1987

A Computational Framework for Constructing Adaptive Grid Domain Mappings

Calvin J. Ribbens

Report Number:
87-673

A COMPUTATTIONAL FRAMEWORK FOR CONSTRUCTING ADAPTIVE GRID DOMAIN MAPPINGS

Calvin J. Ribbens

CSD-TR-673
April 1987

A COMPUTATIONAL FRAMEWORK FOR CONSTRUCTING ADAPTIVE GRID DOMAIN MAPPINGS

Calvin J. Ribbens*
Purdue University

CSD-TR 673
April, 1987

Abstract

This paper describes a computational framework for efficiently constructing and applying *adaptive grid domain mappings* for the numerical solution of partial differential equations (PDEs). These mappings are used to transform a PDE problem so that solving the new problem in the "transformed" domain using a uniform grid is equivalent to solving the original problem using an adapted grid. Thus, it is possible to apply a moving or irregular grid, and yet compute on a uniform grid. We believe that data structures and algorithms based on regular grids are the most promising for the development of good vector and parallel methods. We describe a very efficient method for constructing and evaluating a wide variety of adaptive grid domain mappings, given a redistribution of grid points in the original domain. The extra computation introduced by the mapping is small compared to the cost of solving a typical PDE problem. Using this framework, one can concentrate on the interesting and difficult problems of grid adaption, without having to write new algorithms to solve PDEs on these irregular grids. Good existing software may be used to solve the transformed problem on a uniform grid. Our method was developed as part of an investigation of adaptive grid schemes for two-dimensional, elliptic boundary value problems; it is applicable to grid based methods for other classes of PDEs as well.

1 Introduction

In this paper we consider the efficient construction of *adaptive grid domain mappings* for the numerical solution of two-dimensional partial differential equations. It is well-known that grid adaption is a powerful tool for solving difficult PDE problems. A wide variety of methods have been proposed and studied (see [1] and [12] for surveys). Grid adaption schemes fall into two general categories: grid refinement and grid motion. It is easy to see that a grid motion method may be

*Supported in part by Air Force Office of Scientific Research grant AFOSR 84-0385 and National Science Foundation contract DMS-8301589.

viewed as a mapping between the original domain and a "problem-solving", or computational domain. We refer to such a mapping as an adaptive grid domain mapping. The image under this mapping of a uniform grid in the problem-solving domain is the adapted grid in the original domain. It is typical for computations based on adapted grids to be carried out in the transformed domain, where using a uniform grid is equivalent to solving the original problem using the adapted grid. In this way the complexity introduced by grid adaption is located in the transformed PDE problem rather than in the grid.

By contrast, it is not obvious how to represent a grid refinement method in terms of a domain mapping. The irregularity introduced into the grid by local refinement must instead be handled by a more complicated algorithm, with no change in the PDE. This is done of course, often with good results. It seems likely, however, that good parallel and vector algorithms will be easier to design and implement if we can compute on a uniform grid. Observe also that local grid refinement schemes often introduce new equations and unknowns into the discrete problem in such a way that any special structure in the linear system is likely to be lost.

Our research focuses on adaptive grid domain mapping methods for linear, elliptic boundary-value problems on rectangular domains, although our results are not limited to this particular class of PDEs. In this paper we present a framework in which adaptive grid domain mappings can be quickly constructed, given a redistribution of grid points in the original domain. We show that these mappings can be used to transform and solve PDEs without significant added cost. Efficient methods for generating a good redistribution of grid points are described in [8] and [9]. The effectiveness of our approach for solving difficult PDE problems is addressed in [10]. In Section 2 of the present paper we describe the basic framework in more detail. Sections 3, 4, and 5 discuss the construction, evaluation and inversion of the mappings, respectively. In Section 6 we consider the performance of our framework and compare the costs introduced by adaptive grid domain mappings to the overall cost of a typical computation.

2 Grid adaption by domain mapping

The basic task of an adaptive grid domain mapping scheme is to find a function which maps points in a uniform grid on the computational domain to points in an adapted grid on the original domain. The function is then used as a change of variables in the original PDE. The actual numerical solution is done in the transformed domain. It is obviously a difficult task to generate a function out of thin air which transforms a problem in a useful way. Conceptually, the problem of moving grid points to reduce the error in a numerical method is a more natural one to address, although it is also a difficult one in its own right. Our approach assumes an adapted distribution of grid points in the original domain is given. Based on this set of points, we construct a mapping which approximately reproduces the adapted grid. This is a natural approach and allows us to focus on the interesting and difficult problem of how best to move the grid points, given some information about the PDE. The work of approximately solving the PDE with an adapted grid is taken care of by our framework. Given an adapted grid, we can immediately generate a mapping and solve the transformed problem using available uniform-grid numerical methods.

In the following discussion, we restrict ourselves to *tensor product* grids; that is, grids consisting of the intersection of a set of grid lines in one coordinate direction with a set of grid lines in the other direction. Advantages of regular tensor product grids include: 1) regular grids simplify the programming task and are promising with respect to parallelism, 2) the data structures needed to compute on such grids are relatively simple, 3) the calculations that need to be made during

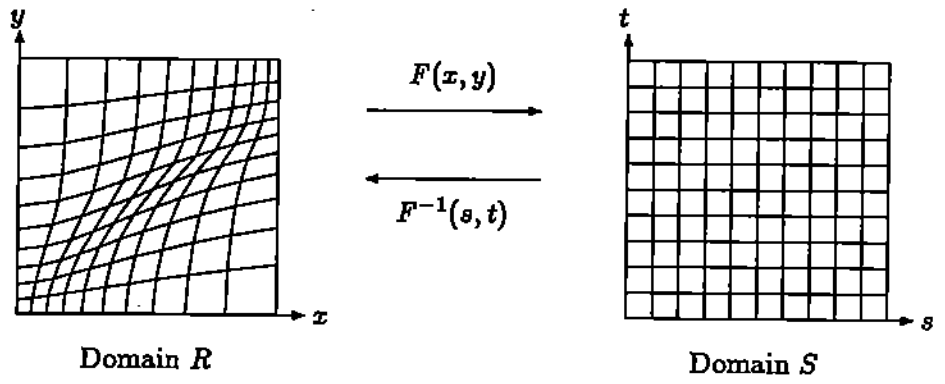

Figure 1: The two domains and grids used in adaptive grid domain mapping.

discretization are usually very similar for each grid element, 4) the linear systems which result are more likely to have some special structure which can be exploited.

We distinguish between two grids and two domains used in solving PDEs within our framework. Figure 1 illustrates the two grids and the two domains. The *mapping grid* is in general a curvilinear grid defined on the problem domain R. The points in this grid are adaptively rearranged and provide the data on which the domain mapping is based. The mapping grid is constrained to be logically a tensor product grid and must conform to the boundary ∂R. Its purpose is to help construct the mapping $F^{-1}(s,t)$. The *discretization grid* is used to discretize the transformed PDE in the solution domain S. It is always taken to be rectangular and uniform. Although in Figure 1 the dimensions of these two grids are the same, this is not required. Typically, the mapping grid need not be as fine as the discretization grid; it must simply include enough points to capture the behavior of the desired adaptive mapping. The original domain R must be rectangular in our present framework. We are considering extensions which will allow Domain R to be nonrectangular.

3 Constructing the mapping

The key step in our framework is to construct a well-behaved function $F^{-1}(s,t)$ which maps points in a uniform grid on Domain S to points in the given mapping grid on Domain R. The representation of F^{-1} is crucial to the performance of our framework. For example, it is important that the mapping not be prohibitively expensive to construct. There are methods which generate smooth mappings such as these, subject to certain adaptive constraints, by solving systems of simple PDEs (see [2] and [13], for example). Our framework avoids the considerable added cost of such an approach. Our choice for representing F^{-1} also permits efficient evaluation of the mapping and its derivatives, an important consideration since solving the transformed PDE requires that this be done many times. We also require that F^{-1} have at least two continuous derivatives with respect to s and t, since derivatives of the mapping of up to second order appear in the coefficients of the transformed PDE. Finally, F^{-1} must be constructed so that it is readily invertible, since we do not have an explicit representation for $F(x,y)$. This section describes a good choice for representing F^{-1} and an efficient construction procedure.

We use a pair of piecewise bicubic spline functions $x(s,t)$ and $y(s,t)$ to represent the two

coordinates of the function

$$F^{-1}(s,t) = (x(s,t), y(s,t)).$$

The number of polynomial pieces k_s and k_t in each direction is small—typically only two or three; the number depends on the complexity of the adaptive mapping. A simple adaption can be represented easily with only one or two pieces in each direction; a more exotic adapted grid may require three or four pieces. The number of points in an $m_s \times m_t$ mapping grid generally exceeds the number of coefficients, or free parameters, in the bicubic spline representation:

$$m_s m_t > (k_s + 3)(k_t + 3).$$

This is by design since it allows us to choose the coefficients so that $F^{-1}(s,t)$ best approximates the mapping grid in a least squares sense. The principal advantage of least squares approximation here is that it smoothes the mapping. We find, in fact, that the accuracy of the solution on the transformed domain is very sensitive to the smoothness of the mapping. Other choices for representing the mapping, such as cubic spline interpolation or hermite quintic interpolation, are not sufficiently smooth in some cases, despite having the required continuity.

We describe the procedure used to construct the function $x(s,t)$. The process for $y(s,t)$ is analogous. We assume the x-coordinates of the mapping grid are given:

$$\{x_{ij} : i = 1, \ldots, m_s;\ j = 1, \ldots, m_t\}.$$

These values may be thought of as the image under $x(s,t)$ of a uniform $m_s \times m_t$ tensor product grid in Domain S:

$$\{(s_i, t_j)\} = \{s_i : i = 1, \ldots, m_s\} \otimes \{t_j : j = 1, \ldots, m_t\}.$$

We seek a piecewise bicubic spline function $x(s,t)$ satisfying

$$\begin{aligned} x(s_i, t_j) \approx x_{ij}, \quad & \text{for} \quad i = 1, \ldots, m_s, \\ & \text{and} \quad j = 1, \ldots, m_t, \end{aligned} \tag{1}$$

where $x(s,t)$ has k_s pieces in the s direction and k_t pieces in the t direction. Let $l_s = k_s + 3$, and $l_t = k_t + 3$. We can write $x(s,t)$ in terms of B-spline basis functions [3] as

$$x(s,t) = \sum_{j=1}^{l_t} \sum_{i=1}^{l_s} \alpha_{ij} \Phi_i(s) \Psi_j(t), \tag{2}$$

where Φ_i is the ith cubic B-spline over a uniform knot sequence in s, and Ψ_j has a similar meaning in the t direction. We seek values for the unknown coefficients

$$\{\alpha_{ij} : i = 1, \ldots, l_s;\ j = 1, \ldots, l_t\}.$$

Assuming $m_s > l_s$ and $m_t > l_t$, we proceed to set up the least squares problem. From the $m_s m_t$ equations (1), and the representation for $x(s,t)$ in (2), we get the following overdetermined system:

$$(T \otimes S)\alpha \approx \mathbf{x}, \tag{3}$$

where

$$\begin{aligned} \alpha &= [\alpha_{11}\, \alpha_{21} \cdots \alpha_{l_s 1}\, \alpha_{12}\, \alpha_{22} \cdots \alpha_{l_s 2} \ \cdots\ \alpha_{1 l_t}\, \alpha_{2 l_t} \cdots \alpha_{l_s l_t}]^T, \\ \mathbf{x} &= [x_{11}\, x_{21} \cdots x_{m_s 1}\, x_{12}\, x_{22} \cdots x_{m_s 2} \ \cdots\ x_{1 m_t}\, x_{2 m_t} \cdots x_{m_s m_t}]^T. \end{aligned}$$

Note that S in (3) is a matrix and not to be confused with the solution domain S. We have written the matrix equation (3) in terms of a tensor product of two matrices T and S. The natural definition (see [5]) of a tensor product, or *Kronecker* product, of two matrices $T \otimes S$, where T is $m_t \times l_t$ and S is $m_s \times l_s$, is the $m_t m_s \times l_t l_s$ matrix

$$T \otimes S = \begin{pmatrix} \tau_{11}S & \tau_{12}S & \dots & \tau_{1l_t}S \\ \tau_{21}S & \tau_{22}S & \dots & \tau_{2l_t}S \\ \vdots & \vdots & \ddots & \vdots \\ \tau_{m_t1}S & \tau_{m_t2}S & \dots & \tau_{m_tl_t}S \end{pmatrix},$$

where $(T)_{ij} = \tau_{ij}$ for $i = 1, \dots, m_t$ and $j = 1, \dots, l_t$. From (2) we see that the elements of T and S in (3) are given by

$$\begin{aligned} (S)_{ij} &= \Phi_j(s_i), \qquad i = 1, \dots, m_s; \quad j = 1, \dots, l_s; \\ (T)_{ij} &= \Psi_j(t_i), \qquad i = 1, \dots, m_t; \quad j = 1, \dots, l_t. \end{aligned}$$

The tensor product formulation of the least squares problem (3) is exploited to make the computation of the coefficients α very efficient. From (3) we form the normal equations

$$(T \otimes S)^T (T \otimes S)\alpha = (T \otimes S)^T \mathbf{x}. \tag{4}$$

By tensor product identities we can rewrite (4) as

$$(T^T T \otimes S^T S)\alpha = (T^T \otimes S^T)\mathbf{x},$$

or

$$\begin{aligned} \alpha &= (T^T T \otimes S^T S)^{-1}(T^T \otimes S^T)\mathbf{x} \\ &= ((T^T T)^{-1} T^T \otimes (S^T S)^{-1} S^T)\mathbf{x} \\ &= (T^+ \otimes S^+)\mathbf{x}, \end{aligned} \tag{5}$$

where T^+ and S^+ are the psuedoinverses of T and S, respectively (see [6]).

Now, let A be the $l_s \times l_t$ matrix whose columns are subvectors of α of length l_s; that is,

$$A = \begin{pmatrix} \alpha_{11} & \alpha_{12} & \dots & \alpha_{1l_t} \\ \alpha_{21} & \alpha_{22} & \dots & \alpha_{2l_t} \\ \vdots & \vdots & \ddots & \vdots \\ \alpha_{l_s1} & \alpha_{l_s2} & \dots & \alpha_{l_sl_t} \end{pmatrix}.$$

Similarly, let X be the $m_s \times m_t$ matrix

$$X = \begin{pmatrix} x_{11} & x_{12} & \dots & x_{1m_t} \\ x_{21} & x_{22} & \dots & x_{2m_t} \\ \vdots & \vdots & \ddots & \vdots \\ x_{m_s1} & x_{m_s2} & \dots & x_{m_sm_t} \end{pmatrix}.$$

From (5) we can write A as follows (see [4]):

$$A = (T^+(S^+X)^T)^T.$$

This suggests an efficient algorithm to compute A (or equivalently α):

1. Solve the least squares matrix equation $SC \approx X$ for C, an $l_s \times m_t$ matrix. This can be done efficiently since the psuedoinverse S^+ is computed once, and then each of the m_t columns of C can be found by a forward and a back substitution. In our implementation we use a modified version of deBoor's L2APPR [3], a one-dimensional least squares routine. We simply do one-dimensional approximation in s, with m_t right sides. We use the method of Cholesky to factor the normal equations.

2. Solve the least squares matrix equation $TD \approx C^T$ for D, an $l_t \times l_s$ matrix. Again, this is essentially a one-dimensional least squares problem, this time with l_s right sides. The solution is then simply $A = D^T$.

The tensor product formulation of the least squares problem (3) results in significant computational savings. Suppose for the sake of simplicity that $m_s = m_t = m$ and $l_s = l_t = l$. Table 1 gives the approximate operation counts for both the tensor product and full system approach. The dominant term in the work for the full system solution is larger than the dominant term for the tensor product formulation by a factor of $l^3/2$. The comparison is even more in favor of the tensor product formulation in our case, since the small support of the cubic B-spline basis functions implies that S and T have bandwidth 4. This bandedness is not present in the full matrix $(T \otimes S)$.

The least squares construction presented above does not guarantee that two important constraints are met by the approximation. Constraints are necessary because it is possible that the bicubic spline which best approximates the data is not a legal mapping. We must guarantee that points on the boundary of the solution domain S map to points on the corresponding side of Domain R, and that no two points in Domain S map to the same point in Domain R (i.e. F^{-1} must be everywhere invertible). In practice we approximate the first constraint by a penalty method. To keep points from moving off the boundary we weigh more heavily the equations in the overdetermined system involving those points. When constructing $x(s,t)$, points on the left and right boundary are given a weight 1000 times that of the remaining equations (10^6 in double precision), and the algorithm proceeds as described above.

The second constraint is harder to guarantee. Our solution is to require that $x(s,t)$ be a strictly increasing function with respect to s (and $y(s,t)$ with respect to t). To check for this, we construct the function $x(s,t)$ as described, and then evaluate the derivative $x_s(s,t)$ at each of the $m_s m_t$ points in Domain S. If this derivative is anywhere less than some small value ϵ (we use 0.1 when S is the unit square), we re-solve the problem with the constraint explicitly present; that is, we solve problem (3) subject to

$$(T \otimes S')\alpha \geq \epsilon, \tag{6}$$

Table 1: Operation counts for two approaches to the least squares problem.

Step of Algorithm	Full System	Tensor Product	
	$(T \otimes S)\alpha \approx \mathbf{x}$	$SC \approx X$	$TD \approx C^T$
Form normal equations	$m^2 l^4/2$	$ml^2/2$	$ml^2/2$
Cholesky factorization	$l^6/6$	$l^3/6$	$l^3/6$
Multiply right side by transpose	$m^2 l^2$	$m^2 l$	ml^2
Forward and back substitution	$l^4 + 2l^2$	$m(l^2 + 2l)$	$l(l^2 + 2l)$
Leading terms of total	$m^2(l^4/2 + l^2) + l^6/6$	$m^2 l + 3ml^2$	

where

$$(S')_{ij} = \Phi'_j(s_i), \quad i = 1, \ldots, m_s; \quad j = 1, \ldots, l_s.$$

Since our implementation currently requires the boundaries of the two domains R and S to be the same, the value ϵ is not scaled in any way. It would be a natural extension of our method to allow R and S to be of different sizes, in which case ϵ would be scaled appropriately.

Lawson and Hanson [6] give an algorithm for least squares with linear inequality constraints. We use a modification of this algorithm designed to take advantage of the tensor product nature of our problem. The least squares problem (3) subject to (6) is equivalent to the *least distance programming* (LDP) problem (see [6, Chapter 23])

$$\begin{aligned} &\text{minimize } \|\mathbf{y}\| \\ &\text{subject to } G\mathbf{y} \geq \mathbf{e}, \end{aligned} \tag{7}$$

where

$$G = (T \otimes S')(R_T \otimes R_S)^{-1}, \tag{8}$$

$$\mathbf{y} = (R_T \otimes R_S)\boldsymbol{\alpha} - (Q_{T1} \otimes Q_{S1})\mathbf{x}, \tag{9}$$

$$\mathbf{e} = \epsilon - (T \otimes S')(R_T \otimes R_S)^{-1}(Q_{T1} \otimes Q_{S1})\mathbf{x}, \tag{10}$$

and the orthogonal factorizations of S and T are

$$S = Q_S R_S = [\underbrace{Q_{S1}^T}_{l_s} \vdots \underbrace{Q_{S2}^T}_{m_s - l_s}],$$

$$T = Q_T R_T = [\underbrace{Q_{T1}^T}_{l_t} \vdots \underbrace{Q_{T2}^T}_{m_t - l_t}].$$

In the equations above, Q_S and Q_T are orthogonal matrices of order $m_s \times m_s$ and $m_t \times m_t$, respectively. R_S and R_T are upper triangular $m_s \times l_s$ and $m_t \times l_t$ matrices, respectively.

We are unable to take full advantage of the tensor product formulation at the central step of the algorithm—solving the LDP problem (7). However, efficient tensor product methods may be used to assemble G and $\mathbf{e}$ according to (8) and (10), and to compute $\boldsymbol{\alpha}$ from the matrix equation (9). Also, we only have to find factorizations of the small matrices S and T instead of the full matrix $T \otimes S$. With these optimizations, the algorithm takes approximately four times longer than the unconstrained tensor product least squares on typical size problems; but it is about four times faster than using the Lawson-Hanson routines on the full system directly. Fortunately, the constraints are violated infrequently—only when a highly irregular grid is to be approximated using a small number of spline pieces. Increasing the number of pieces in the approximation can sometimes help, if the dimension of the mapping grid is increased as well. However, in very bad cases it is possible that the constraints are met at the $m_s m_t$ points, but violated elsewhere. This leads to an illegal mapping and destroys the accuracy of the subsequent solution of the transformed PDE. In practice we find that mapping grids which are this distorted lead to dangerously nonsmooth mappings anyway, even if they are legal.

4 Evaluating the mapping

Since we expect to evaluate $x(s,t)$, $y(s,t)$ and their derivatives many times, it is worth the extra work to convert the approximation from the B-spline representation (2) to a *pp-representation* (see

[3, Chapter 10]). The coefficients in this representation are derivatives of the function at the break points of the piecewise polynomial. More formally, if the break points in the s and t directions are $\{\xi_i : i = 1,\ldots,k_s+1\}$ and $\{\eta_j : j = 1,\ldots,k_t+1\}$, respectively, then the coefficients of the pp-representation are given by

$$c_{i,p,j,q} = D_s^i D_t^j x(\xi_p^+, \eta_q^+) \quad \begin{array}{l} i = 0,\ldots,3; \quad p = 1,\ldots,k_s; \\ j = 0,\ldots,3; \quad q = 1,\ldots,k_t. \end{array} \tag{11}$$

In (11) we use the notation $D_s^i D_t^j x(\xi_p^+, \eta_q^+)$ to indicate the ith "right-derivative" of x with respect to s and the jth with respect to t at the point (ξ_p, η_q); that is, the derivatives in the positive ξ and η direction from (ξ_p, η_q). Since $x(s,t)$ is a bicubic spline, these derivatives are continuous except when $i = 3$ or $j = 3$. With this choice of coefficients, we define $x(s,t)$, for $\xi_p \le s < \xi_{p+1}$ and $\eta_q \le t < \eta_{q+1}$, as

$$x(s,t) = \sum_{j=0}^{3}\sum_{i=0}^{3} \frac{c_{i,p,j,q}(s-\xi_p)^i (t-\eta_q)^j}{(i)!(j)!}. \tag{12}$$

The coefficients $c_{i,p,j,q}$ in (11) are computed efficiently due to the tensor product nature of the B-spline representation in (2). We follow deBoor's suggestion and use a modified version of a routine which converts one-dimensional B-representations to pp-representations; two calls to this subprogram complete the computation of the coefficients.

Given the pp-representation (12), the derivatives of $x(s,t)$ are evaluated efficiently. From (12) the $\rho\sigma$ derivative of x is given by

$$\begin{aligned} D_s^\rho D_t^\sigma x(s,t) &= \sum_{j=\sigma}^{3}\sum_{i=\rho}^{3} \frac{c_{i,p,j,q}(s-\xi_p)^{(i-\rho)}(t-\eta_q)^{(j-\sigma)}}{(i-\rho)!(j-\sigma)!} \\ &= \sum_{j=\sigma}^{3}\left(\sum_{i=\rho}^{3} \frac{c_{i,p,j,q}(s-\xi_p)^{(i-\rho)}}{(i-\rho)!}\right)\frac{(t-\eta_q)^{(j-\sigma)}}{(j-\sigma)!}. \end{aligned} \tag{13}$$

Since we often evaluate $D_s^\rho D_t^\sigma x$ at each point in a grid (typically the uniform discretization grid in Domain S), we take advantage of the fact that the large parenthesized term in (13) is constant for each j, if s and q do not change (if s is constant, p certainly is). Thus, we evaluate along constant s-lines so we only have to compute the inner term once for each interval $[\eta_q, \eta_{q+1}]$. We rearrange the terms to evaluate efficiently along t-lines of a grid as well, if necessary.

In order to solve the transformed problem in S we need the first and second order partial derivatives of F with respect to x and y. These occur in the coefficients of the transformed equation. For example, if $u(x,y)$ is the unknown of a given problem, and $v(s,t)$ is the unknown of the transformed problem, then

$$u(x,y) = v(F(x,y)) = v(s(x,y), t(x,y)).$$

By simple differentiation, suppressing the independent variables, we have

$$\begin{aligned} u_x &= s_x v_s + t_x v_t, \\ u_y &= s_y v_s + t_y v_t, \\ u_{xx} &= s_x^2 v_{ss} + 2 s_x t_x v_{st} + t_x^2 v_{tt} + s_{xx} v_s + t_{xx} v_t, \\ u_{xy} &= s_x s_y v_{ss} + (s_x t_y + t_x s_y) v_{st} + t_x t_y v_{tt} + s_{xy} v_s + t_{xy} v_t, \\ u_{yy} &= s_y^2 v_{ss} + 2 s_y t_y v_{st} + t_y^2 v_{tt} + s_{yy} v_s + t_{yy} v_t. \end{aligned}$$

In our adaptive grid framework, we have an efficient means of evaluating F^{-1} and its derivatives, but we have no explicit representation for F. Thus, it is more efficient to compute the derivatives of F by using the derivatives of F^{-1}. For example, the first order derivatives of F with respect to x and y are given by:

$$\begin{aligned} s_x &= \ \ y_t/J, \\ s_y &= -x_t/J, \\ t_x &= -y_s/J, \\ t_y &= \ \ x_s/J, \end{aligned}$$

where

$$J \ = \ x_s y_t - x_t y_s.$$

5 Inverting the mapping

We have shown that evaluating the mapping F^{-1} or its derivatives can be done efficiently. We have also seen that the derivatives of F may be computed by using the derivatives of F^{-1}. Evaluation of $F(x,y)$ is another matter, however. Note that we do not need to evaluate F in order to solve the transformed problem; it is typically only needed when we want results on the original domain. For example, to evaluate the approximate solution at a point (x,y) in Domain R we need the solution to the transformed problem at the corresponding point $(s,t) = F(x,y)$ in Domain S.

Our solution to this problem is to evaluate F by inverting F^{-1} numerically. We use a special two-dimensional secant method to find a solution (s,t) of the equation

$$F^{-1}(s,t) - (x,y) = 0.$$

The fundamental step of the algorithm goes as follows. Given two previous guesses p_{k-2} and p_{k-1} in Domain S, evaluate F^{-1} at a third point $\hat{p}$ which lies in a direction normal to a line through p_{k-2} and p_{k-1} (see Figure 2). If the coordinates of p_{k-2} and p_{k-1} are (s_{k-2}, t_{k-2}) and (s_{k-1}, t_{k-1}), respectively, and if

$$h_s^{(k-1)} = s_{k-2} - s_{k-1} \qquad \text{and} \qquad h_t^{(k-1)} = t_{k-2} - t_{k-1},$$

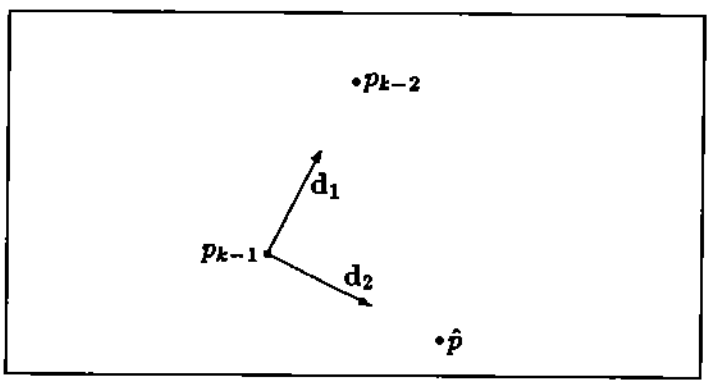

Figure 2: The fundamental step of the two-dimensional secant method.

then the coordinates of $\hat{p}$ are

$$\hat{s} = s_{k-1} + h_t^{(k-1)} \qquad \text{and} \qquad \hat{t} = t_{k-1} - h_s^{(k-1)}.$$

We locate a new guess (s_k, t_k) by finding step sizes from (s_{k-1}, t_{k-1}) in the directions $\mathbf{d}_1$ and $\mathbf{d}_2$ shown in Figure 2. Let these step sizes be h_1 and h_2, respectively. We use the values of F^{-1} at the three points:

$$\begin{aligned} F^{-1}(s_{k-2}, t_{k-2}) &= (x_{k-2}, y_{k-2}), \\ F^{-1}(s_{k-1}, t_{k-1}) &= (x_{k-1}, y_{k-1}), \\ F^{-1}(\hat{s}, \hat{t}) &= (\hat{x}, \hat{y}). \end{aligned}$$

For notational convenience, let the error in each coordinate of F^{-1} at a point (s,t) be given by

$$(\epsilon_x(s,t), \epsilon_y(s,t)) = (F^{-1}(s,t) - (x,y)),$$

where (x,y) is the point in Domain R where we want to evaluate F. Estimates of the first partial derivatives of ϵ_x and ϵ_y at (s_{k-1}, t_{k-1}) in the two directions $\mathbf{d}_1$ and $\mathbf{d}_2$ are given by

$$\begin{aligned} \frac{\partial \epsilon_x}{\partial d_1} &\approx \frac{\Delta \epsilon_x^{(1)}}{r} = \frac{\epsilon_x(s_{k-2}, t_{k-2}) - \epsilon_x(s_{k-1}, t_{k-1})}{r}, \\ \frac{\partial \epsilon_x}{\partial d_2} &\approx \frac{\Delta \epsilon_x^{(2)}}{r} = \frac{\epsilon_x(\hat{s}, \hat{t}) - \epsilon_x(s_{k-1}, t_{k-1})}{r}, \\ \frac{\partial \epsilon_y}{\partial d_1} &\approx \frac{\Delta \epsilon_y^{(1)}}{r} = \frac{\epsilon_y(s_{k-2}, t_{k-2}) - \epsilon_y(s_{k-1}, t_{k-1})}{r}, \\ \frac{\partial \epsilon_y}{\partial d_2} &\approx \frac{\Delta \epsilon_y^{(2)}}{r} = \frac{\epsilon_y(\hat{s}, \hat{t}) - \epsilon_y(s_{k-1}, t_{k-1})}{r}, \end{aligned}$$

where

$$r = \sqrt{\left(h_s^{(k-1)}\right)^2 + \left(h_t^{(k-1)}\right)^2}.$$

Since we want the error in each coordinate to go to zero, a two-dimensional secant step gives

$$\begin{pmatrix} h_1 \\ h_2 \end{pmatrix} = \begin{pmatrix} \frac{\partial \epsilon_x}{\partial d_1} & \frac{\partial \epsilon_x}{\partial d_2} \\ \frac{\partial \epsilon_y}{\partial d_1} & \frac{\partial \epsilon_y}{\partial d_2} \end{pmatrix}^{-1} \begin{pmatrix} \epsilon_x(s_{k-1}, t_{k-1}) \\ \epsilon_y(s_{k-1}, t_{k-1}) \end{pmatrix}.$$

A rotation yields new step sizes $h_s^{(k)}$ and $h_t^{(k)}$ in the s and t directions:

$$\begin{aligned} h_s^{(k)} &= \frac{h_s^{(k-1)}}{r} h_1 + \frac{h_t^{(k-1)}}{r} h_2, \\ h_t^{(k)} &= \frac{h_t^{(k-1)}}{r} h_1 - \frac{h_s^{(k-1)}}{r} h_2. \end{aligned}$$

Finally, the coordinates of the new guess are

$$s_k = s_{k-1} - h_s^{(k-1)} \qquad \text{and} \qquad t_k = t_{k-1} - h_t^{(k-1)}.$$

The inversion algorithm is quite efficient. It takes fewer evaluations of F^{-1} than a standard two-dimensional secant method in which two new function evaluations are needed at each step instead of one. In the present application, it also benefits from very good initial guesses. The approximate root from each completed search is saved so that if the next input point (x, y) is close to the previous one, the last root (s, t) is taken as an initial guess. This situation is common because the typical use for the algorithm is to estimate F on an entire grid of points, moving horizontally or vertically with either x or y fixed. If a free initial guess is unavailable, we still can find a good initial guess by finding the element in the mapping grid which contains the point (x, y). Recall that the mapping grid is a deformed rectangular grid on Domain R (see Figure 1). Once the desired element is found, we immediately know the grid element in the pre-image of the mapping grid on Domain S from which it came. This supplies a good initial guess for the secant method. It is only necessary to do a one-dimensional search when the point (x, y) lies on the boundary of R. This is because we require the mapping F^{-1} (and hence its inverse F) to map points on the boundary of one domain to points on the boundary of the other. This obviously simplifies the search in those cases.

6 The costs of adaptive grid domain mapping

The principle computational costs of adaptive grid domain mapping are the initial cost of adapting the grid and constructing the mapping, and the added cost in solving the transformed problem. Consider first the cost of constructing a domain mapping within our framework. The first step is to define an adapted mapping grid. The amount of work done here will vary according to what method is used. One would like to be able to generate good grids without performing computations which are just as expensive as solving the PDE itself. We report elsewhere (see [8] and [9]) on methods which do this in time that is small compared to the overall work of a typical numerical solution of a PDE.

Given the mapping grid, the tensor product least squares method described in Section 3 is used to compute the coefficients of the mapping F^{-1}. The cost of this computation is relatively small. Computing the bicubic spline coefficients for an $m \times m$ mapping grid with our tensor product least squares implementation takes work $O(km^2)$, where k, the number of spline pieces in each direction, is small. By contrast, assuming a discretization grid of $n \times n$ points, solving a linear system of $O(n^2)$ equations with bandwidth n is an $O(n^4)$ process (using Gauss elimination). As the number of discretization grid points increases, the time to solve the linear system will dominate the computation. Furthermore, we find that in most cases the size of the mapping grid m does not need to grow as fast as the size of the discretization grid n. Thus, the cost of constructing the adaptive mapping is basically fixed, or at worst grows slowly with respect to the dimension of the discretization grid. In our experience, the time taken to compute the adaptive mapping is less than the time for the rest of the computation by at least an order of magnitude, even when constraints must be imposed.

The second major cost introduced by domain mappings is the extra work required to discretize the transformed PDE. Recall that the coefficients of the transformed PDE include derivatives of the mapping F. Hence, one must evaluate these derivatives at many points, making the PDE coefficient evaluation more expensive for the transformed problem than for the original. The relative cost is especially bad when the original operator is simple. The domain mapping transforms an operator with simple coefficients into a very general, variable-coefficient operator. For example, if the original

Table 2: Relative times to discretize the original (R) and transformed (S) PDEs for several example problems.

Example	Time(S) / Time(R)			
	9 × 9 grid	13 × 13 grid	17 × 17 grid	21 × 21 grid
3.1	3.55	3.84	3.81	3.73
3.3	3.07	3.23	3.22	3.17
3.4	2.81	2.94	2.95	2.94
3.7	3.67	3.85	3.96	3.97
3.8	3.62	3.67	3.66	3.63

operator is the Laplacian $u_{xx} + u_{yy}$, the transformed operator is

$$(s_x^2 + s_y^2)v_{ss} + 2(s_x t_x + s_y t_y)v_{st} + (t_x^2 + t_y^2)v_{tt} + (s_{xx} + s_{yy})v_s + (t_{xx} + t_{yy})v_t.$$

We expect the time needed to discretize such an equation to be considerably greater than for the Laplacian. It may even be the case that a different numerical method must be used—because of the variable coefficients, for example.

The size of this added cost is not prohibitive, however, especially if the original operator is a nontrivial one. Our experience is that the time for the discretization phase of a numerical solution may increase by a factor of three to five when the original PDE is replaced by the more general transformed equation, depending on the complexity of the original operator and the discretization method used. For example, Table 2 lists the ratio of time to discretize the transformed PDE to time for the original PDE for four different grid sizes on five test problems from [7]. The elliptic operators represented by these problems range from the Laplacian (Problems 3.1 and 3.7) to a variable-coefficient problem with first derivative terms (Problem 3.4). The discretization module INTERIOR COLLOCATION from the the ELLPACK system [11] is used for each example. The data are from implementations on a Ridge 32 running ROS 3.3. The time for discretization in Domain S is from three to four times greater than the time in Domain R. The relative cost is highest for problems which have particularly simple operators. The important point is that the relative cost remains basically constant as the grid is refined. Hence, the time complexity of the discretization step has not changed, although we do face a considerable increase in the discretization time with the introduction of a domain mapping.

In terms of the entire computation, however, this increase is not as significant as it first appears. The time taken to solve the linear system generally remains fixed for a given grid size, whether the original or transformed equation is discretized. If we take solution time into account, the relative increase in discretization time is not as serious, and in fact decreases in relative importance as the grid is refined. This is not surprising since the linear system solution is typically an $O(n^4)$ process for an $n \times n$ discretization grid, while the discretization is typically an $O(n^2)$ process. Table 3 gives data for the same five examples as considered above, this time comparing the total time for discretization and linear system solution (by band Gauss elimination) for the two problems. The data indicate that the relative cost of the mapping is much less serious when the cost of the entire computation is taken into account. Furthermore, as the grid size grows, the relative cost shrinks toward unity.

We conclude that the costs introduced by domain mappings are not extremely significant. The time taken to construct a mapping is typically an order of magnitude less than that taken

Table 3: Relative times to discretize the original (R) and transformed (S) PDEs and solve the linear systems for five example problems.

	Time(S) / Time(R)			
Example	9 × 9 grid	13 × 13 grid	17 × 17 grid	21 × 21 grid
3.1	1.24	1.16	1.10	1.04
3.3	1.27	1.20	1.16	1.13
3.4	1.29	1.21	1.16	1.13
3.7	1.29	1.15	1.11	1.08
3.8	1.29	1.15	1.10	1.07

to numerically solve the PDE. The extra cost incurred in discretizing the transformed PDE is significant compared with a discretization on the original domain in some cases; but it is much less significant when the time for the entire computation is taken into account. Our experience is that for grids of moderate size, the solution of the transformed PDE typically takes at most 10% longer than the original problem in the same computing environment, and that this relative cost shrinks as the size of the discretization grid grows. Thus, with only a small penalty, we apply moving-grid adaption algorithms, while preserving the grid uniformity and regularity that makes vectorization and parallelization easier.

References

[1] I. Babushka, J. E. Flaherty and J. Chandra (eds.), *Adaptive Computational Methods for PDEs*, SIAM, Philadelphia, 1983.

[2] J. U. Brackbill and J. S. Saltzman, "Adaptive zoning for singular problems in two dimensions", *J. Comp. Phys.*, 46(1982), pp 342–368.

[3] C. deBoor, *A Practical Guide to Splines*, Springer-Verlag, New York, 1978.

[4] W. R. Dyksen, "Tensor product generalized alternating direction implicit methods for solving separable second order linear elliptic partial differential equations", Ph. D. Thesis, Department of Mathematics, Purdue University, 1982.

[5] P. R. Halmos, *Finite-dimensional Vector Spaces*, Van Nostrand, Princeton, New Jersey, 1958.

[6] C. Lawson and R. Hanson, *Solving Least Squares Problems*, Prentice-Hall, Englewood Cliffs, New Jersey, 1974.

[7] C. J. Ribbens, "Domain mappings: a tool for the development of vector algorithms for numerical solutions of partial differential equations", Ph. D. Thesis, Department of Computer Sciences, Purdue University, 1986.

[8] C. J. Ribbens, "A priori grid adaption strategies for elliptic pdes", Purdue University, Computer Science Department Report CSD-TR 667, 1987.

[9] C. J. Ribbens, "A fast grid adaption scheme for elliptic partial differential equations", preprint of paper, 1987.

[10] C. J. Ribbens, "The effectiveness of grid adaption for elliptic partial differential equations", preprint of paper, 1987.

[11] J. R. Rice and R. F. Boisvert, *Solving Elliptic Problems Using ELLPACK*, Springer-Verlag, New York, 1985.

[12] J. F. Thompson, "A survey of dynamically-adaptive grids in the numerical solution of partial differential equations", AIAA Technical Report 84-1606, 1984.

[13] J. F. Thompson, Z. U. A. Warsi and C. W. Mastin, *Numerical Grid Generation*, North Holland, New York, 1985.