Fast Contouring of Solutions to Partial Differential Equations *

E. L. Bradbury and W. H. Enright
Department of Computer Science
University of Toronto
Toronto
Canada, M5S 3G4

Abstract

The application of Differential Equation Interpolants (DEIs) to the visualization of the solutions to Partial Differential Equations (PDEs) is investigated. In particular, we describe how a DEI can be used to generate a fine mesh approximation from a coarse mesh approximation; this fine mesh approximation can then be used by a standard contouring function to render an accurate contour plot of the surface. However, the standard approach has a time complexity equivalent to that of rendering a surface plot, $O(fm^2)$ (where fm is the ratio of the width of the course mesh to the fine mesh). To address this concern three fast contouring algorithms are proposed which compute accurate contour lines directly from the DEI, and have time complexity at most $O(fm)$.

key words: visualization, PDEs, contour, rendering, interpolation

1 Introduction

1.1 Motivation

In scientific computation, visualizing the solution of a large scale simulation on a high resolution monitor is an important task. An accurate graphical representation allows scientists to analyze and share their results effectively. Much effort has been applied to find fast algorithms for computing accurate approximate solutions on relatively coarse meshes but, in order to visualize solutions effectively, approximations on finer meshes are often required. In this investigation, our focus will be on reducing the computational effort to render approximations to the solutions of partial differential equations (PDEs) on high resolution devices.

*This research was supported by the Natural Science and Engineering Research Council of Canada.

There are a number of numerical PDE solvers with varying degrees of accuracy. A high order method will generally have a high computational cost, and produce a discrete approximation on a relatively coarse mesh while a low order method will require a fine mesh for comparable accuracy. The storage costs that a fine mesh would impose can be prohibitive. One resolution of this latter difficulty is to save only a subset of the approximate solution (e.g. only the values associated with a coarse mesh). Whether a high order method or a low order method is used the final result will often be a relatively coarse mesh, accurate, discrete approximation to the solution of the PDE.

It is common practice to visualize solutions from within a Problem Solving Environment (PSE). In such environments, tools are available which allow the user to rotate the surface, zoom in on important features, and use special effects such as lighting and texture. A typical approximation, associated with a 2D underlying problem, might involve a 32×32 coarse rectangular mesh, but, in order to ensure that a surface appears continuous and smooth on a high resolution device (especially after zooming), a 500×500 to 1000×1000 (possibly finer) mesh may be required. In many cases the coarse mesh discrete approximation computed by a numerical method is not appropriate to render a surface acceptably. Current display methods, provided by popular PSEs such as MATLAB, use piecewise linear interpolation to approximate the solution at off-mesh points. The resulting images have noticeable linear artifacts that are distracting. In [3], Enright introduces the concept of a Differential Equation Interpolant (DEI) as an alternative to the more traditional interpolation methods. The DEI approximates the solution of a partial differential equation by an easily computed piecewise polynomial defined over a coarse mesh such that the accuracy of this approximation at off-mesh points is of the same order as the numerical PDE solver provided at the mesh points. The DEI makes it possible to compute an accurate fine mesh approximation from the original coarse mesh approximation that is appropriate for rendering a high resolution surface. This interpolant has been shown to be effective in rendering surface plots by Enright. In this paper, we will investigate the possibility of computing contour lines of a PDE efficiently and directly from a coarse mesh approximation and its associated DEI.

1.2 Partial Differential Equations

In this investigation we will consider approximate solutions to partial differential equations of the form,

$$Lu = g(x, t, u, u_x), \tag{1}$$

where $u_x \equiv \frac{\partial u}{\partial x}$. In particular, we will be considering parabolic PDEs where L has the form

$$L = \frac{\partial}{\partial t} - \beta(x, t)\frac{\partial^2}{\partial x^2}. \tag{2}$$

The approach we develop and algorithms we derive can be applied to a wide class of PDEs, such as the 2D and 3D higher order problems considered in [3].

In our numerical testing we use a numerical method of lines (NMOL) algorithm of Schiesser [7] to compute an approximate solution $\overline{u}(x,t)$ on a coarse rectangular mesh. In general, the computation of a DEI does not require a rectangular mesh as it can be applied to scattered discrete data as illustrated in [6]. Our algorithms and testing have been implemented using MATLAB but the approach and algorithms could be implemented in any PSE.

2 Background

2.1 Differential Equation Interpolant (DEI)

The DEI uses a bivariate piecewise polynomial to approximate the solution at off-mesh points. In this investigation we will restrict ourselves to the use of piecewise bi-cubics. This means that for each mesh element e, the approximate solution is represented by a polynomial of the form

$$p_e(x,t) = \sum_{i=0}^{3}\sum_{j=0}^{3} c_{i,j} x^i t^j. \tag{3}$$

If the mesh is irregular or scattered, triangular elements can be used to define the piecewise polynomials. In this investigation we will only consider rectangular meshes; however, a similar approach could be applied to triangular meshes. If the mesh is rectangular, then an element e is determined by the four 'corner points' $(x_a,t_b),(x_a,t_{b+1}),(x_{a+1},t_b),(x_{a+1},t_{b+1})$ that define the rectangle. In order to determine a cubic bivariate polynomial 16 linear constraints are specified. The first four constraints require that the polynomial p_e interpolate the approximate solution $\overline{u}$ at the four corner points. That is

$$p_e(x,t) = \overline{u}(x,t) \approx u(x,t). \tag{4}$$

where $\overline{u}(x,t)$ denotes the approximate solution (which is defined only at the mesh points) and $u(x,t)$ denotes the underlying true solution to the problem.

In this investigation we will be considering the application of the DEI to the solutions of parabolic PDEs, so in addition to having an approximate solution $\overline{u}$ at the mesh points, we will assume that we have an approximation $\overline{u}_x$ to $u_x(x,t)$ at the mesh points. The next four constraints require that the polynomial interpolate these values at the four corner points,

$$\frac{\partial p_e(x,t)}{\partial x} = \overline{u}_x(x,t) \approx u_x(x,t). \tag{5}$$

To complete the determination of the required piecewise bi-cubic, $p_e(x,t)$, eight points (x_k,t_k) are chosen from within the element, and the solution $u(x_k,t_k)$ and its derivative $u_x(x_k,t_k)$ are initially approximated by piecewise linear interpolation from the mesh data. The final 8 constraints require that the DEI almost satisfy the underlying differential equation (1) at these 8 interior points.

That is

$$Lp_e(x_k, t_k) = g(x_k, t_k, \hat{u}(x_k, t_k), \hat{u}_x(x_k, t_k)), k = 1, \ldots, 8, \tag{6}$$

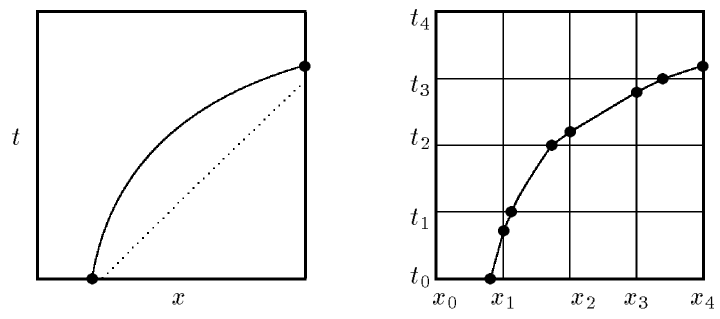
where $\hat{u}$ and $\hat{u}_x$ are the initial approximations to u and u_x respectively. Given these 16 constraints, the DEI algorithm determines the unique polynomial which then is known to accurately approximate the solution of the underlying PDE for any $(x, t) \in e$. Note that if the underlying numerical method does not provide accurate approximations to u_x at meshpoints, then alternative interpolating constraints (such as those based on 'nearby' meshpoints) and/or using more than 8 interior points where constraints are specified can be used to define the DEI. In such cases one must choose the points where the differential equation is to be 'almost' satisfied in a careful way to ensure that these constraints are linearly independent (see Enright [3] for details).

It is possible to evaluate a fine mesh approximation to $u(x, t)$ from a coarse mesh approximation using its DEI. If the coarse mesh is $cm \times cm$ and each element is refined by a factor of fm then the time complexity to evaluate the DEI is $O(fm^2 \cdot cm^2)$ (An $fm \times fm$ refinement of each of the cm^2 elements is required).

2.2 Matlab Contouring Algorithm

Before we discuss the direct computation of contour lines from DEIs, let us first understand how contour lines are computed in a typical PSE. The MATLAB contouring algorithm is described in [5]. The algorithm requires a regularly spaced rectangular mesh, so if one is not provided, it will create one based on the input mesh using piecewise linear interpolation. The algorithm then considers each of the regularly spaced rectangular mesh elements. If the desired contour level falls within the element, the algorithm uses piecewise linear interpolation to determine where the specified contour level intersects the edges of the element; these intersections are then connected with a straight line. Figure 2.2 shows a single coarse mesh element, e, and a typical contour line passing through it. Next to it in figure 2.2 is a depiction of how our MATLAB/DEI *contour* algorithm would approximate the line if the mesh were refined by a factor of four, and the associated fine-mesh approximation were generated (at the associated 25 fine-mesh points) using the DEI.

Given only a coarse mesh with no refinement, like one resulting from a numerical method, the underlying MATLAB contouring algorithm will not produce a smooth contour plot; instead, it will appear as several connected straight line segments (the dotted line of figure 2.2). To avoid these linear artifacts it is necessary to provide an approximation on a much finer mesh. One technique to obtain such an approximation would be to use the numerical method to return the finer mesh approximation; however, the added refinement would come at a high computational and storage cost. A second, more efficient, technique would be to compute the refined mesh approximations directly with a DEI as we have discussed earlier.



(a) True contour line associated with the piecewise polynomial of the DEI and a straight line approximation

(b) MATLAB approximation (refined in x and t directions)

Figure 1: MATLAB/DEI uses linear interpolation and requires a regularly spaced mesh.

The time required to compute the contour is dependent on the dimensions of the mesh. If the coarse mesh were $cm \times cm$, and each element is refined by a factor of fm then the time required to render a contour plot using our MATLAB/DEI approach is $O(fm^2 \cdot cm^2)$; to this we must add the time required to evaluate the DEI at the fine mesh. The total cost for computing a contour using this method is then $O(fm^2 \cdot cm^2)$. For comparison, consider the time complexity for rendering a 3D surface plot. To produce a surface plot, the $(fm \cdot cm)$x$(fm \cdot cm)$ points of the fine mesh approximation must be rendered; this requires $O(fm^2 \cdot cm^2)$ operations. Intuitively, when we are computing contours, we expect a reduction in computational effort corresponding to the reduction in the dimension of the object we are representing (a curve rather than a surface). It is disconcerting therefore that the time complexity of displaying a 2D contour plot for this approach is the same as the time complexity of rendering a 3D surface plot and we will now consider a modification to our approach which has lower complexity.

3 Fast Contouring Algorithm

Grandine explains in [4] how, with a closed-form expression for a surface, it is possible to compute contour lines by solving an associated ordinary differential equation (ODE) with a reliable numerical ODE solver. In scientific simulations, closed-form solutions rarely exist. Instead, we compute accurate approximations on coarse meshes. The DEI can be viewed as providing a closed-form expression for an accurate approximate solution. This raises the question: can the DEI

Table 1: Possible intercept scenarios that our algorithm must consider.

Intercepts	Implication
0	Contour line closed and completely contained within element.
1	Contour line is tangent to boundary.
2	A single contour line passes through element.
3	Part of a contour line passes tangent to an edge.
4	Two contour line segments pass through element.
> 4	More than two contour line segments pass through element.

be used to determine contour lines using an approach similar to that discussed by Grandine? In this investigation, we will show that it is indeed possible to compute approximate contour lines efficiently directly from the piecewise polynomial that defines the DEI.

The fast contouring algorithms we will consider process each element by approximating the local maximum and minimum values. Then for each contour level these values are used to determine whether a contour line passes through the element. If the desired contour level lies between the minimum and maximum values of the element then a segment of the desired contour line must be present within the element. To determine the contour line on elements where it is known to exist, an intersection test is performed on each of the four edges of the element. There are several possible outcomes of these four tests, and each must be treated differently when drawing the resulting segments of the contour line. These possibilities are summarized in Table 1.

In section 4.1 we will see that calculating the extrema information reliably can dominate the total running time of our approach. This is because calculating the extrema information rigorously requires that we determine where $\frac{\partial p_e(x,t)}{\partial x}$ and $\frac{\partial p_e(x,t)}{\partial t}$ are simultaneously zero. Both partial derivatives are bicubic polynomials. We use the built-in MATLAB function $fzero$ to reliably find such zeros, but the resulting computation comprises a significant fraction of the total time required to generate a contour. Note that finding the extreme values of a function approximated by a piecewise polynomial is an interesting problem in its own right and it can be viewed as a degenerate case of the contouring problem (where the desired contour level is equal to the the extreme value). For our purposes, only a crude approximation to the extreme values (and not their respective locations) is required and the MATLAB contouring algorithm uses the extreme values measured over the discrete fine mesh as the crude (but adequate) approximation to the extreme values. One can develop and justify other fast alternative algorithms which can provide similar crude approximations to these extreme values and we have implemented a technique using three steps of an 'orthogonal' descent algorithm starting from the extrema values on the boundary, to determine approximations to these extrema. Computing these crude approximations to local extrema makes it possible to reliably find contour lines even when they are closed and fully contained within an element.

The generic fast contour algorithm is presented in figure 2; it scans the DEI

Figure 2: Generic Fast Contouring Algorithm

for each element e
  lmin(e) $\leftarrow$ estimate of minimum value of $p_e(x,t)$ over $(x,t)\ \epsilon\ e$
  lMax(e) $\leftarrow$ estimate of maximum value of $p_e(x,t)$ over $(x,t)\ \epsilon\ e$
 for each contour level v
  if $lmin(e) \leq v \leq lMax(e)$ **then**
   for each edge s of e
    if v intersects s **then**
     intersections(e) $\leftarrow (x,t)_{intersect}$
    end if
   end for
   count $\leftarrow$ size(intersections(e))
   compute contour segment using FCINT, FCODE or FCODEA
  end if
 end for
end for

data structure, and prepares to compute the contours over each element. In this investigation, we consider in detail three techniques for computing fast contour lines within a single element: the Intercept method (FCINT); the simple ODE method (FCODE); and the ODE method with arclength (FCODEA).

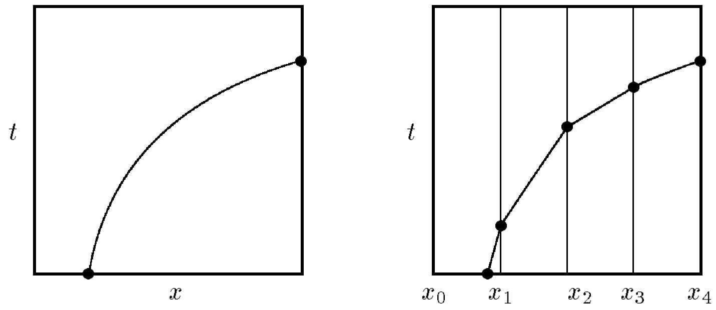
3.1 The Intercept Method

The Intercept method refines each element in one direction only; unlike the MATLAB algorithm, which requires each element to be refined in both the x and t directions. Along each 'line of refinement' (either x is constant or t is constant) $p_e(x,t)$ is reduced to a cubic univariate polynomial. If the refinement is in the x direction, for example (see figure 3), then determining where each line intersects the contour level v is accomplished by finding the roots of the equation,

$$a_3t^3 + a_2t^2 + a_1t + a_0 = v, \tag{7}$$

where the a_j's will depend on the specific constant value of x associated with the fm refinement in the x direction.

If there is more than one intersection per 'line of refinement' then there is no obvious method to determine how the intersection points should be connected. In such cases, instead of plotting a solid line the Intercept method will plot the individual intersection points. In the simple cases, when there is only one intersection per 'line of refinement', then they will be connected in the obvious order (either left to right if the refinement is in the x direction or top to bottom if the refinement is in the t direction).



(a) True contour line associated with the piecewise polynomial of the DEI

(b) FCINT approximation (refined in x only)

Figure 3: FCINT does not require refinement in both directions.

Since this method does not always connect the points, the possibilities listed in Table 1 do not require any extra special treatment. Figure 3.1 illustrates how the intercept method approximates the contour line in figure 3.1 with a refinement of four in the x direction, and figure 4 shows how FCINT handles cases when there are multiple intersections per 'line of refinement'.

It was found through experimentation that the direction of refinement has an effect on how dense the plotted points appear. With this in mind, we developed a heuristic for choosing the direction of refinement. If the element has at least two intersection points along its edges then the algorithm computes the change in x and t directions δx and δt respectively for the first pair of intersection

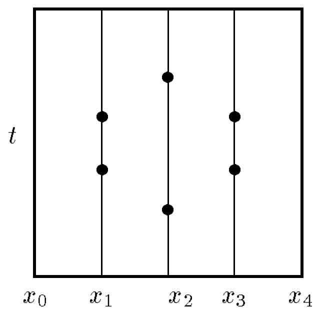


Figure 4: FCINT does not always connect the points.

Figure 5: Intercept method (FCINT) for computing contours

if count $= 0$ **then**
  refine in the x direction
else if count> 1 **then**
  Apply heuristic
  $(x,t)_l \leftarrow intersections(e,1)$
  $(x,t)_r \leftarrow intersections(e,2)$
  $(\delta x, \delta t) \leftarrow (x,t)_r - (x,t)_l$
  if $|\delta x| > |\delta t|$ **then**
    refine in the x direction
  else
    refine in the t direction
  end if
else
  make no refinement (count must be $= 1$)
end if
for each line of refinement
  solve the univariate cubic equation $p_e = v$ for its roots
end for
plot intersection points

points. If $\delta x > \delta t$ then it refines in the x direction otherwise it refines in the t direction. The algorithm in figure 5 provides an overview of how the Intercept method draws contours.

Because this method only requires refinement in one direction, the cost is $O(fm \cdot cm^2)$. This algorithm does not need a refined mesh as input since the refinement is done by the algorithm itself; thus it does not incur the cost of evaluating the DEI on a fine mesh as our MATLAB/DEI method does. Note that the constant associated with this $O(fm \cdot cm^2)$ can be quite large as it involves forming and determining the roots of a cubic polynomial.

3.2 The Simple ODE Method

The contour line can be treated as if it were the solution to an initial value problem (IVP). If we are approximating the contour level v of the PDE solution u we could use the curve $(x, t(x))$ where

$$p_e(x, t(x)) = v. \tag{8}$$

If we differentiate this expression with respect to x and drop the subscript e (where it is obvious) we obtain the ODE,

$$p_x(x, t(x)) + p_t(x, t(x)) \frac{\partial t}{\partial x} = 0, \tag{9}$$

or

$$t_x(x) = -\frac{p_x(x, t(x))}{p_t(x, t(x))}. \tag{10}$$

For any fixed value of x, $t(x)$ can be evaluated (if it exists) by solving a univariate polynomial equation (as in FCINT).

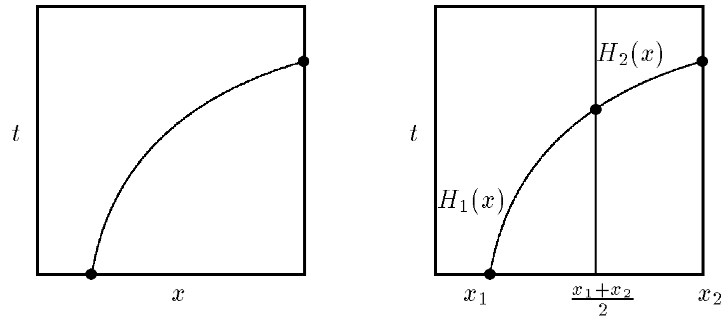
We could alternately express x in terms of t and differentiate with respect to t. In this case we obtain the ODE,

$$x_t(t) = -\frac{p_t(x(t), t)}{p_x(x(t), t)}. \tag{11}$$

If we apply a reliable IVP solver to approximate the solution of (10) with two equal step sizes, $\Delta x = \delta x/2$, then it would provide an approximation to the solution, $t(x)$, and $t_x(x)$ at each end point (the intersection points with the boundary of e) and at the midpoint. But we can also find accurate approximations for these 6 values from the DEI directly without having to apply the IVP solver because the DEI provides not only an approximation $p(x,t) \approx u(x,t)$ but also $p_x(x,t) \approx u_x(x,t)$ and $p_t(x,t) \approx u_t(x,t)$ at any off-mesh point. With these values one can use (10) to determine the value of $t_x(x)$ at the boundary points. To determine $t(x)$ and $t_x(x)$ at the midpoint we use the approach of FCINT to 'refine' along the central x value $\frac{x_1+x_2}{2}$. We can then approximate the desired contour directly with Hermite interpolants $H_1(x) \approx t_1(x)$ defined on $[x_1, \frac{x_1+x_2}{2}]$ and $H_2(x) \approx t_2(x)$ defined on $[\frac{x_1+x_2}{2}, x_2]$. Note that $H_1(x)$ and $H_2(x)$ are interpolants of the exact solution of the IVP associated with (10). If the IVP associated with (11) is used a similar approximation to $x(t)$ is employed. Figure 3.2 shows how FCODE approximates the contour line depicted in figure 3.2. The algorithm outlined in figure 7 gives more details.

We have discussed the most common case of having two intersection points. The other cases listed in table 1 must be given special consideration. For the case when there are no intersections with the element boundary a local extrema $(\hat{x}, \hat{t})$ must exist and FCODE divides the element into four new sub-elements e_1, e_2, e_3, and e_4 as shown in figure 8. The location of the local extrema $(\hat{x}, \hat{t})$ can be used to determine the 'mesh lines' that define this division. A straightforward extension of the 2-intercept procedure can then be used to connect these four points.

For the case when there are four intersections with the element boundary, we make some reasonable assumptions: first, contours do not start and end at the same point; and second, they do not cross each other. Figure 3.2 and 3.2 show the two possible contour line configurations. To discover which configuration to draw, FCODE chooses a test line, either a constant x or constant t, such that two points lie to each side of the line. If the contour lines intersect the test line then the configuration in figure 3.2 should be rendered; otherwise the configuration in figure 3.2 should be rendered. FCODE then applies the normal (2 intercept) procedure to each of the 'matched' pairs of end points. In the most common cases this heuristic is sufficient; however, it is possible (with very coarse meshes) for it to fail. In such cases, an approach similar to that used



(a) True contour line associated with the piecewise polynomial of the DEI

(b) FCODE approximation for two intersection points

Figure 6: FCODE computes $t(x)$ and $t_x(x)$ at the midpoint and end points, and then uses Hermite Interpolants $H_1(x)$ and $H_2(x)$ to connect them.

Figure 7: ODE method for computing contours

if count $= 0$ **then**
  split element up into four sub-elements
  if v intersects new edges **then** compute interpolant for each segment
  else ignore contour display appropriate warning message
else if count$= 2$ **then**
  $(x,t)_l \leftarrow intersections(p,1)$, $(x,t)_r \leftarrow intersections(p,2)$
  $(\delta x, \delta t) \leftarrow (x,t)_r - (x,t)_l$
  if $|\delta x| > |\delta t|$ **then**
    $x_m \leftarrow (x_r + x_l)/2$
    find t_m such that $u(x_m, t_m) = v$
    interpolate between $(x,t)_l$ and $(x,t)_m$, and between $(x,t)_m$ and $(x,t)_r$ (using piecewise Hermite)
  else
    $t_m \leftarrow (t_r + t_l)/2$
    find x_m such that $u(x_m, t_m) = v$
    interpolate between $(x,t)_l$ and $(x,t)_m$, and between $(x,t)_m$ and $(x,t)_r$ (using piecewise Hermite)
  end if
else count$= 4$ **then**
  identify which intersection points are connected within the element
  for each pair proceed as with procedure for two intersection points
end if
evaluate interpolants at fm points and plot

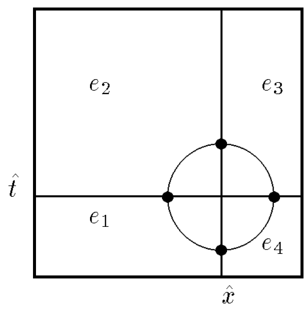


Figure 8: If there are no intersections with the element boundary FCODE subdivides the element into four sub-elements.

in the 0 intercept case might be applied with more success (although in our implementation we halt with a warning message).

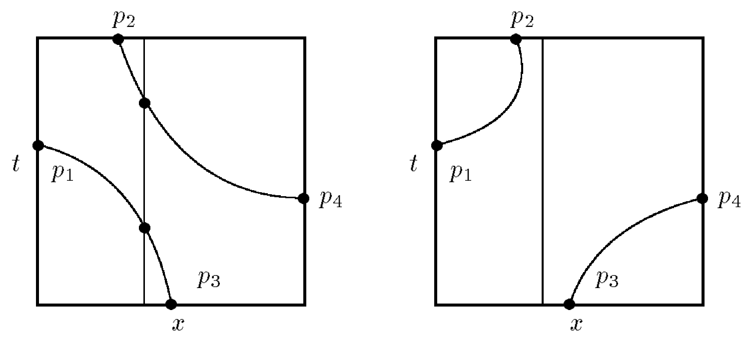
The final three possibilities are: one, there are more than 4 intersections; two, there is 1 intersection; three, there are 3 intersections. These cases raise a warning message in the current version of FCODE. When there are more than four intersection points we assert that the initial mesh approximation must be too coarse because there are more than two contour lines of the same level passing through a single element. FCODE will continue to run but will print a warning that suggests the user should try a finer mesh.

In the case of a single intersection point, the contour line must be tangent to the current element. We then evaluate the number of intersections on a nearby parallel mesh line. If there are 2 intersections we must have a local extrema $(\hat{x}, \hat{t})$ and we can treat it like the 0 intersections case. If the number of intersections on the nearby parallel line is 0 then we ignore this contour line as it will be found when FCODE considers the adjacent element.

When the number of intersections is equal to 3 we compute such contours by counting the point where the contour is tangent to the edge twice, and applying the same strategy as used for 4 intersection points. The tangent point will be the point (of the 3) where the nearby parallel mesh line has 2 rather than 1 intersection points (see figure11).

The cost of evaluating the contour using this technique is quite small. This is because at each element, on average two Hermite polynomials are determined and evaluated at fm points. The time complexity of this algorithm is $O(fm \cdot cm^2)$. It is important to note that the constant associated with the $O(fm \cdot cm^2)$ term is small since evaluating a polynomial at fm points can be relatively inexpensive compared to determining the Hermite polynomial.

It must be acknowledged that this technique will fail if the associated ODEs ((10) and (11)) are both near singular in the range of interest. That is, both $p_t(x, t(x))$ and $p_x(x(t), t)$ are equal to (or close to) zero. In this case we could



(a) The contour line crosses the test line, so p_1 and p_3 are joined, and p_2 and p_4 are joined.

(b) The contour line does not cross the test line, so p_1 and p_2 are joined, and p_3 and p_4 are joined.

Figure 9: With 4 intersection points, FCODE tries to determine which intersection points belong to the same contour line.

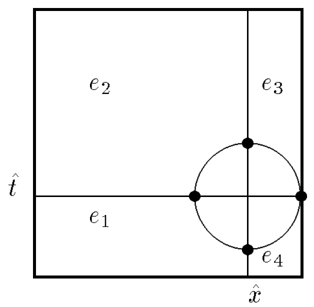


Figure 10: If there is a single intersection with the element boundary and a nearby mesh line intercepts the contour line twice then FCODE subdivides the element into four sub-elements.

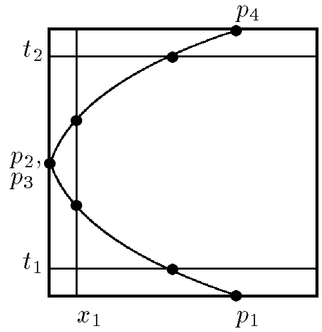


Figure 11: When there are 3 intersection points, we perform intersection tests on 'nearby' mesh lines. An intersection point will be counted twice if its 'nearby' mesh line has 2 intercepts.

use a more robust parameterization to obtain a fast technique.

3.3 The ODE Method with arclength

Instead of using the parameterization in (8), consider parameterization with respect to arclength, (s),

$$p(x(s), t(s)) = v. \tag{12}$$

If we differentiate with respect to arclength we have,

$$p_x x_s + p_t t_s = 0. \tag{13}$$

If we add the condition that arclength be normalized we obtain,

$$(x_s)^2 + (t_s)^2 = 1. \tag{14}$$

From (13) and (14) we can solve for x_s and t_s to obtain the system of ODEs,

$$x_s = \sqrt{1 - (t_s)^2}, \tag{15}$$

$$t_s = \frac{\frac{p_x}{p_t}}{\sqrt{1 + (\frac{p_x}{p_t})^2}}. \tag{16}$$

We could use a reliable IVP solver to approximate $x(s)$ and $t(s)$; however, we can use a similar approach to that employed in FCODE to generate directly approximations to $x(s)$, $t(s)$, $x_s(s)$, and $t_s(s)$ from the DEI. In this case, if we use a two step process we will need to find four Hermite interpolants two each for $x(s)$ and $t(s)$.

One complication with this method is that the value of the total arclength (between the associated boundary points) is not known in advance. It turns out that the accuracy of this approach is not very sensitive to this value and that approximating the total arclength by the minimum distance between the two boundary points is sufficient. (It can be shown that with this assumption the two Hermite interpolants approximating $x(s)$ have an accuracy of $O(\delta x)^4$ while the interpolants approximating $t(s)$ have an accuracy of $O(\delta t)^4$.)

We did not implement this method because the simple ODE approach described in the previous section performs well, and this alternate approach would require at least twice the computational effort since it requires us to compute and render four polynomial interpolants instead of two. The FCODE approach will only behave poorly if both IVPs ((10) and (11)) are nearly singular; the arclength approach will not suffer from this difficulty. It is possible for this situation to arise but it is unlikely and, in any event, it is detectable. Another advantage of the arclength approach is that it is not necessary to determine a direction of refinement.

4 Numerical Results

In this section, we will introduce two test problems and measure the effectiveness of the three contouring algorithms: MATLAB/DEI, FCINT, and FCODE. (We investigate an additional problem in [1].) Since the purpose of the fast contouring algorithm is to improve rendering speed, our comparison will be based primarily the CPU time required to compute and render the contour plot using each algorithm. We will also consider the visual appearance of each contour.

To compare the speed of the algorithms we use MATLAB's *cputime*, which returns the total CPU time used by MATLAB from its start up. To ensure a fair comparison evenly spaced coarse meshes were used in all tests. It is important to note that the MATLAB/DEI approach expects evenly spaced mesh points for the fine mesh, but the DEI has no such inherent requirement and, in particular, all approaches could be applied with arbitrary coarse meshes.

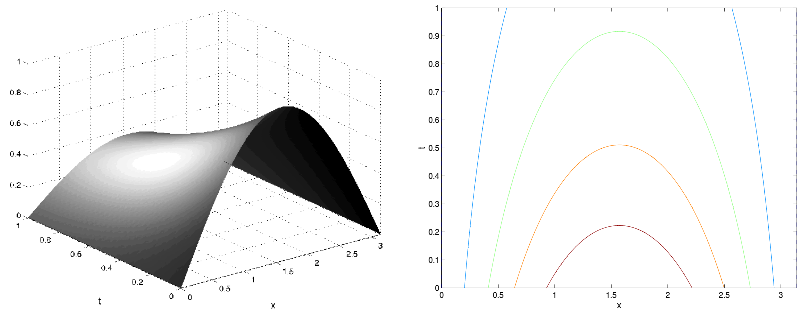
4.1 Test Problems

We will consider two test problems. Since the DEI interpolates based on the assumption of an underlying differential equation, we will consider the solutions to partial differential equations of the form (1), (2). The NMOL package does not provide an approximation to u_x, so we use a second order divided difference scheme to determine the discrete meshpoint approximation, $\overline{u}_x$. Both test problems are linear. Our approach will work with non-linear problems provided the coarse mesh approximation is accurate enough.

The closed-form solutions of the first test problem is known; while a closed form-solution to the second test problem is not known. The problems are defined by equations (17) and (19).

The first is a simple one dimensional heat problem

$$\frac{\partial u}{\partial t} - \frac{\partial^2 u}{\partial x^2} = 0 \tag{17}$$



(a) Surface representation with lighting effects.

(b) Default MATLAB/DEI contour plot.

Figure 12: Solution to the Heat test problem

where

$$\begin{aligned} u(0,t) = u(\pi,t) = 0, &\qquad t > 0, \\ u(x,0) = \sin x, &\quad 0 \le x \le \pi. \end{aligned}$$

The closed-form solution is known to be

$$u(x,t) = e^{-t} \sin x. \tag{18}$$

Figure 12 shows the surface and contour plots of the solution.

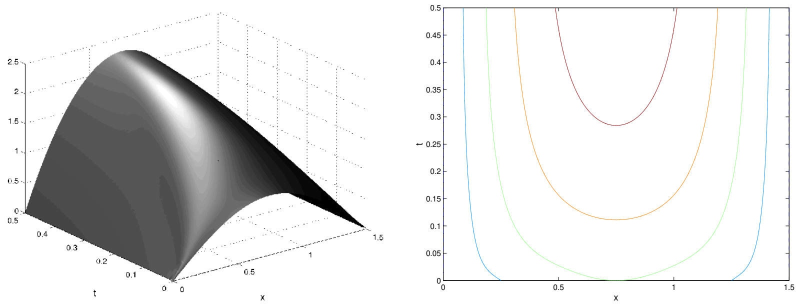
Our second problem, which comes from [2] page 717 question 12, is

$$\frac{\partial^2 u}{\partial x^2} + \frac{\kappa r}{\rho C} = \kappa \frac{\partial u}{\partial t}, \tag{19}$$

where

$$\begin{aligned} u(0,t) = u(l,t) = 0, &\qquad 0 \le x \le l, \\ u(x,0) = \sin \frac{\pi x}{l}, &\qquad t > 0, \\ l = 1.5cm, &\quad \kappa = 1.04cal/cm \cdot deg \cdot s, \\ \rho = 10.6g/cm^3, &\quad C = 0.056cal/g \cdot deg \\ and &\quad r(x,t,u) = 5.0cal/cm^3. \end{aligned}$$

This problem models the temperature of a thin conducting rod such as those used in a nuclear reactor. Here l is the length of the rod, κ is its thermal diffusivity, ρ is the density of the conduction material, C is the specific heat capacity of the conducting material and r is the heat generated per unit volume. The ends of the rod are kept at 0^oC. Because $r(x,t,u)$ is a constant function,



(a) Surface representation with lighting effects.

(b) Default MATLAB/DEI contour plot.

Figure 13: Solution to the Nuclear test problem

(19) is linear. This problem could be made to be non-linear by making r depend on u. The surface and contour plots of the solution to (19) are shown in figure 13.

4.2 Test Results

Using the NMOL package, run on a Unix platform with Solaris 8 and 2@600MHz processors, it was found that a discrete mesh of $(x \times t) = (512 \times 16)$ produced results that were sufficiently accurate (2 or 3 significant digits) to generate our coarse mesh discrete approximations for both test cases. The running time to compute this approximate solution was $1.6s$ and $2.2s$ for the Heat and Nuclear problems respectively. In the next section we will see that only a small fraction of this mesh needs to be stored.

4.2.1 Choosing an appropriate coarse mesh

The choice of the coarse mesh, represented by the integer, cm, is influenced by two opposing factors: a large cm will ensure that the DEI has appropriate accuracy; a small cm will allow for fast determination of the DEI and will save storage. The value of cm is chosen so that it is as small as it can be without compromising the accuracy of the DEI.

To determine an appropriate cm value for the Heat problem, we compare the three contour plots in figure 14. The three DEIs used were based on coarse meshes corresponding to $cm = 2^2, 2^3, 2^4$; they were then evaluated on the same fine mesh. The contour plots were created by using MATLAB's underlying *contour* applied to the respective fine mesh surface approximation associated with each DEI. It is clear that in figure 4.2.1 the contour is not accurate because the contour lines are not continuous. Doubling the coarse mesh to $cm = 2^3$ removes the apparent discontinuities in figure 4.2.1, however figure 4.2.1 has a more pleasing appearance (especially when viewed on a standard monitor); thus we will choose $cm = 2^4$ for the Heat problem.

We perform the same assessment for the Nuclear problem to determine an appropriate value of cm. Figure 15 shows the results of this exercise. Discontinuities are clearly visible in figure 4.2.1. From close inspection of figure 4.2.1 we can still see some discontinuities especially when viewed on a full-screen rendering. Figure 4.2.1 appears to be continuous, so we chose $cm = 2^4$ for this problem as well.

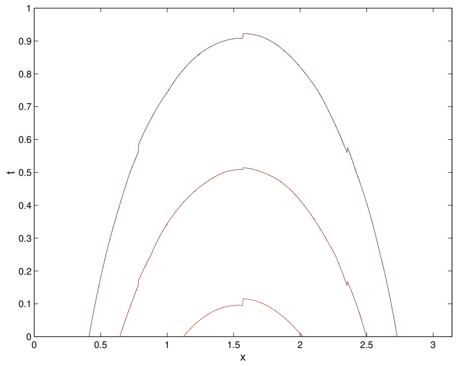
4.2.2 Running time

The mean CPU time (obtained using MATLAB 6.0 run on a Mandrake Linux system with a 2.0GHz pentium 4) for each approach is calculated using an average over ten runs. The timing information in Tables 2 and 3 is based on generating a single contour line. The fm values were chosen based on the resolution required to produce smooth contours on a high resolution computer monitor (this was interpreted as $fm \cdot cm = 2^9$). In some applications one is interested in generating k different contour curves simultaneously (as allowed in the algorithm presented in figure 2). In these cases, the computation of the extrema values and the determination of the DEI need only be done once. Similarly. for MATLAB/DEI, the multi-point evaluation of the DEI (on the fine mesh) need only be done once.

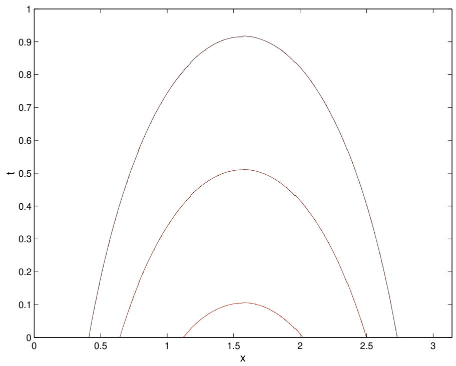
As discussed earlier the fast contouring algorithms could be run with a slow but accurate extrema calculation (SBA) or a faster less accurate extrema calculation (FLA) on each element. The time required to form the DEI is not included in the tables because the DEI need only be determined once and stored for future use, and it is required for each method. The time complexity of the algorithm for computing the DEI is $O(cm^2)$. For both of the test problems we use a coarse mesh of $cm = 2^4$, and the running time for determining the DEI was approximately 4.25s. (Note that determining the DEI could be implemented as part of the solution process by the NMOL package and this would result in a considerable reduction in this computer time – particularly since the polynomials associated with each coarse mesh element can be generated in parallel.)

The time complexities for MATLAB/DEI, FCINT, and FCODE are $O(fm^2 \cdot cm^2)$, $O(fm \cdot cm^2)$, and $O(fm \cdot cm^2)$ respectively for a single contour level (k=1). From this, we can predict that as fm increases the running time for the MATLAB/DEI technique will grow faster than the time required for either fast contour approach. The results reported in tables 2 and 3 are consistent with these predictions.

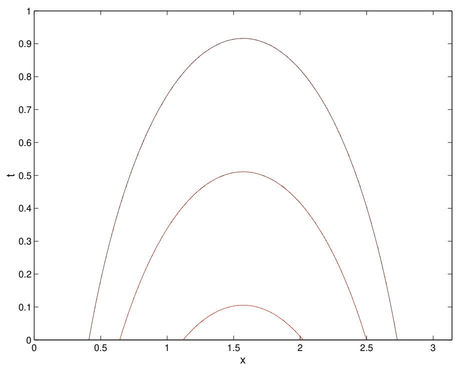
When multiple contour levels are to be determined ($k > 1$) the time complexities for our techniques must be considered more carefully before we can predict the expected relative performance. We assume that the dominant cost of the techniques we have implemented are associated with the determination or evaluation of a cubic (or bi-cubic) polynomial. We also assume that, in the applications we are considering we will have $k < fm$. (Note that if this condition is not satisfied then the resolution of the contour plots may not be sufficient to



(a) $cm = 2^2$

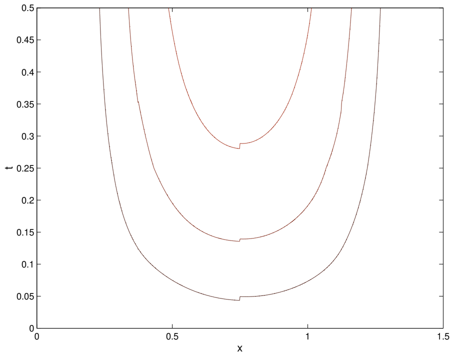


(b) $cm = 2^3$

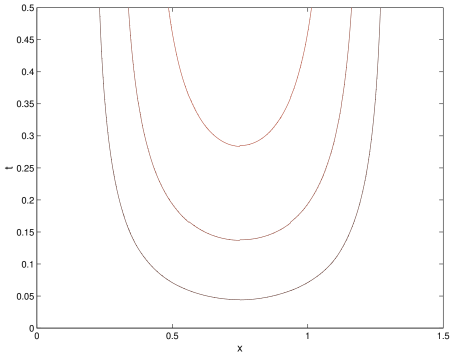


(c) $cm = 2^4$

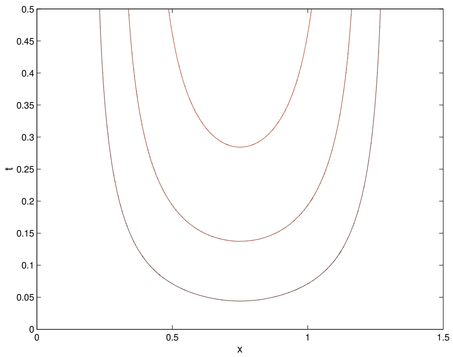
Figure 14: Comparison of Heat solution contours for three coarse meshes



(a) $cm = 2^2$



(b) $cm = 2^3$



(c) $cm = 2^4$

Figure 15: Comparison of Nuclear solution contour for three coarse meshes.

Table 2: Total CPU time (secs) for the Heat problem (with $cm = 2^4$ and $k = 1$)

fm	Algorithm				
	MATLAB/DEI	FCINT		FCODE	
	DEI and *contour*	FLA	SBA	FLA	SBA
2^3	1.236	0.887	1.689	0.924	1.667
2^4	1.865	1.014	1.803	0.924	1.672
2^5	4.202	1.289	2.019	0.921	1.665

Table 3: Total CPU time (secs) for the Nuclear problem (with $cm = 2^4$ and $k = 1$)

fm	Algorithm				
	MATLAB/DEI	FCINT		FCODE	
	DEI and *contour*	FLA	SBA	FLA	SBA
2^3	1.344	0.912	4.074	0.954	4.154
2^4	1.858	1.040	4.172	0.948	4.032
2^5	4.264	1.290	4.378	0.950	4.044

distinguish the different contour levels.) With these assumptions, it is then clear that the maximum per-element time complexities for MATLAB/DEI, FCINT and FCODE are $O((fm+1)fm)+O(fm)$, $O(k \cdot fm)+O(fm)$ and $O(k \cdot fm)+O(fm)$ respectively. The results reported in tables 4 and 5 are consistent with these complexity estimates.

In summary then, after the approximate solution has been determined on a coarse mesh for our two test problems (for example using NMOL this required, as we have noted above, approximately 2s for each problem), the total time required to 'contour the approximate solution' in the MATLAB PSE is approximately 4.25s (to determine the coefficients of the underlying DEI) plus the times reported in tables 2-5 for each of the alternative techniques we have introduced and analyzed.

Table 4: Total CPU time (secs) for the Heat problem (with $cm = 2^4$ and $k = 10$)

fm	Algorithm				
	MATLAB/DEI	FCINT		FCODE	
	DEI and *contour*	FLA	SBA	FLA	SBA
2^3	1.200	2.332	2.908	2.108	2.764
2^4	1.867	3.288	3.684	2.112	2.751
2^5	4.337	5.193	5.219	2.121	2.759

Table 5: Total CPU time (secs) for Nuclear problem (with $cm = 2^4$ and $k = 10$)

fm	Algorithm				
	MATLAB/DEI	FCINT		FCODE	
	DEI and *contour*	FLA	SBA	FLA	SBA
2^3	1.211	2.691	5.424	2.416	5.225
2^4	1.870	3.786	6.307	2.416	5.236
2^5	4.382	5.998	8.118	2.424	5.228

4.2.3 Visualization

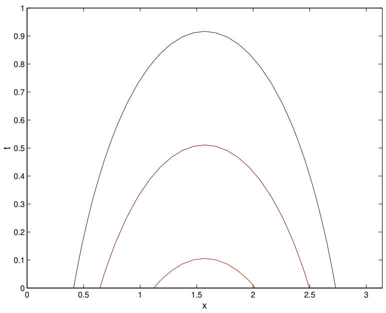
Figures 16 and 17 show side by side comparisons of contour plots at two different resolutions. The left-hand column shows the lowest resolution that produces acceptable results (i.e. non-distracting). The right-hand column shows a higher resolution that produces more appealing contours. The first, second, and third rows show the results from the MATLAB/DEI, FCINT, and FCODE methods respectively.

Careful inspection of the plots and analysis of $\overline{u}(x,t) - u(x,t)$ reveals that FCINT is the most accurate, while FCODE can suffer from small oscillations in the error and MATLAB/DEI can be inaccurate when fm is small. This is due to the fact that all the computed discrete values of the FCINT approach are values where $\overline{u}(x,t) = v$ while with FCODE there is an additional contribution to the error coming from the use of Hermite interpolation, and with MATLAB/DEI a piecewise linear interpolant of $\bar{u}$ is used on the fine mesh. The oscillations of FCODE can be reduced by increasing the value of cm, or by taking four steps instead of two when approximating the solution of the ODE. This would increase the overhead cost of FCODE, but at high resolutions FCODE would still out perform the other two techniques because it has smaller time complexity with respect to fm.

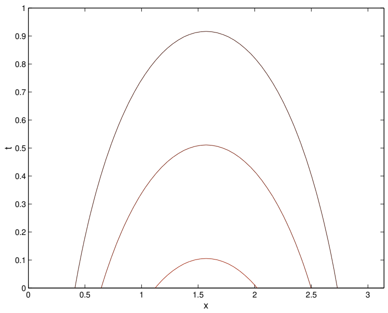
When the solution to a simulation approaches a particular constant value v it is difficult to contour at that specific value, especially if absolute error is used by the numerical PDE solver. The approximate solution is generally not constant in such cases; instead, it oscillates around the true solution. One way to address this difficulty would be to use relative error in the PDE solver; however, this may cause the PDE solver not to converge. The difficulty is inherent in that what is really happening is that the 'contour curve' is degenerating into a contour surface for the specified values of v.

4.3 Conclusions

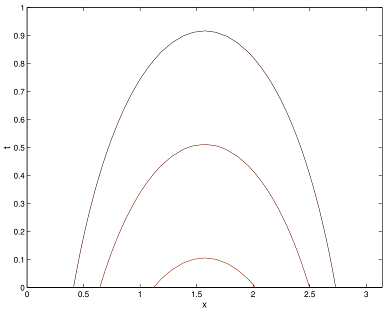
We have introduced efficient contouring algorithms for the solution of PDEs that treat the DEI as a closed-form expression for the surface that is to be contoured. We have compared two algorithms with a straight forward approach using a fine mesh approximation calculated from the DEI and the built-in MATLAB function *contour*. Testing confirmed that the time complexities of the three approaches MATLAB/DEI, FCINT and FCODE, with respect to the refinement fm of the



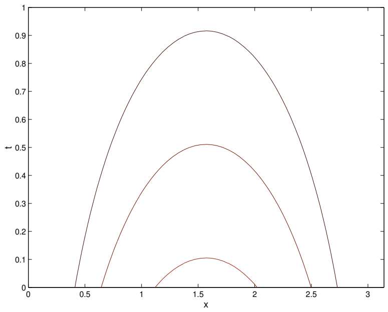
(a) MATLAB/DEI ($fm \cdot cm = 2^5$)



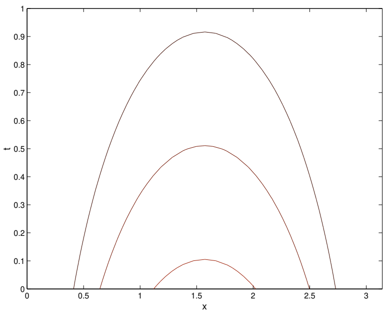
(b) MATLAB/DEI ($fm \cdot cm = 2^6$)



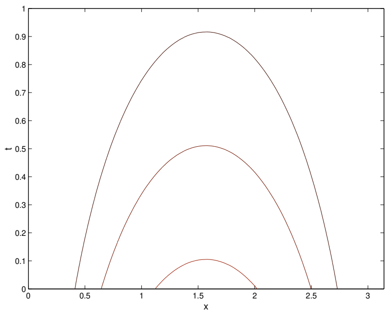
(c) FCINT ($fm \cdot cm = 2^5$)



(d) FCINT ($fm \cdot cm = 2^6$)

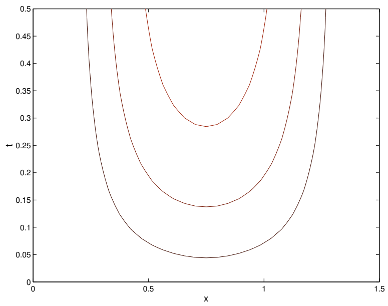


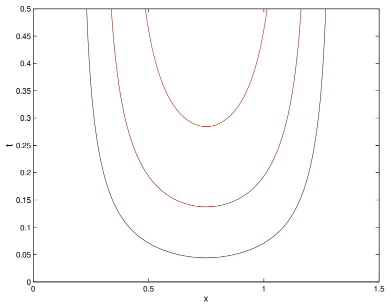
(e) FCODE ($fm \cdot cm = 2^5$)



(f) FCODE ($fm \cdot cm = 2^6$)

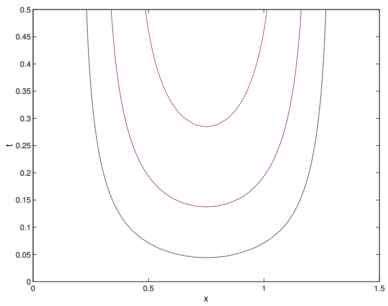
Figure 16: Contours for Heat test problem based on a 2^4x2^4 coarse mesh.

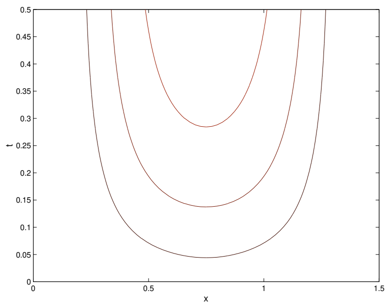




(a) MATLAB/DEI ($fm \cdot cm = 2^5$)

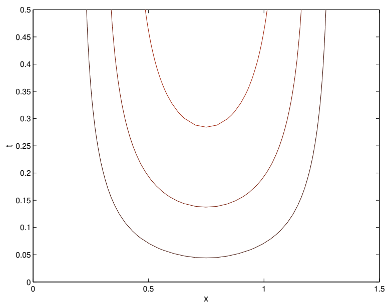
(b) MATLAB/DEI ($fm \cdot cm = 2^6$)

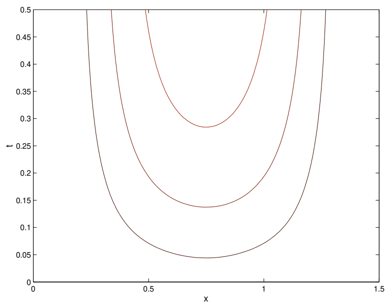




(c) FCINT ($fm \cdot cm = 2^5$)

(d) FCINT ($fm \cdot cm = 2^6$)





(e) FCODE ($fm \cdot cm = 2^5$)

(f) FCODE ($fm \cdot cm = 2^6$)

Figure 17: Contour for the Nuclear test problem based on a 2^4x2^4 coarse mesh.

coarse mesh were $O(fm^2)$, $O(fm)$, and $O(fm)$ respectively. Thus, we have shown that the fast contouring algorithms are less sensitive to the magnitude of fm than the MATLAB/DEI algorithm. This is particularly important if a zooming function is to be applied to the contour plot, or if the image is to be viewed at very high resolution. We have seen that the fast contouring algorithms produce contours that are nearly indistinguishable from the contours computed by the MATLAB/DEI algorithm.

This investigation can be viewed as the initial step in the development of a new approach for efficiently visualizing approximate solutions of PDEs. The fast contouring algorithms may be adapted in a straightforward way to handle irregular meshes, triangular elements and 3 dimensional simulations. We have used MATLAB to implement the fast contouring algorithms because it provides a convenient environment for prototyping numerical algorithms. If they are to reach their full potential, and applied in applications involving high performance simulations, the fast contouring algorithms should be rewritten in a procedural language such as C or FORTRAN instead of the slower interpreted language of MATLAB.

References

[1] Emma L. Bradbury. Fast contouring of solutions to partial differential equations. Master's thesis, Department of Computer Science, University of Toronto, 2002.

[2] Richard L. Burden and J. Douglas Faires. *Numerical Analysis*. Brooks/Cole, 7 edition, 2001.

[3] W. H. Enright. Accurate approximate solution of partial differential equations at off-mesh points. *ACM Transactions on Mathematical Software*, 26(2):274–292, June 2000.

[4] Thomas A. Grandine. Applications of contouring. *SIAM Review*, 42(2):297–316, April 2000.

[5] MathWorks. *MATLAB online documentation*, 12 edition.

[6] Gonzalo A. Ramos and W. H. Enright. Interpolation of surfaces over scattered data. *Proceedings of the IASTED International Conference VISUALIZATION, IMAGING, AND IMAGE PROCESSING*, pages 219–224, 2001.

[7] W.E. Schiesser. *The Numerical Method of Lines Integration of Partial Differential Equations*. Academic Press, 1991.