Transforming systems of PDEs for efficient numerical solution

Andreas Wrangsjö, Peter Fritzson and K. Sheshadri

Department of Computer and Information Science

Linköping University, S−581 83 Linköping, Sweden

Email: {andwr, petfr, shesh}@ida.liu.se

Phone: +46 13 281484, Fax: +46 13 284499

Abstract

A *Mathematica* package to deal with a system of partial differential equations (PDEs) is presented. This package uses explicit finite−difference schemes to handle equations in an arbitrary number of variables that are functions of one spatial variable and time. The code has the flexibility to incorporate any difference approximation specified by the user, and transforms the given system of PDEs into a system of difference equations that can be iteratively solved using the discretized forms of initial and boundary conditions. The iteration is made considerably faster by converting the *Mathematica* code into an optimized C++ code using the MathCode C++ compiler[1]. Examples are presented in which the generated C++ code runs about a thousand times faster than the *Mathematica* code.

1. Introduction

There exist many packages for solving partial differential equations (PDEs) (see references [MinOh95, Ames92, Gust95] for an overview). The motivation for developing yet another package is the following. The existing packages are very efficient, but they are invariably restricted to some particular solution method, and hence to the class of equations for which the solution method is efficient. Often the user has very little freedom to choose an approximation scheme for a particular equation that needs to be solved. There also exists a *Mathematica* package [Ganza96] that can handle a variety of PDEs, but in our opinion it has two limitations: (1) it does not provide a framework for treating an arbitrary PDE, and (2) it uses *Mathematica* for numerical computation, which makes it rather inefficient.

It would be both challenging and useful to develop a PDE solver that has (a) the generality and flexibility to handle, at least in principle, an arbitrary equation with an arbitrary approximation method, and (b) the efficiency and speed to be useful in a real−world situation. We believe that the package that we have developed is a first step towards this goal: we use the symbolic abilities of *Mathematica* to address the first issue, and the idea of code generation to address the second. At present, our package is restricted to systems of PDEs in functions of two variables, and to explicit finite−difference schemes, but can be extended to overcome these limitations in a straightforward manner. Any further limitation is only because the field of PDEs itself is not, and can not be, completely systematized. The high degree of flexibility of our package is because it uses *Mathematica* for the symbolic parts of the problem; its efficiency is because it uses a compiled language (C++ in this case) for numerical computation.

Our main results are the following. We have defined various *Mathematica* functions that symbolically perform the discretization of the variables in the given PDE problem and generate a set of difference equations. This is done based on the approximation method specified in the input. We test these functions on a simple one dimensional parabolic equation $\frac{\partial u(x,t)}{\partial t} = \frac{\partial^2 u(x,t)}{\partial x^2}$ for x=[0,1] and t=[0,∞] with the boundary conditions $u(0,t)=0=u(1,t)$ and the initial condition $u(x,0)=x/2$ for $x<1/2$ and $u(x,t)=1-x/2$ for $x\geq 1/2$. We find that the compiled C++ code runs about a thousand times faster than the interpreted *Mathematica* code for the iteration part.

This paper is organized as follows. In Sec.2 we describe in some detail the way in which we discretize the independent variables to generate a grid and convert the given PDE problem into a system of difference equations to be iteratively solved to obtain the solution at an arbitrary point on the grid. The iteration is done most efficiently by defining a separate function and generating a C++ code for it; the way in which this is done is described in Sec.3. We demonstrate our package for the parabolic equation in Sec.4. Some conclusions and possible future work are discused in Sec.5.

2. Discretizing the PDE Problem

The following format is used for specifying the PDE problem. The problem specification is divided into six lists: (i) *equations*, which contains the PDE, (ii) *initials*, which contains the initial conditions, (iii) *boundaries*, which contains the boundary conditions, (iv) *geometries*, which specifies the system geometry, (v) *variables*, which specifies the problem variable names, the number of grid points and grid spacings, and (vi) *methods*, which specifies the kind of difference approximation and numerical boundary conditions to be used. These lists are given as options to a function called **`DefinePDEProblem`** that we have defined.

At the highest level, our package has a function **`solvePDE`** which performs the iterative numeric computation to obtain the solution at an arbitrary grid point. This iteration uses a set of difference equations, which are obtained by discretizing the given PDE and the initial and boundary conditions. These difference equations are converted into assignment statements for the array of dependent variables: this is a three dimensional array in the present case, the first denoting the name of the dependent variable, and the second and the third, the space and time grid coordinates, respectively.

The assignment statements corresponding to the given PDE are based on its stencil that is obtained by applying the given difference method on the equation. The stencil is generated as a rule that relates the value of the dependent variable at a grid point to its values at a set of other points. The actual set of points is determined by the approximation method, which is specified in the *methods* list. The functions corresponding to the dependent variables are then replaced by the corresponding array variables.

In a similar manner, assignment statements are generated using the discretized forms of the initial and boundary conditions. Together, these three sets of assignment statements are sufficient to iteratively determine the values of the dependent variables at any grid point.

The above process relies on a number of lower level functions which can be classified into two groups:

(1) Information retrieval functions: These functions extract the various global variables, the dependent and independent variables, the approximation orders, and the solution methods from the input lists.

(2) Discretization functions: At the highest level in this category are the functions that generate the stencil and the discretized forms of the initial and boundary conditions. These in turn depend on the functions that constitute the approximation methods, that in turn are constructed from functions that replace derivatives of any order by forward, backward or central difference approximations of a specified order. Finally, there are functions that can transform the independent variables between discrete (grid) and continuous forms.

3. 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 numerical iteration part is done symbolically in the first part of the package, called PDESymbolic. The result of this part are the assignment statements referred to above, along with some global variables that are extracted from the input. This part of the problem is the so called set−up part, and the time it takes is independent of the problem size. The numerical part of the package, PDERuntime, performs the numerical iteration using these assignment statements, and the time it takes increases with the problem size. Evidently, the efficiency of the package is determined by how efficiently the iteration is performed.

We use the idea of code generation to perform efficient numerical computation. We employ the MathCode C++ code generator [Fritz98] which generates efficient C++ code for a suitably stated *Mathematica* task. This requires us to declare the types of all the variables, functions and arrays that the iterator uses, and this we do in the PDERuntime part of our package. We can also directly run the iterator **`solvePDE`** interpretively within *Mathematica.* We find, however, that the generated C++ code for the **`solvePDE`** function runs almost a thousand times faster than the *Mathematica* run. In the next section, we give an example of a parabolic equation, for which we obtain a speedup of about one thousand.

4. An Example: Parabolic Equation

In this section we demonstrate the use of our symbolic functions and code generation for the parabolic equation in one spatial dimension and time:

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

with an initial condition

$u(x,0)=x/2$ for $x<1/2$ and $u(x,t)=1-x/2$ for $x\geq 1/2$,

and boundary conditions

$u(0,t)=0=u(1,t)$.

We use a difference method in which the time derivative is replaced by a first−order forward difference D^+ and the second−order space derivative is replaced by $D_+ D_-$, which is a second−order central difference. As a result, we obtain the difference equation

$$U(1,j,n)=(1-2r)U(1,j,n-1)+r\{U(1,j+1,n-1)+U(1,j-1,n-1)\}$$

for the discretized dependent variable $U(1,j,n)$ (corresponding to the continuous variable $u(x,t)$) at the grid point (j,n); here $r = k/h^2$ where h and k are the space and time steps, respectively. The first index refers to the dependent variable name, and is 1 in the present since we have only one dependent variable u. This difference equation is easily solved by using the discretized forms of the initial and boundary conditions, and it is well known (see [Ames92], for example) that the solution is convergent when $r<0.5$. We now show how this problem is handled by our package.

To begin with, the input is specified in the form of six lists using the function **DefinePDEProblem**:

```
DefinePDEProblem[ parabolic,
                  equations -> {∂u(x, t)/∂t == ∂²u(x, t)/∂x²},
                  initials ->
                  {{t == u[x, t] == If[x < 0.5, x, 1.0 - x]}},
                  boundaries -> {{x == 0, u[x, t] == 0},
                  {x == 1, u[x, t] == 0}},
                  geometries -> {x ≥ 0, x ≤ 1},
                  variables - {{x, 2, 200},
                  {t, 1, 500, 0.0000111111}, {u}},
                  methods -> {SingleStep1D, none, none}
];
```

where *equations*, *initials*, *boundaries*, *geometries*, *variables*, and *methods* are *Mathematica* option parameters. This function keeps the input data in an unevaluated form, by applying the **Hold** construct around the elements of the list in a suitable manner, and stores the problem definition in global variables named *equations*, *initials*, *boundaries*, *geometries*, *variables*, and *methods*.

Note that though the *equations* list has only one element in this case, our package can handle any number of entries in this list.

In the present case, the *initials* list has just one sublist that contains two statements: the first specifies the time point and the second specifies the value of the dependent variable at all space points at this time point. If there were more statements in the initial conditions of the problem, they could be added to this sublist.

In the same way, the *boundaries* list has two sublists for the left and right spatial boundaries of the system, each containing the values of spatial points followed by the function values at these points. More conditions, if there are any, can be added to these sublists.

The *geometries* list specifies the system boundaries, which are simply the left and right ends of the system in the present one−dimensional case.

The *variables* list has three entries in the present case. The first specifies the name of the spatial coordinate, the approximation order to be used for derivatives with respect to this variable, and the number of grid points along this direction. The second entry specifies similar information for the time coordinate; in this case, the discrete step size is also specified. The last element in the list specifies the name of the dependent variable.

Finally, the *methods* list has three entries. The first refers to the approximation method to be used on the PDE: in this case, we have specified **SingleStep1D**, which for the approximation orders 1 and 2, respectively, for time and space, reproduces the difference equation referred to at the beginning of this section. The next two entries are not required in the present example [Ganza96]: the second entry refers to the numerical boundary conditions, and the third refers to the numerical initial conditions.

Our information retrieval functions can then be used to extract data from the input. For example, the values of the x−coordinate at the system boundaries are extracted as follows:

```
GetGeometry1D[geometries, variables, "startx"]
```

```
0
```

```
GetGeometry1D[geometries, variables, "endx"]
```

```
1
```

We give some more examples below.

```
GetDiffMethod1D[variables, methods, "diffmethod"]
```

```
SingleStep1D
```

```
GetDiffMethod1D[variables, methods, "nx"]
```

```
200
```

```
GetDiffMethod1D[variables, methods, "nt"]
```

```
500
```

The difference equation corresponding to the given PDE is obtained in the following way.

```
DifferenceEquations1D[equations,
 GetDiffMethod1D[variables, methods, "diffmethod"],
 GetDiffMethod1D[variables, methods, "xapproxorder"],
 GetDiffMethod1D[variables, methods, "tapproxorder"], h, k]
```

$$\left\{u[x, 1+t] \to \frac{k\,u[-1+x, t] + (h^2 - 2k)\,u[x, t] + k\,u[1+x, t]}{h^2}\right\}$$

Here, *h* and *k* are the discrete step sizes for *x* and *t*, respectively. The function **setupproblem** generates the iteration function **solvePDE** with the assignment statements, and also extracts all the global variables from the input lists. In the following, *dvl* is the number of dependent variables.

```
setupproblem[equations, initials, boundaries, geometries,
variables, methods]
```

```
{HoldPattern[solvePDE[nx$_, nt$_, InteriorBelow$_,
    InteriorLeft$_, InteriorRight$_, dvl$_]] :>
  Module[{Integer j$, Integer n$},
   (U[[1 | dvl$, 1 | nx$, 1 | nt$]] = 0; For[j$ = 1, j$ < nx$ + 1,
     j$++, U[[1, j$, 1]] = If[0.00502513 (-1. + j$) < 0.5,
       0.00502513 (-1. + j$), 1. - 1. (0.00502513 (-1. + j$))]];
    For[n$ = 1, n$ < nt$ + 1, n$++, U[[1, 1, n$]] = 0];
    For[n$ = 1, n$ < nt$ + 1, n$++, U[[1, 200, n$]] = 0];
    For[n$ = InteriorBelow$,
     n$ < nt$ + 1, n$++, For[j$ = InteriorLeft$,
      j$ < InteriorRight$ + 1, j$++, U[[1, j$, n$]] =
       0.440011 U[[1, -1 + j$, -1 + n$]] + 0.119979 U[[1, j$, -1 + n$]] +
        0.440011 U[[1, 1 + j$, -1 + n$]]]);]}
```

This completes the symbolic part of our package. The time taken to run the iteration function **solvePDE** increases with the size of the problem. The following are the results of running the iteration function first on *Mathematica* directly, and then using the generated C++ code from the MathCode compiler. In the following, the function AbsTime measures the time required to run the function which is given as its argument; *timem* and *timemcode* are the times taken to run within *Mathematica*, and the generated C++ code, respectively.

Here is the *Mathematica* run:

```
timem = AbsTime[solvePDE[nx, nt, InteriorBelow, InteriorLeft,
   InteriorRight, dvl];]
```

```
{200.00000 Second, Null}
```

Now we run the iteration function **solvePDE** within MathCode by executing the following sequence of commands:

```
CompilePackage["PDESolver'"]
```

```
Successful compilation to C++: 3 function(s)
```

```
MakeBinary[]
```

```
InstallCode["PDESolver'"]
```

```
PDESolver is installed.
```

```
LinkObject['./PDESolverml.exe', 3, 2]
```

```
timemcode = AbsTime[
  Table[solvePDE[nx, nt, InteriorBelow,
    InteriorLeft, InteriorRight, dvl], {i, 1, 100}];
 ]
```

```
{22.00000 Second, Null}
```

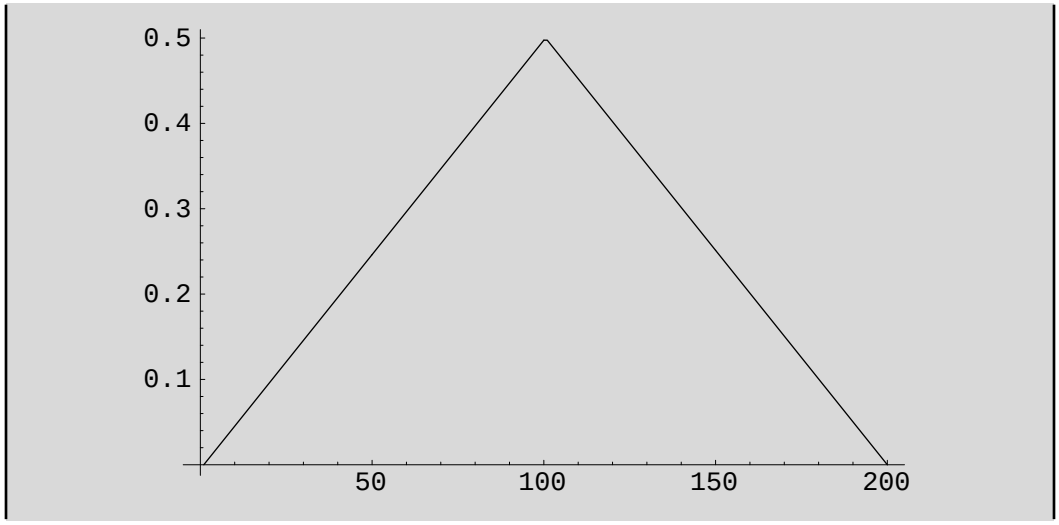
```
timelist={nx,nt,timem[[1]], timemcode[[1]],
100*timem[[1]]/timemcode[[1]]}
```

```
{200, 500, 200.00000 Second, 22.00000 Second, 909.091}
```

In the MathCode part, we run the function 100 times since the time taken for a single run is too small for the time measurement to be accurate. We therefore have to divide the above time by 100 to compare it with the time taken for the *Mathematica* run. We find that for this case the generated C++ code runs about 900 times faster.

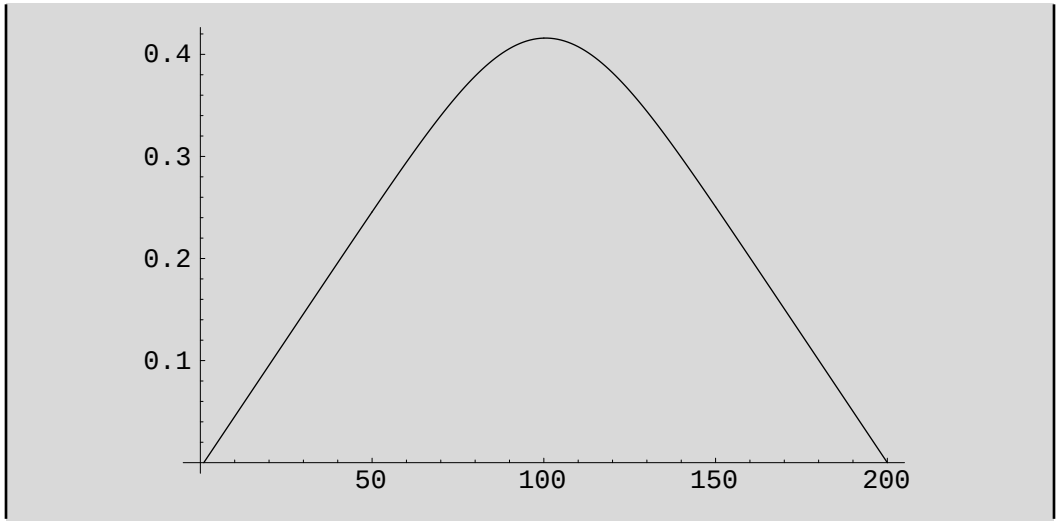
We show below the results from the above runs. For this purpose, we have defined a function **tabmcode** (which is compiled alongwith **solvePDE**) which contains the solution vector for the *i*th dependent variable for the time slice *t*. We can directly plot this vector using the function **ListPlot**. Here is the plot of the initial vector:

```
ListPlot[tabmcode[1,1,nx],PlotJoined->True];
```



Here is the plot of the solution vector for *t=nt*.

```
ListPlot[tabmcode[1,nt,nx],PlotJoined->True];
```



5. Conclusions

As we remarked in the Introduction, any PDE solver can only be incomplete, and the question really is one of the degree of completeness. Our PDE solver is incomplete at present in that it supports only explicit finite−difference schemes and is restricted to one spatial dimension and time. On the other hand, it has some definite advantages. Firstly, it is possible for the user to specify the approximation method, and the solver can transform the PDE based on that. Secondly, using the code generator , we are able to get C++ code that runs several orders faster than the *Mathematica* code. Thirdly, it is possible, in principle, to generate any finite−difference scheme (not necessarily explicit ones) using the functions that our package provides, although we have restricted ourselves to explicit schemes in our numerical computations. As a result, our package provides the user with a high−level interface to specify the PDE problem and the solution technique and generates an efficient C++ code for performing the numerical computation.

In the future, we plan to work along two directions. Our initial motivation for developing a PDE solver was to provide PDE support for the Modelica language [Mode99, Pelab99] within the *Mathematica* environment. Our package 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 [MathMod98]. The second direction of work will focus on developing a PDE *analyzer*, which will perform some preliminary analysis of the PDE problem and select a suitable approximation method by making a pattern matching with a library of PDEs and solution techniques. The analyzer will then employ a *synthesizer*, which will actually perform the numerical computation, and can be any one of the available solvers. In case no available solver suits the suggested technique, the present model of code generation can be employed, thus providing a PDE solver with a high degree of completeness.

References

[Ames92] Willam F. Ames, *Numerical Methods for Partial Differential Equations,* Academic Press, 1992.

[Fritz98] Peter Fritzson, *MathCode C++*, published by MathCore (www.mathcore.com), 1998.

[Ganza96] Victor G. Ganzha and Evgenii V. Vorozhtsov, *Numerical Solutions for Partial Differential Equations: Problems Solving Using Mathematica*, CRC Press, 1996.

[Gust95] Bertil Gustafsson, Heinz−Otto Kreiss and Joseph Oliger, *Time Dependent Problems and Difference Methods*, John Wiley & Sons, Inc., 1995.

[MathMod98] Peter 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.

[MinOh95] Min Oh, *Modelling and simulation of combined lumped and distributed processes*, Ph.D thesis, University of London, 1995.

[Mode99] Modelica homepage (www.modelica.org).

[Pelab99] PELAB − Programming Environment Laboratory, Department of Computer Science, Linköping University, Sweden (www.ida.liu.se/~pelab/modelica).