Finite-Difference Approximations to the Heat Equation

Gerald W. Recktenwald*

January 21, 2004

Abstract

This article provides a practical overview of numerical solutions to the heat equation using the finite difference method. The forward time, centered space (FTCS), the backward time, centered space (BTCS), and Crank-Nicolson schemes are developed, and applied to a simple problem involving the one-dimensional heat equation. Complete, working MATLAB codes for each scheme are presented. The results of running the codes on finer (one-dimensional) meshes, and with smaller time steps is demonstrated. These sample calculations show that the schemes realize theoretical predictions of how their truncation errors depend on mesh spacing and time step. The MATLAB codes are straightforward and allow the reader to see the differences in implementation between explicit method (FTCS) and implicit methods (BTCS and Crank-Nicolson). The codes also allow the reader to experiment with the stability limit of the FTCS scheme.

1 The Heat Equation

The one dimensional heat equation is

$$\frac{\partial \phi}{\partial t} = \alpha \frac{\partial^2 \phi}{\partial x^2}, \qquad 0 \le x \le L, \quad t \ge 0 \tag{1}$$

where $\phi = \phi(x,t)$ is the dependent variable, and α is a constant coefficient. Equation (1) is a model of transient heat conduction in a slab of material with thickness L. The domain of the solution is a semi-infinite strip of width L that continues indefinitely in time. The material property α is the thermal diffusivity. In a practical computation, the solution is obtained only for a finite time, say $t_{\max}$.

Solution to Equation (1) requires specification of boundary conditions at $x = 0$ and $x = L$, and initial conditions at $t = 0$. Simple boundary and initial conditions are

$$\phi(0,t) = \phi_0, \qquad \phi(L,t) = \phi_L \qquad \phi(x,0) = f_0(x). \tag{2}$$

Other boundary conditions, e.g. gradient (Neumann) or mixed conditions, can be specified. To keep the presentation as simple as possible, only the conditions in (2) are considered in this article.

*Associate Professor, Mechanical Engineering Department Portland State University, Portland, Oregon, gerry@me.pdx.edu

2 Finite Difference Method

The finite difference method is one of several techniques for obtaining numerical solutions to Equation (1). In all numerical solutions the continuous partial differential equation (PDE) is replaced with a discrete approximation. In this context the word "discrete" means that the numerical solution is known only at a finite number of points in the physical domain. The number of those points can be selected by the user of the numerical method. In general, increasing the number of points not only increases the resolution (i.e., detail), but also the accuracy of the numerical solution.

The discrete approximation results in a set of algebraic equations that are evaluated (or solved) for the values of the discrete unknowns. Figure 1 is a schematic representation of the numerical solution.

The *mesh* is the set of locations where the discrete solution is computed. These points are called nodes, and if one were to draw lines between adjacent nodes in the domain the resulting image would resemble a net or mesh. Two key parameters of the mesh are Δx, the local distance between adjacent points in space, and Δt, the local distance between adjacent time steps. For the simple examples considered in this article Δx and Δt are uniform throughout the mesh. In Section 2.1 the mesh is defined.

The core idea of the finite-difference method is to replace continuous derivatives with so-called difference formulas that involve only the discrete values associated with positions on the mesh. In Section 2.2 a handful of difference formulas are developed.

Applying the finite-difference method to a differential equation involves replacing all derivatives with difference formulas. In the heat equation there are derivatives with respect to time, and derivatives with respect to space. Using different combinations of mesh points in the difference formulas results in different *schemes*. In the limit as the mesh spacing (Δx and Δt) go to zero, the numerical solution obtained with any useful[1] scheme will approach the true solution to the original differential equation. However, the *rate* at which the numerical solution approaches the true solution varies with the scheme. In addition, there are some practically useful schemes that can fail to yield a solution for bad combinations of Δx and Δt. Three different schemes for the solution to Equation (1) are developed in Section 3.

The numerical solution of the heat equation is discussed in many textbooks. Ames [1], Morton and Mayers [3], and Cooper [2] provide a more mathematical development of finite difference methods. See Cooper [2] for modern introduction to the theory of partial differential equations along with a brief coverage of numerical methods. Fletcher [4], Golub and Ortega [5], and Hoffman [6] take a more applied approach that also introduces implementation issues. Fletcher provides Fortran code for several methods.

2.1 The Discrete Mesh

The finite difference method obtains an approximate solution for $\phi(x, t)$ at a finite set of x and t. For the codes developed in this article the discrete x are

[1] There are plausible schemes that do not exhibit this important property of converging to the true solution. Refer to discussions of *consistency* in references [1, 2].

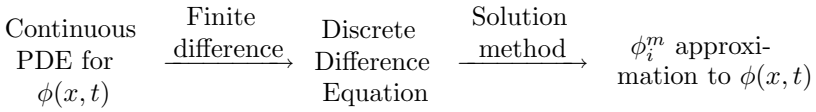

Figure 1: Relationship between continuous and discrete problems.

uniformly spaced in the interval $0 \le x \le L$ such that

$$x_i = (i-1)\Delta x, \qquad i = 1, 2, \ldots N$$

where N is the total number of spatial *nodes*, including those on the boundary. Given L and N, the spacing between the x_i is computed with

$$\Delta x = \frac{L}{N-1}.$$

Similarly, the discrete t are uniformly spaced in $0 \le t \le t_{\max}$:

$$t_m = (m-1)\Delta t, \qquad m = 1, 2, \ldots M$$

where M is the number of time steps and Δt is the size of a time step[2]

$$\Delta t = \frac{t_{\max}}{M-1}.$$

The solution domain is depicted in Figure 2. Table 1 summarizes the notation used to obtain the approximate solution to Equation (1) and to analyze the result.

2.2 Finite Difference Approximations

The finite difference method involves using discrete approximations like

$$\frac{\partial \phi}{\partial x} \approx \frac{\phi_{i+1} - \phi_i}{\Delta x} \tag{3}$$

[2]The first mesh lines in space and time are at $i = 1$ and $m = 1$ to be consistent with the MATLAB requirement that the first row or column index in a vector or matrix is one.

Table 1: Notation

Symbol	Meaning
$\phi(x,t)$	Continuous solution (true solution).
$\phi(x_i, t_m)$	Continuous solution evaluated at the mesh points.
ϕ_i^m	Approximate numerical solution obtained by solving the finite-difference equations.

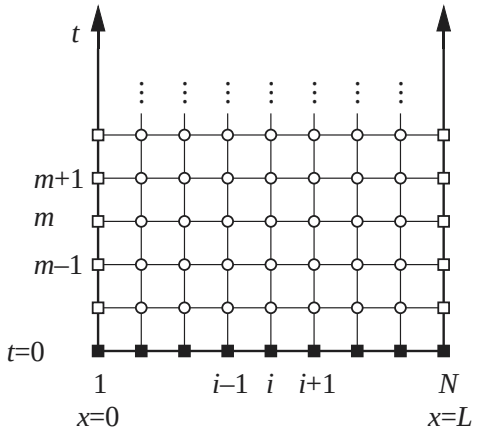

Figure 2: Mesh on a semi-infinite strip used for solution to the one-dimensional heat equation. The solid squares indicate the location of the (known) initial values. The open squares indicate the location of the (known) boundary values. The open circles indicate the position of the interior points where the finite difference approximation is computed.

where the quantities on the right hand side are defined on the finite difference mesh. Approximations to the governing differential equation are obtained by replacing all continuous derivatives by discrete formulas such as those in Equation (3). The relationship between the continuous (exact) solution and the discrete approximation is shown in Figure 1. Note that computing ϕ_i^m from the finite difference model is a distinct step from translating the continuous problem to the discrete problem.

Finite difference formulas are first developed with the dependent variable ϕ as a function of only one independent variable, x, i.e. $\phi = \phi(x)$. The resulting formulas are then used to approximate derivatives with respect to either space or time. By initially working with $\phi = \phi(x)$, the notation is simplified without any loss of generality in the result.

2.3 First Order Forward Difference

Consider a Taylor series expansion $\phi(x)$ about the point x_i

$$\phi(x_i + \delta x) = \phi(x_i) + \delta x \left.\frac{\partial \phi}{\partial x}\right|_{x_i} + \frac{\delta x^2}{2} \left.\frac{\partial^2 \phi}{\partial x^2}\right|_{x_i} + \frac{\delta x^3}{3!} \left.\frac{\partial^3 \phi}{\partial x^3}\right|_{x_i} + \cdots \tag{4}$$

where δx is a change in x relative to x_i. Let $\delta x = \Delta x$ in Equation (4), i.e., consider the value of ϕ at the location of the x_{i+1} mesh line

$$\phi(x_i + \Delta x) = \phi(x_i) + \Delta x \left.\frac{\partial \phi}{\partial x}\right|_{x_i} + \frac{\Delta x^2}{2} \left.\frac{\partial^2 \phi}{\partial x^2}\right|_{x_i} + \frac{\Delta x^3}{3!} \left.\frac{\partial^3 \phi}{\partial x^3}\right|_{x_i} + \cdots$$

Solve for $(\partial \phi / \partial x)_{x_i}$

$$\left.\frac{\partial \phi}{\partial x}\right|_{x_i} = \frac{\phi(x_i + \Delta x) - \phi(x_i)}{\Delta x} - \frac{\Delta x}{2} \left.\frac{\partial^2 \phi}{\partial x^2}\right|_{x_i} - \frac{\Delta x^2}{3!} \left.\frac{\partial^3 \phi}{\partial x^3}\right|_{x_i} + \cdots$$

Notice that the powers of Δx multiplying the partial derivatives on the right hand side have been reduced by one.

Substitute the approximate solution for the exact solution, i.e., use $\phi_i \approx \phi(x_i)$ and $\phi_{i+1} \approx \phi(x_i + \Delta x)$.

$$\left.\frac{\partial \phi}{\partial x}\right|_{x_i} \approx \frac{\phi_{i+1} - \phi_i}{\Delta x} - \frac{\Delta x}{2} \left.\frac{\partial^2 \phi}{\partial x^2}\right|_{x_i} - \frac{\Delta x^2}{3!} \left.\frac{\partial^3 \phi}{\partial x^3}\right|_{x_i} + \cdots \tag{5}$$

The mean value theorem can be used to replace the higher order derivatives (exactly)

$$\frac{\Delta x^2}{2} \left.\frac{\partial^2 \phi}{\partial x^2}\right|_{x_i} + \frac{(\Delta x)^3}{3!} \left.\frac{\partial^3 \phi}{\partial x^3}\right|_{x_i} + \cdots = \frac{\Delta x^2}{2} \left.\frac{\partial^2 \phi}{\partial x^2}\right|_{\xi}$$

where $x_i \le \xi \le x_{i+1}$. Thus

$$\left.\frac{\partial \phi}{\partial x}\right|_{x_i} \approx \frac{\phi_{i+1} - \phi_i}{\Delta x} + \frac{\Delta x^2}{2} \left.\frac{\partial^2 \phi}{\partial x^2}\right|_{\xi}$$

or

$$\left.\frac{\partial \phi}{\partial x}\right|_{x_i} - \frac{\phi_{i+1} - \phi_i}{\Delta x} \approx \frac{\Delta x^2}{2} \left.\frac{\partial^2 \phi}{\partial x^2}\right|_{\xi} \tag{6}$$

The term on the right hand side of Equation (6) is called the *truncation error* of the finite difference approximation. It is the error that results from truncating the series in Equation (5).

In general, ξ is not known. Furthermore, since the function $\phi(x,t)$ is also unknown, $\partial^2\phi/\partial x^2$ cannot be computed. Although the exact magnitude of the truncation error cannot be known (unless the true solution $\phi(x,t)$ is available in analytical form), the "big $\mathcal{O}$" notation can be used to express the dependence of the truncation error on the mesh spacing. Note that the right hand side of Equation (6) contain the mesh parameter Δx, which is chosen by the person using the finite difference simulation. Since this is the only parameter under the user's control that determines the error, the truncation error is simply written

$$\frac{\Delta x^2}{2} \left.\frac{\partial^2 \phi}{\partial x^2}\right|_{\xi} = \mathcal{O}(\Delta x^2)$$

The equals sign in this expression is true in the order of magnitude sense. In other words the "$= \mathcal{O}(\Delta x^2)$" on the right hand side of the expression is not a strict equality. Rather, the expression means that the left hand side is a product of an unknown constant and Δx^2. Although the expression does not give us the exact magnitude of $(\Delta x^2)/2((\partial^2\phi/\partial x^2)_{x_i})_\xi$, it tells us how quickly that term approaches zero as Δx is reduced.

Using big $\mathcal{O}$ notation, Equation (5) can be written

$$\boxed{\left.\frac{\partial \phi}{\partial x}\right|_{x_i} = \frac{\phi_{i+1} - \phi_i}{\Delta x} + \mathcal{O}(\Delta x)} \tag{7}$$

Equation (7) is called the *forward difference* formula for $(\partial\phi/\partial x)_{x_i}$ because it involves nodes x_i and x_{i+1}. The forward difference approximation has a *truncation error* that is $\mathcal{O}(\Delta x)$. The size of the truncation error is (mostly) under our control because we can choose the mesh size Δx. The part of the truncation error that is not under our control is $|\partial\phi/\partial x|_\xi$.

2.4 First Order Backward Difference

An alternative first order finite difference formula is obtained if the Taylor series like that in Equation (4) is written with $\delta x = -\Delta x$. Using the discrete mesh variables in place of all the unknowns, one obtains

$$\phi_{i-1} = \phi_i - \Delta x \left.\frac{\partial \phi}{\partial x}\right|_{x_i} + \frac{\Delta x^2}{2} \left.\frac{\partial^2 \phi}{\partial x^2}\right|_{x_i} - \frac{(\Delta x)^3}{3!} \left.\frac{\partial^3 \phi}{\partial x^3}\right|_{x_i} + \cdots$$

Notice the alternating signs of terms on the right hand side. Solve for $(\partial\phi/\partial x)_{x_i}$ to get

$$\left.\frac{\partial \phi}{\partial x}\right|_{x_i} = \frac{\phi_{i+1} - \phi_i}{\Delta x} + \frac{\Delta x}{2} \left.\frac{\partial^2 \phi}{\partial x^2}\right|_{x_i} - \frac{\Delta x)^2}{3!} \left.\frac{\partial^3 \phi}{\partial x^3}\right|_{x_i} + \cdots$$

Or, using big $\mathcal{O}$ notation

$$\boxed{\left.\frac{\partial \phi}{\partial x}\right|_{x_i} = \frac{\phi_i - \phi_{i-1}}{\Delta x} + \mathcal{O}(\Delta x)} \tag{8}$$

This is called the *backward difference* formula because it involves the values of ϕ at x_i and x_{i-1}.

The order of magnitude of the truncation error for the backward difference approximation is the same as that of the forward difference approximation. Can we obtain a first order difference formula for $(\partial\phi/\partial x)_{x_i}$ with a smaller truncation error? The answer is yes.

2.5 First Order Central Difference

Write the Taylor series expansions for ϕ_{i+1} and ϕ_{i-1}

$$\phi_{i+1} = \phi_i + \Delta x \left.\frac{\partial \phi}{\partial x}\right|_{x_i} + \frac{\Delta x^2}{2} \left.\frac{\partial^2 \phi}{\partial x^2}\right|_{x_i} + \frac{(\Delta x)^3}{3!} \left.\frac{\partial^3 \phi}{\partial x^3}\right|_{x_i} + \cdots \tag{9}$$

$$\phi_{i-1} = \phi_i - \Delta x \left.\frac{\partial \phi}{\partial x}\right|_{x_i} + \frac{\Delta x^2}{2} \left.\frac{\partial^2 \phi}{\partial x^2}\right|_{x_i} - \frac{(\Delta x)^3}{3!} \left.\frac{\partial^3 \phi}{\partial x^3}\right|_{x_i} + \cdots \tag{10}$$

Subtracting Equation (10) from Equation (9) yields

$$\phi_{i+1} - \phi_{i-1} = 2\Delta x \left.\frac{\partial \phi}{\partial x}\right|_{x_i} + \frac{2(\delta x)^3}{3!} \left.\frac{\partial^3 \phi}{\partial x^3}\right|_{x_i} + \cdots$$

Solving for $(\partial\phi/\partial x)_{x_i}$ gives

$$\left.\frac{\partial \phi}{\partial x}\right|_{x_i} = \frac{\phi_{i+1} - \phi_{i-1}}{2\Delta x} - \frac{(\delta x)^3}{3!} \left.\frac{\partial^3 \phi}{\partial x^3}\right|_{x_i} + \cdots$$

or

$$\boxed{\left.\frac{\partial \phi}{\partial x}\right|_{x_i} = \frac{\phi_{i+1} - \phi_{i-1}}{2\Delta x} + \mathcal{O}(\Delta x^2)} \tag{11}$$

This is the *central difference* approximation to $(\partial\phi/\partial x)_{x_i}$. To get good approximations to the continuous problem small Δx is chosen. When $\Delta x \ll 1$,

the truncation error for the central difference approximation goes to zero much faster than the truncation error in Equation (7) or Equation (8).

There is a complication with Equation (11) because it does not include the value for ϕ_i. This may cause problems when the central difference approximation is included in an approximation to a differential equation.

2.6 Second Order Central Difference

Finite difference approximations to higher order derivatives can be obtained with the additional manipulations of the Taylor Series expansion about $\phi(x_i)$. Adding Equation (9) and Equation (10) yields

$$\phi_{i+1} + \phi_{i-1} = 2\phi_i + (\delta x)^2 \left.\frac{\partial^2 \phi}{\partial x^2}\right|_{x_i} + \frac{2(\delta x)^4}{4!} \left.\frac{\partial^4 \phi}{\partial x^4}\right|_{x_i} + \cdots$$

Solving for $(\partial^2\phi/\partial x^2)_{x_i}$ gives

$$\left.\frac{\partial^2 \phi}{\partial x^2}\right|_{x_i} = \frac{\phi_{i+1} - 2\phi_i + \phi_{i+1}}{\Delta x^2} + \frac{(\delta x)^2}{12} \left.\frac{\partial^4 \phi}{\partial x^4}\right|_{x_i} + \cdots$$

Or, using order notation,

$$\boxed{\left.\frac{\partial^2 \phi}{\partial x^2}\right|_{x_i} = \frac{\phi_{i+1} - 2\phi_i + \phi_{i+1}}{\Delta x^2} + \mathcal{O}(\Delta x^2)} \tag{12}$$

This is also called the central difference approximation, but (obviously) it is the approximation to the second derivative, whereas Equation (11) is the central difference approximation to the first derivative.

3 Schemes for the Heat Equation

The finite difference approximations developed in the preceding section are now assembled into a discrete approximation to Equation (1). Both the time and space derivatives are replaced by finite differences. Doing so requires specification both the time and spatial locations of the ϕ values in the finite difference formulas. Therefore, we need to introduce superscript m to designate the time step of the discrete solution. Though this seems like only is a bookkeeping issue, we shall soon see that choosing the time step at which the spatial derivatives are evaluated will have a large impact on the performance and ease of implementation of the finite difference model.

3.1 Forward Time, Centered Space

Approximate the time derivative in Equation (1) with a *forward difference*

$$\left.\frac{\partial \phi}{\partial t}\right|_{t_{m+1}, x_i} = \frac{\phi_i^{m+1} - \phi_i^m}{\Delta t} + \mathcal{O}(\Delta t) \tag{13}$$

Note that terms on the right hand side only involve ϕ at $x = x_i$.

Use the central difference approximation to $(\partial^2\phi/\partial x^2)_{x_i}$ and evaluate all terms at time m.

$$\left.\frac{\partial^2\phi}{\partial x^2}\right|_{x_i} = \frac{\phi_{i-1}^m - 2\phi_i^m + \phi_{i+1}^m}{\Delta x^2} + \mathcal{O}(\Delta x^2). \tag{14}$$

Substitute Equation (13) into the left hand side of Equation (1); substitute Equation (14) into the right hand side of Equation (1); and collect the truncation error terms to get

$$\frac{\phi_i^{m+1} - \phi_i^m}{\Delta t} = \alpha\frac{\phi_{i-1}^m - 2\phi_i^m + \phi_{i+1}^m}{\Delta x^2} + \mathcal{O}(\Delta t) + \mathcal{O}(\Delta x^2) \tag{15}$$

The temporal errors and the spatial errors have different orders. Also notice that we can explicitly solve for ϕ_i^{m+1} in terms of the other values of ϕ. Drop the truncation error terms from Equation (15) and solve for ϕ_i^{m+1} to get

$$\boxed{\phi_i^{m+1} = \phi_i^m + \frac{\alpha\Delta t}{\Delta x^2}(\phi_{i+1}^m - 2\phi_i^m + \phi_{i-1}^m)} \tag{16}$$

Equation (16) is called the *Forward Time, Centered Space* or FTCS approximation to the heat equation. A slight improvement in computational efficiency can be obtained with a small rearrangement of Equation (16)

$$\phi_i^{m+1} = r\phi_{i+1}^m + (1-2r)\phi_i^m + r\phi_{i-1}^m \tag{17}$$

where $r = \alpha\Delta t/\Delta x^2$.

The FTCS scheme is easy to implement because the values of ϕ_i^{m+1} can be updated independently of each other. The entire solution is contained in two loops: an outer loop over all time steps, and an inner loop over all interior nodes. The code fragment in Listing 1 shows how easy it is to implement the FTCS scheme[3].

Equation (16) can be represented by the *computational molecule* or *stencil* in the left third of Figure 3. Notice that the value of ϕ_i^{m+1} does not depend on ϕ_{i-1}^{m+1} or ϕ_{i+1}^{m+1} for the FTCS scheme. Thus, this scheme behaves more like a the solution to a hyperbolic differential equation than a parabolic differential equation.

The solutions to Equation (1) subject to the initial and boundary conditions in Equation (2) are all bounded, decaying functions. In other words, the magnitude of the solution will decrease from the initial condition to a constant. The FTCS can yield unstable solutions that oscillate and grow if Δt is too large. Stable solutions with the FTCS scheme are only obtained if

$$r = \frac{\alpha\Delta t}{\Delta x^2} < \frac{1}{2}. \tag{18}$$

[3]The inner `for` loop could be vectorized by replacing it with

```
u(2:n-1) = r*uold(1:n-2) + r2*uold(2:nx-1) + r*uold(3:nx)
```

Furthermore, since the right hand side of this expression is evaluated in its entirety before assigning it to the left hand side, the `uold` variable can be eliminated completely. Although these code optimizations will reduce the execution time, the less efficient code makes the intent of the loop more clear.

```
% --- Define constants and initial condition
L = ...             %  length of domain in x direction
tmax = ...          %  end time
nx = ...            %  number of nodes in x direction
nt = ...            %  number of time steps
dx = L/(nx-1);
dt = tmax/(nt-1);
r = alpha*dt/dx^2;    r2 = 1 - 2*r;

% --- Loop over time steps
t = 0
u = ...             %   initial condition
for m=1:nt
  uold = u;         %  prepare for next step
  t = t + dt;
  for i=2:nx-1
    u(i) = r*uold(i-1) + r2*uold(i) + r*uold(i+1);
  end
end
```

Listing 1: Code snippet for MATLAB implementation of the FTCS scheme for solution to the heat equation. The `u` and `uold` variables are vectors.

See, e.g.,[3] or [5] for a proof that Equation (18) gives the stability limit for the FTCS scheme.

Finally, we note that Equation (17) can be expressed as a matrix multiplication.

$$\phi^{(m+1)} = A\phi^{(m)}$$

where A is the tridiagonal matrix

$$A = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ r & (1-2r) & r & 0 & 0 & 0 \\ 0 & r & (1-2r) & r & 0 & 0 \\ 0 & 0 & \ddots & \ddots & \ddots & 0 \\ 0 & 0 & 0 & r & (1-2r) & r \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

$\phi^{(m+1)}$ is the vector of ϕ values of at time step $m+1$, and $\phi^{(m)}$ is the vector of ϕ values at time step m. The first and last rows of A are adjusted so that the boundary values of ϕ are not changed when the matrix-vector product is computed.

3.2 Backward Time, Centered Space

In the derivation of Equation (16), the forward difference was used to approximate the time derivative on the left hand side of Equation (1). Now, choose the backward difference,

$$\left.\frac{\partial \phi}{\partial t}\right|_{t_{m+1},x_i} = \frac{\phi_i^m - \phi_i^{m-1}}{\Delta t} + \mathcal{O}(\Delta t) \tag{19}$$

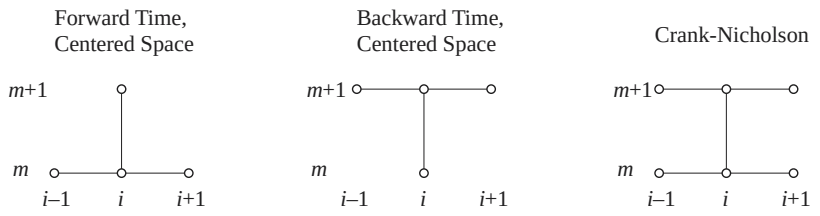

Figure 3: Computational molecules for finite-difference approximations to the heat equation.

Substitute Equation (19) into the left hand side of Equation (1); substitute Equation (14) into the right hand side of Equation (1); and collect the truncation error terms to get

$$\frac{\phi_i^m - \phi_i^{m-1}}{\Delta t} = \alpha \frac{\phi_{i-1}^m - 2\phi_i^m + \phi_{i+1}^m}{\Delta x^2} + \mathcal{O}(\Delta t) + \mathcal{O}(\Delta x^2) \tag{20}$$

The truncation errors in this approximation have the same order of magnitude as the truncation errors in Equation (15). Unlike Equation (15), however, with Equation (20) cannot be rearranged to obtain a simple algebraic formula for computing for ϕ_i^m in terms of its neighbors ϕ_{i+1}^m, ϕ_{i-1}^m, and ϕ_i^{m-1}. Developing a similar equation for ϕ_{i+1}^m (or simply adding 1 to each spatial subscript in Equation (20)) shows that ϕ_{i+1}^m depends on ϕ_{i+2}^m and ϕ_i^m. Thus, Equation (20) is one equation in a *system of equations* for the values of ϕ at the internal nodes of the spatial mesh ($i = 2, 3, \ldots, N-1$).

To see the system of equations more clearly, drop the truncation error terms from Equation (20) and rearrange the resulting equation to get

$$-\frac{\alpha}{\Delta x^2}\phi_{i-1}^m + \left(\frac{1}{\Delta t} + \frac{2\alpha}{\Delta x^2}\right)\phi_i^m - \frac{\alpha}{\Delta x^2}\phi_{i+1}^m = \frac{1}{\Delta t}\phi_i^{m-1} \tag{21}$$

The system of equations can be represented in matrix form as

$$\begin{bmatrix} b_1 & c_1 & 0 & 0 & 0 & 0 \\ a_2 & b_2 & c_2 & 0 & 0 & 0 \\ 0 & a_3 & b_3 & c_3 & 0 & 0 \\ 0 & 0 & \ddots & \ddots & \ddots & 0 \\ 0 & 0 & 0 & a_{N-1} & b_{N-1} & c_{N-1} \\ 0 & 0 & 0 & 0 & a_N & b_N \end{bmatrix} \begin{bmatrix} \phi_1 \\ \phi_2 \\ \phi_3 \\ \vdots \\ \phi_{N-1} \\ \phi_N \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \\ d_3 \\ \vdots \\ d_{N-1} \\ d_N \end{bmatrix} \tag{22}$$

where the coefficients of the interior nodes are

$$\begin{aligned} a_i &= -\alpha/\Delta x^2, & i = 2, 3, \ldots, N-1 \\ b_i &= (1/\Delta t) + (2\alpha/\Delta x^2), \\ c_i &= -\alpha/\Delta x^2, \\ d_i &= (1/\Delta t)\phi_i^{m-1}. \end{aligned}$$

For the Dirichlet boundary conditions in Equation (2)

$$\begin{aligned} b_1 = 1, \quad c_1 = 0, \quad d_1 = \phi_0 \\ a_N = 0, \quad b_N = 1, \quad d_n = \phi_L \end{aligned}$$

Implementation of the BTCS scheme requires solving a system of equations at each time step. In addition to the complication of developing the code, the computational effort per time step for the BTCS scheme is greater than the computational effort per time step of the FTCS scheme[4]. The BTCS scheme has one huge advantage over the FTCS scheme: it is unconditional stable (for solutions to the heat equation). The BTCS scheme is just as accurate as the FTCS scheme. Therefore, with some extra effort, the BTCS scheme yields a computational model that is robust to choices of Δt and Δx. This advantage is not always overwhelming, however, and the FTCS scheme is still useful for some problems.

3.3 Crank-Nicolson

The FTCS and BTCS schemes have a temporal truncation error of $\mathcal{O}(\Delta t)$. When time-accurate solutions are important, the Crank-Nicolson scheme has significant advantages. The Crank-Nicolson scheme is not significantly more difficult to implement than the BTCS scheme, and it has a temporal truncation error that is $\mathcal{O}(\Delta t^2)$. The Crank-Nicolson scheme is implicit, like BTCS, it is also unconditional stable [1, 7, 8].

The left hand side of the heat equation is approximated with the backward time difference used in the BTCS scheme, i.e. Equation (19). The right hand side of the heat equation is approximated with the *average* of the central difference scheme evaluated at the current and the previous time step. Thus, Equation (1) is approximated with

$$\frac{\phi_i^m - \phi_i^{m-1}}{\Delta t} = \frac{\alpha}{2}\left[\frac{\phi_{i-1}^m - 2\phi_i^m + \phi_{i+1}^m}{\Delta x^2} + \frac{\phi_{i-1}^{m-1} - 2\phi_i^{m-1} + \phi_{i+1}^{m-1}}{\Delta x^2}\right] \tag{23}$$

Notice that values of ϕ from time step m and time step $m-1$ appear on the right hand side. Equation (23) is used to predict the values of ϕ at time m, so all values of ϕ at time $m-1$ are assumed to be known. Rearranging Equation (23) so that values of ϕ at time m are on the left, and values of ϕ at time $m-1$ are on the right gives.

$$-\frac{\alpha}{2\Delta x^2}\phi_{i-1}^m + \left(\frac{1}{\Delta t} + \frac{\alpha}{\Delta x^2}\right)\phi_i^m - \frac{\alpha}{2\Delta x^2}\phi_{i+1}^m = \\ +\frac{\alpha}{2\Delta x^2}\phi_{i-1}^{m-1} + \left(\frac{1}{\Delta t} - \frac{\alpha}{\Delta x^2}\right)\phi_i^{m-1} + \frac{\alpha}{2\Delta x^2}\phi_{i+1}^{m-1} \tag{24}$$

The Crank-Nicolson scheme is implicit, and as a result a system of equations for the ϕ must be solved at each time step. The system of equations is identical

[4]For transient problems in one space dimension, the computation cost difference is not too important. For transient problems with two or three space dimensions, however, the computational effort per time step for BTCS is much greater than the computational effort per time step of FTCS. Nonetheless, the superior stability of BTCS usually provides an overall computational advantage.

in form to Equation (22) but with different coefficients:

$$
\begin{aligned}
a_i &= -\alpha/(2\Delta x^2), && i = 2, 3, \ldots, N-1 \\
b_i &= (1/\Delta t) + (\alpha/\Delta x^2), \\
c_i &= -\alpha/(2\Delta x^2), \\
d_i &= (1/\Delta t)\phi_i^{m-1} - a_i\phi_{i-1}^{m-1} + (a_i + c_i)\phi_i^{m-1} - c_i\phi_{i+1}^{m-1}.
\end{aligned}
$$

Note that the a_i, b_i, and c_i for the Crank-Nicolson scheme differ from their counterparts in the BTCS scheme by a factor of two. Apart from this minor difference, the only significant implementation difference between the BTCS and Crank-Nicolson scheme is the appearance of the extra ϕ^{m-1} terms in the d_i.

Algorithmically, the BTCS and Crank-Nicholson scheme are very similar. The Crank-Nicolson scheme has a truncation error of $\mathcal{O}(\Delta t^2)+\mathcal{O}(\Delta x^2)$, i.e., the temporal truncation error is significantly smaller than the temporal truncation error of the BTCS scheme.

4 Implementation and Verification

MATLAB versions of the FTCS, BTCS, and Crank-Nicolson schemes are presented and demonstrated in this section.

4.1 Test Problem

The finite difference codes are tested by solving the heat equation with boundary conditions $\phi(0,t) = \phi(L,t) = 0$, and initial condition $f_0(x) = \sin(\pi x/L)$. The exact solution for this problem (which is derived in Appendix B) is

$$\phi(x,t) = \sin\left(\frac{\pi x}{L}\right)\exp\left(-\frac{\alpha\pi^2 t}{L^2}\right).$$

4.2 Implementation

The FTCS and BTCS schemes are implemented in the MATLAB functions `heatFTCS` and `heatBTCS` in Listing 2 and Listing 3, respectively. The system of equations for the BTCS method is solved with the `tridiagLU` and `tridiagLUsolve` functions in Listing 6 and Listing 7. Appendix C gives a brief derivation of the solution method for a tridiagonal system of equations.

Running `heatFTCS` with the default parameters gives

```
>> heatFTCS

Norm of error =   2.404e-002 at t =     0.500
    dt, dx, r =   5.556e-002    5.263e-002       2.006
```

and the plot in Figure 4. Note that the plot only shows the *last* time step in the solution for $\phi(x,t)$.

Similar results are obtained with the heatBTCS and `heatFTCS` codes

```
>> heatBTCS

Norm of error =   2.676e-002 at t =     0.500
```

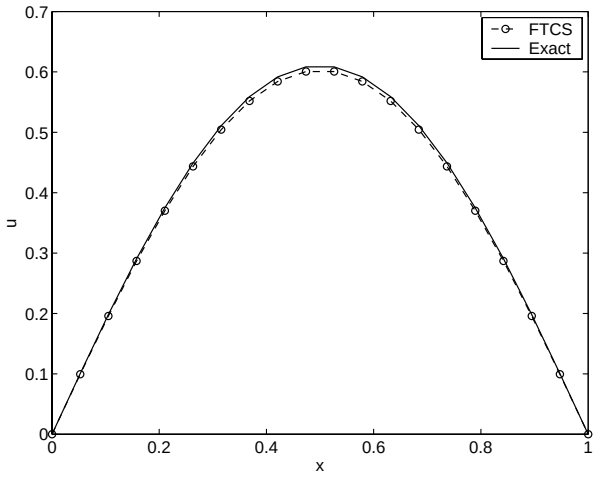

Figure 4: FTCS solution to the heat equation at $t = 0.5$ obtained with $r = 2$. The solution seems to be stable, but only because the duration of the simulation is not long enough for the instability to become apparent.

```
    dt, dx, r =   5.556e-002    5.263e-002      2.006

>> heatCN

Norm of error =   1.883e-003 at t =    0.500
    dt, dx, r =   5.556e-002    5.263e-002      2.006
```

Note that the L_2 norm of the error for the FTCS scheme and BTCS scheme are comparable: 0.024 and 0.027. The error of the Crank-Nicolson scheme is 0.0018, which is more than an order of magnitude smaller than either the FTCS and BTCS schemes. In § 4.4 below, the variation of the errors with Δx and Δt is explored.

4.3 Stability

The heat equation is a model of diffusive systems. For all problems with constant boundary values, the solutions of the heat equation decay from an initial state to a non-varying steady state condition. The transient behavior of these solutions are smooth and bounded: the solution does not develop local or global maxima that are outside the range of the initial data.

In section 3.1 it was asserted that the FTCS scheme can exhibit instability. An unstable numerical solution is one in which the values of ϕ at the nodes may oscillate or the magnitude grow outside the bounds of the initial and boundary conditions. According to Equation (18) the solution in Figure 4 should be unstable because the value of $r = 2$ for this simulation, an r value that is significantly higher than the stability limit of 0.5. The FTCS scheme *is* unstable for $r > 0.5$. By increasing the length of the simulation time to 1.0 (and correspondingly increasing `nt` to maintain the same value of Δt), the unstable behavior is made apparent. In the next MATLAB session, the values of `nt` and `tmax` are

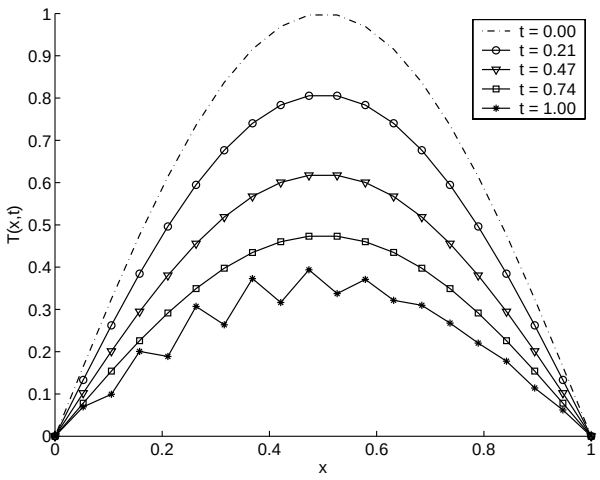

Figure 5: FTCS solution to the heat equation at $t = 1$ obtained with $r = 2$. The instability in the solution is now obvious.

both increased by a factor of two, and the solution from the `heatFTCS` is saved in the `x`, `t`, and `T` variables. function

```
>> [err,x,t,Phi] = heatFTCS(20,20,0.1,1,1);

Norm of error =   1.069e-001 at t =    1.000
    dt, dx, r =   5.263e-002    5.263e-002      1.900
```

The error at the end of the simulation has increased significantly. The `showTheat` function can be used to plot the `T` fields at selected time steps.

```
>> showTheat(x,t([1 5 10 15 20]),Phi(:,[1 5 10 15 20]) )
```

The expression `t([1 5 10 15 20])` selects the values of t_i for $i = 1, 5, 10, 15, 20$. The expression `Phi(:,[1 5 10 15 20])` selects columns 1, 5, 10, 15, and 20 from the `Phi` matrix.

The plot created by `showTheat` is shown in Figure 5. Note that the solution appears smooth as late as $t = 0.74$. However, by $t = 1$ significant oscillation in the solution is evidence of instability in the FTCS scheme.

In contrast to the FTCS scheme, the BTCS and Crank-Nicholson schemes are unconditionally stable. Repeating the preceding calculations with the BTCS scheme gives this text output

```
>> [err,x,t,Phi] = heatBTCS(20,20,0.1,1,1);

Norm of error =   3.134e-002 at t =    1.000
    dt, dx, r =   5.263e-002    5.263e-002      1.900
```

To view curves of $\phi(x, t)$ use the `showTheat` command

```
>> showTheat(x,t([1 5 10 15 20]),Phi(:,[1 5 10 15 20]) )
```

which produces the plot in Figure 6. Repeating the calculations with the Crank-Nicolson scheme would give qualitatively similar results, but with much greater absolute accuracy (smaller errors).

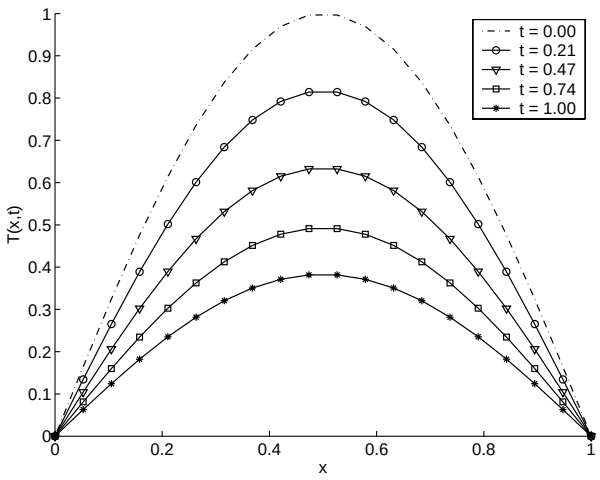

Figure 6: Stable BTCS solution to the heat equation at $t = 1$ obtained with $r = 2$.

4.4 Verifying Truncation Error Estimates

The truncation error for the FTCS or BTCS schemes is $\mathcal{O}(\Delta t) + \mathcal{O}(\Delta x^2)$. The big $\mathcal{O}$ notation expresses the *rate* at which the truncation error goes to zero.

Usually we are only interested in the order of magnitude of the truncation error. For code validation, however, we need to work with the magnitude of the truncation error. Let TE denote the true magnitude of the truncation error for a given Δt, Δx, and other problem parameters (α, L, f_0, etc.). As $\Delta t \to 0$ and $\Delta x \to 0$, the true magnitude of the truncation error is

$$\mathrm{TE} = K_t \Delta t + K_x \Delta x^2 \tag{25}$$

where K_t and K_x are constants that depend on the accuracy of the finite difference approximations and the problem being solved[5]. To make TE arbitrarily small, both Δt and Δx must approach zero.

To verify that a FTCS or BTCS code is working properly, we wish to determine whether a linear reduction in Δt causes a linear reduction in TE. Similarly, we wish to determine whether a linear reduction in Δx causes a quadratic reduction in TE. To perform these tests, we must vary only Δt or Δx. The Crank-Nicolson scheme should show a quadratic reduction in TE with Δt.

Suppose two numerical solutions are obtained: one with Δt_1 and Δx_1, and the other with Δt_2 and Δx_2. The ratio of truncation errors for the two solutions is

$$\frac{\mathrm{TE}_2}{\mathrm{TE}_1} = \frac{K_t \Delta t_2 + K_x \Delta x_2^2}{K_t \Delta t_1 + K_x \Delta x_1^2} \tag{26}$$

To measure the dependence of TE on Δt, choose a small value of Δx, and hold it constant as Δt is systematically reduced. If Δx is small enough, i.e. if

[5]The K_t and K_x for FTCS are different from the K_t and K_x for BTCS.

$K_x \Delta x^2 \ll K_t \Delta t_1$ and $K_x \Delta x^2 \ll K_t \Delta t_2$ then

$$\left.\frac{\mathrm{TE}_2}{\mathrm{TE}_1}\right|_{\Delta x = \text{ constant}} \approx \frac{K_t \Delta t_2}{K_t \Delta t_1} = \frac{\Delta t_2}{\Delta t_1} \tag{27}$$

Given the numerical solution ϕ_i^m, $i = 1, \ldots, N$ at time step m, one scalar measure of the truncation error as

$$e = \|\phi_i^m - \phi(x_i, t_m)\|_2. \tag{28}$$

One could also use the L_1 or L_∞ norms. The value of e is used to verify that the FTCS and BTCS solutions have truncation errors that decrease as $\mathcal{O}(\Delta t) + \mathcal{O}(\Delta x^2)$. In other words, e can be used as TE in Equations (25), (26), or (27).

4.5 Truncation Error Measurements

Figure 7 shows how the truncation error of the FTCS and BTCS codes varies when Δt is reduced and Δx is fixed. The solutions were obtained for $\alpha = 0.1$, $L = 1$, $t_{\max} = 0.5$. The dashed line shows $e \sim \Delta t$, which is the theoretically predicted variation of the truncation error with Δt for the FTCS and BTCS schemes. The BTCS scheme can obtain solutions for much larger Δt than the FTCS scheme because the BTCS scheme is unconditionally stable. For $\Delta x = 3.448 \times 10^{-2}$ and $\alpha = 0.1$, the stability limit for the FTCS scheme is $\Delta t = 5.94 \times 10^{-3}$. As $\Delta t \to 0$, the truncation errors for the FTCS and BTCS schemes approach constant values. Further reduction in Δt do not reduce the error because the contribution of the spatial error is fixed (when Δx is fixed). In other words, as $\Delta t \to 0$, Equation (27) does not hold because the constant $K_x \Delta x$ terms in the numerator and denominator of Equation (26) are not negligible. Thus, as $\Delta t \to 0$ for finite Δx, the TE of BTCS and FTCS approach constants. For intermediate Δt, Equation (27) holds.

Figure 8 shows that the dependence of the truncation error for both schemes is nearly identical except at very small Δx. As Δx becomes very small, the $K_t \Delta t$ terms in the numerator and denominator of Equation (26) become important.

References

[1] William F. Ames. *Numerical Methods for Partial Differential Equations.* Academic Press, Inc., Boston, third edition, 1992.

[2] Jeffery Cooper. *Introductin to Partial Differential Equations with* MATLAB. Birkhäuser, Boston, 1998.

[3] K.W. Morton and D.F. Mayers. *Numerical Solution of Partial Differential Equations: An Introduction.* Cambridge University Press, Cambridge, England, 1994.

[4] Clive A.J. Fletcher. *Computational Techniquess for Fluid Dynamics.* Springer-Verlag, Berlin, 1988.

[5] Gene Golub and James M. Ortega. *Scientific Computing: An Introduction with Parallel Computing.* Academic Press, Inc., Boston, 1993.

[6] Joe D. Hoffman. *Numerical Methods for Engineers and Scientists*. McGraw-Hill, New York, 1992.

[7] R. L. Burden and J. D. Faires. *Numerical Analysis*. Brooks/Cole Publishing Co., New York, sixth edition, 1997.

[8] Eugene Isaacson and Herbert Bishop Keller. *Analysis of Numerical Methods*. Dover, New York, 1994.

[9] Erwin Kreyszig. *Advanced Engineering Mathematics*. Wiley, New York, seventh edition, 1993.

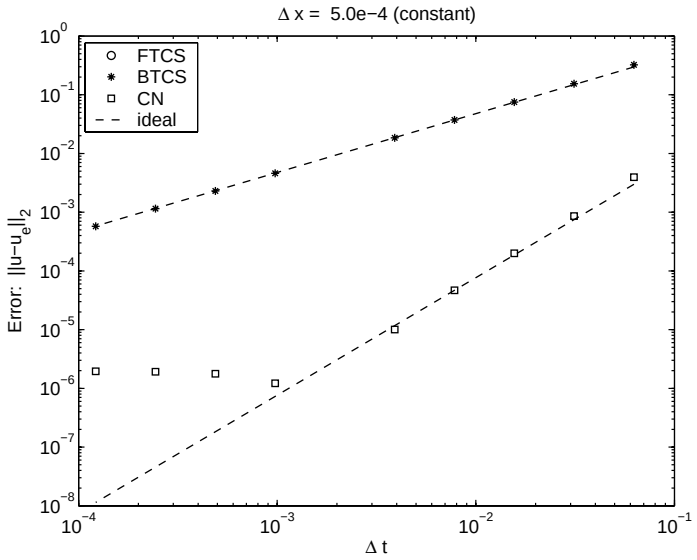

Figure 7: Comparison of truncation errors for the FTCS, BTCS, and Crank-Nicolson schemes as a function of Δt for fixed Δx.

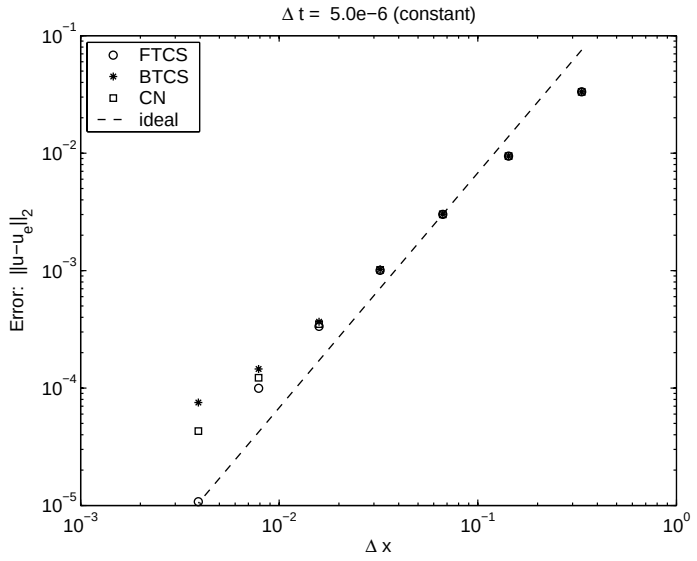

Figure 8: Comparison of truncation errors for FTCS, BTCS, and Crank-Nicolson schemes as a function of Δx for fixed Δt.

Appendix A: Order Notation

The formal definition of $\mathcal{O}(\ )$ is as follows. We say that some function $f(s)$ is $\mathcal{O}(\psi(s))$ for $s \in \Omega$ if

$$|f(s)| \leq C|\psi(s)| \quad \forall\ s \in \Omega$$

Suppose for concreteness that we have an approximation with a truncation error that is $\mathcal{O}(\Delta x^2)$. By this we mean that the upper limit for the truncation error is $C\Delta x^2$ where C is an unknown constant.

The value of C does not need to be known for us to make good use of the truncation error estimate. For the particular case where the truncation error is $\mathcal{O}(\Delta x^2)$ we have

$$\frac{\text{Error for } \Delta x_2}{\text{Error for } \Delta x_1} = \frac{C|(\Delta x_2)^2|}{C|(\Delta x_1)^2|} = \left[\frac{\Delta x_2}{\Delta x_1}\right]^2$$

So as long as we work with the ratio of errors, the value of the constant does not matter.

Appendix B: Exact Solution

The general solution of Equation (1) with the boundary and initial conditions

$$\phi(0,t) = \phi(L,t) = 0, \qquad \phi(x,0) = f_0(x) \tag{29}$$

is (See e.g., Kreyszig [9, §11.5].)

$$\phi(x,t) = \sum_{n=1}^{\infty} b_n \sin\left(\frac{n\pi x}{L}\right) \exp\left(-\frac{\alpha n^2 \pi^2 t}{L^2}\right) \tag{30}$$

where the b_n are obtained from

$$b_n = \frac{2}{L}\int_0^L f_0(x) \sin\left(\frac{n\pi x}{L}\right)\, dx \tag{31}$$

Example: $f_0(x) = \sin(\pi x/L)$.
Compute

$$b_n = \frac{2}{L}\int_0^L \sin\left(\frac{\pi x}{L}\right) \sin\left(\frac{n\pi x}{L}\right)\, dx$$

but

$$\int_0^L \sin\left(\frac{\pi x}{L}\right) \sin\left(\frac{n\pi x}{L}\right)\, dx = \begin{cases} L/2 & \text{if } n = 1 \\ 0 & \text{otherwise} \end{cases}$$

Therefore,

$$\begin{aligned} b_1 &= 1, \\ b_n &= 0, \qquad n = 2, 3, \ldots \end{aligned}$$

The exact solution to Equation (1) subject to initial and bound conditions in Equation (29), and with $f_0(x) = \sin((\pi x)/(L))$ is

$$\phi(x,t) = \sin\left(\frac{\pi x}{L}\right) \exp\left(-\frac{\alpha \pi^2 t}{L^2}\right).$$

Appendix C: Solving a Tridiagonal System

The system of equations for the BTCS scheme can be represented in matrix form as $Av = d$ where

$$A = \begin{bmatrix} b_1 & c_1 & 0 & 0 & 0 & 0 \\ a_2 & b_2 & c_2 & 0 & 0 & 0 \\ 0 & a_3 & b_3 & c_3 & 0 & 0 \\ 0 & 0 & \ddots & \ddots & \ddots & 0 \\ 0 & 0 & 0 & a_{N-1} & b_{N-1} & c_{N-1} \\ 0 & 0 & 0 & 0 & a_N & b_N \end{bmatrix}, \quad v = \begin{bmatrix} \phi_1 \\ \phi_2 \\ \phi_3 \\ \vdots \\ \phi_{N-1} \\ \phi_N \end{bmatrix}, \quad d = \begin{bmatrix} d_1 \\ d_2 \\ d_3 \\ \vdots \\ d_{N-1} \\ d_N \end{bmatrix}$$

This system can be efficiently solved using LU factorization with backward substitution. Since the coefficient matrix is known to by positive definite (and symmetric in the case of conduction or diffusion without convection), we can use LU factorization without pivoting.

If the coefficient matrix is A, we wish to find L and U such that $A = LU$. It turns out that L and U have the form

$$L = \begin{bmatrix} e_1 & & & & & \\ a_2 & e_2 & & & & \\ & a_3 & e_3 & & & \\ & & \ddots & \ddots & & \\ & & & a_{n-1} & e_{n-1} & \\ & & & & a_n & e_n \end{bmatrix} \quad U = \begin{bmatrix} 1 & f_1 & & & & \\ & 1 & f_2 & & & \\ & & 1 & f_3 & & \\ & & & \ddots & \ddots & \\ & & & & 1 & f_{n-1} \\ & & & & & 1 \end{bmatrix}$$

Evaluating each nonzero term in the product LU and setting it equal to the corresponding entry in A gives

$$\begin{aligned} e_1 &= b_1 \\ e_1 f_1 &= c_1 \\ \\ a_2 &= a_2 \\ a_2 f_1 + e_2 &= b_2 \\ e_2 f_2 &= c_2 \\ \\ a_i &= a_i \\ a_i f_{i-1} + e_i &= b_i \\ e_i f_i &= c_i \\ \vdots &\quad \vdots \\ a_n &= a_n \\ a_n f_{n-1} + e_n &= b_n \end{aligned}$$

Solving for the unknown e_i and f_i gives

$$f_1 = c_1/e_1 = c_1/b_1$$

$$e_2 = b_2 - a_2/f_1$$
$$f_2 = c_2/e_2$$

$$e_i = b_i - a_i f_{i-1}$$
$$f_i = c_i/e_i$$

$$e_n = b_n - a_n f_{n-1}$$

The equivalent MATLAB code is (with `a`, `b`, `c` given, and `n` = `length(a)` = `length(b)` = `length(c)`)

```
e(1) = b(1);
f(1) = c(1)/b(1);
for i=2:n
  e(i) = b(i) - a(i)*f(i-1);
  f(i) = c(i)/e(i);
end
```

Since $A = LU$, the system $Av = d$ is equivalent to $(LU)v = d$ or $L(Uv) = d$. Introducing the w vector defined as $w = Uv$ the system of equations becomes $Lw = d$. Since L is a lower triangular matrix, w can easily[6] be obtained by solving $Lw = d$. Then with w known, v is computed (again, easily because U is triangular) by solving $Uv = w$. Thus, once L and U have been found, the v vector can be computed in the two step process

$$\begin{aligned} \text{solve} \quad & Lw = d \\ \text{solve} \quad & Uv = w \end{aligned}$$

Since L is lower triangular, the first solve is a forward substitution. Since U is upper triangular, the second solve is a backward substitution.

Applying the BTCS scheme to (the constant coefficient) heat equation yields a coefficient matrix that does not change from time step to time step. Thus, the L and U factors need to be obtained only once at the beginning of the simulation. The triangular solves are performed only once per time step. Given the a, e, and f vectors that define the L and U matrices, the solution algorithm can be embodied in two loops.

```
% --- Forward substitution to solve L*w = d
w(1) = d(1)/e(1);
for i=2:n
  w(i) = (d(i) - a(i)*w(i-1))/e(i);
end

% --- Backward substitution to solve U*v = w
v(n) = w(n);
for i=n-1:-1:1
  v(i) = w(i) - f(i)*w(i+1);
end
```

[6] Recall that the solution of a lower triangular systems is easy to compute because it only requires a forward substitution loop.

Careful examination of the two preceding loops shows that `v` is used only after `w` is created, and that `v(i)` is obtained only after `v(i+1)` is found. These facts enable us to eliminate the `w` vector entirely and use `v` in its place. This saves us the task of creating the temporary `w` vector. The revised algorithm is in Listing 7.

Appendix D: Code Listings

`heatBTCS`	Use the BTCS scheme to solve the problem defined in § 4.1.
`heatCN`	Use the Crank-Nicolson scheme to solve the problem defined in § 4.1.
`heatFTCS`	Use the FTCS scheme to solve the problem defined in § 4.1.
`showTheat`	Plot the results of solving the heat equation with either `heatFTCS`, `heatBTCS` or `heatCN`.
`tridiagLU`	Obtain the LU factorization of a tridiagonal system of equations if the coefficient matrix that is positive definite.
`tridiagLUsolve`	Solve a tridiagonal system of equations when the LU factorization of that system is available. `tridiagLUsolve` is used by `heatBTCS`.

```
function [errout,xo,to,Uo] = heatFTCS(nt,nx,alpha,L,tmax,errPlots)
% heatBTCS  Solve 1D heat equation with the BTCS scheme
%
% Synopsis:  heatFTCS
%            heatFTCS(nt)
%            heatFTCS(nt,nx)
%            heatFTCS(nt,nx,alpha)
%            heatFTCS(nt,nx,alpha,L)
%            heatFTCS(nt,nx,alpha,L,tmax)
%            heatFTCS(nt,nx,alpha,L,tmax,errPlots)
%            err = heatFTCS(...)
%            [err,x,t,U] = heatFTCS(...)
%
% Input:  nt = number of steps.  Default:  nt = 10;
%         nx = number of mesh points in x direction.  Default: nx=20
%         alpha = diffusivity.  Default:  alpha = 0.1
%         L = length of the domain.  Default:  L = 1;
%         tmax = maximum time for the simulation.  Default:  tmax = 0.5
%         errPlots = flag (1 or 0) to control whether plots of the solution
%                    and the error at the last time step are created.
%                    Default:  errPlots = 1 (make the plots)
%
% Output: err = L2 norm of error evaluated at the spatial nodes on last time step
%         x = location of finite difference nodes
%         t = values of time at which solution is obtained (time nodes)
%         U = matrix of solutions:  U(:,j) is U(x) at t = t(j)

if nargin<1, nt = 10;     end
if nargin<2, nx = 20;     end
if nargin<3, alpha = 0.1; end
if nargin<4, L = 1;       end
if nargin<5, tmax = 0.5;  end
if nargin<6, errPlots=1;  end

% --- Compute mesh spacing and time step
dx = L/(nx-1);
dt = tmax/(nt-1);
r = alpha*dt/dx^2;    r2 = 1 - 2*r;

% --- Create arrays to save data for export
x = linspace(0,L,nx)';
t = linspace(0,tmax,nt);
U = zeros(nx,nt);

% --- Set IC and BC
U(:,1) = sin(pi*x/L);   %  implies u0 = 0;  uL = 0;
u0 = 0;   uL = 0;       %  needed to apply BC inside time step loop

% --- Loop over time steps
for m=2:nt
  for i=2:nx-1
    U(i,m) = r*U(i-1,m-1) + r2*U(i,m-1) + r*U(i+1,m-1);
  end
end

% --- Compare with exact solution at end of simulation
ue = sin(pi*x/L)*exp(-t(nt)*alpha*(pi/L)^2);
err = norm(U(:,nt)-ue);

% --- Set values of optional output variables
if nargout>0,  errout = err;              end
if nargout>1,  xo = x;  to = t;  Uo = U;  end

% --- Plot error in solution at the last time step
if ~errPlots,  return;  end
fprintf('\nNorm of error = %12.3e at t = %8.3f\n',err,t(nt));
fprintf('\tdt, dx, r = %12.3e  %12.3e   %8.3f\n',dt,dx,r);
figure;  plot(x,U(:,nt),'o--',x,ue,'-');   xlabel('x');  ylabel('u');
legend('FTCS','Exact');
figure;  plot(x,U(:,nt)-ue,'o--');         xlabel('x');  ylabel('u - u_e');
```

Listing 2: The `heatFTCS` function solves the one-dimensional heat equation with the FTCS scheme.

```
function [errout,xo,to,Uo] = heatBTCS(nt,nx,alpha,L,tmax,errPlots)
% heatBTCS  Solve 1D heat equation with the BTCS scheme
%
% Synopsis:  heatBTCS
%            heatBTCS(nt)
%            heatBTCS(nt,nx)
%            heatBTCS(nt,nx,alpha)
%            heatBTCS(nt,nx,alpha,L)
%            heatBTCS(nt,nx,alpha,L,tmax)
%            heatBTCS(nt,nx,alpha,L,tmax,errPlots)
%            err = heatBTCS(...)
%            [err,x,t,U] = heatBTCS(...)
%
% Input:  nt = number of steps.  Default:  nt = 10;
%         nx = number of mesh points in x direction.  Default: nx=20
%         alpha = diffusion coefficient.  Default:  alpha = 0.1
%         L = length of the domain.  Default:  L = 1;
%         tmax = maximum time for the simulation.  Default:  tmax = 0.5
%         errPlots = flag (1 or 0) to control whether plots of the solution
%                    and the error at the last time step are created.
%                    Default:  errPlots = 1 (make the plots)
%
% Output: err = L2 norm of error evaluated at the spatial nodes on last time step
%         x = location of finite difference nodes
%         t = values of time at which solution is obtained (time nodes)
%         U = matrix of solutions:  U(:,j) is U(x) at t = t(j)

if nargin<1, nt = 10;     end
if nargin<2, nx = 20;     end
if nargin<3, alpha = 0.1; end   %  diffusivity
if nargin<4, L = 1;       end
if nargin<5, tmax = 0.5;  end
if nargin<6, errPlots=1;  end   %  flag: no error plots if errPlots=0

% --- Compute mesh spacing and time step
dx = L/(nx-1);
dt = tmax/(nt-1);

% --- Create arrays to save data for export
x = linspace(0,L,nx)';
t = linspace(0,tmax,nt);
U = zeros(nx,nt);

% --- Set IC and BC
U(:,1) = sin(pi*x/L);   %  implies u0 = 0;  uL = 0;
u0 = 0;   uL = 0;

% --- Coefficients of the tridiagonal system
a = (-alpha/dx^2)*ones(nx,1);     %  subdiagonal a: coefficients of phi(i-1)
c = a;                            %  superdiagonal c: coefficients of phi(i+1)
b = (1/dt)*ones(nx,1) - 2*a;      %  diagonal b: coefficients of phi(i)
b(1) = 1;     c(1) = 0;           %  Fix coefficients of boundary nodes
b(end) = 1;   a(end) = 0;
[e,f] = tridiagLU(a,b,c);         %  Get LU factorization of coefficient matrix

% --- Loop over time steps
for m=2:nt
  d = U(:,m-1)/dt;                            %  update right hand side
  d(1) = u0;  d(end) = uL;                    %  overwrite BC values
  U(:,m) = tridiagLUSolve(d,a,e,f,U(:,m-1));   %  solve the system
end

% --- Compare with exact solution at end of simulation

% ... Remainder of code is the same as heatFTCS
```

Listing 3: The `heatBTCS` function solves the one-dimensional heat equation with the BTCS scheme.

```
function [errout,xo,to,Uo] = heatCN(nt,nx,alpha,L,tmax,errPlots)
% heatBTCS  Solve 1D heat equation with the Crank-Nicolson scheme
%
% Synopsis:  heatCN
%            heatCN(nt)
%            heatCN(nt,nx)
%            heatCN(nt,nx,alpha)
%            heatCN(nt,nx,alpha,L)
%            heatCN(nt,nx,alpha,L,tmax)
%            heatCN(nt,nx,alpha,L,tmax,errPlots)
%            err = heatCN(...)
%            [err,x,t,U] = heatCN(...)
%
% Input:  nt = number of steps.  Default:  nt = 10;
%         nx = number of mesh points in x direction.  Default: nx=20
%         alpha = diffusion coefficient.  Default:  alpha = 0.1
%         L = length of the domain.  Default:  L = 1;
%         tmax = maximum time for the simulation.  Default:  tmax = 0.5
%         errPlots = flag (1 or 0) to control whether plots of the solution
%                    and the error at the last time step are created.
%                    Default:  errPlots = 1 (make the plots)
%
% Output: err = L2 norm of error evaluated at the spatial nodes on last time step
%         x = location of finite difference nodes
%         t = values of time at which solution is obtained (time nodes)
%         U = matrix of solutions:  U(:,j) is U(x) at t = t(j)

if nargin<1, nt = 10;     end
if nargin<2, nx = 20;     end
if nargin<3, alpha = 0.1; end   %  diffusivity
if nargin<4, L = 1;       end
if nargin<5, tmax = 0.5;  end
if nargin<6, errPlots=1;  end   %  flag: no error plots if errPlots=0

% --- Compute mesh spacing and time step
dx = L/(nx-1);
dt = tmax/(nt-1);

% --- Create arrays to save data for export
x = linspace(0,L,nx)';
t = linspace(0,tmax,nt);
U = zeros(nx,nt);

% --- Set IC and BC
U(:,1) = sin(pi*x/L);   %  implies u0 = 0;  uL = 0;
u0 = 0;   uL = 0;       %  needed to apply BC inside time step loop

% --- Coefficients of the tridiagonal system
a = (-alpha/2/dx^2)*ones(nx,1);     %  subdiagonal a: coefficients of phi(i-1)
c = a;                              %  superdiagonal c: coefficients of phi(i+1)
b = (1/dt)*ones(nx,1) - (a+c);      %  diagonal b: coefficients of phi(i)
b(1) = 1;     c(1) = 0;             %  Fix coefficients of boundary nodes
b(end) = 1;   a(end) = 0;
[e,f] = tridiagLU(a,b,c);           %  Get LU factorization of coefficient matrix

% --- Loop over time steps
for m=2:nt
  %  Right hand side includes time derivative and CN terms
  d = U(:,m-1)/dt - [0; a(2:end-1).*U(1:end-2,m-1); 0] ...
                  + [0; (a(2:end-1)+c(2:end-1)).*U(2:end-1,m-1); 0] ...
                  - [0; c(2:end-1).*U(3:end,m-1);   0];
  d(1) = u0;  d(end) = uL;                  %  overwrite BC values
  U(:,m) = tridiagLUSolve(d,a,e,f,U(:,m-1)); %  solve the system
end

% --- Compare with exact solution at end of simulation

% ... Remainder of code is the same as heatFTCS
```

Listing 4: The `heatCN` function solves the one-dimensional heat equation with the Crank-Nicolson scheme.

```
function showTheat(x,t,T,pflag)
% showTheat  Plot and print solutions to the heat equation
%
% Synopsis:  showTheat(x,t,T)
%            showTheat(x,t,T,pflag)
%
%  Input:  x = vector of positions at which temperature is known
%          t = vector of times at which solution is to be evaluated
%          T = matrix of T(x,t) values.  T(:,t(j)) is T(x) at time t(j)
%          pflag = flag to control printing of results.
%                  Default:  pflag = 0, do not print results

% --- Define string matrix of line styles that can be reused.
%     nsymb is the total number of line style strings (plot symbols)
%     In plot loop the statement isymb = 1 + rem(j-1,nsymb) is an
%     index in the range 1 <= isymb <= nsymb
lineSymb = strvcat('b-.','k-o','m-v','b-s','k*-','m-d','b-+','k-<','m-h');
nsymb = size(lineSymb,1);

% --- Plot T(x,t):  each time is a different curve with different symbol
for j=1:length(t)
  hold on
  isymb = 1 + rem(j-1,nsymb);        %  cyclic index for line styles.
  plot(x,T(:,j),lineSymb(isymb,:));
  % --- Build string matrix for legend.  Each row is a legend string.
  s = sprintf('t = %-4.2f',t(j));
  if j==1
    legstr = s;
  else
    legstr = strvcat(legstr,s);
  end
end
hold off

legend(legstr,2);   xlabel('x');   ylabel('T(x,t)');
```

Listing 5: The `showTheat` plots the $\phi(x,t)$ curves from solutions of the heat equation.

```
function [e,f] = tridiagLU(a,b,c)
% tridiagLU  Obtain the LU factorization of a tridiagonal matrix
%
% Synopsis:  [e,f] = tridiag(a,b,c)
%
% Input:  a,b,c = vectors defining the tridiagonal matrix.  a is the
%                 subdiagonal, b is the main diagonal, and c is the superdiagonal
%
% Output:  e,f = vectors defining the L and U factors of the tridiagonal matrix

n = length(a);
e = zeros(n,1);  f = e;

e(1) = b(1);
f(1) = c(1)/b(1);
for i=2:n
    e(i) = b(i) - a(i)*f(i-1);
    f(i) = c(i)/e(i);
end
```

Listing 6: The `tridiagLU` function obtains the LU factorization of a tridiagonal system of equations if the coefficient matrix is positive definite.

```
function v = tridiagLUsolve(d,a,e,f,v)
% tridiagLUsolve  Solve (LU)*v = d where L and U are LU factors of a tridiagonal
%                 matrix
%
% Synopsis:  v = tridiagLUsolve(d,e,f)
%            v = tridiagLUsolve(d,e,f,v)
%
% Input:   d = right hand side vector of the system of equatoins
%        e,f = vectors defining the L and U factors of the tridiagonal matrix.
%              e and f are obtained with the tridiagLU function
%          v = solution vector.  If v is supplied, the elements of v are over-
%              written (thereby saving the memory allocation step).  If v is not
%              supplied, it is created.  v is used as a scratch vector in the
%              forward solve.
%
% Output: v = solution vector

n = length(d);
if nargin<5,  v = zeros(n,1);  end

% --- Forward substitution to solve L*w = d
v(1) = d(1)/e(1);
for i=2:n
  v(i) = (d(i) - a(i)*v(i-1))/e(i);
end

% --- Backward substitution to solve U*v = w
for i=n-1:-1:1
  v(i) = v(i) - f(i)*v(i+1);
end
```

Listing 7: The `tridiagLUSolve` function solves a tridiagonal system of equations when the LU factorization of that system is available.