Journal of Computational and Applied Mathematics 39 (1992) 213–225
North-Holland

Fast direct solvers for the heat equation in simple geometries

James F. Epperson

Department of Mathematical Sciences, University of Alabama in Huntsville, Huntsville, AL 35899, United States

Received 10 August 1990
Revised 5 February 1991

Abstract

Epperson, J.F., Fast direct solvers for the heat equation in simple geometries, Journal of Computational and Applied Mathematics 39 (1992) 213–225.

In this note we present a method for reducing a multi-dimensional heat equation on a rectangle to a family of one-dimensional heat equations, thus raising the possibility of substantially lowering the computational cost. The method is shown to be stable and examples from $\mathbb{R}^2$ and $\mathbb{R}^3$ are presented, along with execution times. Operation count and storage estimates are also given.

Keywords: Parabolic equations, numerical methods, Green's functions, ADI methods.

1. Introduction

Consider the heat equation with Dirichlet data, posed on the unit square:

$$\begin{aligned} &u_t = \Delta u + f(x, y, t), && (x, y) \in R = (0, 1) \times (0, 1), \\ &u(x, y, t) = 0, && (x, y) \in \partial R, \\ &u(x, y, 0) = u_0(x, y). \end{aligned} \tag{P}$$

In this note we present a method for solving (P), which requires the factorization of only a single tridiagonal matrix. The method is stable and extends easily to $\mathbb{R}^3$. No FFTs are required, and no iteration is performed. The algorithm appears to be very amenable to vector or parallel implementation. If the domain is rectangular but not square, or if different diffusion coefficients are used on u_{xx} and u_{yy}, then we are still able to apply the algorithm but we now need to solve two tridiagonal systems.

This work is based on the use of heat kernels to partially invert the PDE in (P) before discretization. For earlier work along these lines, see [4,5]; of related interest is the computational Green's function method [3,6,8]. It appears that the method here can be extended to more general operators and domains; see [7] for a more complete study along these lines.

The present paper is organized as follows. The basic algorithm is described in Section 2 with examples and extensions, including timing tests and 3D versions, being given in Section 3. In Section 4 we present rough operation and memory costs for each algorithm, followed by two appendices giving the proof of a technical lemma needed for stability of the 3D algorithm and details of the comparison programs from Section 3. Finally, there is a brief remark on the extension of these ideas to problems involving more general operators on more general domains.

2. Derivation of the algorithm

We begin by treating the u_{yy} term in (P) as an additional source term to derive a one-parameter family of heat equations in x, only:

$$\begin{aligned} &u_t = u_{xx} + f + Y, && 0 < x < 1, \text{ each } y \in (0, 1), \\ &u(0, y, t) = u(1, y, t) = 0, && \text{each } y \in (0, 1), \\ &u(x, y, t) = u_0(x, y), && 0 < x < 1, \text{ each } y \in (0, 1). \end{aligned} \tag{P'}$$

Here $Y = u_{yy}$. Now let $K(x, t; \xi)$ be the kernel, or Green's function, for the one-dimensional heat equation on (0, 1). Then, for a given $\Delta t > 0$, we can "solve" (P') as follows:

$$u(x, y, t) = \int_0^1 K(x, \Delta t; \xi) u(\xi, y, t - \Delta t)\, d\xi + \int_{t-\Delta t}^{t} \int_0^1 K(x, t - s; \xi)\big(f(\xi, y, s) + u_{yy}(\xi, y, s)\big)\, d\xi\, ds.$$

This may not appear immediately useful, but we nevertheless press on. Use the trapezoid rule to approximate the time integral to get

$$\begin{aligned} &u(x, y, t) - \tfrac{1}{2}\Delta t\big(f(x, y, t) + u_{yy}(x, y, t)\big) \\ &\quad = \int_0^1 K(x, \Delta t; \xi)\big(u(\xi, y, t - \Delta t) + \tfrac{1}{2}\Delta t\big(f(\xi, y, t - \Delta t) \\ &\qquad + u_{yy}(\xi, y, t - \Delta t)\big)\big)\, d\xi + \mathcal{O}(\Delta t^3). \end{aligned}$$

Call the integral on the right side $\phi(x, y, t)$; then rewrite the above to get

$$\begin{aligned} &-\tfrac{1}{2}\Delta t\, u_{yy} + u = \tfrac{1}{2}\Delta t\, f + \phi + \mathcal{O}(\Delta t^3), && \text{each } x \in (0, 1), \\ &u(x, 0, t) = u(x, 1, t) = 0, && \text{each } x \in (0, 1). \end{aligned} \tag{1}$$

This is nothing more than a one-parameter family of two-point boundary value problems in y. Assuming that the right-hand side functions are known, we can easily compute the solution to get $u(x, y, t)$ for each $x \in (0, 1)$. But how do we compute the function ϕ?

Recall that the operation ot integrating a known function against the heat kernel is equivalent to solving the heat equation forwards with the known function as the initial data. In

other words, for fixed t_0, ϕ is simply the solution of the following one-dimensional heat equation problem:

$$\begin{aligned} &\phi_t = \phi_{xx}, && 0 < x < 1, \text{ each } y \in (0, 1), \ t > t_0, \\ &\phi(0, y, t) = \phi(1, y, t) = 0, && \text{each } y \in (0, 1), \ t > t_0, \\ &\phi(x, y, t_0) = u(x, y, t_0) + \tfrac{1}{2}\,\Delta t\big(f(x, y, t_0) + u_{yy}(x, y, t_0)\big). \end{aligned} \tag{2}$$

The basic algorithm can now be spelled out. Given an array $\{u_{ij}(t - \Delta t)\}$, $u_{ij}(t - \Delta t) \approx u(x_i, y_j, t - \Delta t)$ on a uniform grid $\{(x_i, y_j)\}$, with $x_i = ih$ for $h = 1/N$, and similarly for y, we first compute the array $\{\phi_{ij}(t)\}$ by approximating (2) using ordinary Crank–Nicolson methods, then we use $\{\phi_{ij}(t)\}$ to compute $\{u_{ij}(t)\}$ by solving (1) using central differences. More precisely, we have the following algorithm.

Algorithm 1.

(1) Form the array $\{r_{ij}\}$ from $\{u_{ij}(t_{n-1})\}$ according to

$$r_{ij} = \tfrac{1}{2}\,\Delta t\, f(x_i, y_j, t_{n-1}) + \lambda u_{i,j-1}(t_{n-1}) + (1 - 2\lambda)u_{ij}(t_{n-1}) + \lambda u_{i,j+1}(t_{n-1}),$$

where $\lambda = \Delta t/2h^2$. This corresponds to computing the initial condition in (2).

(2) Form the array $\{\phi_{ij}\}$ from $\{r_{ij}\}$ by solving the system of equations

$$-\lambda\phi_{i-1,j} + (1 + 2\lambda)\phi_{ij} - \lambda\phi_{i+1,j} = \lambda r_{i-1,j} + (1 - 2\lambda)r_{ij} + \lambda r_{i+1,j}.$$

This corresponds to solving the heat equation (2) for a single time step.

(3) Compute $\{u_{ij}(t_n)\}$ from $\{\phi_{ij}\}$ by solving the system:

$$-\lambda u_{i,j-1}(t_n) + (1 + 2\lambda)u_{ij}(t_n) - \lambda u_{i,j+1}(t_n) = \phi_{ij} + \tfrac{1}{2}\,\Delta t\, f(x_i, y_j, t).$$

This corresponds to solving the system of two-point boundary value problems (1).

In matrix notation the method takes on a very interesting form. Define two orderings of the $\{u_{ij}(t)\}$ array, as follows:

$$\hat{u}(t) = (u_{11}(t), u_{12}(t), \ldots, u_{1,N-1}(t), u_{21}(t), \ldots)^{\mathrm{T}},$$

$$\tilde{u}(t) = (u_{11}(t), u_{21}(t), \ldots, u_{N-1,1}(t), u_{12}(t), \ldots)^{\mathrm{T}}$$

(note that the first is a row-wise ordering and the second is a column-wise ordering) and let Q be the orthogonal matrix mapping from one to the other:

$$\hat{u}(t) = Q\tilde{u}(t), \qquad \tilde{u}(t) = Q^{\mathrm{T}}\hat{u}(t).$$

Define further the *block diagonal stiffness matrix*

$$\mathbb{K} = \lambda \begin{bmatrix} K & & \\ & \ddots & \\ & & K \end{bmatrix},$$

where the K matrix is the usual one-dimensional finite-difference stiffness matrix:

$$K = \text{tridiag}(-1, 2, -1)$$

and $\lambda = \Delta t/2h^2$. Then, under the assumption that $f \equiv 0$ (this is for simplicity only), the algorithm above can be succinctly written in matrix form as follows:

$$\hat{u}(t) = (I + \mathbb{K})^{-1} Q (I + \mathbb{K})^{-1} (I - \mathbb{K}) Q^{\mathrm{T}} (I - \mathbb{K}) \hat{u}(t - \Delta t).$$

If we define the matrix G_2 as

$$G_2 = (I + \mathbb{K})^{-1} Q (I + \mathbb{K})^{-1} (I - \mathbb{K}) Q^{\mathrm{T}} (I - \mathbb{K}),$$

then we have the following theorem.

Theorem 2. *For any $h > 0$, $\Delta t > 0$, we have $\rho(G_2) < 1$ so that Algorithm 1 is unconditionally stable.*

Proof. We have that G_2 is similar to H, where H is defined by

$$H = Q (I + \mathbb{K})^{-1} (I - \mathbb{K}) Q^{\mathrm{T}} (I - \mathbb{K}) (I + \mathbb{K})^{-1}.$$

Then we have

$$\rho(G_2) = \rho(H) \leqslant \| H \|_2 \leqslant \| (I - \mathbb{K})(I + \mathbb{K})^{-1} \|_2^2 < 1,$$

since $\mathbb{K}$ is symmetric and positive definite for all h and Δt. □

3. Examples and extensions

In this section we present a series of example computations with this algorithm, as well as some extensions to the basic method. The codes were written in FORTRAN and run on a Sun4/260 and a CRAY X-MP/24, both at the University of Alabama in Huntsville. The assistance of the Alabama Supercomputer Network is hereby acknowledged.

Example 3. We first look at the simplest model to demonstrate that the method does work and achieves the expected accuracy. Consider the problem

$$\begin{aligned} &u_t = \Delta u, && (x, y) \in R = (0, 1) \times (0, 1), \\ &u(x, y, t) = 0, && (x, y) \in \partial R, \\ &u(x, y, 0) = u_0(x, y), \end{aligned} \tag{E$_1$}$$

where

$$u_0(x, y) = \sin \pi x \sin \pi y.$$

Then the exact solution is

$$u(x, y, t) = \exp\{-2\pi^2 t\} \sin \pi x \sin \pi y.$$

Algorithm 1 was used to solve this problem forward in time and the approximate solution was compared to the exact solution. Table 1 shows the behavior of the error which drops by nearly the expected factor of 4 as h and Δt are both cut in half. These runs were made with $\Delta t = \frac{1}{2}h$ and the errors are computed at $t = \frac{1}{4}$.

Table 1
Error data from 2D kernel code

$N = 1/h$	L^∞ error	L^2 error
8	$3.6 \cdot 10^{-4}$	$1.8 \cdot 10^{-4}$
16	$9.2 \cdot 10^{-5}$	$4.8 \cdot 10^{-5}$
32	$2.4 \cdot 10^{-5}$	$1.2 \cdot 10^{-5}$
64	$6.4 \cdot 10^{-6}$	$3.1 \cdot 10^{-6}$

We next looked at execution time comparisons. Two new programs were written, using "standard" methods for solving the linear systems associated with problems of this type. The first code was written to use the IMSL routine FPS2H (which is a second-order fast Poisson solver based on [1,2]) to approximate solutions of (E_1). The second code used the routine SSORCG (symmetric SOR with conjugate gradient acceleration) from the ITPACK [11] library of iterative linear system solvers. Details of these comparison codes are given in Appendix B at the end of this paper. Table 2 gives the overall CPU-time and also the average CPU-time per time step, for all three codes. This data was obtained on a Sun 4/260, using IMSL routines to do the time checks. The codes both used $\Delta t = \frac{1}{2}h$, and ran until $t = \frac{1}{8}$ was reached.

Example 4. Consider now the following anisotropic problem:

$$\begin{aligned} &u_t = au_{xx} + bu_{yy}, && (x, y) \in R = (0, 1) \times (0, 1), \\ &u(x, y, t) = 0, && (x, y) \in \partial R, \\ &u(x, y, 0) = u_0(x, y), \end{aligned} \tag{E_2}$$

where a and b are positive constants, not necessarily equal.

In this case Algorithm 1 changes somewhat, since the constant λ no longer has the same value. We have the next algorithm.

Algorithm 5.

(1) Form the array $\{r_{ij}\}$ from $\{u_{ij}(t_{n-1})\}$ according to

$$r_{ij} = \lambda_b u_{i,j-1}(t_{n-1}) + (1 - 2\lambda_b)u_{ij}(t_{n-1}) + \lambda_b u_{i,j+1}(t_{n-1}),$$

where $\lambda_b = b\,\Delta t/2h^2$.

Table 2
Timing data from 2D codes

$N = 1/h$	Kernel algorithm		FPS2H		SSORCG	
	Total	Per step	Total	Per step	Total	Per step
8	$1.0 \cdot 10^{-2}$	$5.0 \cdot 10^{-3}$	$1.0 \cdot 10^{-2}$	$5.0 \cdot 10^{-3}$	$1.6 \cdot 10^{-3}$	$8.0 \cdot 10^{-2}$
16	$6.0 \cdot 10^{-2}$	$1.5 \cdot 10^{-2}$	$9.0 \cdot 10^{-2}$	$2.3 \cdot 10^{-2}$	$7.6 \cdot 10^{-1}$	$1.9 \cdot 10^{-1}$
32	$4.4 \cdot 10^{-1}$	$5.5 \cdot 10^{-2}$	$6.9 \cdot 10^{-1}$	$8.6 \cdot 10^{-2}$	$4.4 \cdot 10^{-0}$	$5.5 \cdot 10^{-1}$
64	$3.0 \cdot 10^{-0}$	$1.9 \cdot 10^{-1}$	$5.3 \cdot 10^{-0}$	$3.2 \cdot 10^{-1}$	$3.4 \cdot 10^{+1}$	$2.1 \cdot 10^{-0}$

(2) Form the array $\{\phi_{ij}\}$ from $\{r_{ij}\}$ by solving the system of equations

$$-\lambda_a\phi_{i-1,j}+(1+2\lambda_a)\phi_{ij}-\lambda_a\phi_{i+1,j}=\lambda_a r_{i-1,j}+(1-2\lambda_a)r_{ij}+\lambda_a r_{i+1,j}.$$

Here $\lambda_a = a\,\Delta t/2h^2$.

(3) Compute $\{u_{ij}(t_n)\}$ from $\{\phi_{ij}\}$ by solving the system

$$-\lambda_b u_{i,j-1}(t_n)+(1+2\lambda_b)u_{ij}(t_n)-\lambda_b u_{i,j+1}(t_n)=\phi_{ij},$$

The matrix form of the algorithm now is

$$\hat{u}(t)=(I+\mathbb{K}_b)^{-1}Q(I+\mathbb{K}_a)^{-1}(I-\mathbb{K}_a)Q^{\mathrm{T}}(I-\mathbb{K}_b)\hat{u}(t-\Delta t), \tag{3}$$

where $\mathbb{K}_i$, $i=a, b$, means that λ_i, $i=a, b$, has been used in the computation. The stability theorem still holds, more or less exactly as before. As can be seen from (3), we now must factor two different tridiagional systems. The method is still very fast, since the only additional work is the factorization of a second tridiagonal system, and a set of test runs similar to what was done in Example 3 produced essentially the same values.

Example 6. We now consider a problem in three dimensions:

$$\begin{aligned} &u_t = au_{xx}+bu_{yy}+cu_{zz}, && (x, y, z)\in R=(0, 1)^3,\\ &u(x, y, z, t)=0, && (x, y, z)\in \partial R,\\ &u(x, y, z, 0)=u_0(x, y, z). \end{aligned} \tag{E$_3$}$$

The algorithm is basically the same as before, but longer.

Algorithm 7.

(1) Form the array $\{r_{ijk}\}$ from $\{u_{ijk}(t_{n-1})\}$ according to

$$r_{ijk}=\lambda_c u_{i,j,k-1}(t_{n-1})+(1-2\lambda_c)u_{i,j,k}(t_{n-1})+\lambda_c u_{i,j,k+1}(t_{n-1}),$$

where $\lambda_c = c\,\Delta t/2h^2$.

(2) Form the array $\{F_{ijk}\}$ from $\{r_{ijk}\}$ according to

$$F_{ijk}=\lambda_b r_{i,j-1,k}+(1-2\lambda_b)r_{ijk}+\lambda_b r_{i,j+1,k}.$$

(3) Compute $\{\psi_{ijk}\}$ by solving the system

$$-\lambda_a\psi_{i-1,j,k}+(1+2\lambda_a)\psi_{ijk}-\lambda_a\psi_{i+1,j,k}=\lambda_a F_{i-1,j,k}+(1-2\lambda_a)F_{ijk}+\lambda_a F_{i+1,j,k}.$$

(4) Compute $\{\phi_{ijk}\}$ by solving the system

$$-\lambda_b\phi_{i,j-1,k}+(1+2\lambda_b)\phi_{ijk}-\lambda_b\phi_{i,j+1,k}=\psi_{ijk}.$$

(5) Compute $\{u_{ijk}(t_n)\}$ from $\{\phi_{ijk}\}$ by solving the system

$$-\lambda_c u_{i,j,k-1}(t_n)+(1+2\lambda_c)u_{ijk}(t_n)-\lambda_c u_{i,j,k+1}(t_n)=\phi_{ij}.$$

To write the algorithm in matrix form, we must now consider three different orderings of the $\{u_{ijk}(t)\}$ array:

$$\hat{u}(t) = (u_{111}(t), u_{112}(t), \ldots, u_{1,1,N-1}(t), u_{211}(t), \ldots)^{\mathrm{T}},$$

$$\tilde{u}(t) = (u_{111}(t), u_{121}(t), \ldots, u_{1,N-1,1}(t), u_{112}(t), \ldots)^{\mathrm{T}},$$

$$\bar{u}(t) = (u_{111}(t), u_{211}(t), \ldots, u_{N-1,1,1}(t), u_{121}(t), \ldots)^{\mathrm{T}}.$$

We again define orthogonal matrices mappings from one ordering to another, as follows:

$$\hat{u}(t) = Q\tilde{u}(t), \qquad \tilde{u}(t) = Q^{\mathrm{T}}\hat{u}(t), \qquad \hat{u}(t) = P\bar{u}(t), \qquad \bar{u}(t) = P^{\mathrm{T}}\hat{u}(t).$$

In addition, the product $Q^{\mathrm{T}}P$ maps between orderings as follows:

$$\hat{u}(t) = Q^{\mathrm{T}}P\bar{u}(t), \qquad \bar{u}(t) = P^{\mathrm{T}}Q\hat{u}(t).$$

If we define the block diagonal matrices $\mathbb{K}_i$, $i = a, b, c$, analogous to Example 4, then we have the following matrix form:

$$\hat{u}(t) = (I + \mathbb{K}_c)^{-1}Q(I + \mathbb{K}_b)^{-1}Q^{\mathrm{T}}P(I + \mathbb{K}_a)^{-1} \times (I - \mathbb{K}_a)P^{\mathrm{T}}Q(I - \mathbb{K}_b)Q^{\mathrm{T}}(I - \mathbb{K}_c)\hat{u}(t - \Delta t).$$

For simpliciy, define the matrix G_3 as the coefficient matrix in the above:

$$G_3 = (I + \mathbb{K}_c)^{-1}Q(I + \mathbb{K}_b)^{-1}Q^{\mathrm{T}}P(I + \mathbb{K}_a)^{-1}(I - \mathbb{K}_a)P^{\mathrm{T}}Q(I - \mathbb{K}_b)Q^{\mathrm{T}}(I - \mathbb{K}_c).$$

The stability of the method in this case is somewhat more difficult to establish, but a rigorous proof can be obtained by a close look at the structure of the reordering matrices P and Q. This we do in a preliminary lemma, whose proof is deferred to Appendix A.

Lemma 8. *The matrix*

$$M = Q(I + \mathbb{K}_b)^{-1}Q^{\mathrm{T}}P(I + \mathbb{K}_a)^{-1}(I - \mathbb{K}_a)P^{\mathrm{T}}Q(I - \mathbb{K}_b)Q^{\mathrm{T}} \tag{4}$$

is symmetric.

Proof. See Appendix A. □

Given the lemma, we can easily establish a stability result, more or less as before.

Theorem 9. *For any $h > 0$, $\Delta t > 0$, we have $\rho(G_3) < 1$ so that Algorithm 7 is unconditionally stable.*

Proof. We can write G_3 in terms of the matrix M as follows:

$$G_3 = (I + \mathbb{K}_c)^{-1}M(I - \mathbb{K}_c).$$

If we use similarity to define H as was done before, we get

$$H = M(I - \mathbb{K}_c)(I + \mathbb{K}_c)^{-1},$$

so that

$$\rho(G_3) = \rho(H) \leqslant \| H \|_2 \leqslant \| M \|_2 \| (I - \mathbb{K}_c)(I + \mathbb{K}_c)^{-1} \|_2.$$

Table 3
Error data from 3D kernel code

$N = 1/h$	L^∞ error	L^2 error
8	$4.0 \cdot 10^{-3}$	$1.4 \cdot 10^{-3}$
16	$1.0 \cdot 10^{-3}$	$3.5 \cdot 10^{-4}$
32	$2.5 \cdot 10^{-4}$	$8.8 \cdot 10^{-5}$
64	$5.8 \cdot 10^{-5}$	$2.0 \cdot 10^{-5}$

Table 4
Timing data from 3D codes

$N = 1/h$	Kernel algorithm		FPS3H		SSORCG	
	Total	Per step	Total	Per step	Total	Per step
8	$5.0 \cdot 10^{-2}$	$2.5 \cdot 10^{-2}$	$2.0 \cdot 10^{-1}$	$1.0 \cdot 10^{-1}$	$9.1 \cdot 10^{-1}$	$4.6 \cdot 10^{-1}$
16	$7.9 \cdot 10^{-1}$	$2.0 \cdot 10^{-1}$	$2.5 \cdot 10^{+0}$	$6.2 \cdot 10^{-1}$	$1.3 \cdot 10^{+1}$	$3.2 \cdot 10^{+0}$
32	$1.3 \cdot 10^{+1}$	$1.6 \cdot 10^{+1}$	$4.1 \cdot 10^{+1}$	$5.1 \cdot 10^{+0}$	$1.8 \cdot 10^{+2}$	$2.2 \cdot 10^{+1}$
64	$2.2 \cdot 10^{+2}$	$1.4 \cdot 10^{+1}$	$6.0 \cdot 10^{+2}$	$3.8 \cdot 10^{+1}$	—	—

The matrices in the norms on the right are both symmetric, hence their 2-norms are equal to their spectral radii which are easily shown to be less than one. □

For comparison purposes we wrote a heat solver based on the IMSL routine FPS3H, a 3D version of FPS2H, as well as a three-dimensional version of the SSORCG code used above. These were compared to a program based on Algorithm 7 and all three codes were run on (E_3), using $a = b = c = 1$. As before, we used a sequence of h values, with $\Delta t = \frac{1}{2}h$, and the runs were timed over $(0, \frac{1}{8})$. Tables 3 and 4 give the error and timing results on a Sun 4/260. The gap in the SSORCG data is due to the large memory requirements for workspace when running that program — it was not practical to run that case on the Sun 4.

Finally, we give the results of the same tests with the 2D codes, made on a CRAY X-MP/24. The primary purpose here was to see how well the kernel algorithm would perform in a vector environment. The code for these tests was no different from the code as run on the Sun. The vectorization of the solution steps was accomplished by organizing the loops to do them in parallel. The ITPACK code was written using ITPACKV, a version of ITPACK designed for vector processors like the CRAY. It is entirely plausible that this comparison is unfair to the FPS2H code, since we were not able to get reliable information on whether the IMSL code had been optimized for vectorization on the CRAY. Table 5 gives the results for (E_1), using $\Delta t = \frac{1}{2}h$, and $t_{\max} = \frac{1}{8}$.

Table 5
Timing data from 2D codes on a CRAY X-MP/24

$N = 1/h$	Kernel algorithm		FPS2H		SSORCG	
	Total	Per step	Total	Per step	Total	Per step
8	$1.2 \cdot 10^{-3}$	$6.0 \cdot 10^{-4}$	$3.6 \cdot 10^{-3}$	$1.8 \cdot 10^{-3}$	$4.2 \cdot 10^{-3}$	$2.1 \cdot 10^{-3}$
16	$5.5 \cdot 10^{-3}$	$1.4 \cdot 10^{-3}$	$2.2 \cdot 10^{-2}$	$5.5 \cdot 10^{-2}$	$2.1 \cdot 10^{-2}$	$5.3 \cdot 10^{-3}$
32	$2.5 \cdot 10^{-2}$	$3.1 \cdot 10^{-3}$	$1.2 \cdot 10^{-1}$	$1.5 \cdot 10^{-2}$	$1.3 \cdot 10^{-1}$	$1.7 \cdot 10^{-2}$
64	$1.4 \cdot 10^{-1}$	$8.6 \cdot 10^{-3}$	$7.9 \cdot 10^{-1}$	$4.9 \cdot 10^{-2}$	$8.6 \cdot 10^{-1}$	$5.4 \cdot 10^{-2}$

4. Operation counts and storage estimates

In this section we will estimate the number of floating-point operations required for the algorithms presented in the previous sections, as well as the amount of storage. For simplicity's sake we will assume homogeneous Dirichlet data on the boundary and, further, that the forcing term is not present: $f(x, y, t) \equiv 0$, and similarly for the three-dimensional case. The operation count estimates are for multiplications and divisions only.

We will assume that the grid points along each axis are labeled from 0 to N; thus the mesh is $h = 1/N$, and there are $(N-1)^m$ unknowns in $\mathbb{R}^m$.

Two-dimensional Algorithm. If we assume the diffusion is isotropic, then there is only a single tridiagonal factorization to be performed. If we assume further that a modified Choleski scheme is used, then the cost of factorization can be estimated as

$$C_{\text{factor}} = 3N + \mathcal{O}(1).$$

The cost per time step can be readily estimated from Algorithm 1. We have

$$C_{\text{step}} = 6(N-1)^2 + 2(N-1)C_{\text{solve}} + \mathcal{O}(N),$$

where C_{solve} is the cost involved in doing the forward/backward substitution steps associated with the Choleski factorization. We have

$$C_{\text{solve}} = 2N + \mathcal{O}(1),$$

so that, finally,

$$C_{\text{step}} = 10N^2 + \mathcal{O}(N). \tag{5}$$

Note that the factorization costs are a full order of magnitude less than the cost per step.

The major storage requirements are for the tridiagonal matrix, which takes only about $2N$ words (assuming symmetry), and the solution array itself, which takes about N^2 words. The codes we used required a second copy of the solution array, making for a total of

$$M = 2N^2 + \mathcal{O}(N)$$

words of storage. It may be possible to avoid using the second complete copy of the solution array.

If the diffusion is anisotropic, then the only significant change is that we must factor two tridiagonal matrices instead of just one. This only changes the lower-order terms, so (5) stands as the two-dimensional operation count estimate. The memory requirements are similarly unaffected.

Three-dimensional Algorithm. Working from Algorithm 7 we have

$$C_{\text{step}} = 9(N-1)^2 + 3(N-1)C_{\text{solve}} + \mathcal{O}(N),$$

where C_{solve} is as before. Hence

$$C_{\text{step}} = 15N^2 + \mathcal{O}(N).$$

The memory requirements are analogous to the 2D case, the bulk of the storage being used for the solution arrays. Again, we used two copies of the solution for a total storage requirement of

$$M = 2N^3 + \mathcal{O}(N^2)$$

words, but might be able to get by with less than that.

Again, anisotropic diffusion does not significantly affect the operation count, nor the memory requirements.

Appendix A

Proof of Lemma 8. If we multiply out the definition of the matrix M as given in (4), then we have

$$M = (I + \mathbb{M}_b)^{-1}(I + \mathbb{M}_a)^{-1}(I - \mathbb{M}_a)(I - \mathbb{M}_b),$$

where $\mathbb{M}_a = P\mathbb{K}_a P^{\mathrm{T}}$ and $\mathbb{M}_b = Q\mathbb{K}_b Q^{\mathrm{T}}$. It is clear, then, that M is symmetric if and only if the $\mathbb{M}_i$ matrices have the same eigenvectors. This requires a detailed examination of the reordering matrices. Fortunately, there is no loss of generality in considering only the case of $N = 4$, in which case P and Q are 27×27. However, the block structure allows us to view them as 9×9, with 3×3 blocks. Define the matrices J_{ij} to be zero everywhere, except for a one at the (i, j) entry. For example,

$$J_{12} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}.$$

In this notation we can write out P and Q explicitly:

$$P = \left[\begin{array}{ccc:ccc:ccc} J_{11} & 0 & 0 & J_{21} & 0 & 0 & J_{31} & 0 & 0 \\ 0 & J_{11} & 0 & 0 & J_{21} & 0 & 0 & J_{31} & 0 \\ 0 & 0 & J_{11} & 0 & 0 & J_{21} & 0 & 0 & J_{31} \\ \hdashline J_{12} & 0 & 0 & J_{22} & 0 & 0 & J_{32} & 0 & 0 \\ 0 & J_{12} & 0 & 0 & J_{22} & 0 & 0 & J_{32} & 0 \\ 0 & 0 & J_{12} & 0 & 0 & J_{22} & 0 & 0 & J_{32} \\ \hdashline J_{13} & 0 & 0 & J_{23} & 0 & 0 & J_{33} & 0 & 0 \\ 0 & J_{13} & 0 & 0 & J_{23} & 0 & 0 & J_{33} & 0 \\ 0 & 0 & J_{13} & 0 & 0 & J_{23} & 0 & 0 & J_{33} \end{array}\right],$$

$$Q = \left[\begin{array}{ccc|ccc|ccc} J_{11} & J_{21} & J_{31} & 0 & 0 & 0 & 0 & 0 & 0 \\ J_{12} & J_{22} & J_{32} & 0 & 0 & 0 & 0 & 0 & 0 \\ J_{13} & J_{23} & J_{33} & 0 & 0 & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & J_{11} & J_{21} & J_{31} & 0 & 0 & 0 \\ 0 & 0 & 0 & J_{12} & J_{22} & J_{32} & 0 & 0 & 0 \\ 0 & 0 & 0 & J_{13} & J_{23} & J_{33} & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & 0 & 0 & J_{11} & J_{21} & J_{31} \\ 0 & 0 & 0 & 0 & 0 & 0 & J_{12} & J_{22} & J_{32} \\ 0 & 0 & 0 & 0 & 0 & 0 & J_{13} & J_{23} & J_{33} \end{array}\right].$$

Now, since the $\mathbb{K}_i$ matrices are block diagonal, the blocks being proportional to the tridiagonal matrix $K = \text{tridiag}(-1, 2, -1)$, it follows that the eigenvectors of $\mathbb{K}_i$ are formed from the eigenvectors of K. To be specific, let ξ be an eigenvector of K; then the corresponding eigenvector of $\mathbb{K}_i$ is given by

$$x = (\xi, \xi, \xi, \xi, \xi, \xi, \xi, \xi, \xi)^{\mathrm{T}}.$$

Then we can explicitly form the products $z_a = Px$ and $z_b = Qx$, which give the eigenvectors of the corresponding $\mathbb{M}_i$, and see that we do, in fact, have $z_a = z_b$. Therefore the eigenvectors of $\mathbb{M}_a$ and $\mathbb{M}_b$ are equal, hence $\mathbb{M}_a$ and $\mathbb{M}_b$ commute and M is symmetric. □

Appendix B. Details of comparison codes

In this section we (briefly) discuss the codes used for the comparisons of Section 3.

FPSnH Codes. As mentioned above, these codes are part of the IMSL package and are based on the work in [1,2]. Since they are designed to approximate solutions to elliptic problems on rectangles or boxes, the heat equation was written in time discretized form to yield the sequence of elliptic equations:

$$-\Delta u(t_{n+1}) + \tfrac{1}{2}\,\Delta t\, u(t_{n+1}) = \Delta u(t_n) + \tfrac{1}{2}\,\Delta t\, u(t_n) + \mathcal{O}(\Delta t^3). \tag{6}$$

A function PRHS was written to compute the (discretized) right-hand side of (6), and a second function BRHS was written to provide the boundary data. Other parameters in the FPSnH calls were specified as follows:

COEFU: $\frac{1}{2}\,\Delta t$;
NX: 9, 17, 33, 65; depending on the case;
NY: 9, 17, 33, 65; depending on the case;
NZ: 9, 17, 33, 65; depending on the case (3D, only);
AX: 0.0;
BX: 1.0;
AY: 0.0;
BY: 1.0;
AZ: 0.0 (3D, only);
BZ: 1.0 (3D, only);

IBCTY: 1, 1, 1, 1 (vector of size 4 in 2D routine);
1, 1, 1, 1, 1, 1 (vector of size 6 in 3 routine);
IORDER: 2.
For details on the meaning of these parameters the reader is referred to the IMSL documentation.

ITPACK Codes. For precise details about the use of this package, the reader is referred to [11,12], which are available in TeX form from the NETLIB service [10]. Based on the information in [11] we decided to use the SSORCG routine for our tests here. The initial guess at each time step was taken to be the final value from the previous time step. Default values for all parameters were used.

ITPACK and ITPACKV use rather special storage schemes which make construction of the matrix somewhat involved. In the scalar version, several routines are supplied to assist the user in this connection, and these were used on the Sun. In both environments, the cost of forming and loading the matrix is included in the timing data given in Section 3.

Remark on further development of the algorithm

There appears to be no reason not to extend this algorithm to more general operators on more general domains [7]. If we consider, for example, the equation

$$u_t = \operatorname{div}(a(x, y)\,\nabla u)$$

on the unit square, then we get the algebraic system

$$\hat{u}(t) = (I + \mathbb{K}_2)^{-1} Q (I + \mathbb{K}_1)^{-1} (I - \mathbb{K}_1) Q^{\mathrm{T}} (I - \mathbb{K}_2) \hat{u}(t - \Delta t),$$

where the $\mathbb{K}_i$ are still block diagonal, but the blocks are no longer identical. We have

$$\mathbb{K}_1 = \lambda \text{ block diag}(K(a, x_i, \cdot)), \qquad \mathbb{K}_2 = \lambda \text{ block diag}(K(a, \cdot, y_i)),$$

where $K(a, x_i, \cdot)$ is the matrix obtained by applying a central difference approximation (in the y-direction) to $\operatorname{div}(a(x_i, y)\,\nabla u)$, and $K(a, \cdot, y_i)$ is the matrix obtained by applying a central difference approximation (in the x-direction) to $\operatorname{div}(a(x, y_i)\,\nabla u)$. This requires more storage than the previous algorithms, as well as the solution of $2(N-1)$ different tridiagonal systems, instead of just one or two. Estimating the operation count is difficult, since the cost of *forming* the systems depends heavily on how the coefficient functions are constructed. However, the additional memory requirements are fairly straightforward: instead of storing a single tridiagonal system, we must store two sets of $N-1$ tridiagonal systems. This results in a memory usage of

$$M = 6N^2 + \mathcal{O}(N)$$

words, in 2D, and

$$M = 4N^3 + \mathcal{O}(N)$$

words, in 3D.

While these figures do represent an increase over the constant-coefficient case, it is not a significant increase. In particular, these figures are much less than what would be expected using traditional methods.

The extension to more general domains is trickier, largely due to the complications introduced by the geometry of the domain, but it appears to be possible to apply these "dimensional separation" ideas at least to convex domains. That is the objective of the forthcoming work [7], still in progress.

References

[1] R. Boisvert, A fourth order accurate fast direct method for the Helmholtz equation, in: G. Birkhoff and A. Schoenstadt, Eds., *Elliptic Problem Solvers II* (Academic Press, New York, 1984) 35–44.
[2] R. Boisvert, A fourth order accurate Fourier method for the Helmholtz equation in three dimensions, *ACM Trans. Math. Software* **13** (3) (1987) 221–234.
[3] J. Epperson, On the use of Green's functions for approximating nonlinear parabolic PDE's, *Appl. Math. Lett.* **2** (3) (1989) 293–296.
[4] J. Epperson, Semi-group linearization for nonlinear parabolic equations, *Numer. Methods Partial Differential Equations* **7** (1991) 147–163.
[5] J.F. Epperson, A kernel-based method for parabolic equations with nonlinear convection terms, *J. Comput. Appl. Math.* **36** (3) (1991) 275–288.
[6] J. Epperson, Efficient computation with heat equation kernels, *SIAM J. Sci. Statist. Comput.*, to appear.
[7] J. Epperson, All parabolic equations are (numerically) one dimensional, in preparation.
[8] J. Epperson, Computational Green's function methods for nonlinear parabolic PDE's, in preparation.
[9] J. Epperson, EVOLVE: A set of subroutines for rapid solution of parabolic equations on a rectangle, in preparation.
[10] E. Grosse, Netlib news: greetings, *SIAM News* **23** (6) (1990) 14
[11] D. Kincaid et al., ITPACK 2C: A FORTRAN package for solving large sparse linear systems by adaptive accelerated iterative methods, typescript documentation.
[12] D. Kincaid et al., ITPACKV 2C user's guide, Report CNA-191, Center for Numer. Anal., Univ. Texas, Austin, 1984.