Generating Spectral Method Solvers for Partial Differential Equations

Gal Berkooz
Paul Chew
Jim Cremer
Rick Palmer
Richard Zippel

TR 92-1308
October 1992

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501
Generating Spectral Method Solvers for Partial Differential Equations

Gal Berkooz, Paul Chew, Jim Cremer, Rick Palmer and Richard Zippel
Cornell University
Ithaca, NY 14853

October 22, 1992

Abstract

A major cost in scientific computing is the creation of software that performs the numerical computations. This paper presents preliminary results on research to build a framework for automating the construction of numerical solvers for differential equations. Within this framework, the scientific computing problem is described using a very high level programming language that captures the original differential equations in a natural fashion. A sequence of code "transformers" are used to gradually refine the high level description of the problem into a concrete, executable form. Numerical techniques like the finite element method, the spectral method and the Crank-Nicolson discretization scheme are encoded in these transformers and once so encoded can be applied to a wide variety of different problems. This framework provides a natural environment for coarse scale parallelization based on relatively abstract properties of the specific equations and methods.

1 Introduction

One of the most difficult aspects of large scale scientific computations is generating the programs that perform the numerical computations. These programs often involve extensive, intricate mathematical computations whose coding can be quite error prone. In certain domains, such as turbulent fluid flow, it can take years to write efficient scientific software. These problems are exacerbated when parallelism is involved. This problem has been recognized by a number of authors [1, 2, 4, 6–8, 10–12].

We are developing a framework to automate the generation of this type of numerical software. Our approach translates the problem of code generation into one of program transformations. This enables us to use existing programming language, program transformation and compiler technology. The intermediate structures produced are combinations of mathematical statements and programming constructs. These novel structures allow us to incorporate deductive reasoning and type inferencing technology to ensure the resulting numerical programs are correct. As with many of the previous approaches we also use symbolic computing, although in a slightly different fashion.

2 Burgers' Equation

To illustrate our approach, we consider the following very simple example; a driven version of Burgers' equation:

\[ \frac{\partial u}{\partial t} + u \frac{\partial u}{\partial x} = \frac{\partial^2 u}{\partial x^2} + f(x, t), \]  

(1)
Burgers1 := 
declare u(\cdot, \cdot) in Hilb([0, T] \times D \to \mathbb{R});
solve(\forall z \in D. \{ u(0, x) = u_0(x) \},
\forall z \in D. \{ u(0, x) \});

\begin{align*}
\text{solved}(\forall \{ t \in (0, T) \} \land (z \in D). \{ \frac{\partial u(t, x)}{\partial t} + u(t, x) \frac{\partial u(t, x)}{\partial x} = \frac{\partial^2 u(t, x)}{\partial x^2} + f(x, t) \},
\forall \{ t \in (0, T) \} \land (z \in D). \{ u(t, x) \});
\end{align*}

Figure 1: Original program for Burgers' equation

where \( f(x, t) \) is a driving force (provided numerically) and where \( u \) satisfies the periodic boundary condition \( u(t, x) = u(t, x + L) \) and the initial condition \( u(0, x) = u_0(x) \).

We want to translate this mathematical problem into an executable program that solves Burgers' equation using a "pseudo-spectral co-location method" [5]. Several important issues are stated only implicitly in the mathematical formulation of the problem. To create a computer program for the problem these issues must be made explicit. The most important of the unstated issues is the "type" of \( u \): \( u \) is a real valued function of time and space. To capture the periodic boundary condition \( u(t, x) = u(t, x + L) \), we make the spatial domain be the periodic interval \([0, L]\), which we denote by \( D = \text{Per}(0, L) \). Thus \( u \) is an element of the space of twice differentiable functions:

\[ u \in \text{Hilb}([0, T] \times D \to \mathbb{R}). \tag{2} \]

We call this function space the domain of \( u \).

The top half of Figure 1 gives a graphical representation of these domains. We find that this graphical representation gives the clearest image of the progress of the code transformations and will most likely be central to the user interface of the code generation system we are building.

The original differential equation problem (1) is given in the bottom half of Figure 1 in a readable version of the data structures that our system manipulates. To distinguish the programming fragments (which are executable) and mathematical expressions (which need further conversion), the typewriter font is used for all programming expressions.

The declare statement indicates the mathematical type of \( u \), i.e., the domain of which \( u \) is a member. In this statement we have used a dot (\( \cdot \)) to indicate a variable that we do not wish to name, but that is required by conventional mathematical notation. In this case it would be more natural to use the syntax of (2), but an additional statement would be needed to indicate where to place the arguments of \( u \). Notice that this declaration provides information about the analytic character of \( u \).

The "for all" (\( \forall \)) statements are to be interpreted as mathematical expressions that represent sets. Thus,

\[ \forall \{ t \in (0, T) \} \land (x \in D). \{ \frac{\partial u(t, x)}{\partial t} + u(t, x) \frac{\partial u(t, x)}{\partial x} = \frac{\partial^2 u(t, x)}{\partial x^2} + f(x, t) \}, \]

is interpreted as a set of "incremental constraints" on \( u \) that hold at each point in \((0, T) \times D\). Notice that the mathematical types of \( t \) and \( x \) may be determined from their domains \((0, T)\) and
Burgers1 := 
\{ 
  declare u(.,.) \in \text{Hüb}(0, T) \times D \rightarrow \mathbb{R};
  solve(\forall z \in D . \{ u(0, z) = u_0(z) \}) ,
    \forall z \in D . \{ u(0, z) \}) ;
  solve(\forall (t \in (0, T]) \land (z \in D) . \{ \frac{\partial u(t, z)}{\partial t} + u(t, z) \frac{\partial u(t, z)}{\partial x} = \frac{\partial^2 u(t, z)}{\partial x^2} + f(x, t) \}) ,
    \forall (t \in (0, T]) \land (z \in D) . \{ u(t, z) \}) ;
  \forall (t \in [0, T]) \land (z \in D) . \{ \int_0^L u(t, z)u(t, z + y) dy \}) ;
\}

Figure 2: Burgers' equation with Autocorrelation Computation

$D$ respectively. The occurrence of $t$ and $x$ in the denominator of the derivatives is not a reference to points in $(0, T]$ and $D$, but rather to directions. This is an issue that conventional mathematical notation obscures.

The solve function takes two arguments. The first is a set of equations (or constraints) and the second is a set of quantities to be determined. The statement

solve(\forall (t \in (0, T]) \land (z \in D) . \{ \frac{\partial u(t, z)}{\partial t} + u(t, z) \frac{\partial u(t, z)}{\partial x} = \frac{\partial^2 u(t, z)}{\partial x^2} + f(x, t) \}) ,
\forall (t \in (0, T]) \land (z \in D) . \{ u(t, z) \}) ;

indicates that we are to determine the values of $u(t, z)$ for all $z \in D$ and for all time, $t \in (0, T]$, subject to constraint (1). The first solve expression in Burgers1 is easily converted to an assignment statement later.

It is not always the case when solving a differential equation that the solution is the final goal. Sometimes, additional computations are done with the solution as shown in Figure 2, where we are interested in the spatial autocorrelation of the solution of Burgers' equation. An additional advantage of our requirement that the problem be stated in a high level programming language is that discretizations of the PDE's will also discretize the additional side computations.

The "program" given in Figure 1 contains a complete description of the desired computation. It accurately and succinctly conveys the desires of the mathematician. What it does not contain is a specification of how to accomplish the computation. Our goal is to convert this mathematically intuitive "program" into an executable code. In spirit, our approach is similar to that of a compiler, however there are differences that will become apparent as we proceed.

The most important difference is that unlike a compiler, we expect the programmer to indicate which techniques should be used to convert the mathematical program into a more concrete, executable one. The techniques we are currently considering can be viewed as successively discretizing the function space of $u$. Initially the domain of $u$ consists of real valued functions on a continuous space $([0, T] \times D)$ and as each dimension is discretized, the domain of the function becomes continuous in fewer dimensions until ultimately, the domain of $u$ is a discrete set of points. At this stage, $u$ can be viewed as an array of values (one for each point in the domain of $u$) and the constraints are just numerical equations that are to be solved.

The discretization process is specified by the user who indicates which numerical method is to be used in each dimension. The system then applies the numerical method to the "program" to produce a new program with fewer continuous variables. As each discretization method is applied the program becomes less mathematical and more concrete, until finally we are left with a conventional numerical program.
Burgers2 :=

\begin{align*}
\text{declare } u(t) & \in \text{Hilb}(\{0, \ldots, T/\Delta t\} \times D \rightarrow \mathbb{R}); \\
\forall x \in D. \{u^{(n)}(x) - u_0(x)\}; \\
\text{for } n \in \{1, 2, \ldots, T/\Delta t\} \text{ do } \{ \\
\text{solve}(\forall x \in D. \{ \\
\frac{u^{(n+1)} - u^{(n)}}{\Delta t} = \frac{1}{2} \left[ \frac{\partial^2 u^{(n+1)}}{\partial x^2} + \frac{\partial^2 u^{(n)}}{\partial x^2} + f(x, (n+1)\Delta t) + f(x, n\Delta t) \right] \\
- \frac{1}{2} \left[ 3u^{(n)} \frac{\partial u^{(n)}}{\partial x} - u^{(n-1)} \frac{\partial u^{(n-1)}}{\partial x} \right] \\
\}, \\
\forall x \in D. \{u^{(n+1)}(x)\}; \}
\end{align*}

Figure 3: Time discretized program

For this simple, one dimension problem four steps are taken to generate the concrete program. First, time is discretized into equal sized steps. Effectively, this converts \( u(t, x) \) into a set of functions \( u^{(n)}(x) = u(n\Delta t, x) \). Second, \( u^{(n)}(x) \) is expanded in an infinite Fourier series. This introduces a dual Hilbert space of functions on frequencies. Due to the non-linearity of the equation, the transformed program includes variables defined over both spatial and frequency domains. Third, the Fourier series needs to be truncated and the Fourier transform converted a discrete Fourier transform. At this point, the constraints are discrete, but are still expressed as equality constraints and not assignment statements. Finally, the concrete code is generated and fast Fourier transforms introduced. In the following presentation we have indicated the beginning of the discussion of each step in bold face.

In this example, some code motion and and optimization is performed between the second and third steps. In general, optimizations, parallelizations and more sophisticated mathematical transformations could be performed throughout the process. This should be contrasted with the standard approaches taken by compilers, which only have access to the final code.

**Step 1** For this example, the time interval is divided into equal steps of length \( \Delta t \). This converts the continuous interval \([0, T]\) into a set of the discrete time values \(\{1, 2, \ldots, T/\Delta t\}\). This effect is graphically represented in the passage from Figure 1 to Figure 3. Denoting the time discretized version of \( u \) by \( u_{\text{time}} \), we have

\[ u_{\text{time}} \in \text{Hilb}(\{0, \ldots, T/\Delta t\} \times D \rightarrow \mathbb{R}). \]

It is conventional to write \( u_{\text{time}}(n, x) \) as \( u^{(n)}(x) \).

The effect of this transformation on the code is relatively straightforward. Instead of having a continuous set of constraints on \( u(t, x) \) for all \( t \in (0, T] \), we have a discrete set of constraints on \( u^{(n)}(x) \).

The differential equation itself is transformed by a combination of two different techniques.
First, the \( u_t \) term is discretized as
\[
\frac{\partial u(t, x)}{\partial t} \rightarrow \frac{u^{(n+1)}(x) - u^{(n)}(x)}{\Delta t}.
\]

The linear terms are then transformed by the Crank-Nicolson approach:
\[
g(u(t, x)) \rightarrow \frac{g(u^{(n+1)}(x)) + g(u^{(n)}(x))}{2},
\]
while a second order Adams-Bashforth technique is used for the non-linear terms:
\[
h(u(t, x)) \rightarrow \frac{1}{2} \left[ 3h(u^{(n)}(x)) - h(u^{(n-1)}(x)) \right].
\]

Applying these transformations to the previous code module gives the code in the bottom half of Figure 3.

The solve statement used to express the initial conditions has been converted to an assignment statement (the \( \leftarrow \) is used for assignment to distinguish it from the equality constraint used in the solve statement). Notice that the solve statement has been moved inside the loop over the discrete time values. That is, we have used a serializing transformation that converts
\[
\text{solve}(\forall n \in \{1, 2, \ldots, T/\Delta t\}) \land (x \in D). \{\ldots\}
\]
to
\[
\text{for } n \text{ in } \{1, 2, \ldots, T/\Delta t\} \text{ do } \{ \\
\quad \text{solve}(\forall x \in D. \{\ldots\}) \\
\}
\]

This transformation indicates that a parallel system of equations can be solved one at a time. Needless to say, this transformation is not always valid, though when it is valid can be determined automatically. In this particular case, the serialization transformation is a component of the time discretization method.

**Step 2** The next step is to expand \( u^{(n)}(x) \) as a Fourier series with unknown coefficients. When Burgers2 is transformed in this fashion, the new program will determine the coefficients of the Fourier expansion of \( u^{(n)}(x) \) instead of solving for \( u^{(n)}(x) \). Since each \( u^{(n)}(x) \) is periodic (\( D \) is periodic), by Fourier's theorem, we can write
\[
\begin{align*}
\frac{u^{(n)}(x)}{2\pi} & = \sum_{k=-\infty}^{\infty} \tilde{u}_k^{(n)} e^{2\pi i k x / L}, \\
\end{align*}
\]
where
\[
\tilde{u}_k^{(n)} = \frac{1}{L} \int_0^L u^{(n)}(x) e^{2\pi i k x / L} \, dx.
\]

We denote the relationship between \( u^{(n)}(x) \) and \( \tilde{u}_k^{(n)} \) by
\[
\tilde{u}_k^{(n)} = \mathcal{F}_k(u^{(n)}(x)).
\]
Burgers3 := |
declare \( u^{(\cdot)}(\cdot) \in \) Hilb\(\{0,\ldots,T/\Delta t\} \times D \to \mathbb{R}\);
declare \( \bar{u}^{(\cdot)}(\cdot) \in \) Hilb\(\{0,\ldots,T/\Delta t\} \times \hat{D} \to \mathbb{R}\);
\(\forall x \in D, \{u^{(0)}(x) \to u_0(x)\}\);
\(\forall k \in \{0,1,2,\ldots\}, \{\bar{u}_k^{(0)} \mapsto \mathcal{F}_k(\forall x \in D u_0(x))\}\);
for \(n\) in \(\{1,2,\ldots,T/\Delta t\}\) do |
\(\text{solve}(\forall k \in \{0,1,2,\ldots\}, \{\)
\(\frac{1}{\Delta t} + \frac{2x^2k^2}{L^2} \bar{u}_k^{(n+1)} = \frac{1}{2} \left[ \frac{2}{\Delta t} \bar{u}_k^{(n)} - \frac{2x^2k^2}{L^2} \bar{u}_k^{(n)} + \mathcal{F}_k(f(x,(n+1)\Delta t) + f(x,n\Delta t)) \right] \)
\(- \frac{3}{2} \mathcal{F}_k \left\{ u^{(n)}(x) \frac{\partial u^{(n)}(x)}{\partial x} \right\} + \frac{1}{2} \mathcal{F}_k \left\{ u^{(n-1)}(x) \frac{\partial u^{(n-1)}(x)}{\partial x} \right\} \}
\},
\(\forall k \in \hat{D}, \{\bar{u}_k^{(n+1)}\}\);
\(u^{(n+1)}(x) = \mathcal{F}^{-1}(\forall k \in \{0,1,2,\ldots\}, \bar{u}_k^{(n+1)}\}\);
\}

\textbf{Figure 4: Fourier transformed program}

The effect of the Fourier expansion is to introduce a frequency domain \(\hat{D}\) that is dual to the spatial domain \(D\). The element of \(\hat{D}\) are discrete frequencies,

\[\hat{D} = \mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, 3, \ldots\}.\]

The Fourier transform \(\mathcal{F}\) is a functor between the two Hilbert spaces

\[
\begin{align*}
\text{Hilb}(D \to \mathbb{R}) & \quad \xrightarrow{\text{Hilb}(\hat{D} \to \mathbb{R})} \\
x \mapsto u^{(n)}(x) & \quad \mapsto k \mapsto \mathcal{F}_k\{u^{(n)}(x)\}.
\end{align*}
\]

Since \(u^{(n)}(x)\) is a real valued function, the Fourier coefficients are symmetric, i.e., \(\bar{u}_k^{(n)} = \bar{u}_{-k}^{(n)}\).

Thus we only need to determine the positive Fourier coefficients and we can restrict \(\hat{D}\) to the non-negative integers as is shown in Figure 4.\(^1\)

To apply the Fourier series technique, equations involving \(u^{(n)}(x)\) are converted, via (3), into equations in \(\bar{u}_k^{(n)}\) by equating coefficients in the exponentials.\(^2\) The only part of the transformation

\(^1\)This is a good example of where deductive reasoning can be used to advantage.

\(^2\)More accurately, a Galerkin projection is applied, but since the exponentials are orthogonal this is equivalent to equating the exponentials.
that needs to be developed is that for $u_{xx}$. Taking the second derivative of (3) gives

$$ \frac{\partial^2 u^{(n)}(x)}{\partial x^2} = \sum_{k=-\infty}^{\infty} \tilde{u}_k^{(n)} \left( \frac{2\pi ik}{L} \right)^2 e^{2\pi ikx/L}. $$

Applying this transformation to the main equation in Burgers2 gives

$$ \frac{u_k^{(n+1)} - u_k^{(n)}}{\Delta t} = \frac{1}{2} \left[ \left( \frac{2\pi ik}{L} \right)^2 \tilde{u}_k^{(n+1)} + \left( \frac{2\pi ik}{L} \right)^2 \tilde{u}_k^{(n)} + \mathcal{F}_k \{ f(x, (n + 1)\Delta t) \} + \mathcal{F}_k \{ f(x, n\Delta t) \} \right] $$

$$ - \frac{3}{2} \mathcal{F}_k \left\{ u^{(n)}(x) \frac{\partial u^{(n)}(x)}{\partial x} \right\} + \frac{1}{2} \mathcal{F}_k \left\{ u^{(n-1)}(x) \frac{\partial u^{(n-1)}(x)}{\partial x} \right\} $$

where $\mathcal{F}_k \{ g(x) \}$ indicates the $k^{th}$ coefficient of the Fourier expansion of $g(x)$.

At this point it is also appropriate to isolate the $u_k^{(n+1)}$ terms on the left hand side of the equation using symbolic techniques. Notice that an inverse Fourier transform needed to be added to the end of Burgers3 to ensure that the behavior of the program was unchanged from Burgers2. While a formula like (3) could have been used here, it seems preferable to leave the $\mathcal{F}^{-1}$ thus preserving the intent of the operation for future optimizations.

A certain amount of optimization, common subexpression elimination and strength reduction can be accomplished at this stage. Because of the compactness of the code in Burgers3, it is significantly easier to perform these optimizations now than waiting until the (much larger) executable code is available. The latter approach is used by compiler oriented optimizers since they have no other choice.

It is easy to recognize that the terms $\mathcal{F}_k \{ u^{(n)} u_x^{(n)} \}$ and $\mathcal{F}_k \{ u^{(n-1)} u_x^{(n-1)} \}$ are closely related and only one needs to be computed at each time step. The representation we use for the intermediate code allows such optimizations to be efficiently detected. This optimization is particularly valuable for problems involving more complicated PDE's and for producing code to be run on a parallel machine.

The particular optimization we want to apply depends upon recognizing structures like

```plaintext
for n in ... do {
    ... f(n) ... f(n+1)
}
```

where $f$ is a computationally expensive expression. When constructs like this are encountered, the repeated computations are cached in other variables as follows

```plaintext
f_0 ← f(0);
for n in ... do {
    f_{n+1} ← f(n+1);
    ... f_n ... f_{n+1}
}
```

When applied to Burgers3, this technique yields the code in Figure 5.
Burgers3b := {
dec {u^{n}(\cdot)} \in \mathcal{H}ib((0,\ldots,T) \times D \rightarrow \mathbb{R});
dec {\phi^{n}(\cdot)} \in \mathcal{H}ib((0,\ldots,T) \times \hat{D} \rightarrow \mathbb{R});
dec \hat{f}(\cdot,\cdot), \hat{\phi}(\cdot,\cdot) \in \mathcal{H}ib((0,\ldots,T) \times \hat{D} \rightarrow \mathbb{R});
\forall x \in D. \{u^{0}(x) = u_{0}(x)\};
\forall k \in \{0,1,2,\ldots\}. \{u^{(n)}_{k} = \mathcal{F}_{k} \{\forall x \in D. u_{0}(x)\}\};
for n in \{1,2,\ldots,T/\Delta t\} do {
\forall k \in \{0,1,2,\ldots\}. \{\hat{f}(n,k) = \mathcal{F}_{k} \{f(x,n\Delta t)\}\};
\forall k \in \{0,1,2,\ldots\}. \{\hat{\phi}(n,k) = \mathcal{F}_{k} \{u^{(n)}(x) \cdot \partial u^{(n)}(x)/\partial x\}\};
\text{solve}(\forall k \in \{0,1,2,\ldots\}.
\left(\frac{1}{\Delta t} + \frac{2\pi^{2}k^{2}}{L^{2}}\right)\hat{u}^{(n+1)}_{k} = \frac{1}{\Delta t}\hat{u}^{(n)}_{k} - \frac{2\pi^{2}k^{2}}{L^{2}}\hat{u}^{(n)}_{k} + \frac{1}{2}\hat{f}(n+1,k) + \frac{1}{2}\hat{f}(n,k) - \frac{3}{2}\hat{\phi}(n,k) + \frac{1}{2}\hat{\phi}(n-1,k))
}.
\forall k \in \{0,1,2,\ldots\}. \{u^{(n+1)}_{k}\};
\text{undo}(\forall k \in \{0,1,2,\ldots\}. \{u^{(n+1)}_{k}\}\};
}\}

Figure 5: Optimized, Fourier transformed program

**Step 3** At this point the constraints are no longer continuous. The number of equations and unknowns, however, is still infinite. The next step is to truncate the spectrum, which induces a discretization on $D$ as shown in Figure 6.

The most interesting part of this transformation is the effect on the line

$\forall k \in \{0,1,2,\ldots\}. \{\hat{\phi}(n,k) = \mathcal{F}_{k} \{u^{(n)}(x) \cdot u_{x}^{(n)}(x)\}\}$

where the $x$ subscript indicates a partial derivative in the $x$ direction. No ordering of the assignments is implied by this statement. When the frequency spectrum is truncated at $K$ modes, we effectively have

$$
\begin{align*}
\begin{bmatrix}
\hat{\phi}(n,0) \\
\hat{\phi}(n,1) \\
\vdots \\
\hat{\phi}(n,K-1)
\end{bmatrix} & \Rightarrow
\begin{bmatrix}
\mathcal{F}_{0} \{u^{(n)}(x) \cdot u_{x}^{(n)}(x)\} \\
\mathcal{F}_{1} \{u^{(n)}(x) \cdot u_{x}^{(n)}(x)\} \\
\vdots \\
\mathcal{F}_{K-1} \{u^{(n)}(x) \cdot u_{x}^{(n)}(x)\}
\end{bmatrix}
\end{align*}
$$

The fast Fourier transform algorithm allows us to compute all of the Fourier transforms on the right hand side sharing a great number of common computations. Graphically, this transformation
Burgers 4 := 
\[
\begin{align*}
\{0, \ldots, T/\Delta t\} \times D &= \text{Per}(0, L) \\
\{0, \ldots, T/\Delta t\} \times \tilde{D} &= \{0, 1, \ldots, K - 1\} \to \mathbb{R}
\end{align*}
\]

\[
\begin{align*}
\text{for } n \in \{1, 2, \ldots, T/\Delta t\} \text{ do } \\
\quad \text{for } \ell \in \{0, 1, \ldots, K - 1\} \text{ do } \\
\quad \quad \text{temp}[\ell] \leftarrow f(\ell L/K, n \Delta t); \\
\quad \quad \tilde{f}(n, \cdot) \leftarrow \mathcal{F}\{\text{temp}[\cdot]\}; \\
\quad \quad \text{for } \ell \in \{0, 1, \ldots, K - 1\} \text{ do } \\
\quad \quad \quad \text{temp}[\ell] \leftarrow (2\pi i \ell/L) \cdot \tilde{u}^{(n)}_{k}; \\
\quad \quad \quad \text{tempb}[\ell] \leftarrow \mathcal{F}^{-1}\{\text{temp}[\cdot]\}; \\
\quad \quad \quad \tilde{\phi}(n, \cdot) \leftarrow \mathcal{F}\{\text{tempb}[\cdot]\}; \\
\quad \quad \text{for } k \in \{0, \ldots, K - 1\} \text{ do } \\
\quad \quad \quad \text{solve} & \left\{ \begin{array}{l}
\frac{1}{\Delta t} \tilde{u}^{(n+1)}_{k} = \frac{2\pi^2 k^2}{L^2} \tilde{u}^{(n)}_{k} + \frac{1}{2} \tilde{f}(n, k) + \frac{1}{2} \tilde{f}(n + 1, k) + \frac{1}{2} \tilde{f}(n, k) \\
-3/2 \phi(n, k) + \frac{1}{2} \phi(n - 1, k),
\end{array} \right. \\
\quad \quad \quad \tilde{u}^{(n+1)}_{k}; \\
\quad \quad \tilde{u}^{(n+1)}_{\ell} \leftarrow \mathcal{F}^{-1}\{\tilde{u}^{(n+1)}_{k}\}; \\
\end{align*}
\]

Figure 6: Spatially and temporally discretized program

can be expressed as
\[
\begin{align*}
\begin{pmatrix}
\tilde{\phi}(n, 0) \\
\tilde{\phi}(n, 1) \\
\vdots \\
\tilde{\phi}(n, K - 1)
\end{pmatrix} & \iff \begin{pmatrix}
u^{(n)}(0) \cdot u_{x}^{(n)}(0) \\
\vdots \end{pmatrix} \\
\begin{pmatrix}
u^{(n)}(1-K/L) \cdot u_{x}^{(n)}(1-K/L) \\
\vdots \end{pmatrix} \\
\begin{pmatrix}
u^{(n)}(K-1-K/L) \cdot u_{x}^{(n)}(K-1-K/L) \\
\vdots \end{pmatrix}
\end{align*}
\]

This expression would be linearly represented as
\[
\left( \forall k \in \{0, 1, \ldots, K - 1\}. \tilde{\phi}(n, k) \right) \iff \mathcal{F}\{\forall \ell \in \{0, 1, \ldots, K - 1\}. u^{(n)}(\ell / K L) \cdot \partial_{x}^{(n)}(\ell / K L)\}
\]

When this expression is converted to code we have
Burgers5 := { 
  declare u[1](·) ∈ H(0,...,T/Δt) × {0,...,K−1} → ℝ;
  declare u̇[1](·) ∈ H(0,...,T/Δt) × {0,...,K−1} → ℝ;
  declare f₀[·], f_n[·], phi₀[·], phi_n[·] ∈ H(0,...,2π(K−1)/L) → ℝ;
  for k in {0,...,K−1} do u[0](k) ← u₀(2πi(k−1)/(L));
  u̇[0] ← FFT_{K}(u₀);
  for n in {1,2,...,T/Δt} {
    f_n ← f_n;
    phi_n ← phi_n;
    for ℓ in {0,...,K−1} do temp[ℓ] ← f(ℓL/K,nΔt);
    f_n ← FFT_{K}(temp);
    for k in {0,...,K−1} do temp[k] ← (2πiK/L)u_n[k];
    temp[b] ← IFFT_{K}(temp);
    for ℓ in {0,...,K−1} do tempc[ℓ] ← u^{(n)}(ℓL/K) · temp[b];
    phi_n ← FFT_{K}(tempc);
    for k in {0,...,K−1} do
      u^{(n+1)}[k] ← \left( \frac{1}{Δt} + \frac{2πk^2}{L^2} \right)^{-1} \left[ \frac{1}{Δt} u^{(n)}[k] - \frac{2πk^2}{L^2} u^{(n)}[k] + \frac{f_n[k]}{2} + \frac{f_0[k]}{2} - \frac{3}{2}\phi_n(k) + \frac{phi_0(k)}{2} \right];
    u^{(n+1)} ← IFFT_{K}(u^{(n+1)});
  }
}

Figure 7: Final Program

for ℓ ∈ {0,1,...,K−1} do tempc[ℓ] ← u^{(n)}(ℓL/K) · u^{(n)}_x(ℓL/K);
φ(n,·) ← S{tempc[·]}

(This is all that needs to be done to generate the code that computes f(n,·).)

Next, the computation of the derivatives in the spatial domain is converted to a computation in the frequency domain using Fourier transforms.

for ℓ ∈ {0,1,...,K−1} do tempa[ℓ] ← (2πiℓ/L) · u^{(n)}_x;
temp[b] ← S^{-1}\{tempa[·]\};
for ℓ ∈ {0,1,...,K−1} do tempc[ℓ] ← u^{(n)}(ℓL/K) · temp[b];
φ(n,·) ← S\{tempc[·]\}

The bottom half of Figure 6 shows the result of truncating the infinite Fourier series in Burgers3b. In general, the system of equations inside the solve could involve any sort of coupling. In this case, the transformation system recognizes that the system of equations to be solved is diagonal and merely moves the loop over k outside of the solve statement.

In the more general case, the transformation system could decide which sort of linear equation solver to use—general, tridiagonal, etc. Since the linear equations are known explicitly at this point, the system could also decide precisely what type of data storage mechanisms ought to be used.

Step 4 The program Burgers4 is nearly executable. The only major step required to turn it into an executable program is to convert the solve statement into an assignment. This can be done symbolically, since the equation is linear. In addition the Fourier transform can be replaced by a call to a specific fast Fourier transform routine since the number of points is now known.
In addition, the optimizer has recognized that it is not necessary to preserve the values of \( f \) and \( \phi \) for all of time, only two different values of each are needed. Thus what might have been two dimensional arrays \( f \) and \( \phi \) are instead \( f_0 \) (old) and \( f_n \) (new) for \( f \) and \( \phi_0 \) and \( \phi_n \) for \( \phi \).

It should be noted that in higher dimensional (more realistic) problems additional discretization steps will be required and more PDE's would be involved leading to significantly more complex codes. Some of the optimizations we have performed would be much more difficult to perform with the final codes than with the intermediate structures we have used.

3 Implementation Technique

The set of transformations described here requires a number of different facilities: source to source code transformations, symbolic computing and reasoning about domains. These disparate needs were met by implementing the transformations in Common Lisp and using the symbolic computing substrate Weyl [13]. The flexibility of the Common Lisp's macro language allowed us to try out a variety of programming notations quickly and its free syntax simplified the construction of code walkers and transformers.

The mathematical portion of the programs are represented using the internal representation system of Weyl, a computer algebra substrate we are developing. Unlike most computer algebra systems, one does not use Weyl via a special language. Instead, Weyl extends a high level programming language, in this case Common Lisp, to have symbolic computing facilities. Thus we did not have to sacrifice the advantages of Common Lisp for the computer algebra facilities. In addition, Weyl's domain theoretic organization provided the basis for the construction and manipulation of function spaces. In the future, we expect domains to play a role in determining which discretization methods are appropriate and gluing together solutions from different differential equations.

The transformations themselves were implemented in a rather ad hoc fashion. Representations were developed for the code modules, including the domains and any parameters that arise. As we gain more experience with the transformation structure we expect that certain idioms and patterns of use will arise that can be captured. However, the complexity of some of the transformations (e.g., the Crank-Nicolson/Adams-Bashforth transformation used for time) encouraged us to make a complete programming language available to the transform writer.

We would like the transformations used in this process to actually be “equivalence” transformations. For this to be the case, it would be necessary to include some error information. For instance, if a forward Euler time discretization approach were used, we would need to transform Burgers’ equation into

\[
\frac{u^{(n+1)}(x) - u^{(n)}(x)}{\Delta t} + E_n(x) + u^{(n)}(x) \cdot \frac{\partial u^{(n)}}{\partial x}(x) = \frac{\partial^2 u^{(n)}}{\partial x^2}(x) + f(x, n\Delta t),
\]

where \( E_n \) represents the discretization error. In the process of producing code, the \( E_n \) term would be dropped, but then it would be necessary to indicate how this changes what is computed. In general, maintaining the equivalence of programs is quite difficult.

If the original “program,” which is provided by the scientist or engineer is correct then the final program will also be correct. We found that the transform philosophy made it much easier to understand what the structure and role of the transformation modules should be. It also clarified the modularity of the transformations and indicated how the transformation modules could be reused and recombined for different problems.

In Section 2 three different transformers were discussed: (1) a uniform step size, Crank-Nicolson/Adams-Bashforth discretization method, (2) a Fourier transform method, (3) a uniform
stepsizes, continuous to discrete method. None of these methods was designed especially for Burgers’ equation. The same set of transformations has been successfully applied to the Boussinesq equation

$$\frac{\partial u}{\partial t} + 3 \frac{\partial^2 (u^2)}{\partial x^2} = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^4 u}{\partial x^4}$$

to produce a numerical solver. In addition these discretization techniques can be mixed and matched with other transformation methods to produce more sophisticated solvers. For instance, the hybrid Crank-Nicolson/Adams-Bashforth method could be combined with a wavelet based spatial discretization method, or the Fourier decomposition transformations could be combined with a Runge-Kutta time advancement strategy. We feel that the ability to reuse different transformation modules is a significant accomplishment of this work.

The language of Burgers should be viewed as a very high level scientific programming language. Thus far we have only used it to state problems of solving differential equations. It will be interesting to see how it evolves as it is used for problems whose final result is more complex—results that involve statistical averages or convolutions of numerical solutions, for instance.

4 Conclusions

We feel this research combines several key ideas, which are given below:

- High level programming language that mixes mathematical and programming constructs.
- Interpretation of PDE’s as the more computational concept of “continuous” constraints.
- Encapsulation of numerical techniques as code transformers.
- Use of domains as an organizing principle.
- Use of compiler optimization techniques on high level programming structures.

The most important idea in this work is the mixed mathematical/computational programming language. Its use of mathematical domains (Hilbert spaces) as a “type system” provides a powerful framework for describing scientific computations and their transformation into effective programs.

There are several major areas requiring future research. First, the mixed mathematical/programming language used throughout the code transformations needs to be formalized. The semantics and types of variables must be understood more thoroughly. In most earlier work, semanticists were free to design the programming language or correct language details that were at variance with their logical formalisms. The language we are using makes use of a fair amount of pre-existing mathematics, which does not necessarily match the current logical formalisms. Reconciling these two perspectives, the logical formalism of programming languages and the mathematical formalism of partial differential equations, appears to be challenging for both sides.

Second, we expect a large number of transformation methods to be created that discretize continuous mathematical programs. Each of these transformation methods should be able to decide whether or not they are applicable to a particular code module. This would allow the user to, somewhat imprecisely, specify how the code mode should be implemented, and the system would suggest only those transformation modules that are applicable. For example, the user might specify that a spectral method be used to solve a problem, but allow the system to decide what set of basis functions ought to be used given the boundary conditions. Or the user might let the system choose
the order the different discretizations to achieve a specified level overall accuracy. Also a theorem prover could help recognize that
\[ \int_0^L u(t, x)u(t, x + y)\,dy = |\bar{u}|^2. \]
thus simplifying the computation of the integral in Figure 2. To support these processes some deductive reasoning mechanism like Nuprl [3] or Ontic [9] is needed.

This work was supported in part by the Advanced Research Projects Agency of the Department of Defense under ONR contracts N00014–88–K–0591 and N00014–89–J–1946, NSF Grant IRI–9006137, Air Force Office of Sponsored Research grants AFOSR–91–0328 and AFOSR 49620–92–J–0287 and in part by the U.S. Army Research Office through the Mathematical Science Institute of Cornell University

References