (10)\\
&\text{Backward Euler} && A_{\mathrm{BE}}\mathbf{u}^{(j+1)} = \mathbf{u}^{(j)} && (11)\\
&\text{Crank-Nicholson} && A_{\mathrm{CN}}\mathbf{u}^{(j+1)} = B_{\mathrm{CN}}\mathbf{u}^{(j)} && (12)
\end{aligned}
$$

where $\mathbf{u}^{(j)} = (u_{1,j}, u_{2,j}, \cdots, u_{m-1,j})^T$ represents the solution of the equation at the inner mesh points at time j. The matrices are all tridiagonal [1]. Backward Euler and Crank-Nicholson are both implicit methods, since they require a system of equations to be solved at each time step. Schemes for hyperbolic and elliptic equations are created similarly. Considering other PDE problems with different boundary conditions or source functions leads to modifications of these schemes by adding extra terms or matrix entries.

Note: The central difference used for the time derivative in the Crank-Nicholson scheme leads to times half way between mesh points. Where these appear in the central difference for space we take the average of the time points either side.

3 Design

The software consists of

1. objects for specifying PDE problems
2. implementations of finite difference schemes
3. various extra functions for plotting, animation, calculating errors etc.

The three files `parabolicpde.py`, `hyperbolicpde.py` and `ellipticpde.py` contain the objects and solvers relevant to each class of problem. Some functions and objects could be used by more than one class of problem, and these are separated into the files:

- `boundary.py` - containing objects for specifying boundary conditions. For example `Dirichlet` and `Neumann` can be used for both parabolic and elliptic problems.
- `visualizations.py` - containing the function `plot_solution` for plotting a numerical and exact solution together, and `animate_tsunami` for animating travelling waves.
- `helpers.py` - containing other useful functions. `tridiag` creates a tridiagonal matrix given the entries on the diagonals, `get_error` gives you the error between an exact and numerical solution and the functions `numpify` and `numpify_many` are for vectorizing constants or sympy expressions to allow for flexible input to the solvers.

This separation represents a compromise between combining common features of the problems and allowing future modifications to one class of problems without affecting the others.

Example of Usage

The user can create a `ParabolicProblem` object. This is a readable way of entering problems that minimizes the number of extra functions that have to be written.

[1] $A_{\mathrm{FE}} = \mathrm{tridiag}(\lambda, 1-2\lambda, \lambda)$, $B_{\mathrm{FE}} = \mathrm{tridiag}(-\lambda, 1+2\lambda, -\lambda)$, $A_{\mathrm{CN}} = \mathrm{tridiag}(-\lambda/2, 1+\lambda, -\lambda/2)$, $B_{\mathrm{CN}} = \mathrm{tridiag}(\lambda/2, 1-\lambda, \lambda/2)$, where $\lambda = \kappa\Delta t/\Delta x^2$

```
from sympy import *
from parabolicpde import ParabolicProblem
from boundary import Dirichlet

# specify the problem
dp = ParabolicProblem(kappa=1, L=1,
                     ic=sin(pi*x),
                     lbc=Dirichlet(0,0), rbc=Dirichlet(1,0),
                     source=0)

# print the problem using LaTeX
dp.pprint()

# exact solution
u = exp(-(pi**2)*t)*sin(pi*x)

# numerical parameters
mx = 15      # number of grid points in space
mt = 100     # number of grid points in time
T = 0.5      # time to stop the integration

# solve the PDE at time T = 0.5 with forward Euler method
uT, err, lmbda = dp1.solve_at_T(0.5, mx, mt, 'FE')
```

This creates an object specifying the problem in Equations (1)-(4). The initial condition and source functions are specified with `sympy` expressions or constants. The boundary conditions are specified by `Dirichlet` objects that live in `boundary.py`, with the x value as the first parameter and a `sympy` function or constant as the second.

The method `pprint` will print the problem in latex form, which is attractive and useful for checking that you have specified the problem correctly. `solve_at_T` will solve the problem, plot the results and return the solution at time T, the absolute error (if an expression for the exact solution `u_exact` is given), and the value of λ. For representative examples of solving hyperbolic and elliptic problems see examples 7 and 10 respectively.

4 Worksheet 1: Parabolic PDEs

4.1 Question 2. (a) and (b): Solving Diffusion Problems

There are three functions in the file `parabolicsolvers` for solving parabolic PDE problems such as in Equations (1) - (4). These are `forwardeuler`, `backwardeuler` and `cranknichol-son` (corresponding to the three methods outlined earlier). The three functions all have the same calling signature, e.g.

```
def forwardeuler(mx, mt, L, T,
                 kappa, source,
                 ic, lbc, rbc, lbctype, rbctype):
```

The parameters for producing the numerical solution are: `mx` - the number of mesh points in space, `mt` - the number of grid points in time, `L` - the width of the interval and `T` - the time at which to terminate the integration. Next are modifications to the PDE problem - the diffusion coefficient `kappa` and a source function `source`. These may be passed as constant values, `sympy` expressions or vectorized functions that take `numpy` arrays. The final row takes the initial conditions and boundary conditions. Again, the software is flexible to the type of initial condition (`ic`) and left and right boundary conditions (`lbc` and `rbc`) that are passed to it. The types of the boundary conditions must also be specified as strings with `lbctype` and `rbctype`. The software is capable of handling Dirichlet or Neumann boundary

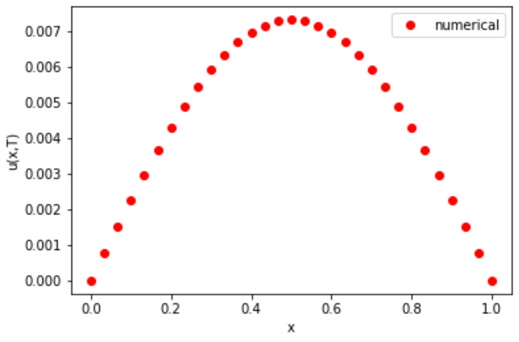

Figure 1: Numerical solution of Equations (1)-(4) at time $T = 0.5$. Obtained using the function `plot_solution` in the file `visualizations.py`.

conditions (pass 'Dirichlet' or 'Neumann' to the solver). The function returns an array of values `xs`, the numerical solution values `uT` at those x values, and the value of $\lambda = c\Delta t/\Delta x^2$.

Let's now use the software to numerically solve Equations (1)-(4) for a bar of length $L = 1$ up to a time $T = 0.5$ using the backward Euler method.

```
import numpy as np
from parabolicpde import backwardeuler
from visualizations import plot_solution

mx = 30; mt = 1000; L = 1; T = 0.5; kappa = 1

# initial condition
def uI(x):
    return np.sin(np.pi*x/L)

xs, uT = backwardeuler(mx, mt, L, T,
                       kappa, 0,
                       uI, 0, 0, 'Dirichlet', 'Dirichlet')

plot_solution(xs, uT)
```

The results are shown in Figure (1).

Examples

There are many solved diffusion problems in the Examples. [2]. Problems are specified using a `ParabolicProblem` object for convenience, as described in the previous section. These examples demonstrate the different classes of problems that can be solved with the software, compared with analytical (exact or approximate) solutions where possible.

1. The solution to Equations (1)-(4).
2. An inhomogeneous Dirichlet boundary condition on one side.
3. Neumann boundary conditions.
4. A constant heat source.
5. A spatially varying heat source.

[2]Examples taken from *Partial Differential Equations for Scientists and Engineers* Stanley Farlow (Wiley, 1982; Dover, 1993)

6. Mixed boundary conditions.

Example (6) uses `backwardeuler2`, a second implementation of the backward Euler scheme that takes mixed boundary conditions such as

$$\alpha u + \beta \frac{\partial u}{\partial x} = g(t). \tag{13}$$

Discretizing the space derivative with a forward derivative at the left boundary gives

$$\begin{aligned} \alpha_1 u_{0,j} + \beta_1 \frac{u_{1,j} - u_{0,j}}{\Delta x} &= g_j \\ \implies (\alpha_1 \Delta x - \beta_1) u_{0,j} + \beta_1 u_{1,j} &= \Delta x q_j. \end{aligned}$$

We can obtain a similar expression for the right boundary using a backward difference. These two equations are then added as the top and bottom rows of the backward Euler matrix equation to solve for the boundaries $u_{0,j+1}$ and $u_{mx,j+1}$.

4.2 Question 2. (c): Improvements

Improvements to the user interface and structure of the code have already been described in the Design section.

Computational Improvements

The software takes advantage of the tridiagonal matrices by using the sparse matrix representation in the `scipy` package. In the file `helpers.py` there is a function `tridiag`, which takes lists corresponding to the lower, main and upper diagonals and returns a sparse matrix representation. The solvers then use the function `spsolve` from `scipy.sparse.linalg` to speed up the calculations.

4.3 Question 3: Error Analysis

In the solutions we see comparisons of the errors produced by the three schemes. There are three code boxes under the heading "Worksheet 1 Question 3". Each one can be modified by varying parameters or trying a different problem (some are commented out). We start the analysis with backward Euler and Crank-Nicholson (so we can ignore stability for a moment). The theoretical results suggest that the truncation error due to these schemes will be

$$\begin{aligned} \text{Backward Euler} \quad & E = O(\Delta t) + O(\Delta x^2) \\ \text{Crank-Nicholson} \quad & E = O(\Delta t^2) + O(\Delta x^2) \end{aligned}$$

Code box 1: Varying Δt

We set $\Delta x = 0.001$ so that the $O(\Delta x^2)$ term in the error should be overwhelmed by the $O(\Delta t)$ term that we are investigating. The table underneath the code box in the solutions shows how the error decreases as Δt decreases. Figure (2) shows a log-log plot of the error (calculated using the L^2 norm) as Δt is varied across 4 orders of magnitude.

For backward Euler we see a straight line. The slope of this line looks to be very close to 1, which we should expect because (ignoring the $O(\Delta x^2)$ term)

$$\begin{aligned} E &\sim c\Delta t \\ \implies \log(E) &\sim \log(\Delta t) + \log(c) \end{aligned}$$

Crank-Nicholson shows a deviation from a straight line at the bottom left where the term of $O(\Delta x^2)$ is becoming important. In the region where this term is not important it's clear to see that the slope is twice that of the line for backward Euler.

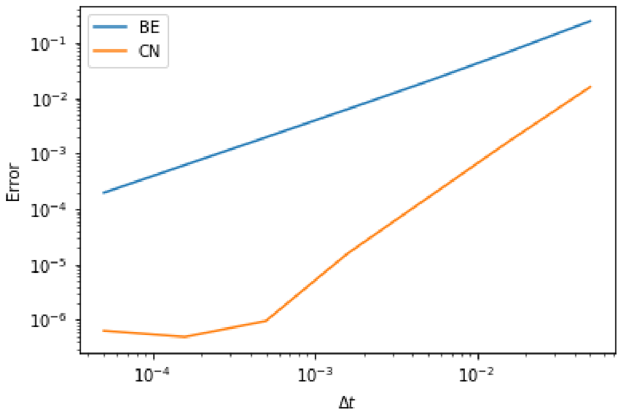

Figure 2: Comparison of errors in numerical solutions using the backward Euler and Crank-Nicholson schemes varying Δt. The slope of the Crank-Nicholson curve in the region $\Delta t > 10^{-5}$ is twice that of backward Euler.

Code box 2: Varying Δx

Figure (3) shows the errors produced when varying Δx, keeping a fixed $\Delta t = 10^{-5}$. We need such a small Δt in this case since the error due to Δt should be small in comparison to the error due to Δx^2 (for backward Euler). Since we are varying the number of points in space, and thus the amount of terms in the calculation of the L^2 norm, we will use the L^∞ norm for a fairer comparison.

The slope of the line for Crank-Nicholson is clearly 2, displaying the $O(\Delta x^2)$ dependency of the truncation error. The curve for backward Euler is more affected by the Δt term (as expected since it isn't squared) but the slope of the curve certainly approaches that of Crank-Nicholson as Δx increases.

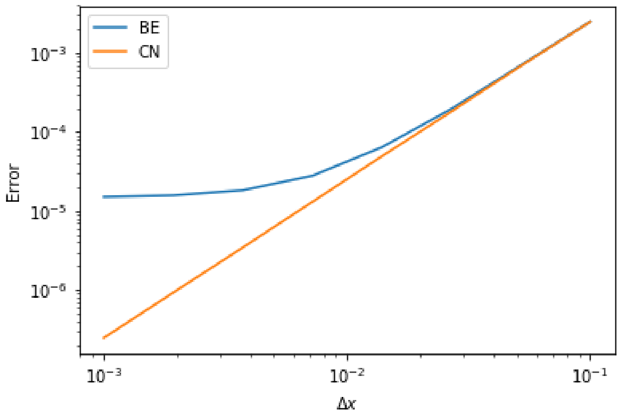

Figure 3: Comparison of errors in numerical solutions using the backward Euler and Crank-Nicholson schemes varying Δx. The slope of the Crank-Nicholson line is clearly 2, and the backward Euler curve approaches this slope as the truncation error due to Δt becomes unimportant.

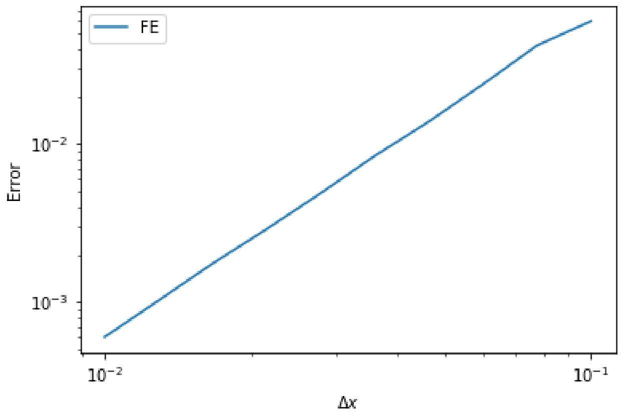

Figure 4: Error of forward Euler scheme due to varying Δx

Code box 3: Forward Euler

Investigating the error due to Δt requires a very small Δx, and $\lambda = \kappa \Delta t / \Delta x^2$ must be greater than 1/2 for stability of the scheme. Therefore, we will focus on showing how the error in Δx varies with changes in Δt. Figure (4) shows a test over a stable range of the forward Euler scheme, clearly showing the slope of 2 consistent with an error of order Δx^2.

5 Worksheet 2: Hyperbolic PDEs

Examples 7-9 show how hyperbolic problems can be specified and solved in a similar manner to parabolic problems. The examples demonstrate the explicit and implicit methods using both Dirichlet and Neumann boundary conditions.

5.1 Question 1. (a): Stability and Accuracy

The solutions (under "Worksheet 2 Question 1") show the absolute errors resulting from varying λ using the explicit method. We can see that the error is minimum at $\lambda = 1$, the 'magic timestep' where the truncation error is zero (so any remaining error is due to roundoff). Beyond this the error quickly scales out of control as the scheme loses stability.

5.2 Question 1. (b): Implicit Method

The next code box shows the same investigation for the implicit method, for which we see that there is no loss of stability.

5.3 Question 2: Non-Constant Wavespeed

Due to the specific nature of this problem I created a `TsunamiProblem` object for studying travelling waves, which calls the solver `tsunami_solve` in `hyperbolicpde.py`. `tsunami_solve` is based on the explicit hyperbolic solver with some slight differences:

1. Since the wave initially splits into right and left moving parts the boundary conditions are set to be open at the start. Once the left moving wave has left the domain it becomes a periodic boundary that is set to whatever the right boundary was at the previous timestep. This allows the initially right moving wave to continually propagate across the domain from left to right.

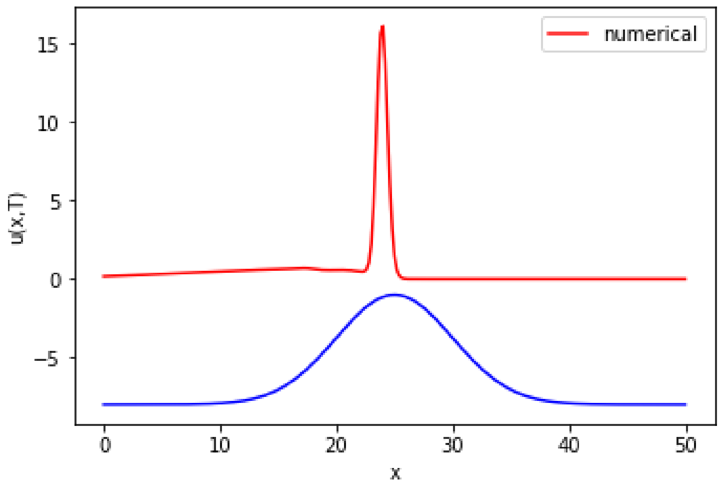

Figure 5: The tsunami at its highest point. As the wave passes the subsea hill (blue) it slows down at the front and increases in height.

2. The speed of the wave is now locally $\sqrt{h}$, requiring changes to the rows of the matrix. Where the seabed is flat the matrix looks the same as in the explicit solver.

The solutions (under "Worksheet 2 Question 2") show the code for creating an animation of the wave that goes around the domain twice. This animation is included in the repository (tsunami.mp4), created with the `solve_at_T` method of the object at time $T = 40$ and the keyword argument `animate` set to `True`. We can see that as the wave passes the region around $x = 25$ it becomes narrower and higher.

Figure (5) shows the cause of this deformation. There is a subsea hill that decreases the height between the undisturbed sea and the seabed, reducing the speed that the wave can travel with at this point. We see the wave slow down as the front reaches the hill, causing it to bunch up and increase in height (from around $u = 10$ to $u = 15$). As the front of the wave leaves the hill it begins to move faster than the trailing part, and the wave returns to a shape similar to the one it had before initially encountering the hill. A real wave coming into shore has an ever-decreasing height above the seabed that causes it to continually grow, creating dangerously large tidal waves or tsunamis from an initial disturbance such as an earthquake.

6 Worksheet 3: Elliptic PDEs

Elliptic PDEs are also implemented with their own object `EllipticProblem` and solver `SOR`.

6.1 Question 1: The SOR Method

Example 10 shows that the SOR method reduces the number of iterations required to reach a solution to 123 (from 477 using the solver in `myJacobi2DLaplace.py`), and the error from the true solution is an order or magnitude better (10^{-3} rather than 10^{-2}).

These results were achieved using a value of $\omega = 1.5$. Figure (6) shows how the number of iterations varies as ω varies between 1 and 2 (code in solutions under "Worksheet 3 Question 1"). There looks to be a minimum value around $\omega = 1.8$, after which the number of iterations increases sharply.

The following code box uses the function `minimize_scalar` in the package `scipy.optimize` to minimize the function `num_iterations` over the interval $\omega \in (1, 2)$. The results are shown

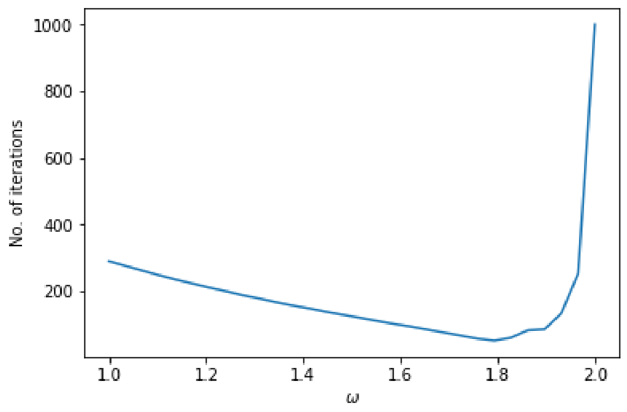

Figure 6: Number of iterations required to reach a numerical solution of Laplace's equation using the SOR method, for various values of ω

below the code box, giving a result for the minimum as $\omega = 1.78$ to 2 d.p.

6.2 Learning Log

This project presented a range of different challenges. There are many software engineering principles and mathematical concepts that I have learned or improved in the process.

- This was a useful time to refresh my knowledge of PDEs, having not studied them for a while. Solving the textbook problems on paper and then with the software was satisfying and gave me confidence that I was producing the correct answers.
- There are many aspects of programming with Python that I was able to explore and improve my skills with during this project. I learned about object-oriented programming and scientific computing with `scipy`. I gained more experience of vectorizing calculations with `numpy` and furthered my plotting skills with `matplotlib`. I certainly feel confident about tackling further projects in scientific computing after completing this unit.
- Having learned about test-driven development in the first part of the course, I found it very useful to devote a file to examples. Whenever I made changes to the code I could run all the examples to check if anything had broken.
- Starting with the diffusion-type problems, I set about designing the software with the big picture in mind. I had become accustomed in previous software engineering projects to writing specific pieces of code to solve a problem, and then attempting to organise and generalise the code into useful functions and separate files. The function `solve_diffusion_pde` in the file `discretesolve.py` is an example of this kind of attempt. All three schemes are represented by one matrix equation, with calls to functions for each scheme to get the matrices and boundary conditions. While this is a nice concise solution, it is hard to anticipate how the next boundary condition or problem description might be different. As with analytical solutions of PDEs, there is no general numerical way that applies across all types of problems. Generalizing the code early on caused me to spend time undoing what I had already done. I often found that I had to modify previously written code when starting any new piece of the software. It became a time-consuming process of continual combining and breaking apart pieces of code.

- I learned a lot about the schemes by investigating the errors in question 3 on the first worksheet. Keeping the parameters small increased the run time, but increasing them led to ambiguous results or stability issues. Playing around with the edge cases gives you a better idea of the optimal parameter values for a particular scheme.

- If I could start the project again I would focus more carefully on the worksheet questions, rather than producing a perfect piece of software. The test of the software is ultimately that it works on the cases that it's supposed to work on. There is no end to improvements that can be made, and as the deadline approached I started having to make compromises. In software engineering companies this tension between making something perfect and elegant versus something that works must be a constant issue.

- A major difference between this project and the first coursework in the unit was the concurrent workload due to other units and exam revision. This meant leaving the project for periods of time and coming back to it later. This showed me the value of commenting code carefully, as it can be very hard to understand code that that you haven't seen for a while. The git commit history is also a very helpful way of seeing the progression of the project and what you were previously working on.

- I have learned that there is a whole world of methods for producing numerical solutions to problems in mathematics. The issues of stability, accuracy, truncation error, round-off error and running time always have to be taken into consideration and tradeoffs between them are often necessary. Even among finite difference methods many exist for each small subset of the problems you might be interested in, so it becomes a matter of choosing the right solver for the specific problem. Having gained this knowledge from the unit I will be much more sensitive to the particular choice of method when using software that solves equations numerically.