Scientific Programming 11 (2003) 39–56
IOS Press

A general symbolic PDE solver generator: Explicit schemes

K. Sheshadri and Peter Fritzson
Department of Computer and Information Science, Linköping University, S-581 83 Linköping, Sweden
Tel.: +46 13 281484; Fax: +46 13 284499; E-mail: {shesh, petfr}@ida.liu.se

Abstract. A symbolic solver generator to deal with a system of partial differential equations (PDEs) in functions of an arbitrary number of variables is presented; it can also handle arbitrary domains (geometries) of the independent variables. Given a system of PDEs, the solver generates a set of explicit finite-difference methods to any specified order, and a Fourier stability criterion for each method. For a method that is stable, an iteration function is generated symbolically using the PDE and its initial and boundary conditions. This iteration function is dynamically generated for every PDE problem, and its evaluation provides a solution to the PDE problem. A C++/Fortran 90 code for the iteration function is generated using the MathCode system, which results in a performance gain of the order of a thousand over Mathematica, the language that has been used to code the solver generator. Examples of stability criteria are presented that agree with known criteria; examples that demonstrate the generality of the solver and the speed enhancement of the generated C++ and Fortran 90 codes are also presented.

1. Introduction

The solution of partial differential equations (PDEs) is arguably one of most important tasks in the modeling and simulation of complex physical and engineering systems. The ubiquity of PDEs owes to the circumstance that often the laws governing a particular situation express the relation between various physical quantities in terms of their spatial and temporal variations. Very few known PDEs can be solved exactly, and therefore a variety of approximate numerical techniques have been developed over many decades to deal with equations that arise in realistic situations. Examples of numerical techniques include finite-differences, finite volumes, finite elements, and so on. Each has its own strengths, weaknesses, and domains of applicability.

Our motivation for this work is twofold. Firstly, many computer packages have been developed over the years presenting the user with a wide range of solvers to choose from depending on the nature and complexity of the PDE problem. Diffpack (see [2,9]) presents an object-oriented problem solving environment for numerical solution of PDE's. It implements finite difference as well as finite element methods and provides C++ modules with a wide selection of interchangeable and application-independent components. ELLPACK (see [3]) presents a high level interface language for formulating elliptic PDE problems, and presents over 50 problem solving modules for handling complex elliptic boundary value problems. It is implemented as a FORTRAN preprocessor and can handle a variety of system geometries in two dimensions (both finite difference and finite elements), and rectangular geometries in three dimensions (finite differences only). Two more efforts include Cogito and COMPOSE, developed at Uppsala University, Sweden (see [16–18]). Both implement finite-difference schemes for time dependent problems and exploit object-oriented technology to develop a new kind of software library with parts that can be flexibly combined, enhancing easy construction and modification of programs. Cogito is a high performance solver that comprises the three layers Parallel, Grid and Solver, the lower two layers being Fortran 90 parallel code, while the Solver is written in C++. COMPOSE is a C++ object-oriented system that exploits the Overture system [13] for grid generation. The work on numerical PDE libraries is far too extensive for us to be able to present an exhaustive review here (see [1,6,8,11] for overviews). Our work is a contri-

ISSN 1058-9244/03/$8.00

bution to this important field; the main features of our work include high-level interactivity and exploitation of the power of symbolic programming.

A second motivation for our work is the following. In the last few years, there have been several efforts to develop computer languages for modeling complex engineering systems from various domains. One such language, called Modelica (see [12]), is particularly interesting because of its conceptual soundness: it is an object-oriented modeling language that is equation based and acausal. For such a language to be useful for multi-domain modeling (i.e., modeling of multi-physics systems, ex., electrical, mechanical, thermodynamic, etc.), it is necessary to provide it with support for efficient solution of systems of equations encountered in such models. At present, there is no PDE support available for Modelica, and our work is partly motivated by this. Further, there is a Mathematica environment for the Modelica language, a preliminary version of which is available now [10,7], and our solver generator can be easily integrated with this system.

Our chief design goals are flexibility and generality. An important requirement for a PDE solver is flexibility: it should be capable of offering a wide range of solution schemes from which the user might choose the solution technique that is most appropriate for the given PDE problem. The generality might here be interpreted as the ability to handle systems of PDE's for functions in arbitrary number of variables and geometries, as well as the generation of solution schemes to any order of accuracy, at least in principle. We have been able to meet these requirements to a large extent using the symbolic power of Mathematica language. In addition, we are able to achieve appreciable computational efficiency by employing the idea of code generation.

Our preliminary results were reported in the references [19,15]. Here is a brief summary of our results in the present work. Our solver generaor can accept the PDE and the initial and boundary conditions, along with the geometry of the system, in a simple format. In principle, there could be any number of equations, and the system could have an arbitrary dimensionality and geometry. The solver generator can generate all possible finite-difference schemes to a specified approximation order and the Fourier stability condition for each scheme. The user can then choose the scheme that is most stable/appropriate. The solver generator then symbolically produces an iteration function using the stencil (i.e., the chosen finite-difference scheme) and the initial and boundary conditions. Evaluation of this function leads to the solution of the given PDE problem, and can be done directly in Mathematica. However, this stage involves only numerical computation, and it is much more efficient to use the computational power of a compiled language. We use the MathCode C++ code generator [4] to translate the iteration function into optimized C++ code. The generated C++ code runs about a thousand times faster than the Mathematica function. We give several examples that demonstrate the flexibility and efficiency of our solver generator.

The work presented in this paper is restricted to explicit finite difference methods. However, the overall design of the solver generator is such that other solution techniques can be straightforwardly incorporated. We have done a preliminary implementation of implicit schemes within our framework, but this will be published elsewhere. We also plan to take up finite element methods in the near future.

This paper is organized as follows. In Section 2 we describe the symbolic aspects of the solver: in particular, we discuss the symbolic generation of all possible difference schemes for the given PDE and their stability criteria, the generation of the iteration function, which is essentially a loop nest containing a sequence of assignment statements for the array of the dependent variables, using the discretized PDE, and the initial and boundary conditions. We have to make type declarations of all the variables and arrays appearing in the iteration function to generate C++ or Fortran90 code for it using the MathCode compiler. These code generation aspects are explained in Section 3. In Section 4 we give several examples to illustrate discretization of the PDE and the initial and boundary conditions, generation of finite-differences, stencils and the Fourier stability conditions, flexibility to handle an arbitrary number of dependent and independent variables, and finally code generation and speed enhancement. Some conclusions and possible future work are discussed in Section 5.

2. Symbolic aspects

2.1. Format

To begin with, the problem is specified as a list of three lists. To illustrate the format, we begin with the example of a simple one-dimensional PDE problem, the diffusion equation:

$$\frac{\partial u(x,t)}{\partial t} = \frac{\partial^2 u(x,t)}{\partial x^2}$$

```
parabolic1D = {
   { (* equations *)
     {∂{t,1}u[x, t] = ∂{x,2}u[x, t]}, (* PDE *)
     {x == 0, u[x, t] == 0}, (* boundary condition *)
     {x == 1, u[x, t] == 0}, (* boundary condition *)
     {t == 0, u[x, t] == If[x < 0.15, x/2, 1 - x/2]}
     (* initial condition *)
   },
   { (* geometries *)
     {{x, 0, 1}}, {{t, 0, tmax}}
   },
   { (* methods *)
     {u[x[{}, {2, 2}], t[{1, 1}]]},
     {u[t[], x[{1}]]},
     {u[t[], x[{1}]]},
     {u[x[], t[{1}]]}
   }
 };
```

Fig. 1.

with an initial condition

$$u(x,0) = x/2 \text{ for } x < 0.15 \text{ and}$$

$$u(x,t) = 1 - x/2 \text{ for } x \geqslant 0.15,$$

and boundary conditions

$$u(0,t) = 0 = u(1,t) \text{ for } t \geqslant 0.$$

This problem is presented to the solver as a list of three lists in Fig. 1.

The list parabolic1D has three sublists: equations, geometries, and methods lists, respectively. The geometries list has a further sublist for each independent variable; each element in this sublist is a list of the form {variable, min, max} and specifies the limits of the independent variables in a subdomain (i.e., a continuous segment along the variable, see Example 3 in Section 2.6) of that independent variable in the standard Mathematica iterator notation. Further, min and max here could depend on other independent variables: this is how we are able to describe arbitrary geometries. In the present case, there are just two independent variables, one space variable x and time variable t, but there could be any number of independent variables in general; each variable can in general have any number of subdomains (although there is only one for each in the present simple example).

The equations and methods lists have the same number of elements: there is a method (see the discussion below) for each equation. The equations list contains all the equations in the problem: the PDE followed by the initial and boundary conditions. The PDE could be a single equation or a system of equations separated by commas. In the present case there is just one equation.

Note the slight change in format between the PDE and the initial and boundary conditions in the equations list: the latter have to have the boundary specified; the first element in these sublists specifies the boundary, and the remaining elements, the conditions that hold at this boundary.

We also note here that this format allows for a specification of numerical boundary conditions as well. These arise, for example, when a high-order approximation method is used for the PDE, resulting in a stencil that requires values of the solution vector outside the problem domain to be known for computing the solution vector inside the domain, but close to a boundary. In such cases, we need to provide the solution vector val-

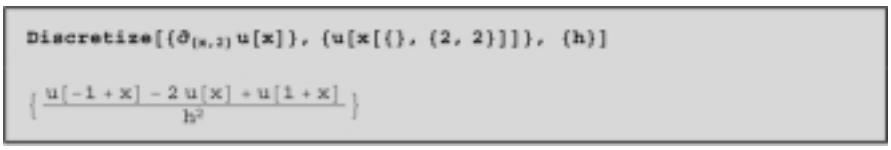

Fig. 2.

ues outside the domain, at a finite number of grid point layers near a boundary. In our format, these values are specified by simply adding more statements to the relevant boundary condition list. An example will clarify this. Suppose in the above problem `parabolic1D` the boundary condition $\{x = 0, u[x,t] = 0\}$ required a modification to include the condition $u[x,t] = nbc$ at values of x one grid layer to the left of the boundary $x = 0$. The modified boundary condition that takes this numerical boundary condition into account is simply $\{x = 0, u[x,t] = 0, u[x-1,t] = nbc\}$. Note that nbc can in general be a function of all the other independent variables (in the present case, a function of t). The choice and form of numerical boundary conditions depend on the kind of method used for the PDE.

As we mentioned above, there is an element (that we call a method) in the methods list for each element in the equations list. In particular, the first element in the methods list corresponds to the PDE. This first element has only one element, since there is only one PDE in this case, and is $u[x[\{\},\{2,2\}],t[\{1,1\}]]$. In general, if the largest derivative order of the dependent variablewith respect to the independent variable x_i is O_i, then the general form of a methods element is $u_1[x_1[\{m_1,n_1\},\ldots,\{m_{O_1},n_{O_1}\}],$ $\ldots,\{m_{O_2},n_{O_2}\}],\ldots]$. Here, an element $\{m,n\}$ appearing as the kth argument of x, which in turn is an argument of u, specifies the way in which the kth derivative of u with respect to x is to be treated by the finite-difference method: m is the approximation order, and n is an integer from 0 to $m+1$: $n = 0$ leaves the kth derivative intact, $n = 1$ corresponds to forward difference, $n = m+1$ corresponds to backward difference, and $n = 2$ to m correspond to the various central differences. We hereafter refer to lists such as $\{m,n\}$ as approximation specifications, since they specify how the individual derivatives appearing in a PDE have to handled.

The first approximation specification of x in the above example is $\{\}$: since the PDE has no first derivative of u with respect to x, the solver ignores the contents of this list, so we have kept it empty; it could instead have been $\{-1,0\}$, for instance. However, it is important that this list be the first argument of x even though there is no first derivative of u with respect to x: this ensures that $\{2,2\}$, the second argument of x, is the approximation specification for the second derivative of u with respect to x. We have $\{2,2\}$ here because we have chosen to replace the second derivative of u with respect to x by a second-order central difference. The following shows the effect of the approximation specification $\{2,2\}$ on a second derivative as shown in Fig. 2.

Here, h is the step size. Also note that in the resulting expression, (1) x is the discretized variable corresponding to the independent variable x; this is done so as not to generate more variables, and should result in no confusion, and (2) the arguments of u are expressed in units of h, the step size.

In general, if the largest-ordered derivative with respect to an independent variable in the equation is L, then the number of approximation-specification arguments of this variable must be exactly L, some of which are possibly empty lists. However, if the PDE has kth derivative appearing, then the kth basic method of the corresponding independent variable has to be nonempty.

A similar explanation holds for the time derivatives in the PDE. However, one comment about the order of arguments of u in the methods list is in order. Some methods can result in an explicit finite-difference scheme, in which case one of the independent variables acts as the marching variable. It is important to specify the marching variable as the last argument of u. In the present case, we have a time-marching method, so the time variable t is the last argument.

Finally, the same comments hold for the methods to be used for the initial and boundary conditions, with just one addition: in these, the last argument of the dependent variable is the independent variable for which the equation of the boundary has to be solved. Further, we also need to specify in which subdomain of the independent variable this condition holds, and this is just an integer; the latter is specified as the last argument of the independent variable in the corresponding element of the methods list. In the present example, since these conditions have no derivatives appearing (i.e., $L = 0$), the independent variables in the corresponding meth-

ods lists have no arguments, except the last independent variable in each element, which has a single-element sublist specifying an integer ($= 1$ in our example of rectangular geometry).

2.2. Methods generation

Now it can be seen why the above seemingly-intricate format for methods specification has an undoubted advantage. Since the format has an approximation specification for each distinct-ordered derivative appearing in the equation, it is very structured, and can therefore be easily automated: we simply need to specify the accuracy (which is the first element of the approximation specification list) for each distinct ordered derivative appearing in the PDE to generate all possible regular finite-difference schemes to that accuracy. Here, by a regular finite-difference scheme we mean one that can be obtained by simply replacing the derivatives appearing in the equation by finite differences. There are, however, some schemes (e.g. Lax-Wendroff, see [8]) that are not obtainable in this manner, and this approach obviously fails to generate such schemes, since these are very special. But when we know some irregular scheme for a problem, we can explicitly specify the difference equation for that scheme in place of the PDE, and specify the corresponding method as simply $\{u[x[], \ldots, t[]]\}$. In this case, the solver functions as a high-level interface to generate a C++ code for the given PDE problem.

The idea behind automatically generating all the regular methods is very simple. In the approximation specification $\{m, n\}$ for each derivative, we have two integers: the first specifies the order of accuracy and the second, the kind of finite difference (forward, backward, central, or identity). If the first integer is m, then the second can assume the $m+2$ values $0, 1, \ldots, m+1$. As a result, the number of possible regular methods is simply $\prod_{i=1}^{d}[m(i) + 2]$ where $m(i)$ is the order of accuracy (and $m(i) + 2$ the number of approximation specifications) for the ith distinct-ordered derivative, and d is the number of distinct-ordered derivatives in the given equation.

To illustrate for the above 1D parabolic example, we have two distinct-order derivatives: $\partial_{XX}u(x,t)(i = 1)$ and $\partial_{XX}u(x,t)(i = 2)$, so $d = 2$. If we choose $m(1) = 2$ and $m(2) = 1$, then the total number of regular methods is $(2 + 2) * (1 + 2) = 12$, one of which is the standard method $u[x[\{\}, \{2, 2\}], t[\{1, 1\}]]$ used for this equation. In the next subsection we describe how stability conditions for the generated methods are obtained; it will be demonstrated in Section 4 that the stability criterion we obtain for the above standard method agrees with the known criterion for this scheme, and that the remaining 11 methods are all unstable.

2.3. Discretization of the equations

Once we have generated all the regular methods to a certain order of accuracy, we can operate them on the PDE and other equations (the initial and/or boundary conditions) to obtain difference equations.

First we have to discretize the space of independent variables by introducing the step sizes: the step size for a variable is the smallest amount by which it can vary. The set of all the points in the space of independent variables that can be reached in an integer number of steps from an origin is known as the grid. Subsequent to this, all the functions of independent variables will be defined only on the grid.

The discretization of an equation then involves (i) identifying all derivatives appearing in the given equation, and (ii) replacing each derivative by a suitable finite difference as specified in the method. The step (i) is a simple instance of pattern recognition that can be easily performed using Mathematica. Step (ii) can be done in various (equivalent) ways, but we have followed the method of Lagrange polynomials (see [11] for a discussion) that helps us obtain a finite-difference approximation of a derivative of an arbitrary order to an arbitrary accuracy. By doing this we approximate the derivative by an expression that involves the values of the dependent variable at the given and neighboring grid points, and the various step sizes. Substituting such expressions for all the derivatives appearing in the equation results in a difference equation. The difference equation corresponding to the PDE can be solved for the dependent variable at the largest value of the marching variable appearing in the equation. The relation that results is known as the stencil. Solving the discretized initial and boundary conditions expresses the dependent variable at points close to the boundaries in terms of the parameters of the problem.

At this stage we can turn to the symbolic generation of the iteration function for the system of difference equations, providing the method leads to a stable solution.

2.4. Stability analysis

We showed above how we can generate all possible regular methods for a given PDE. However, we will

Fig. 3.

have to choose from these methods one that leads to a stable (and hence convergent) solution to the problem. Moreover, there is no guarantee that one or more of the regular methods will lead to a stable solution. It is therefore desirable to perform a stability analysis before generating the iteration function for the problem. There are a few such stability techniques that are available, one of the simplest being the Fourier stability analysis (see [1], for example) that we have used in the present work.

The Fourier stability condition is arrived at on the basis of the following simple idea (see e.g. [1]). The stencil for the PDE obtained using a method expresses the value of the dependent variable at a given grid point in terms of its values at a few neighboring grid points. For explicit schemes, the given grid point corresponds to the largest value of the marching variable, and the neighboring points correspond to smaller values. In the Fourier stability analysis, one assumes a solution to this stencil in which the independent variables are seperated, and demands that the absolute value of the right-hand side of the stencil be between zero and unity at all grid points, for the solution to be stable and to converge to the correct solution of the PDE. This directly gives a relation between the various step sizes that has to be satisfied for the method to be stable.

We can easily write a Mathematica function for performing this analysis. We show in Section 4 that this approach generates the correct Fourier stability conditions for some well known equations. This gives us confidence to use our solver generator for equations that we have not previously encountered.

2.5. Symbolic generation of the iteration function

We now turn to the last phase of the symbolic part of our solver, where a Mathematica code for the function that performs the iteration of the stencil is generated. We first note that the details of this code are problem dependent, and only the structure of the code can be ascertained a priori. The structure of the code is a sequence of nested iteration loops, the body of which involves discretized initial and/or boundary conditions. The number of nested loops is equal to the number of conditions specified in the problem. The last nested loop has as its body the stencil of the given PDE. We remark here that at this stage, the dependent variables are replaced by arrays, and the discretized equations are simply assignment statements for the array elements. There are no symbolic manipulations left to be done, and all assignments involve purely arithmetic operations. The number of assignment statements in each nested loop depends on the nature of the initial/boundary conditions and the PDE. As a result, a different iteration function has to be generated for every PDE problem.

The generated iteration function takes as its arguments the numbers of discrete steps for all the independent variables and the step size for the marching variable; the step sizes for the other variables are calculated from these arguments and the geometry.

Finally, we note that the body of the symbolically generated iteration function also has the type declarations of all the local variables, since we intend to generate a C++ or Fortran90 code for the iteration function using the MathCode compiler [4].

2.6. A note on system geometry

As explained in Section 2.1, we are able to describe arbitrary domains of independent variables. We have managed to do this by making the "independent" variables depend on one another in general. A few examples to explain this are in order.

Fig. 3 *A Rectangular Geometry:* A geometry list of the form $\{\{\{x, 0, 2\}\}, \{\{y, 0, 1\}\}\}$ describes a simple rectangle of sides 2 and 1 in the xy-plane.

Fig. 4 *A Circular Geometry:* A geometry list of the form $\{\{\{x, -\texttt{Sqrt}\,[1 - y\texttt{\^{}}2],\ \texttt{Sqrt}\,[1 - y\texttt{\^{}}2]\}\}, \{\{y, -1, 1\}\}\}$ describes a circle of radius 1 in the xy-plane. The following figure shows a plot of the geometry:

Fig. 5 *An Irregular Geometry:* A geometry list of the form $\{\{\{x, 0,\ \texttt{If}\,[y \leqslant y1, x1, 1]\}, \{x,\ \texttt{If}\,[y \leqslant y1, x2, 1.0], 1.0\}\}, \{\{y, 0, 1\}\}, \{\{t, 0, tmax\}\}\}$ describes an irregular geometry: x has two disconnected subdomains ("rectangular wells") up to a certain value $y1$ of y, and only one beyond that.

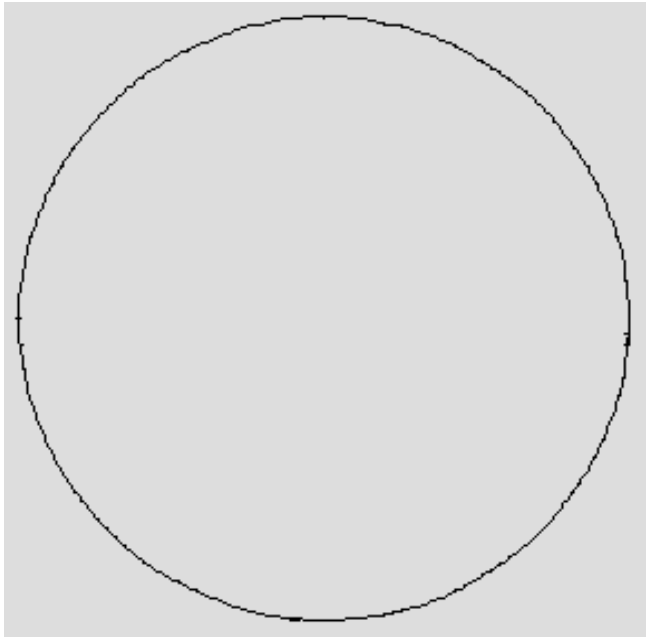

Fig. 4.

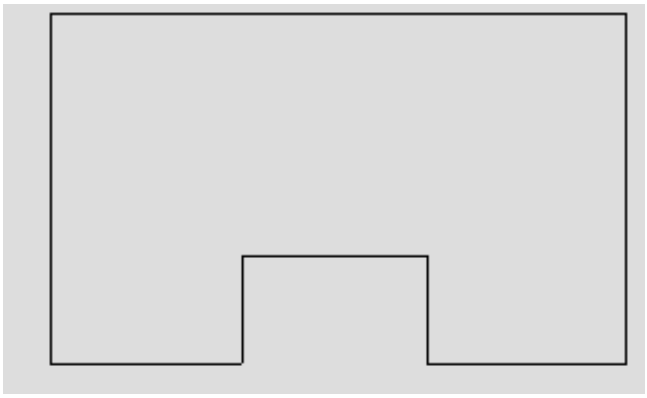

Fig. 5.

It is clear from the above three examples that we are able to describe fairly general geometries in this manner. However, we should note that we can obtain only an approximation to the true boundary of the domain by our approach, since we generate our domain by choosing a subset of points from a rectangular distribution of grid points. For certain geometries (like for instance one in which the nonrectangular domain boundary is made of straight line segments that are parallel to the coordinate axes) this approximation becomes exact; for others, the accuracy improves with decreasing step sizes in the difference scheme used.

3. C++ code generation

We have separated the symbolic and numerical aspects of the problem with the idea of code generation in mind. Accordingly, everything except the iterative numerical computation part is done symbolically in the first part of the package, the set-up part. The result of this part is the iteration function referred to above. The numerical part of the package, namely the run-time part, performs the iteration and the time it takes increases linearly with the problem size. Evidently, the efficiency of the package is determined by how efficiently the iteration is performed.

For the numerical part, we employ the MathCode C++ code generator [4] which generates optimized C++ code for a suitably stated Mathematica task. The Mathematica code generated for the iteration function has the type declarations for all the local variables, as we mentioned in the previous section. However, the type declaration for the iteration function itself, and for the solution array, has to be done separately; this part is also problem specific, since the number of arguments of the iteration function and the array dimension depend on the problem dimensionality. We make these declarations in the run-time part, after evaluating the set-up part.

Once these declarations are done, we are ready to compile and run the iteration function. The compilation results in a C++ code that can be run transparently as if the code was executed within Mathematica. This is achieved by MathCode by loading the C++ code into a separate process and automatically generating communication code to make it callable. We can also run the iteration function interpretively within Mathematica before generating the C++ code. However, the C++ code runs considerably faster, and we get a speed enhancement of about a factor of one thousand.

4. Examples

In this section we demonstrate some important features of our solver, including discretization of derivatives, PDE, and the initial and boundary conditions. We also demonstrate the automatic generation of regular methods and the stability conditions for these; we show how the iteration function is symbolically generated, and finally demonstrate the generation of C++ code for the iteration function and the speed enhancement over Mathematica.

We would like to emphasize that our PDE solving system provides an interactive problem solving environment for PDE's. The system presents a whole gamut of tools for analyzing various aspects of the problem to be solved: one can explore the PDE, the initial and boundary conditions and different discretizations; one can construct a method based on this analysis; one can even automatically generate all possible discretization schemes to a given order of approximation; one can obtain a Fourier stability condition for each of these methods, and select an appropriate method. Alternatively, if one already knows that a particular method works for the given problem, one can then use the system to directly generate a program for implementing it, bypassing a preliminary analysis. By virtue of the advantages offered by the Mathematica environment, one can then make a plot of the results obtained.

The plan for the rest of this section is the following. We first describe some exploratory aspects of the PDE solving system by demonstrating the use of the various functions the solver is based on. We then present a few complete PDE examples using the system: a one-dimensional diffusion equation, a two-dimensional diffusion equation in circular and dumb-bell geometries, and a wave equation in a dumb-bell geometry.

4.1. Examples of tools available in the package

In Fig. 6 is an example of approximations to a first derivative to an accuracy order 1 (note that there are two approximations, one forward and one backward, in this case).

The result is presented as a function with a head $\left(\mathtt{u}^{(1,0)}[\mathtt{x},\mathtt{t}]\right)$ which is the derivative being approximated, with arguments whose number is one greater than the accuracy order: the first argument is the forward difference, the last the backward difference, and the intervening ones (none in this case) the various central differences. The head of the output is made to be the derivative itself so that we can extract the zeroth element of the output when no approximation is desired. (The indexing function `Part` in Mathematica returns the head of an indexed object when using zero as the index. For example if `x` denotes then `h2[a,b,c]` gives `h2`.) We could use this function to discretize derivatives of any order to any accuracy.

Discretization of an equation involves replacing all the derivatives appearing in the equation by suitable finite difference approximations as specified by the method. The function we have defined for discretizing the PDE, namely `Discretize`, takes as its two arguments the equation and the method, and returns the stencil. Figure 7 is a simple example of the one-dimensional advection equation.

We have used the standard method for this equation which consists in replacing the space derivative by a first-order backward difference and the time derivative by a first-order forward difference; we will show below, using stability analysis, that the resulting stencil leads to a stable solution. In the above example, h and k are the step sizes for x and t, respectively. The `Discretize` function works for any dimensionality of the geometric domain.

We now give an example of discretizing the boundary conditions. Discretizing these is the same as discretiz-

```
DiffApprox[∂x u[x, t], {1}, {h}]
u^(1,0)[x, t][(-u[x, t] + u[1 + x, t])/h, (-u[-1 + x, t] + u[x, t])/h]
```

Fig. 6.

```
Discretize[{∂t u[x, t] + ∂x u[x, t] == 0},
 {u[x[{1, 2}], t[{1, 1}]]}, {h, k}]
{(-u[-1 + x, t] + u[x, t])/h + (-u[x, t] + u[x, 1 + t])/k == 0}
```

Fig. 7.

```
Discretize2[{x == 0, u[x, y, z, t] == val1, ∂x u[x, y, z, t] == der1},
 {u[y, z, t, x[{1, 1}, {1}]]}, {h1, h2, h3, k}]
{u[1 + x, y, z, t] → der1 k + val1, u[x, y, z, t] → val1}
```

Fig. 8.

```
Discretize2[{x == 1, u[x, y, z, t] == val1, ∂x u[x, y, z, t] == der1},
 {u[y, z, t, x[{1, 2}, {1}]]}, {h1, h2, h3, k}]
{u[-1 + x, y, z, t] → -der1 k + val1, u[x, y, z, t] → val1}
```

Fig. 9.

```
DiffMethods[{∂t u[x, t] == ∂{x,2} u[x, t]},
 {{u[x, 2], 2}, {u[t, 1], 1}}, t]
{{u[x[{-1, 0}, {2, 0}], t[{1, 0}]]},
 {u[x[{-1, 0}, {2, 1}], t[{1, 0}]]},
 {u[x[{-1, 0}, {2, 2}], t[{1, 0}]]},
 {u[x[{-1, 0}, {2, 3}], t[{1, 0}]]},
 {u[x[{-1, 0}, {2, 0}], t[{1, 1}]]},
 {u[x[{-1, 0}, {2, 1}], t[{1, 1}]]},
 {u[x[{-1, 0}, {2, 2}], t[{1, 1}]]},
 {u[x[{-1, 0}, {2, 3}], t[{1, 1}]]},
 {u[x[{-1, 0}, {2, 0}], t[{1, 2}]]},
 {u[x[{-1, 0}, {2, 1}], t[{1, 2}]]},
 {u[x[{-1, 0}, {2, 2}], t[{1, 2}]]},
 {u[x[{-1, 0}, {2, 3}], t[{1, 2}]]}}
```

Fig. 10.

ing the PDE, except that in this case no stencil is generated, but the dependent variable at grid points close to the boundary is expressed in terms of the problem parameters, and the equations are accompanied by the equation of the boundary where they hold. For these reasons, we have defined a different function called `Discretize2` to be used for the initial and boundary functions. In Figs 8 and 9 we give two examples, one for the left boundary ($x = 0$) and another for the right ($x = 1$).

We now turn to examples that show how the methods can be automatically generated, and stability conditions for the stencils obtained. The methods are generated using a function that we have defined, namely DiffMethods. This function takes as arguments the equation and a list specifying the order of accuracy. Figure 10 shows how all regular methods (see Section 2) for the one-dimensional diffusion equation are

```
StableF[{∂{t,1} u[x, t] == ∂{x,2} u[x, t]},
 {{u[x, 2], 2}, {u[t, 1], 1}}, {h, k}, t]
method no. 7: {u[x[{-1, 0}, {2, 2}], t[{1, 1}]]}
Stencil:
 {u[x, t] → -1/h^2 (-k u[-1 + x, -1 + t] - h^2 u[x, -1 + t] + 2 k u[x, -1 + t] -
       k u[1 + x, -1 + t])}
stability condition(s):
0 ≤ Sqrt[(1 - 2 k/h^2)^2] ≤ 1
```

Fig. 11.

```
GenerateCodeSolvePDE[parabolic1D, {0, 0}, {1, tmax},
  {1, 201, 501}, {tmax}, {Real}, "C:\\pdesolver\\test\\"];

MathCode C++ ver  1.10 loaded.
Successful compilation to C++: 2 function(s)
FiniteDiff is installed.

A C++ code has been generated for the function solvePDE.

Now you can obtain the solution by
   evaluating solvePDE. Make sure that  the sizes are:
  (1) smaller than the maximum sizes you have specified (
 {1, 201, 501}), and
  (2) compatible with the stability
   requirements for the FD scheme used:

      {0 ≤ Sqrt[(-1 + Nt - 2 (-1 + Nx)^2 tmax)^2 / (-1 + Nt)^2] ≤ 1}

The function slice contains the solution
   array at any specified value of the march variable.
```

Fig. 12.

generated to order 2 in space and 1 in time (there are consequently 12 regular methods).

Now we can use each of these methods to generate a stencil for the equation, and obtain the Fourier stability condition for each. This is done by using the function `StableF` that we have defined. We show in Fig. 11 the stability conditions for the above example; since the output is too long, we have shown only the stability conditions for the stable methods.

It turns out that only one of the 12 methods generated is stable, and it can be seen that this is precisely the method that is used often for this simple one-dimensional diffusion equation. Further, from the above stability condition, we see at once that this method is stable only if $k/h^2 \leqslant 1/2$, a well known result [1,8]. In the same manner, we are able to generate stable methods and their correct Fourier stability conditions ($k/h \leqslant 1$, see [1,8]) for other equations, e.g. the advection and wave equations.

4.2. Diffusion equation in 1D

We illustrate the solution of the one-dimensional diffusion equation as our first example. This problem has been defined as a list parabolic1D in Section 2.1. The function GenerateCodeSolvePDE takes the problem list and generates a C++/Fortran 90 code for solving it using the specified solution scheme; it also generates a few comments about the stability condition as shown in Fig. 12.

We also need to specify the bounding rectangle: this is the smallest rectangle that completely encloses the

```
nx = 100; nt = 100; ht = 0.005;
timem = AbsTime[
        solvePDE[nx, nt, ht];
    ]
{12.00000 Second, Null}
```

Fig. 13.

```
nx = 100; nt = 100; ht = 0.005;
timemcode = AbsTime[
        Table[solvePDE[nx, nt, ht], {i, 1, 500}];
    ];
timelist = {nx, nt, timem[[1]],
  timemcode[[1]], 500 * timem[[1]] / timemcode[[1]]}
{100, 100, 12.00000 Second, 4.00000 Second, 1500.000}
```

Fig. 14.

```
ex1 = slice[1, nx, 1];
ListPlot[ex1, PlotJoined → True];
```

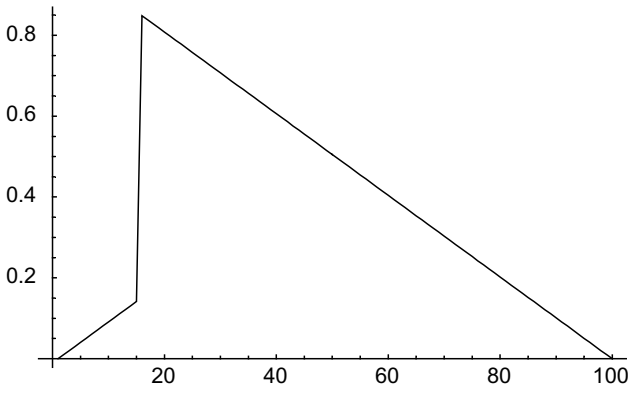

Fig. 15.

system, and for the present simple case this is simply the system itself. The second and third arguments above do this, while the next argument specifies the maximum problem size, useful for array declarations in the C++ code.

We also define a list of parameters whose values we need not fix until runtime. These and their data types are the next two arguments, while the last argument specifies the folder name where the C++ code is saved. Here, `slice` is a function to extract the solution array for a particular value of the marching variable.

Once this is done, we can run the iteration function on Mathematica. Figures 13 and 14 show the runs, respectively, of the iteration function executed interpretively within Mathematica and of the compiled C++ code.

We have run the iteration function 500 times in the second box to get a good estimate for the time taken for one run: this is because the latter is too small to be accurately measured by the function `AbsTime`. We see that the C++ code for the function `solvePDE` speeds up the Mathematica evaluation by about 1500 times. We have also generated Fortran 90 code for the iteration function in the same way as above, using the MathCode

```
ex1 = slice[1, nx, nt];
ListPlot[ex1, PlotJoined → True];
```

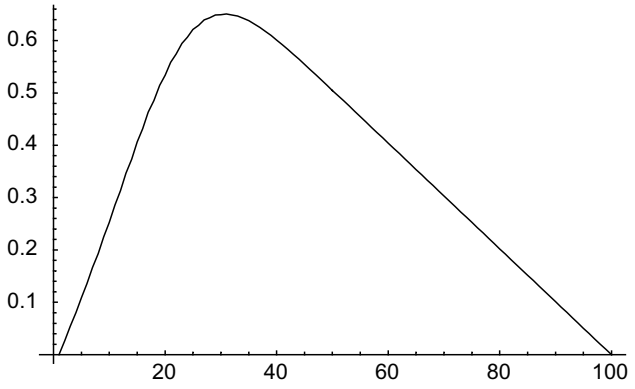

Fig. 16.

```
testprob = {
   {
    {∂{t,1} u[x, y, t] == ∂{x,2} u[x, y, t] + ∂{y,2} u[x, y, t]},
    {x == -Sqrt[1 - y^2], u[x, y, t] == delta},
    {x == Sqrt[1 - y^2], u[x, y, t] == delta},
    {t == 0,
     u[x, y, t] == If[x^2 + y^2 < 0.4, 0, Sqrt[delta + 1 - x^2 - y^2]]}
   },
   {
    {{x, -Sqrt[1 - y^2], Sqrt[1 - y^2]}}, {{y, -1, 1}}, {{t, 0, tmax}}
   },
   {
    {u[x[{}, {2, 2}], y[{}, {2, 2}], t[{1, 1}]]},
    {u[y[], t[], x[{1}]]},
    {u[y[], t[], x[{1}]]},
    {u[x[], y[], t[{1}]]}
   }
  };
```

Fig. 17.

Fortran90 translator that has recently become available, and achieve the same speed enhancement.

Finally, we plot the results obtained for the solution array U. For this purpose, we have defined a function `slice` (which is compiled alongwith `solvePDE`): `slice[i, nx, t]` contains the solution vector for the ith dependent variable for the time slice t. We can directly plot this vector using the function `ListPlot`. In Fig. 15 is the plot of the initial vector.

In Fig. 16 we show the plot of the solution vector for the last time slice.

4.3. Diffusion equation in 2D: Circular geometry

We now present an example of a non-rectangular geometry: the two-dimensional diffusion equation in a circular geometry. The dependent variable is delta at the boundaries at all times, and has an initial profile that is a hemisphere with a coaxial cylinder removed. The solution will then show how this profile evolves with time. First we define the problem list testprob in our notation as shown in Fig. 17.

```
ex1 = slice[1, 100, 100, 1]; (*the solution at t=0*)ListPlot3D[ex1];
```

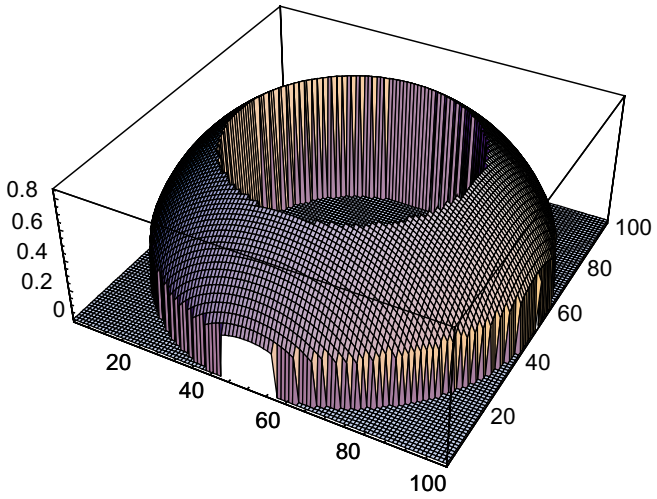

Fig. 18.

```
ex1 = slice[1, 100, 100, 200];
(*the solution at t=0.01*)ListPlot3D[ex1];
```

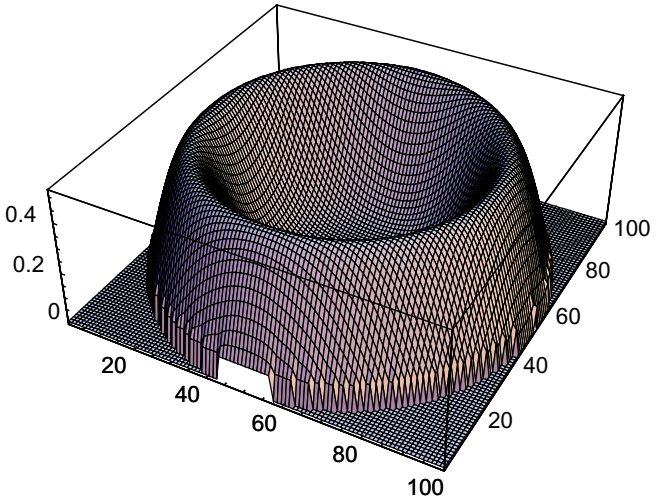

Fig. 19.

We then follow a procedure very similar to the previous example to generate a C++ code by employing the MathCode compiler, and run it. We do not show those details here, but only the results. Figures 18 and 19 show the intial profile and the profile at a later time.

The "hole" in the plot above is because of the fact that the underlying grid is rectangular, although the system geometry is circular. The size of the hole can

```
testprob = {
   {
    {∂{t,1} u[x, y, t] == ∂{x,2} u[x, y, t] + ∂{y,2} u[x, y, t]},
    {x == If[y < 0,
       If[y < -(a - 2 b + c), -Sqrt[b^2 - (y + a - b)^2],
        -Sqrt[2 b c - c^2]], If[y > (a - 2 b + c),
        -Sqrt[b^2 - (y - a + b)^2], -Sqrt[2 b c - c^2]]
      ], u[x, y, t] == 0},
    {x == If[y < 0,
       If[y < -(a - 2 b + c), Sqrt[b^2 - (y + a - b)^2], Sqrt[2 b c - c^2]],
       If[y > (a - 2 b + c), Sqrt[b^2 - (y - a + b)^2], Sqrt[2 b c - c^2]]
      ], u[x, y, t] == 0},
    {t == 0, u[x, y, t] == If[x < 0, (x - If[y < 0,
           If[y < -(a - 2 b + c), -Sqrt[b^2 - (y + a - b)^2],
            -Sqrt[2 b c - c^2]], If[y > (a - 2 b + c),
            -Sqrt[b^2 - (y - a + b)^2], -Sqrt[2 b c - c^2]]
          ])^2, (x - If[y < 0,
           If[y < -(a - 2 b + c), Sqrt[b^2 - (y + a - b)^2],
            Sqrt[2 b c - c^2]], If[y > (a - 2 b + c),
            Sqrt[b^2 - (y - a + b)^2], Sqrt[2 b c - c^2]]
          ])^2]}
   },
   {
    {{x, If[y < 0,
       If[y < -(a - 2 b + c), -Sqrt[b^2 - (y + a - b)^2],
        -Sqrt[2 b c - c^2]], If[y > (a - 2 b + c),
        -Sqrt[b^2 - (y - a + b)^2], -Sqrt[2 b c - c^2]]
      ], If[y < 0,
       If[y < -(a - 2 b + c), Sqrt[b^2 - (y + a - b)^2], Sqrt[2 b c - c^2]],
       If[y > (a - 2 b + c), Sqrt[b^2 - (y - a + b)^2], Sqrt[2 b c - c^2]]
      ]}},
    {{y, -a, a}}, {{t, 0, tmax}}
   },
   {
    {u[x[{}, {2, 2}], y[{}, {2, 2}], t[{1, 1}]]},
    {u[y[], t[], x[{1}]]},
    {u[y[], t[], x[{1}]]},
    {u[x[], y[], t[{1}]]}
   }
  };
```

Fig. 20.

be reduced by making the grid size smaller.

4.4. Diffusion equation in 2D: Dumb-bell geometry

We now present another example of a non-rectangular geometry, this time a somewhat irregular, dumb-bell geometry. The problem list is as defined in Fig. 20.

Again, we do not show those details of code generation here, but only the results. Two plots (Figs 21 and 22) show the intial profile and the profile at a later time.

4.5. Wave equation in 2D: Circular geometry

We now present an example of a wave equation in a circular geometry (Fig. 23).

Figures 24 and 25 show the intial profile and the profile at a later time.

5. Conclusions

In this section we discuss the strengths and limitations of our PDE solving system. The system has three

```
ex1 = slice[1, 100, 100, 1]; ListPlot3D[ex1];
```

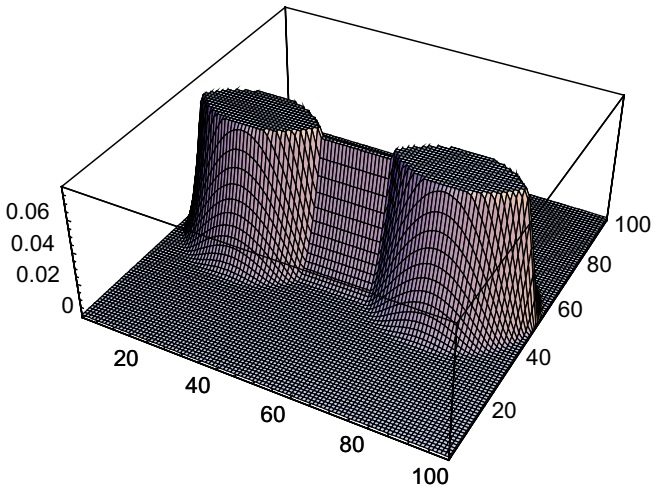

Fig. 21.

```
ex1 = slice[1, 100, 100, 200]; ListPlot3D[ex1];
```

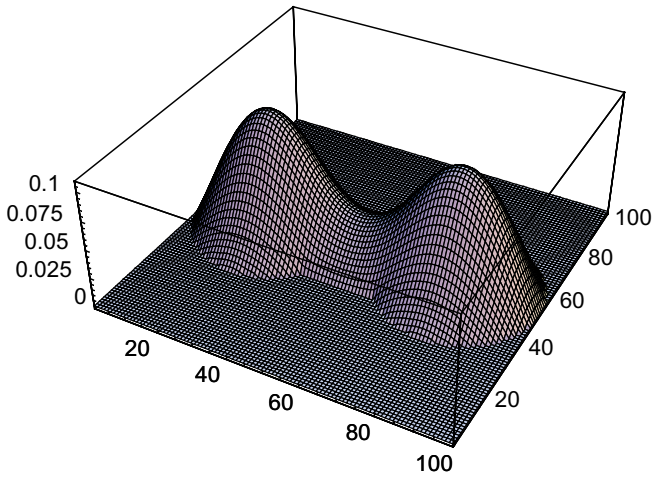

Fig. 22.

important merits. First, since it is based on symbolic manipulation, it has a flexibility that makes it ideal for multi-physics applications where the exact nature of the problem is not known in advance. Further, because of the symbolic power of Mathematica, we are able to generate a set of methods for the given equation, and analyze the stability properties of the stencil that results on applying a method to the equation. However, as we have noted in Section 2, the solver generator can still be used for a PDE for which the user already knows a stable difference method: this further adds to the flexibility of the system. We also note that our system can handle equations in any number of spatial dimensions.

The second important merit of our solver generator

```
testprob = {
   {
    {∂{t,2} u[x, y, t] == ∂{x,2} u[x, y, t] + ∂{y,2} u[x, y, t]},
    {t == 0, u[x, y, t] == Sin[2 Pi x] Sin[2 Pi y], ∂{t,1} u[x, y, t] == 0},
    {x == -Sqrt[1 - y^2], u[x, y, t] == 0,
     ∂{x,1} u[x, y, t] == 2 Pi Cos[2 Pi x] Sin[2 Pi y],
     ∂{y,1} u[x, y, t] == 2 Pi Sin[2 Pi x] Cos[2 Pi y]},
    {x == Sqrt[1 - y^2], u[x, y, t] == 0,
     ∂{x,1} u[x, y, t] == 2 Pi Cos[2 Pi x] Sin[2 Pi y],
     ∂{y,1} u[x, y, t] == 2 Pi Sin[2 Pi x] Cos[2 Pi y]}
   },
   {
    {{x, -Sqrt[1 - y^2], Sqrt[1 - y^2]}}, {{y, -1, 1}}, {{t, 0, tmax}}
   },
   {
    {u[x[{-1, 0}, {2, 2}], y[{-1, 0}, {2, 2}], t[{-1, 0}, {2, 1}]]},
    {u[x[], y[], t[{1, 1}, {1}]]},
    {u[t[], y[{1, 1}], x[{1, 1}, {1}]]},
    {u[t[], y[{1, 1}], x[{1, 2}, {1}]]}
   }
  };
```

Fig. 23.

```
ex1 = slice[1, 100, 100, 1]; ListPlot3D[ex1];
```

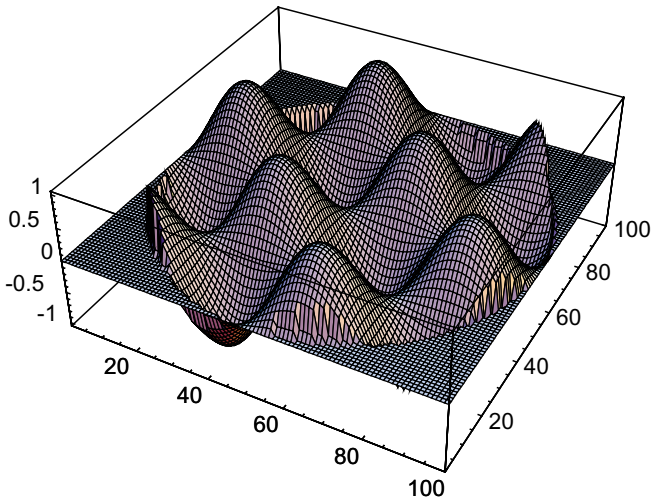

Fig. 24.

is the efficiency of generated solvers in performing numerical computations. This is enabled by each generated solver being a compact efficient numerical code for a particular combination of equation and boundary conditions, and by our use of the MathCode compiler that generates code for the iteration function in a compiled language. The resulting code, as we have demonstrated, runs considerably faster than the Mathematica code. These two features combine the strengths of symbolic programming and numerical computation in a complementary way, resulting in a reasonably general-purpose PDE solving system. The fact that we are using

```
ex1 = slice[1, 100, 100, 100]; ListPlot3D[ex1];
```

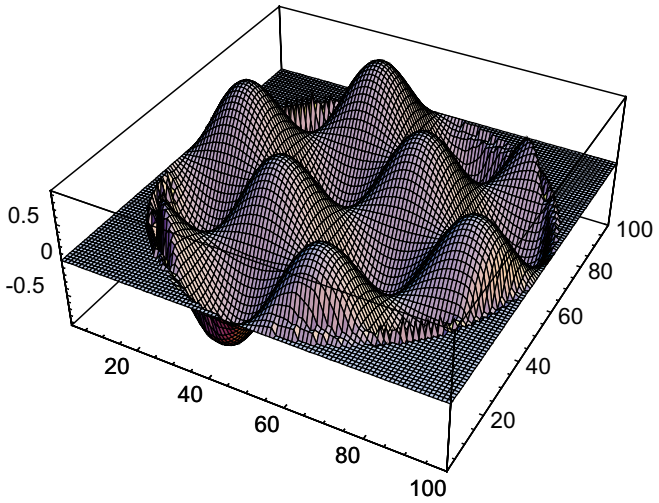

Fig. 25.

a high-level language like Mathematica means that the system is very easy to use, which is of some practical importance.

Thirdly, we have given examples of very different geometries that the solver can handle. In fact, it is clear from the way we handle system geometry that our solver can describe any geometry, not necessarily convex ones, subject however to the limitation that the boundary of the region is approximated by a saw-tooth curve, as remarked in Section 2.6. This we believe is a very powerful practical feature of our system.

There are some limitations of our solving system for which there is scope, and need, for improvement. One of these is the limitation to explicit finite-difference schemes. This is one thing we plan to work on in the near future. There is also the important issue of numerical boundary conditions that arises when we use finite-difference schemes (see [8,5]). There is no general resolution of this issue, and there are a few rules of thumb. Our solver allows for a specification of certain kinds of numerical boundary conditions, as discussed in Section 2.1.

Our immediate goal is to provide PDE support for the Modelica language [12,14] within the Mathematica environment. Our solver can serve this purpose since it is written in Mathematica, and there is presently some work in progress on developing Modelica syntax in a Mathematica environment [10]. However, our package can also be used as a stand-alone PDE solver. We have demonstrated how it works for some example problems in the last section. We have not yet tested our solving system for nonlinear problems, but are presently working on some examples. We believe that nonlinear PDE's in which the derivatives with respect to the marching variable appear to first degree can be handled without much difficulty by our solver, although there could be numerical difficulties in this case.

References

[1] W.F. Ames, *Numerical Methods for Partial Differential Equations*, Academic Press, 1992.

[2] Numerical Objects AS Online (http://www.nobjects.com/prodserv/diffpack/).

[3] J.R. Rice and R.F. Boisvert, Solving Elliptic Problems Using ELLPACK, Springer, 1984, see the ELLPACK webpage http://www.cs.purdue.edu/ellpack/ellpack.html.

[4] P. Fritzson, MathCode C++, published by MathCore (http://www.mathcore.com), 1998.

[5] V.G. Ganzha and E.V. Vorozhtsov, *Numerical Solutions for Partial Differential Equations: Problems Solving Using Mathematica*, CRC Press, 1996.

[6] "Applied Numerical Analysis" (5th ed.) by C.F. Gerald and P.O. Wheatley, Addison-Wesley, 1994.

[7] M. Jirstrand, Johan Gunnarsson and Peter Fritzson, MathModelica – A New Modeling and Simulation Environment for Mathematica, in: *Proceedings of the International Mathematica Symposium* (*IMS99*), Linz, Austria, August 1999 (available at http://south.rotol.ramk.fi/~keranen/IMS99/paper12/MathModelica-IMS99.nb).

[8] B. Gustafsson, Heinz-Otto Kreiss and Joseph Oliger, *Time Dependent Problems and Difference Methods*, John Wiley & Sons, Inc., 1995.
[9] H.P. Langtangen, *Computational Partial Differential Equations – Numerical Methods and Diffpack Programming*, Springer-Verlag, 1999.
[10] P. Fritzson, Vadim Engelson and Johan Gunnarsson, An Integrated Modelica Environment for Modeling, Documentation and Simulation, in: *Proceedings of SCSC-98* (*Summer Computer Simulation Conference*), Reno, Nevada, July 1998.
[11] M. Oh, *Modelling and simulation of combined lumped and distributed processes*, Ph.D thesis, University of London, 1995.
[12] Modelica Association homepage (http://www.modelica.org).
[13] W.D. Henshaw, Overture Documentation, LLNL Overlapping Grid Project, see http://www.llnl.gov/CASC/Overture/henshaw/overtureDocumentation/overtureDocumentation.html.
[14] PELAB – Programming Environment Laboratory, Department of Computer Science, Linköping University, Sweden, http://www.ida.liu.se/~pelab/modelica.
[15] K. Sheshadri and P. Fritzson, A Mathematica-based PDE Solver Generator, *Proceedings of SIMS'99, The 1999 Conference of the Scandinavian Simulation Society*, Linköping, Sweden, pp. 66–78.
[16] E. Mossberg, K. Otto and M. Thuné, Object-oriented software tools for the construction of preconditioners, *Scientific Programming* **6** (1997), 285–295.
[17] K. Åhlander, An Object-Oriented Framework for PDE Solvers, PhD thesis, Dept. of Scientific Computing, Uppsala University, Sweden, 1999.
[18] See http://www.tdb.uu.se/research/swtools/.
[19] A. Wrangsjö, P. Fritzson and K. Sheshadri, Transforming Systems of PDEs for Efficient Numerical Solution, *Proceedings of the International Mathematica Symposium* (*IMS99*), Linz, Austria, 1999, http://south.rotol.ramk.fi/~keranen/IMS99/ paper7/Transformations.pdf.