Scientific Programming 11 (2003) 225–235
IOS Press

A general symbolic PDE solver generator: Beyond explicit schemes

K. Sheshadri and Peter Fritzson
Programming Environment Laboratory, Department of Computer and Information Science, Link öping University, S-581 83 Linköping, Sweden
E-mail: {shesh,petfr}@ida.liu.se

Abstract. This paper presents an extension of our Mathematica- and MathCode-based symbolic-numeric framework for solving a variety of partial differential equation (PDE) problems. The main features of our earlier work, which implemented explicit finite-difference schemes, include the ability to handle (1) arbitrary number of dependent variables, (2) arbitrary dimensionality, and (3) arbitrary geometry, as well as (4) developing finite-difference schemes to any desired order of approximation. In the present paper, extensions of this framework to implicit schemes and the method of lines are discussed. While C++ code is generated, using the MathCode system for the implicit method, Modelica code is generated for the method of lines. The latter provides a preliminary PDE support for the Modelica language. Examples illustrating the various aspects of the solver generator are presented.

1. Introduction

The numerical solution of partial differential equations is a very extensively researched field in the past few decades. The reasons for this are fairly obvious, if one considers the frequent occurrence of PDEs in most fields of science and engineering. Furthermore, few PDEs, especially the practically interesting ones, are amenable to analytical solution techniques, and as a result a wide range of numerical techniques have been developed to study them. Given the constant increase in the power of the digital computer, an increasing number of these techniques have been implemented.

This paper is an extension of the symbolic-numeric framework for the solution of PDEs that we have developed recently (see [27,25,24]). Our earlier work was restricted to explicit finite-difference schemes, while otherwise being fairly general: it can develop approximation schemes to any desired order for PDE problems in arbitrary number of variables, dimensions and geometries. We presented a Mathematica-based solver generator that performs a series of symbolic transformations on the given PDE problem and employs the MathCode translator (see [6]) to generate optimized C++/Fortran 90 code for iteratively obtaining numerical solutions to PDEs. This framework has the obvious advantage of combining the symbolic power of Mathematica and the computational efficiency of compiled languages like C++ and Fortran into a flexible system for generating highly efficient compiled solvers. The present work extends the solver generator to treat of implicit finite-difference schemes and the method of lines.

The motivation for our work comes largely from two independent active fields of research. Firstly, we view our efforts as a contribution to the vast amount of work that is going on to develop efficient numerical solvers for PDE problems. To place our work in perspective, we make a brief survey of related work. Diffpack (see [2,13]) 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 and PELLPACK (see [22,5, 21]) are problem solving environments (PSEs) with a high level interface language for formulating elliptic PDE problems, that contain over 50 problem solving modules for handling complex elliptic boundary value problems. They are implemented as a FORTRAN

ISSN 1058-9244/03/$8.00

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). The main feature of these PSEs is the reuse of powerful static modules. They also offer help with analyzing various aspects of the PDE problem. The description of our solving system in later sections shows the ways in which our system compares and contrasts with extant PSEs for handling PDE problems. Briefly, our system has a range of Mathematica modules that offer help in analyzing a PDE problem, and we use a dynamic library of modules that are tailored for the given PDE problem. Two more efforts include Cogito and COMPOSE, developed at Uppsala University, Sweden (see [15,19,26]). Both implement finite-difference schemes for time dependent problems and exploit the 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 [20] 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 [1,11,12,17] for overviews). Our work is a contribution to this important field, and distinguishes itself by the approach we have taken, as outlined above. We believe that Mathematica, by virtue of its flexibility and rich and unique library of functions, is specially suited to perform certain symbolic transformations that a general-purpose equation-solving system calls for. We augment this feature with the computational efficiency of C++/Fortran by employing the MathCode translator, that seamlessly integrates with Mathematica.

Secondly, we are inspired by efforts on the development of a high-level language called Modelica for modeling a broad class of engineering and physical systems [18,7]. Modelica is an object-oriented language that is equation based and acausal, and presents a very sound methodology for modeling of complex systems. However, presently there is no simulation support available in Modelica for handling PDE problems, (however, a preliminary prototype has been developed [23]), and only ordinary differential equations (ODEs) are supported. With this in mind, we have worked on an implementation of the method of lines, which is one kind of finite difference method that transforms a PDE into a system of coupled ODEs. In addition, the solver generator generates Modelica code for solving the resulting system of ODEs, which can be incorporated into a Modelica program as part of a class declaration. We thus achive a preliminary PDE support to the Modelica language without any change in its syntax. Further, the MathModelica system, an environment for programming in the Modelica language within Mathematica is presently available [7,9,14,16], and our solver generator can be straightforwardly integrated with this system with a little more effort.

The results presented in this paper are the following. We have implemented implicit finite difference methods, and give examples of implicit stencils obtained using our solver generator, and show how we can automatically generare implicit methods to a given approximation order. We then present the solution of the one-dimensional diffusion equation using the implicit method, and contrast the stable solution thus obtained with the unstable solution that one would obtain for the same problem using the explicit method. This brings out an advantage of the implicit method over the explicit method very clearly: we do not have to obey the stability criteria in the former case; we also comment on the considerations involved in choosing between explicit and implicit methods. We then discuss the implementation of the method of lines that transforms a PDE problem to an ODE one, and give an example of Modelica code generated for the transformed problem. The resulting code can be used in a Modelica program as a PDE model class without any syntax extensions of the language.

This paper is organized as follows. In Section 2 we describe the symbolic transformations involved in implementing the implicit finite-difference scheme, followed by a description of the method of lines in Section 3. These sections describe the symbolic and numeric aspects of the solver generator in detail, and present a few examples of applying the various tools developed. We make some concluding remarks in Section 4.

2. The implicit method

In the finite difference method, the independent variables are regarded as discrete and the domain becomes a grid. The derivatives of the dependent variables then automatically become differences between values at a combination of these grid points; the actual combination depends on the nature of the difference approximation. After such an approximation is performed,

we have no derivatives present in the system, only the functions at the grid points; this resulting system of relations between the values of dependent variables at a set of neighboring grid points is referred to as the stencil for the PDE system. Applying the stencil to all the grid points results in a system of coupled algebraic equations. Solution of this system of algebraic equations involves iteration from the boundaries, where the function values and/or derivatives are known. Often, one of the independent variables is singled out as the marching variable, and the solution is progressed along this direction: from the function values up to a certain value of the marching variable, the function values at the next value of the marching variable are computed using the stencil.

The stencil at any grid point, as we said above, is a relation between the values of the dependent variable at this point and its neighbors. If the approximation order used for the derivative with respect to the marching variable is n, then the stencil has dependent variables at $n+1$ values of the marching variable appearing: $u[i], u[i+1], \ldots, u[i+n]$ (where the arguments of the dependent variable u are the discrete values of the marching variable; we have suppressed other independent variables to keep the notation simple). If we can solve the stencil for $u[i+n]$ explicitly in terms of $u[i], u[i+1], \ldots, u[i+n-1]$, the difference scheme is called explicit; otherwise, it is called implicit.

The solution method can involve more computation in an implicit scheme than in an explicit scheme, because we have to solve a matrix system for each value of the marching variable in the implicit case. However, there is one definite advantage: the implicit schemes do not have the stablity problems that often plague explicit methods. This means that there is no criterion involving the step sizes of the independent variables that needs to be satisfied for the resulting solution to be stable. This can contribute to a reduction in the computational complexity, since we can choose larger step sizes for the marching variable without the danger of an unstable solution.

In the following, we discuss various symbolic aspects involved in implementing implicit schemes. This discussion parallels the one in our earlier paper on explicit schemes [24], to which we refer the reader for further details.

2.1. Format

The general format for specifying a PDE problem to our solver generator is the following:

```
testproblem={equations, geometry,
approximation specifications}
```

Here, equations is a list of all the equations in the problem, namely the PDE and all the intial and boundary conditions; geometry is a list that specifies the problem domain; finally, approximation specifications is a list that contains the difference method to be used for each member of the equations list. Each of these lists can have arbitrary numbers of sublists depending on the complexity of the problem. Note that with this format, it is possible to unambiguously state any PDE problem.

We illustrate the input format with an example of a simple one-dimensional PDE problem, the diffusion equation:

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

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 generator as follows:

```
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, 1 - x]}
    (*initial condition*)
   },
   {
    {{x, 0, 1}}, {{t, 0, tmax}} (*geometry*)
   },
   { (*difference approximation
      specifications:*)
    {u[x[{}, {2, 2, {0, 1}}],
      t[{1, 1, {0, 0}}]]},
    {u[t[], x[{1}]]},
    {u[t[], x[{1}]]},
    {u[x[], t[{1}]]}
   }
  };
```

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: the latter have to have the space/time boundary specified; the first element in these sublists specifies the boundary, and the remaining elements specify the conditions that hold at this boundary.

The geometry list has as many sublists as there are independent variables. Each of these sublists can have a number of further sublists, if the system geometry breaks up into disconnected subdomains in any direction. We can specify arbitrary geometries within this format; a few examples are given in Section 2.3.

There is an element (that we call a method) in the approximation specifications list for each element in the equations list. In particular, the first element in the approximation specifications list corresponds to the PDE. This first element has only one element, since there is only one PDE in this case, and has the form $u[x[\{\}, \{2, 2, \{0, 1\}\}], t[\{\{1, 1\}\}]$. This is an implicit approximation specification, and the meaning of this format is as follows. An element $\{m, n, \{\Delta x, \Delta t\}\}$ 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 $\{m, n, \{\Delta x, \Delta t\}\}$ as basic approximation specifiers, since they specify how the individual derivatives appearing in a PDE have to handled. The list $\{\Delta x, \Delta t\}$ is optional: if it is absent, the method is explicit, and its presence here indicates that we are dealing with an implicit method. Its meaning is as follows. It contains the instruction that, after applying the explicit difference approximation $\{m, n\}$, we should replace x by $x + \Delta x$ and t by $t + \Delta t$ in the resulting finite difference; these replacements have the effect of making the stencil implicit.

Note that any implicit method that involves merely replacing every derivative in the PDE by the result of an approximation specification like $\{m, n, \{\Delta x, \Delta t\}\}$ can be expressed in the form

$$
\begin{aligned}
&u_1(x_1\{m_1, n_1, \{\Delta x_1, \Delta x_2, \ldots\}\}, \ldots,\\
&\quad \{m_{o_1}, n_{o_1}, \{\delta x_1, \Delta x_2, \ldots\}\}],\\
&x_2[\{m_2, n_2, \{\Delta x_1, \Delta x_2, \ldots\}\}, \ldots,\\
&\quad \{m_{o_2}, n_{o_2}, \{\Delta x_1, \Delta x_2, \ldots\}\}], \ldots],.
\end{aligned}
$$

which is completely specified by a set of integers m_i and n_i. As a result, we can generate all possible such approximation specifications in an automated way, by simply generating all possible sets of integers allowed in the approximation specifications. The latter are fairly unambiguous (for a detailed discussion see our earlier paper [24]). Even the generation of a stencil by applying such an approximation specification can be easily automated.

The first basic approximation specifier of x in the above example is $\{\}$: since the PDE has no first derivative of u with respect to x, the solver generator 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 $\{2, 2, \{0, 1\}\}$ (corresponding to $m = 2 = n$, $\Delta x = 0$ and $\Delta t = 0$ in $\{m, n, \{\Delta x, \Delta t\}\}$), the second argument of x, is the basic approximation specifier for the second derivative of u with respect to x. We have $\{2, 2, \{0, 1\}\}$ here because we have chosen to replace the second derivative of u with respect to x by a second-order central difference, and add to the discretized t the integer 1 (and leave the discretized x variable intact; that is, the finite-difference approximation of the second order space derivative is replaced by its value one time step later) in the resulting difference, to make the method implicit. The following example demonstrates what we mean here:

$$
\begin{aligned}
&\text{DiscretizeImplicit}[\\
&\{\partial_{\{t,1\}} u[x, t] = \partial_{\{x,2\}} u[x, t]\},\\
&\{u[x[\{\}, \{2, 2, \{0, 1\}\}],\\
&t[\{1, 1, \{0, 0\}\}]]\}, \text{ step}]\\
&\left\{\frac{-u[x, t] + u[x, 1 + t]}{\text{step}[2]} =\right.\\
&\frac{1}{\text{step}[1]^2}(u[-1 + x, 1 + t]\\
&-2u[x, 1 + t] + u[1 + x, 1 + t])\}
\end{aligned}
$$

It can be seen that we can't solve the resulting stencil for the dependent variable (u) at the highest value ($t +$

1) of the marching variable, hence this is an implicit method. (Note that in the above, step is a list of step sizes of the discretized independent variables.)

In general, if the largest-ordered derivative with respect to an independent variable in the equation is L, then the number of arguments (of the form $\{m, n, \{\Delta x, \delta t\}\}$) of this variable must be L, some of which are possibly empty lists. However, if the PDE has kth derivative appearing, then the kth basic approximation specifier 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 approximation specifications list is in order. Some approximation specifications can result in a 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 approximation specifications 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 approximation specifications list. In the present example, since these conditions have no derivatives appearing, the independent variables in the corresponding approximation specifications 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. Iteration scheme: The sweep method

The main difference between explicit and implicit schemes is in the solution procedure. This is because in the implicit method, the stencil does not permit a closed form expression for the dependent variable at the latest value of the marching variable in terms of its values at earlier marching variable values. A variety of solution schemes have been developed to solve the system of algebraic equations, and we shall restrict ourselves to just one in this paper, namely the Sweep method (see, for instance [3]).

A brief description of the Sweep method is as follows. Suppose we are concerned with solving the problem parabolic1D, presented in Section 2.1 (α and β are the boundary values of u in the following). The application of the implicit method results in the following system of equations that must be solved at every time instant:

$$u[0, n+1] = \alpha;$$
$$(1+2r)u[k, n+1] - ru[k+1, n+1] -$$
$$ru[k-1, n+1] = u[k, n];$$
$$u[K, n+1] = \beta;$$

for $k = 1, 2, \ldots, K-1$. Here, $r = \Delta t/\Delta x^2$ is the ratio of step sizes (Δx and Δt are, respectively, the x and t step sizes), and K is the system size in the x-direction. The middle equation above has three unknowns for every k, namely $u[k-1, n+1]$, $u[k, n+1]$ and $u[k+1, n+1]$. Now, since $u[0, n+1]$ is given, we can get a relation between $u[1, n+1]$ and $u[2, n+1]$ for $k = 1$. Similarly, we can get a relation between $u[2, n+1]$ and $u[3, n+1]$ for $k = 2$, and so on for $k = 3, 4, \ldots, K-1$. Suppose we assume the following form, where we introduce the coefficients $L[k]$ and $M[k]$,

$$u[k-1, n+1] = L[k]u[k, n+1] + M[k];$$

for the relationship between $u[k-1, n+1]$ and $u[k, n+1]$. In the same way, we can write

$$u[k, n+1] =$$
$$L[k+1]u[k+1, n+1] + M[k+1];$$

Substituting for $u[k-1, n+1]$ and $u[k, n+1]$ using these two equations in the second of the three equations above and rearranging terms, we obtain

$$u[k, n+1] =$$
$$\left(\frac{r}{1+2r-rL[k]}\right) u[k+1, n+1]$$
$$+ \left(\frac{u[k, n] + rM[k]}{1+2r-rL[k]}\right).$$

Comparison of the last two equations shows that we can write

$$L[k+1] = \frac{r}{1+2r-rL[k]},$$
$$M[k+1] = \frac{u[k, n] + rM[k]}{1+2r-rL[k]}.$$

These two equations serve as recursion relations for L and M. Using these recursion relations and the

boundary conditions, we can compute $L[k]$ and $M[k]$ for all values $k = 1, 2, \ldots, K-1$, and from these, we can compute u in the entire domain.

We have implemented this method in our solver generator, where Mathematica code for the function that performs the solution procedure of the linear system is generated. The details of this code are problem dependent, and only the structure of the code can be ascertained a priori. 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 inside nested loops. The character and number of assignment statements in each nested loop depend on the nature of the initial/boundary conditions and the PDE. As a result, a different iteration function is generated for each PDE problem. (Our use of the term 'iteration function' is not to be confused with the terms 'direct' and 'iterative', used to refer to solution schemes for linear algebraic systems.) This iteration function is specified for the particular PDE problem and can therefore be made rather computationally efficient. It has type declarations of all the local variables, since we intend to generate C++ or Fortran90 code for the iteration function using the MathCode compiler [6].

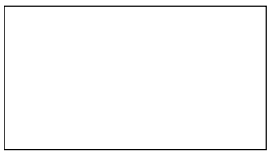
2.3. A note on system geometry

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.

1. *A Rectangular Geometry:* A geometry list of the form

   $$\{\{\{x, 0, 2\}\},\ \{\{y, 0, 1\}\}\}$$

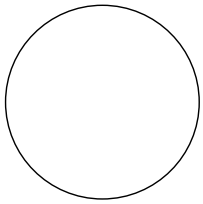
   describes a simple rectangle of sides 2 and 1 in the xy-plane.



2. *A Circular Geometry:* A geometry list of the form

   $$\{\{\{x, -\text{Sqrt}[1 - y\hat{}2],$$
   $$\text{Sqrt}[1 - y\hat{}2]\}\},\ \{\{y, -1, 1\}\}\}$$

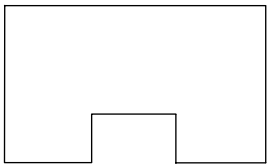
   describes a circle of radius 1 in the xy-plane. The following figure shows a plot of the geometry:



3. *An Irregular Geometry:* A geometry list of the form

   $$\{\{\{x, 0, \text{If}[y \leqslant y1, x1, 1]\},$$
   $$\{x, \text{If}[y \leqslant y1, x2, 1.0], 1.0\}\},$$
   $$\{\{y, 0, 1\}\}, \{\{t, 0, t\,\text{max}\}\}\}$$

   describes an irregular geometry: x has two disconnected subdomains ("rectangular wells") up to a certain value $y1$ of y, and only one beyond that.



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.

2.4. C++ code generation

For the numerical part, we employ the MathCode code generator [6] which generates optimized C++/Fortran 90 code for a suitably stated Mathematica task. The Mathematica code generated for the iteration function has 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 for the iteration function and the array dimension de-

pend 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++/Fortran 90 code that can be run transparently as if the code was executed within Mathematica. This is achieved by MathCode by loading the generated 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 by a factor of 150. We found that the performance of the generated C++ code was the same as that of the generated Fortran 90 code; further, the compiled language can be specified as a simple option (Language->"Fortran 90" or Language->"C++") to the BuildCode function that is available in the MathCode system.

2.5. Examples

We give examples of application of some Mathematica modules that we have written to implement the implicit method. For a more detailed exposition of our solver generator, we refer the reader to our earlier work [24].

Here is the way the implicit stencil is generated:

$$
\begin{aligned}
&\text{DiscretizeImplicit}[\\
&\{\partial_{\{t,1\}}u[x,t] = \partial_{\{x,2\}}u[x,t]\},\\
&\{u[x[\{\},\{2,2,\{0,1\}\}],\\
&t[\{1,1,\{0,0\}\}]]\},\ \text{step}]\\
&\left\{\frac{-u[x,t]+u[x,1+t]}{\text{step}[2]} =\right.\\
&\frac{1}{\text{step}[1]^2}(u[-1+x,1+t]-\\
&2u[x,1+t]+u[1+x,1+t])\}
\end{aligned}
$$

Here, step is a list of step sizes of the independent variables. If we used the same approximation specification but with the function Discretize, which generates an explicit stencil, here is what we would obtain:

$$
\begin{aligned}
&\text{Discretize}[\\
&\{\partial_{\{t,1\}}u[x,t] = \partial_{\{x,2\}}u[x,t]\},\\
&\{u[x[\{\},\{2,2,\{0,1\}\}],\\
&t[\{1,1,\{0,0\}\}]]\},\ \text{step}]\\
&\left\{\frac{-u[x,t]+u[x,1+t]}{\text{step}[2]} =\right.\\
&(u[-1+x,t]-2u[x,t]\\
&+u[1+x,1+t])/\text{step}[1]^2\}
\end{aligned}
$$

We can automate the generation of implicit approximation specifications such as the one used above. We give below an example in which we generate a set of finite-difference approximation specifications to order 2 in x and order 1 in t. Each of these approximation specifications is implicit: each replaces $\partial_{\{x,2\}}u[x,t]$ by its explicit finite difference at $t+1$.

We show below the iteration function using the Sweep method (Section 2.2), generated for the one-dimensional diffusion equation of Section 2.1.

Note that the body of the module implements the Sweep method. The reader might notice the Mathematica Round function in many parts of the module. This function arises when we calculate the limits for the loop variables from the geometry list in the given problem: the round function ensures that the limits are always integers. For example, if the system extends from $x\min$ to $x\max$ in the x direction, and the number of steps in x is nx, the integer corresponding to a value $x1$ of x is $1+\text{Round}\left[\frac{(-1+nx)(x1-x\min)}{x\max-x\min}\right]$. When $x1 = x\max = 1$, $x\min = 0$, this is Round$[nx]$, corresponding to the upper limit in the above module. When we execute this module, we obtain the solution of the PDE problem. For this end, we generate a C++ code using the MathCode translator, which can be run within Mathematica. We will not give the details here, but only present the results.

Suppose nx and nt define the problem size, and the lists $\{0,1\}$ and $\{0,t\max\}$ define the system size in the x and t directions. The step sizes are then $h = 1/(nx-1)$ and $\tau = t\max/(nt-1)$, respectively, in the x and t directions. These numbers can be specified at runtime.

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

```
{w[x[{-1, 0, {0, 0}}, {2, 2, {1, 0}}],
  t[{1, 0, {0, 0}}]]},
{w[x[{-1, 0, {0, 0}}, {2, 3, {1, 0}}],
  t[{1, 0, {0, 0}}]]},
{w[x[{-1, 0, {0, 0}}, {2, 0, {1, 0}}],
  t[{1, 1, {0, 0}}]]},
{w[x[{-1, 0, {0, 0}}, {2, 1, {1, 0}}],
  t[{1, 1, {0, 0}}]]},
{w[x[{-1, 0, {0, 0}}, {2, 2, {1, 0}}],
  t[{1, 1, {0, 0}}]]},
{w[x[{-1, 0, {0, 0}}, {2, 3, {1, 0}}],
  t[{1, 1, {0, 0}}]]},
{w[x[{-1, 0, {0, 0}}, {2, 0, {1, 0}}],
  t[{1, 2, {0, 0}}]]},
{w[x[{-1, 0, {0, 0}}, {2, 1, {1, 0}}],
  t[{1, 2, {0, 0}}]]},
{w[x[{-1, 0, {0, 0}}, {2, 2, {1, 0}}],
  t[{1, 2, {0, 0}}]]},
{w[x[{-1, 0, {0, 0}}, {2, 3, {1, 0}}],
  t[{1, 2, {0, 0}}]]}}
```

The choice $\{nx, nt, t\max\} = \{100, 100, 0.01\}$ corresponds to $r = \hat{o}/h^2 = 0.99$, which clearly violates the criterion $r \leqslant 1/2$ for

```
Implicit[parabolic1D, {0, 0},
 {0, tmax}, {tmax}, {1, 201, 501}]

HoldPattern[solvePDE[Nx_, Nt_, tmax_]] :>
 Module[{Integer Ix, Integer It},
  {M1[[1 ;; 1, 1, 1 ;; 501]] = 0; L1[[1 ;; 1, 1, 1 ;; 501]] = 0;
   U[[1 ;; 1, 1, 1 ;; 201, 1 ;; 501]] = 0;
   For[It = 1, It < 1 + Round[Nt],
    It = 1 + It, U[[1, 1, It]] = 0];
   For[It = 1, It < 1 + Round[Nt],
    It = 1 + It, U[[1, Round[Nx], It]] = 0];
   For[Ix = 1, Ix < 1 + Round[Nx], Ix = 1 + Ix,
    U[[1, Ix, 1]] = If[(-1 + Ix)/(-1 + Nx) < 0.15,
      (-1 + Ix)/(-1 + Nx), 1 - (-1 + Ix)/(-1 + Nx)]];
   For[It = 2, It < 1 + Round[Nt], It = 1 + It,
    (L1[[1, 1]] = 0; M1[[1, 1]] = U[[1, 1, It]]);
    {For[Ix = 2, Ix < 1 + Round[Nx],
      Ix = 1 + Ix, L1[[1, Ix]] =
       -1 / ((2 (-1 + Nx)^2 + (-1 + Nt)/tmax)/(-1 + Nx)^2 +
          L1[[1, -1 + Ix]])];
     M1[[1, Ix]] = -(M1[[1, -1 + Ix]] +
        ((-1 + Nt) U[[1, -1 + Ix, -1 + It]]) / ((-1 + Nx)^2 tmax)) /
       ((-2 (-1 + Nx)^2 + (-1 + Nt)/tmax)/(-1 + Nx)^2 +
        L1[[1, -1 + Ix]])];
   For[Ix = -1 + Round[Nx], Ix > 1,
    Ix = -1 + Ix, CompoundExpression[
     U[[1, Ix, It]] = M1[[1, 1 + Ix]] + L1[[1,
        1 + Ix]] U[[1, 1 + Ix, It]]]]};]
```

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

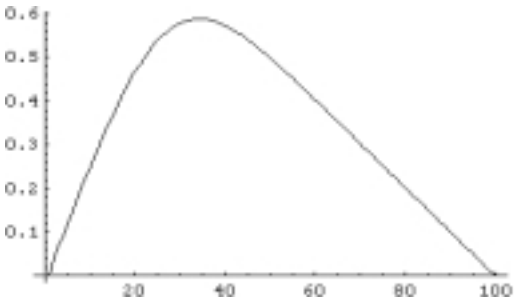


Fig. 1.

an explicit method. However, we can see below that the solution obtained by the implicit method is stable for this choice, as expected. The following plot is the solution for the last time slice.

The explicit method gave the following solution to the same problem, which we show below just to bring out the difference.

This shows us that we can obtain stable solutions without regard to the choice of step sizes, and can thus reach a desired value of the march variable in fewer steps, illustrating an advantage of the implicit method over the explicit method. This is because the stability criterion does not arise for implicit methods. However, there is a trade-off here. A larger step size results in a larger truncation error, which is obviated by the stability requirement in explicit methods. Further, for each value of the march variable, typically one needs to perform a larger amount of computation in implicit methods than in explicit methods: consider for instance

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

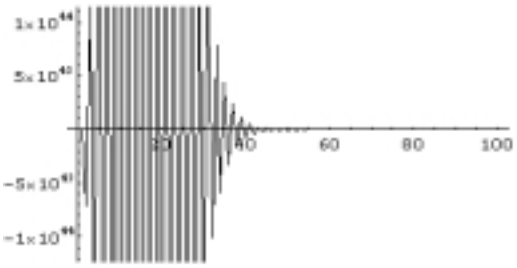


Fig. 2.

the need for solving the recursion relations for L and M illustrated in Section 2.2; the amount of computation is far less for explicit methods. The choice between explicit and implicit methods is therefore determined by considerations of truncation error and computational efficiency.

3. The method of lines

The method of lines is a particular kind of finite-difference method that transforms a PDE into a system of coupled ordinary differential equations (ODEs) (see, for example, [17]). This is achieved by discretizing all except one independent variable (usually time), which is left intact. Consequently, the only derivatives that remain are those with respect to this variable, and hence the system that results is a system of ODEs. Our motivation for implementing this method within our solver generator comes from our desire to provide PDE support to the Modelica language [18], that presently supports only ODEs. Our approach here is to perform the method of lines on a given PDE problem, and then generate Modelica code that can be integrated into a Modelica program as a PDE model. This doesn't require any changes in the existing Modelica syntax, and is therefore very straightforward.

3.1. The method

We illustrate the method of lines with the following problem:

Note in the above that in the approximation specifications part for the PDE, we have specified no discretization for the variable t. This corresponds to the method of lines and results in a system of ODEs for the variables $\{u[k,t]|k = 2,3,\ldots,K-1\}$:

```
parabolic1D =
  {
   { (*equations*)
    {D[u[x, t], {t, 1}] ==
      D[u[x, t], {x, 2}]},
    {x == 0, u[x, t] == 0},
    {x == 1, u[x, t] == 0},
    {t == t0,
     u[x, t] == If[x < 0.15, x, 1 - x]}
   },
   { (*geometry*)
    {{x, 0, 1}}, {{t, 0, tmax}}
   },
   { (*approximation specifications*)
    {u[x[{}, {2, 2}], t[{-1, 0}]]},
    (*central diff for space,
     time left intact*)
    {u[t[], x[{1}]]},
    {u[t[], x[{1}]]}, {u[x[], t[]]}
   }
  };
```

$$\partial_t u[k,t] = u[k-1,t] - 2u[k,t] + u[k+1,t];$$

The above equation for $k = 2,3,\ldots,K-1$ becomes a system of coupled ODEs for the $K-2$ variables $u[k,t]$ that can be presented to an ODE solver. The method of lines therefore becomes a particular instance for the solver generator, with the added benefit that we are able to describe fairly arbitrary geometries.

3.2. Modelica code generation

We treat the resulting ODE problem as a Modelica model and generate a code for it. This is done by a series of symbolic transformations. The variable $u[k,t]$ becomes simply $u[k]$; time dependence is suppressed since every variable in Modelica is automatically time dependent. In the following we show how code for a Modelica model is generated for a two-dimensional diffusion equation in a simple rectangular geometry (the problem list parabolic2D is a two-dimensional generalization of parabolic1D, and is not shown here), with just three grid points in each of the two spatial directions, while time is left undiscretized:

The spatial discretization chosen is the two dimensional central difference scheme (specified in parabolic2D, not shown here). This program can be compiled by a Modelica translator (for instance the Dymola com-

```
LinesModExtended[parabolic2D, PDE[3, 3],
 {-1, -1}, {1, 1}, der,
 {left, right, below, above},
 {Real, Real, Real, Real}, {0., 0., 0., 0.}]
 model PDE

   parameter Real left = 0.;
   parameter Real right = 0.;
   parameter Real below = 0.;
   parameter Real above = 0.;

   Real U[1, 2] (start = 1.);
   Real U[2, 1] (start = 1.);
   Real U[2, 2] (start = 0.);
   Real U[2, 3] (start = 1.);
   Real U[3, 2] (start = 1.);

   equation
      "der(U[2, 2])" = U[1, 2] + U[2, 1]
        - 4*U[2, 2] + U[2, 3] + U[3, 2];
      U[1, 2] = left;
      U[2, 1] = below;
      U[2, 3] = above;
      U[3, 2] = right;

 end PDE;
```

piler [4], the MathModelica compiler [9,14,16], or the Open Source Modelica compiler [8]) and shown to give correct results.

4. Conclusions

The solver generator that we have built is based on a symbolic-numeric framework: we use the symbolic power of Mathematica to make certain transformations on the given PDE problem, and generate code for its numerical solution in a compiled language like C++. This combination results in a flexible solution tool that is also very computationally efficient. In our earlier work [24], in which we developed this framework, we implemented only explicit finite-difference methods and stated that we could extend it to implicit methods. In the present paper we have demonstrated this extension.

The extension to implicit methods further strengthens the framework, since this gives us access to methods that are free from stability problems, thus giving us a wider range of solution schemes to choose from. Further, if chosen judiciously, this can also result in a reduction in the amount of computation in many cases, enhancing the efficiency of the solver.

We have also implemented the method of lines, and have shown that the solver generator can generate code for a Modelica PDE model in this case. This code can be directly incorporated into a Modelica program without any change in the syntax of the language. This provides a preliminary level of PDE support to the Modelica language, exploiting the ODE support that already exists. However, this severely underutilizes the possibilities of our solver generator, restricting PDE support to the method of lines. This is a gap that we are presently working to bridge.

The basic framework within which we have implemented the various difference methods has a high degree of generality, which we have discussed at length in our earlier paper [24]: it can handle arbitrary numbers of dependent and independent variables, and any approximation order and geometry. This is mainly due to the fact that we have chosen to develop our solver generator in Mathematica; a by-product of this choice is that we can use the rich library of functions available in Mathematica, and the interactive and graphical aspects of its notebook environment.

In future, we plan to work on an extended PDE support to the Modelica language. A Mathematica environment for this language, known as MathModelica, is presently available (see [7,9,14,16]), and we believe that our solver generator would easily integrate with this system. Another issue we would like to address is to test our solver generator with realistic example problems. This requires cooperation with research groups that work on specialized problems that involve solving PDE systems, and we are presently pursuing this line. Finally, we are thinking hard about the possibilities and limitations of the solver generator in treating the finite-element methods.

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] V.F. D'yachenko, *Basic Computational Mathematics,* Mir Publishers, Moscow, 1979.
[4] The Dynasim website, http://www.dynasim.se.
[5] The ELLPACK webpage, http://www.cs.purdue.edu/ellpack/ellpack.html.
[6] P. Fritzson, MathCode C++, published by MathCore (http://www.mathcore.com/), 1998.

[7] P. Fritzson, Principles of Object Oriented Modeling and Simulation – with Modelica 2.0, first chapter of book draft available at (http://www.ida.liu.se/labs/pelab/modelica/).

[8] P. Fritzson, P. Aronsson, P. Bunus, V. Engelson, H. Johansson, A. Karström and L. Saldamli, The Open Source Modelica Project, in *Proceedings of the 2nd International Modelica Conference,* March 18–19, 2002, Munich, Germany. (Paper at http://www.modelica.org. More info at www.ida.liu.se/labs/pelab/modelica).

[9] P. Fritzson, V. Engelson and J. Gunnarsson, An Integrated Modelica Environment for Modeling, Documentation and Simulation, in *Proceedings of SCSC-98* (*Summer Computer Simulation Conference*), Reno, Nevada, July 1998.

[10] P. Fritzson, J. Gunnarsson and M. Jirstrand, MathModelica – An Extensible Modeling and Simulation Environment with Integrated Graphics and Literate Programming, in *Proceedings of the 2nd International Modelica Conference,* March 18–19, 2002, Munich, Germany. Short version available at http://www.modelica.org/. Long version available at http://www.ida.liu.se/labs/pelab/modelica/.

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

[12] B. Gustafsson, H.-O. Kreiss and J. Oliger, *Time Dependent Problems and Difference Methods,* John Wiley & Sons, Inc., 1995.

[13] H.P. Langtangen, Computational Partial Differential Equations – Numerical Methods and Diffpack Programming, Springer-Verlag, 1999.

[14] M. Jirstrand, J. Gunnarsson and P. 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).

[15] An Object-Oriented Framework for PDE Solvers, Krister Åhlander, PhD thesis, Dept. of Scientific Computing, Uppsala University, Sweden, 1999.

[16] The MathCore website, http://www.mathcore.se.

[17] M. Oh, Modelling and simulation of combined lumped and distributed processes, Ph.D thesis, University of London, 1995.

[18] Modelica Association homepage (http://www.modelica.org/).

[19] E. Mossberg, K. Otto and M. Thuné, Object-oriented software tools for the construction of preconditioners, *Scientific Programming* **6** (1997), 285–295.

[20] W.D. Henshaw, Overture Documentation, LLNL Overlapping Grid Project. http://www.llnl.gov/CASC/Overture/henshaw/overtureDocumentation/overtureDocumentation.html.

[21] The PELLPACK webpage http://purdue.cs.purdue.edu./.

[22] J.R. Rice and R.F. Boisvert, *Solving Elliptic Problems Using ELLPACK,* Springer, 1984.

[23] L. Saldamli, P. Fritzson and B. Bachmann, Extending Modelica for Partial Differential Equations, in *Proceedings of the 2nd International Modelica Conference,* March 18–19, 2002, Munich, Germany, (www.modelica.org).

[24] K. Sheshadri and P. Fritzson, A General Symbolic PDE Solver Generator: Explicit Schemes, accepted for publication in Scientific Programming, 2002; A Mathematica notebook version of the paper is available upon request from the authors.

[25] K. Sheshadri and P. Fritzson, A Mathematica-based PDE Solver Generator, pages 66–78, Proceedings of SIMS'99, The 1999 Conference of the Scandinavian Simulation Society, Linköping, Sweden. (www.scansims.org).

[26] Dept. Scientific Computation, Uppsala University, website. http://www.tdb.uu.se/research/swtools/.

[27] 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).