SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2002; **00**:1–6 *Prepared using* speauth.cls *[Version: 2000/03/16 v2.12]*

A little language for modularizing numerical PDE solvers

Michael H. Hohn[†]

Department of Mathematics, University of Utah, Salt Lake City, Utah 84112, USA

SUMMARY

This paper describes a concise specification language for linear partial differential equations (PDEs) on a union of rectangles, along with three tools: a pretty-printer, TEX generator, and a code generator. The pretty-printer and TEX generator help users by allowing equations to be specified (and read) in their natural form, while the code generator allows implementors to separate their numerical solver from the input equations, and greatly simplifies testing.

KEY WORDS: domain-specific language; code generation; pretty-printing; partial differential equations

Introduction

In the implementation and use of numerical solvers for a general class of equations, one recurring problem is the transition from the current mathematical equations to an easily tested program, or the input to such a program. The purpose of this paper is to illustrate the derivation of a little language for a fairly broad class of problems, and demonstrate its utility via one of the simplest realistic sample problems for this class.

Among existing solvers, some provision is made to simplify equation input. In [3], this input takes the form of tables provided in the implementation language of the system, FORTRAN, while other solvers, e.g. [7], provide a programming API for specifying the equations and the method of solution; such an API, whether object-oriented or other, is equivalent to providing a table of functions and satisfies the demands of a numerical solver.

The fundamental difficulties with user-provided table input are correctness and legibility. Correctly providing hundreds of coefficient functions in a table form (or some equivalent) is tedious and difficult. While the original mathematical equations can be made as legible as desirable, by typesetting them,

[†]E-mail: hohn@math.utah.edu

Received 11 September 2002

the resulting table will bear no resemblance to these equations, and its legibility may be considered close to zero.

Using a customized little language is better for several reasons. The first is programmability; the input, using such a language, is just text and can be generated easily by other programs, including a GUI, and more importantly, computer algebra systems (CAS). Common equation parts can be factored using a CAS, or simply using a macro expander such as m4, further simplifying the input; such a language is human-readable and can be used to produce pretty-printed or typeset output, all code needed by the main solver, and even full diagrams of the input equations. This multi-use approach is one of the most useful aspects the intermediate language.

The idea of domain-specific languages is common in computer science, and this was the motivation for the present language. With the exception of numerical linear algebra languages, domain-specific languages seem to be a rarity in numerical analysis. In one of the oldest PDE packages [6] a small language addition was made which translates into FORTRAN. This latter translation uses a very simple language and directly produces method-specific output, without exposing intermediate representations for other uses. This approach is not flexible enough for the problems described here, and the mathematical problems dealt with are also simpler; the system itself only deals with scalar, continuous-coefficient equations, using specific finite-difference approximations. Nevertheless, this little language allows for very concise input of equations handled by the solver.

The rest of this paper proceeds as follows. First, the sample problem's equations are shown[*], followed by a more abstract description of their structure, and their use as input to a numerical solver. Following this, a short summary of the author's experience with a primitive prototype implementation of such a solver is given; this summary lists numerous problems encountered. Next, a closer look is given to the requirements of a numerical solver, which directly lead to the above mentioned table-driven approach. The use of tables removes some problems, but leaves many others. Then, the needs of users and the structure of the equations are considered, and used to derive the input language. This little language is declarative, contains all equations in full, and provides modularity not only for a numerical solver, but also for the input to such a solver; for illustration, this resultant solver and equation destructuring are shown last.

An illustrative mathematical problem

The problems handled here are linear, variable-coefficient elliptic PDEs defined on a finite union of rectangles[†]; for each rectangle, there are interior equations and boundary conditions. In the following, the focus is on a two-rectangle problem taken from elasticity; this problem is one of the simplest realistic examples, and serves to illustrate the language. Details can be found in [2].

For this sample problem typical boundary conditions have the forms

[*]In this paper, equations are only considered from a structural point of view; mathematical meaning is only mentioned when necessary for the language design.

[†]An extension for other classes of equations, and higher dimensions, would be straightforward, because only the syntax of equations matters here.

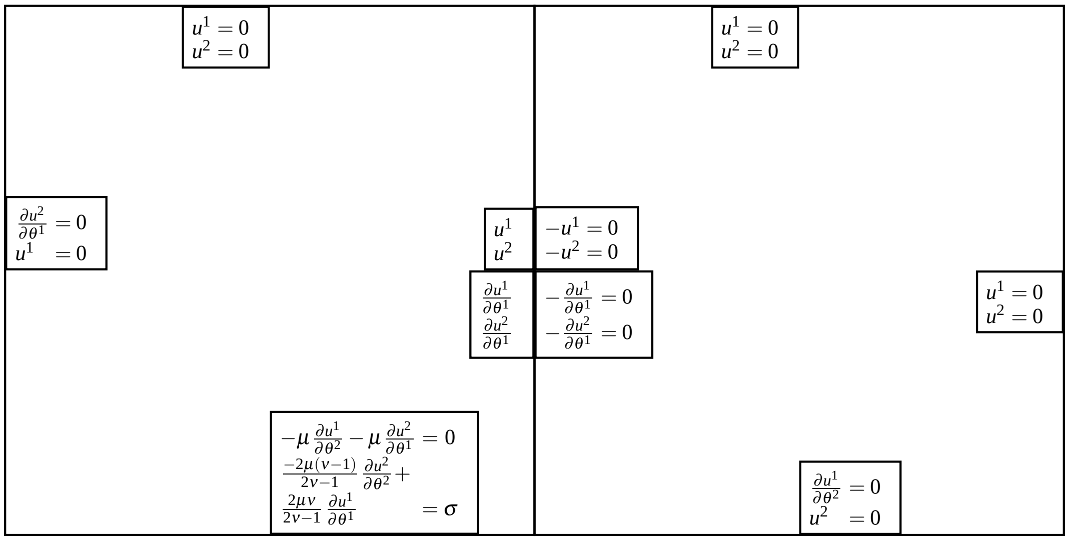
$u^1 = 0$, $u^2 = 0$
$u^1 = 0$, $u^2 = 0$
$\frac{\partial u^2}{\partial \theta^1} = 0$, $u^1 = 0$
u^1, u^2
$-u^1 = 0$, $-u^2 = 0$
$\frac{\partial u^1}{\partial \theta^1}$, $\frac{\partial u^2}{\partial \theta^1}$
$-\frac{\partial u^1}{\partial \theta^1} = 0$, $-\frac{\partial u^2}{\partial \theta^1} = 0$
$u^1 = 0$, $u^2 = 0$
$-\mu\frac{\partial u^1}{\partial \theta^2} - \mu\frac{\partial u^2}{\partial \theta^1} = 0$, $\frac{-2\mu(\nu-1)}{2\nu-1}\frac{\partial u^2}{\partial \theta^2} + \frac{2\mu\nu}{2\nu-1}\frac{\partial u^1}{\partial \theta^1} = \sigma$
$\frac{\partial u^1}{\partial \theta^2} = 0$, $u^2 = 0$

Figure 2. Boundary conditions for a simple but realistic two-rectangle problem. The main (interior) equations are shown in equation 3

$$\begin{aligned} -\mu\frac{\partial u^1}{\partial \theta^2} - \mu\frac{\partial u^2}{\partial \theta^1} &= 0, \\ \frac{-2\mu(\nu-1)}{2\nu-1}\frac{\partial u^2}{\partial \theta^2} + \frac{2\mu\nu}{2\nu-1}\frac{\partial u^1}{\partial \theta^1} &= \sigma, \\ u^1 &= 0, \\ \frac{\partial u^2}{\partial \theta^1} &= 0 \end{aligned} \tag{2}$$

as shown in figure 2, while the interior equations, as a second-order system, are

$$\begin{aligned} (-2\nu+2)\frac{\partial^2 \mathrm{u}^1}{(\partial\theta^1)^2} + (1-2\nu)\frac{\partial^2 \mathrm{u}^1}{(\partial\theta^2)^2} + \frac{\partial^2 \mathrm{u}^2}{\partial\theta^1\partial\theta^2} &= 0 \\ (1-2\nu)\frac{\partial^2 \mathrm{u}^2}{(\partial\theta^1)^2} + (-2\nu+2)\frac{\partial^2 \mathrm{u}^2}{(\partial\theta^2)^2} + \frac{\partial^2 \mathrm{u}^1}{\partial\theta^1\partial\theta^2} &= 0 \end{aligned} \tag{3}$$

For the author's numerical method, a first-order system is needed and these equations are rewritten as

$$\begin{aligned} (-2\nu+2)\frac{\partial u^{11}}{\partial\theta^1} + (1-2\nu)\frac{\partial u^{12}}{\partial\theta^2} + \frac{\partial u^{21}}{\partial\theta^2} &= 0, \\ (1-2\nu)\frac{\partial u^{21}}{\partial\theta^1} + (2-2\nu)\frac{\partial u^{22}}{\partial\theta^2} + \frac{\partial u^{11}}{\partial\theta^2} &= 0 \end{aligned} \tag{4}$$

by introducing the artificial unknowns u^{11}, u^{12}, u^{21} and u^{22} via

$$\begin{aligned}\frac{\partial u^1}{\partial \theta^1} - u^{11} = 0, \frac{\partial u^1}{\partial \theta^2} - u^{12} = 0,\\ \frac{\partial u^2}{\partial \theta^1} - u^{21} = 0, \frac{\partial u^2}{\partial \theta^2} - u^{22} = 0\end{aligned} \tag{5}$$

These latter equations are then also used on the boundaries; for flexibility, both first- and second-order systems are to be handled.

General mathematical structure

To get a more detailed idea of the allowable format, examine the structure of this problem inside-out; first consider a single term. Being linear variable-coefficient equations, the form of any left-hand-side term is one of

$$f(x,y)u^i \tag{6}$$

or

$$f(x,y)\frac{\partial u^i}{\partial \theta^j} \tag{7}$$

where $f(x,y)$ contains no u^i. The coefficient $f(x,y)$ must evaluate to a number, and so cannot contain symbolic operations such as integrals and derivatives; only the arithmetic operations of the target language and function calls are allowed. Thus, the $f(x,y)$ follows just a standard expression form as found in many programming languages. The unknown u^i can only appear by itself, or be differentiated in one or more directions.

Next, consider a single equation. The left-hand-side can only contain a sum or difference of such terms, and an unknown can appear multiple times with different operators, e.g., $\frac{\partial u^1}{\partial \theta^1} + \frac{\partial u^1}{\partial \theta^2}$, while a right-hand-side can contain no u^i at all and is thus a single expression.

For a collocation-based numerical method, a discrete linear system

$$A\mathbf{u} = \mathbf{b} \tag{8}$$

is set up and solved, where $\mathbf{u}$ contains values of the u^i at specific points. To make the connection between the equations and their linear system form, consider equations 4 written in a matrix form:

$$\begin{pmatrix} (-2\nu+2)\frac{\partial}{\partial\theta^1} & (1-2\nu)\frac{\partial}{\partial\theta^2} & \frac{\partial}{\partial\theta^2} & 0 \\ \frac{\partial}{\partial\theta^2} & 0 & (1-2\nu)\frac{\partial}{\partial\theta^1} & (2-2\nu)\frac{\partial}{\partial\theta^2} \end{pmatrix} \begin{pmatrix} u^{11} \\ u^{12} \\ u^{21} \\ u^{22} \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix} \tag{9}$$

Each block of A thus derives from a term

$$f_i(x,y)\mathrm{D}_i u^j, \tag{10}$$

where D_i is one of the operators $\frac{\partial}{\partial\theta^1}$, $\frac{\partial}{\partial\theta^2}$ or I, so its entries are determined by f_i and D_i, while u^j is only a reference point.

The general input format for an equation thus has to be

$$f_1 \frac{\partial u^1}{\partial \theta^1} + f_2 \frac{\partial u^1}{\partial \theta^2} + f_3 u^1 + \cdots + f_i \frac{\partial u^j}{\partial \theta^1} + f_{i+1} \frac{\partial u^j}{\partial \theta^2} + f_{i+2} u^j = g \tag{11}$$

where most of the f_i are zero.

Next in the structure, each equation belongs to a region. The regions are top, bottom, left, right, and interior, as well as the overlap variants for top, bottom, left and right. Each region has one or more equations, and is part of a domain. Same-name constants and variables appearing in different domains are different, and need to be treated separately. Connected domain boundaries are special, because these equations, while appearing in separate domains, form a single equation. Therefore, the order of equations matters and must be consistent on both sides of such a boundary. E.g., in the first equation in the middle of figure 2, $-u^1 = 0$ is *not* the same as $u^1 = 0$; using the former and indicating domains by subscripts, the globally correct equation $u_1^1 - u_2^1 = 0$ is produced; using the latter, this becomes $u_1^1 + u_2^1 = 0$.

Insight from a prototype

Initially, the following direct approach to the use of the equations in the production of the matrix A and right-hand-side $\mathbf{b}$ of equation 8 was tried:

- Manually implement a monolithic solver which only handles a specific equation by combining the equations' functions $f_i(x,y)$ with the operator (D_i) implementation[‡].
- Because of the sparse structure of the system of equations, cover only non-zero parts of the matrix.
- To avoid complex intermediate data structures, insert matrix blocks into the matrix as they are formed.

This approach is the most straightforward, and very quickly produces working programs for simple one- and two-dimensional problems; it provides insight into the block structure of the matrix A and vector $\mathbf{b}$; it gives an idea of what pieces a modular solver might consist of; and it identifies many implementation problems.

In collocation, the discrete- and continuous-operator block structures are identical and can simply be referred to as the block structure. The block structure of simple problems is analogous to equation 9; in particular, a block's row is indexed by domain, region and equation while the column is indexed by domain and unknown. Each block's structure is as shown in equation 10, and blocks are manually aligned in a full matrix A.

Among the implementation problems are:

1. Explicitly using the sparsity of the equations in a program results in many "special" cases in code.

[‡]This is especially tempting in finite-difference code, where most of the D_i are trivial.

2. Implicitly forming the matrix blocks implies that key structural information about the blocks is unavailable elsewhere.
3. Testing of individual components of the solver is impossible; there is one one component.
4. Incremental testing of the solver, using different equations, is impossible; every new system of equations requires a substantial rewrite.
5. As the equations' components are now embedded in the solver, it is hard to verify their correctness, or to test them in another way.
6. The equations embedded in the program are of no use in documentation; there may be inconsistencies between implemented equations and the documentation.

All of these problem are due to lack of modularity, and the use of *manually* produced different forms of the same equations. As for any complex software system, modularity is critical; for systems implementing mathematics, this is aggravated by the density of the data (a single sign switch can be fatal), the fact that many of the data are functions (which are not first-class in most languages), and the difficulty of testing isolated parts (correct values of intermediate structures/results are not usually known).

Easily identified components are the equations, the solver core, the matrix blocks, the linear solver, and a solution extractor. The first three are of interest here, and are dealt with next.

Solver requirements

A first step in modularizing the prototype is the separation of the equations, the solver core, and the matrix blocks.

To determine a format acceptable to the PDE solver, first consider the structure the solver will have to produce. The complete sample problem consists of the equations shown in figure 2 together with equations 4 and 5 in the interior. By following the same row and column indexing, and block-structuring the problem analogously to equation 9, one arrives at the block matrix structure of figure 3. This matrix contains 114 separate blocks, each of which is sparse and depends on the use of the correct coefficient functions $f_i(x,y)$ and operators D_i of the block's equation component (as in equation 10). Ignoring the order of the blocks, the full matrix is simply made up of a collection of indexed blocks, and each block holds one or more operator/function pairs. Thus, a table mapping (domain, region, equation, side) to a operator/function list is one natural choice for this problem[§]; with this, the solver core simply iterates over all possible block positions, but only generates entries when a corresponding table entry exists.

Such a table can be directly constructed for functional languages like Ocaml and LISP, but for FORTRAN or C, a collection of functions and a separate table are needed.

In the table, the implementation of the function should be no problem; it will only use common operators and functions, and allow calls to user-defined functions.

[§]There are at least two ways of splitting the equations; another possible mapping is from (domain, region, equation, side) to a (operator list)/function pair. Both cases have to handle multiple operators applied to an unknown in a single equation, but to simplify the solver core, the former approach is taken here.

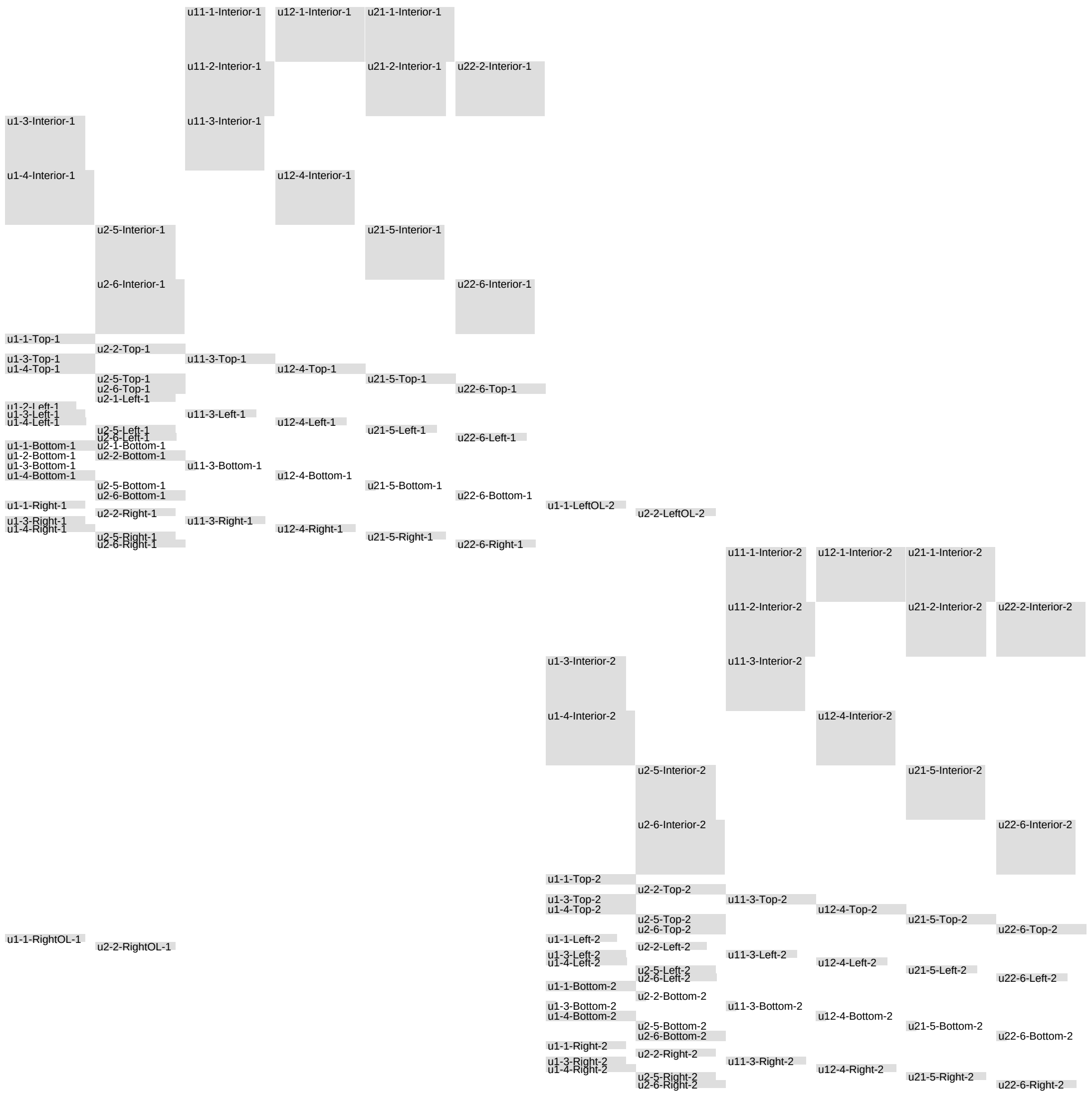

Figure 3. An example of the matrix block structure, with 114 blocks in 60 rows; the 64 blocks in the right-hand side are not shown. A block's row is indexed by domain, region and equation; the column is indexed by domain and unknown. The indexing of values within the blocks is not relevant here. See also figure 4.

```
let _gensym_f1 x y =  1.0
let _gensym_f2 x y =  0.0
let _gensym_f3 x y =  1.0
let _gensym_f4 x y =  0.0
let _gensym_f5 x y =  1.0
let _gensym_f6 x y =  -1.0
...
let _gensym_f30 x y =  -. (!Dom_1._mu)
let _gensym_f31 x y =  -. (!Dom_1._mu)
...
let _gensym_f36 x y =
    ((-2.0*. (!Dom_1._mu))*. (-1.0+. (!Dom_1._nu)))/.
       (-1.0+. 2.0*. (!Dom_1._nu))
let _gensym_f37 x y =
    ((2.0*. (!Dom_1._mu))*. (!Dom_1._nu))/. (-1.0+. 2.0*. (!Dom_1._nu))
let _gensym_f38 x y =  (!Dom_1._sigma)
...
let _gensym_f178 x y =  0.0
```

Figure 4. A subset of the 178 generated coefficient functions. These correspond to 114 matrix blocks, plus 64 right-hand-side blocks; see the assembled matrix in figure 3.

The operator poses a more interesting problem. Should it be provided for every table entry? If so, what arguments should it take, and how should its return values be handled? There are many mathematical ways of encoding such an operator, some of which require substantial information about the matrix block. All of them require further indexing information *within* a given block. And of course, there are only three distinct operators in the present problem¶, namely I, D_1, and D_2. It therefore makes sense to simply specify which operator is to be used, and leave the implementation for another module.

At this point, the original matrix block structure can be directly generated using a table and associated functions. Using this table‖ already avoids several of the preceding problems (1 and 2), and lessens others (3, 4), but it does not eliminate them all. It remains to find a way of producing such tables and avoiding the remaining problems.

For illustration, examples of such files are shown in figures 4 (functions), 5 (domain-specific parameters) and 6 (table); these were produced automatically. While manually written versions could have better names and other readability enhancements, these would not improve the legibility significantly.

The scoping of domain-specific variables can be dealt with in many ways; here, it is done using Ocaml modules. For C code, a simple long name may be preferable, although this introduces global variables; another choice would be the definition of an appropriate struct, to be passed to every function.

¶For more general problems, adding the z-direction operator $\frac{\partial}{\partial \theta^3}$, along with a partial in time would be straightforward. Of course, the solver would also have to handle them

‖At least one other solver, TTGU [3], takes this approach to providing input equation, but goes no further.

```
module Dom_1 = struct
     let _nu = ref 0.33
     let _mu = ref 12345.0
     let _sigma = ref 1.0
     end
module Dom_2 = struct
     let _nu = ref 0.33
     end
let domainNames = [ (1, ["_nu"; "_mu"; "_sigma"]) ; (2, ["_nu"]) ]
```

Figure 5. Values for the domain-specific parameters.

```
let ins key value = Hashtbl.add cd_dom_table key value ;;

let _ = ins (1, Top, 1, Lhs) [
        {cd_f = _gensym_f1 ; cd_op = I ; cd_u = "u1" }];;
let _ = ins (1, Top, 1, Rhs) [
        {cd_f = _gensym_f2 ; cd_op = I ; cd_u = "" }];;
...
let _ = ins (1, Right, 1, Lhs) [
        {cd_f = _gensym_f48 ; cd_op = D1 ; cd_u = "u1" }];;
...
let _ = ins (1, RightOL, 1, Lhs) [
        {cd_f = _gensym_f87 ; cd_op = I ; cd_u = "u1" }];;
let _ = ins (1, RightOL, 1, Rhs) [
        {cd_f = _gensym_f88 ; cd_op = I ; cd_u = "" }];;
...
let _ = ins (2, LeftOL, 1, Rhs) [
        {cd_f = _gensym_f176 ; cd_op = I ; cd_u = "" }];;
let _ = ins (2, LeftOL, 2, Lhs) [
        {cd_f = _gensym_f177 ; cd_op = D1 ; cd_u = "u2" }];;
let _ = ins (2, LeftOL, 2, Rhs) [
        {cd_f = _gensym_f178 ; cd_op = I ; cd_u = "" }];;
```

Figure 6. The mapping from (domain, region, equation, side) to the appropriate function(s).

For C++, defining callable objects in which the constructor receives needed arguments is another choice. In all cases, the correct values of variables are provided by the user.

Input language requirements

For users, a pretty-printed listing is the first step in verifying the input equations' correctness; for complex equations, typeset output may be more appropriate. Equally important is the use of the same equations in documenting the numerical results, which also requires typeset output.

To be practical in this way, the input language must resemble the mathematical equations as much as possible. One way of accomplishing this is to provide a translator which produces typeset output, and structure the input language to correspond as much as possible to this typeset output. Complicated equations are likely to be derived using a computer algebra system (CAS), so syntactic compatibility with one of these is desirable. Here, a superset of MAPLE's expression syntax is used. For compatibility, all new operators have a function-call prefix form so expressions can be read and written by MAPLE; a translator to another language would be a simple addition.

To be of use to the solver, the equations given in the input language must be readily translatable to the preceding table format. The easiest way to achieve this is for the nested expression format to have levels corresponding to the indices of the table.

The structure for user input, pretty-printed output, and typeset output is mostly provided by the problem itself: every problem has multiple domains (enumerated), every domain has multiple regions (named), every region contains a list of equations (enumerated), and every equation has the structure shown in equation 11. Recall that the structure for the solver is a table mapping (domain, region, equation, side) to a operator/function list, and thus the index of the table corresponds one-to-one to the user input structure; using nested key-value pairs, the outer structure can be written as

```
equation_specs=[
   unknowns=[u1, u2, u11, u12, u21, u22],
   domains=[
      1=[
         regions=[
            Interior=[ <equation>, ..],
            ..] ],
      .. ] ]
```

To arrive at the form individual `equation` terms need to take, move inside-out. At the lexical level, the Greek letters have to be dealt with; using MAPLE's long-identifier syntax (enclosing symbols in back quotes) allows the use of arbitrary names. This allows direct use of TEX names in the expressions, e.g. ν is `\nu`, μ is `\mu`, σ is `\sigma`, etc.

The form of any left-hand-side term, from equation 11, is $f(x,y)u^i$ or $f(x,y)\frac{\partial u^i}{\partial \theta^j}$, where $f(x,y)$ contains no u^i. The derivatives are already taken in numbered directions; to further mechanize the symbolics, introduce an infix operator to denote differentiation (here, the backslash \\), and simply append all index superscripts to the unknown name.

With these changes, equations 4 can be written in the form

```
((-2*‘\nu‘+2) * u11\ 1 + (1-2*‘\nu‘) * u12\ 2) + u21\ 2 = 0,
((1-2*‘\nu‘) * u21\ 1 + (2-2*‘\nu‘) * u22\ 2) + u11\ 2 = 0
```
(12)

while equation 11 takes the form

$$f_1 \cdot u^1\backslash 1 + f_2 \cdot u^1\backslash 2 + f_3 \cdot u^1 + \cdots + f_i \cdot u^j\backslash 1 + f_{i+1} \cdot u^j\backslash 2 + f_{i+2} \cdot u^j = g \tag{13}$$

Every equation must have the format of equation 13, and the translator has to recognize each f_i, u^j, g and $\backslash k$ correctly, and should reject non-conforming terms. The operator/function list structure needed for the solver can be clearly seen in equation 13, which is a sum (list) of function-unknown/operator products. This structure is less obvious in equations 12, and hence problematic for program identification.

The key here is to realize the semantic difference between the `*` and `+` operators in and between the general terms $f_1 \cdot u^1\backslash 1 + \cdots$, and those of the simple coefficient functions, e.g. `-2*‘\nu‘+2`. The latter are just addition and multiplication, while the former also provide structural information. To get easily identifiable input equations in the form of equation 13, this structure can be made explicit by introducing four new infix operators. In typeset form, these are $\oplus$, $\ominus$, $\doteq$, and $\odot$. while the ASCII forms are `+~`, `-~`, `=~`, and `*~`.

Using these, equation 12 becomes

```
(-2*‘\nu‘+2) *~ u11\ 1 +~ (1-2*‘\nu‘) *~ u12\ 2 +~ u21\ 2 =~ 0,
(1-2*‘\nu‘) *~ u21\ 1 +~ (2-2*‘\nu‘) *~ u22\ 2 +~ u11\ 2 =~ 0
```
(14)

and equation 13 can be written as

$$f_1 \odot u^1\backslash 1 \oplus f_2 \odot u^1\backslash 2 \oplus f_3 \odot u^1 \oplus \cdots \oplus f_i \odot u^j\backslash 1 \oplus f_{i+1} \odot u^j\backslash 2 \oplus f_{i+2} \odot u^j \doteq g \tag{15}$$

This notation reduces every equation to a tree, with variations only in the leaves, allowing the translator to precisely check the structure[**] of the equations, and providing an explicit reminder of the expected structure to the user. The conversion of this tree to a list of (function, operator,unknown) tuples is straightforward; combining the earlier nested lists with these new ones gives the final syntax shown in figure 7.

Overall, this produces a hierarchy of views of the same data, from most legible to least:

1. typeset mathematical equations;
2. typeset mathematical equations with a few special operators;
3. ASCII input form, infix with special operators;
4. prefix ASCII input form, compatible with MAPLE and other computer algebra packages;
5. a collection of functions and a table indexing them, for use by the numerical core.

For the present example the typeset version and the table forms were shown previously; the typeset form with special operators, and common equations factored, is shown in figure 8, the pretty-printed infix form in figure 9, and the prefix form in figure 10. The latter three figures were of course generated, with input as in figure 9 without the indentation. Because the typeset version structurally corresponds to the pretty-printed infix version, the two can be used very well in the verification of the input equations. The poor readability of the prefix form illustrates why a specialized grammar is needed here.

A modular solver and modular testing for implementors

A comparison of figures 4 and 6 (table input) with 8 and 9 (little language input) clearly shows the advantage of the little language to the user of a solver. The development of a solver requires testing it,

[**]This is strictly a structural check which will not enforce mathematical validity of equations; even so, it has been found to be very useful.

```
nonterminals:   <bar>
terminals:      BAR
Literals:       bar + - * ** ^ \  +~ -~ *~ =~
Lists:          [list <of> ITEMS]
one or more:    +
zero or one:    ?
zero or more:   ×
groups:         {  }
alternation:    |

<struct>      ::=  equation_specs = [<specs>  <specs>]
<specs>       ::=  unknowns = [SYMBOL+] | domains = [<domain>+]
<domain>      ::=  INT = [ regions = [<region>+] ]
<region>      ::=  { Interior
                     | Top       | TopOL
                     | Left      | LeftOL
                     | Bottom    | BottomOL
                     | Right     | RightOL  } = [<equation>+]
<equation>    ::=  <expr> =~ <plain-expr>
<expr>        ::=  {<plain-expr> *~}? SYMBOL {\ INT}? {{+~|-~} <expr>}×
<plain-expr>  ::=  {-|+}? <prod-expr> {{+|-} <plain-expr>}×
<prod-expr>   ::=  <pow-expr> {{*|/} <prod-expr>}×
<pow-expr>    ::=  <elem-expr> {{**|^} <pow-expr>}×
<elem-expr>   ::=  FLOAT | INT | SYMBOL | <function-call> | <indexed-access>
```

Figure 7. A BNF grammar for the equation input format. The key to automatic input equation conversions are the domain, region, equation and expr rules. The rules for plain-expr are standard expression syntax.

which means using it. Thus, the language is immediately useful for solver developers by simplifying the test input. However, using the code generator allows for much more refined and detailed testing than is possible without it. To see why, one must distinguish three approaches to testing a solver: the mathematical way, the programmer's way, and a hybrid.

Before describing these three testing approaches, an overview of the solver is needed. The solver structure is shown in figure 11. Proceeding from left to right, equations for the specific problem are provided; from these, program code, LaTeX, and plain ASCII text can be produced. The solver core then combines the code with equation parameters and other problem parameters to form the matrix blocks. These are assembled and passed to the linear solver, producing the solution. At this point, the LaTeX output from the type generator can be used in the documentation along the computed solution. The grey boxes show user-provided data when using the little language; without it, the equations would not be needed, but the functions, lookup table, parameter template and TeX source would be.

Mathematical testing means using a problem with known answer, providing appropriate solver input, and comparing the computed solution to the exact one. There are several problems with this. First, looking at figure 11, this would imply testing all components of the solver at once – which was problem 3 encountered with the prototype. Second, using a real problem as an initial test is hopeless; recall the present sample problem has 178 functions and blocks, making it too large to find errors easily. Viewed this way, testing of the solver core implies testing the whole solver – a very difficult problem. Further, testing such a problem does not verify all (or even many) interactions between functions and operators,

$$
\begin{array}{l}
\text{equation\_specs}=[\\
\quad \text{unknowns}=[\text{u1},\quad \text{u2},\quad \text{u11},\quad \text{u12},\quad \text{u21},\quad \text{u22}],\\
\quad \text{domains}=[\\
\quad\quad 1=[\\
\quad\quad\quad \text{regions}=[\\
\quad\quad\quad\quad \text{Interior}=[\\
\quad\quad\quad\quad\quad (((-2)\cdot\nu+2)\odot\text{u11}\backslash 1\oplus(1-2\cdot\nu)\odot\text{u12}\backslash 2)\oplus\text{u21}\backslash 2\doteq 0,\\
\quad\quad\quad\quad\quad ((1-2\cdot\nu)\odot\text{u21}\backslash 1\oplus((-2)\cdot\nu+2)\odot\text{u22}\backslash 2)\oplus\text{u11}\backslash 2\doteq 0,\\
\quad\quad\quad\quad\quad \text{system}],\\
\quad\quad\quad\quad \text{Bottom}=[\\
\quad\quad\quad\quad\quad -\mu\odot\text{u1}\backslash 2\oplus-\mu\odot\text{u2}\backslash 1\doteq 0,\\
\quad\quad\quad\quad\quad \frac{(-2)\cdot\mu\cdot((-1)+\nu)}{(-1)+2\cdot\nu}\odot\text{u2}\backslash 2\oplus\frac{2\cdot\mu\cdot\nu}{(-1)+2\cdot\nu}\odot\text{u1}\backslash 1\doteq\\
\quad\quad\quad\quad\quad \sigma,\\
\quad\quad\quad\quad\quad \text{system}],\\
\quad\quad\quad\quad \text{Left}=[\text{u21}\doteq 0,\quad \text{u1}\doteq 0,\quad \text{system}],\\
\quad\quad\quad\quad \text{Top}=[\text{u1}\doteq 0,\quad \text{u2}\doteq 0,\quad \text{system}],\\
\quad\quad\quad\quad \text{Right}=[\text{u11}\doteq 0,\quad \text{u21}\doteq 0,\quad \text{system}],\\
\quad\quad\quad\quad \text{RightOL}=[\text{u1}\doteq 0,\quad \text{u2}\doteq 0]]],\\
\quad\quad 2=[\\
\quad\quad\quad \text{regions}=[\\
\quad\quad\quad\quad \text{LeftOL}=[(-1)\odot\text{u11}\doteq 0,\quad (-1)\odot\text{u21}\doteq 0],\\
\quad\quad\quad\quad \text{Left}=[(-1)\odot\text{u1}\doteq 0,\quad (-1)\odot\text{u2}\doteq 0,\quad \text{system}],\\
\quad\quad\quad\quad \text{Top}=[\text{u1}\doteq 0,\quad \text{u2}\doteq 0,\quad \text{system}],\\
\quad\quad\quad\quad \text{Right}=[\text{u1}\doteq 0,\quad \text{u2}\doteq 0,\quad \text{system}],\\
\quad\quad\quad\quad \text{Interior}=[\\
\quad\quad\quad\quad\quad (((-2)\cdot\nu+2)\odot\text{u11}\backslash 1\oplus(1-2\cdot\nu)\odot\text{u12}\backslash 2)\oplus\text{u21}\backslash 2\doteq 0,\\
\quad\quad\quad\quad\quad ((1-2\cdot\nu)\odot\text{u21}\backslash 1\oplus((-2)\cdot\nu+2)\odot\text{u22}\backslash 2)\oplus\text{u11}\backslash 2\doteq 0,\\
\quad\quad\quad\quad\quad \text{system}],\\
\quad\quad\quad\quad \text{Bottom}=[\text{u12}\doteq 0,\quad \text{u2}\doteq 0,\quad \text{system}]]]]]\\
\text{system}=[\text{u1}\backslash 1\ominus\text{u11}\doteq 0,\quad \text{u1}\backslash 2\ominus\text{u12}\doteq 0,\quad \text{u2}\backslash 1\ominus\text{u21}\doteq 0,\quad \text{u2}\backslash 2\ominus\text{u22}\doteq 0]
\end{array}
$$

Figure 8. Typeset infix form with automatically factored common equations.

or regions and domains, or proper matrix block alignment, etc. Thus, even a working test problem gives little confidence in the solver as a whole.

The programmer's approach to testing is to write a modular program, and test modules independently. The solver is now modular, and it is possible to use properly selected input to verify the correctness of specific components, e.g., the code generator. The question is: what constitutes useful testing input? The obvious choice is a simple mathematical problem with known solution, but this has the previously mentioned problems.

A practical approach is a hybrid of the mathematical and programmer's approaches. The key to this is to notice that while the solver's modularity allows testing of individual modules, the language itself, and especially the BNF grammar in figure 7, provide separation and orthogonality of equation components, and using these, the structure of the *input* to the solver can be treated as "modular",

```
equation_specs=[
   unknowns=[u1, u2, u11, u12, u21, u22],
   domains=[
      1=[
         regions=[
            Interior=[
               ((-2*‘\nu‘+2)*˜ u11\ 1+˜ (1-2*‘\nu‘)*˜ u12\ 2)+˜
               1*˜ u21\ 2=˜0,
               ((1-2*‘\nu‘)*˜ u21\ 1+˜ (-2*‘\nu‘+2)*˜ u22\ 2)+˜
               1*˜ u11\ 2=˜0, 1*˜ u1\ 1+˜ -1*˜ u11=˜0,
               1*˜ u1\ 2+˜ -1*˜ u12=˜0, 1*˜ u2\ 1+˜ -1*˜ u21=˜0,
               1*˜ u2\ 2+˜ -1*˜ u22=˜0],
        ...
            Right=[
               1*˜ u11=˜0, 1*˜ u21=˜0, 1*˜ u1\ 1+˜ -1*˜ u11=˜0,
               1*˜ u1\ 2+˜ -1*˜ u12=˜0, 1*˜ u2\ 1+˜ -1*˜ u21=˜0,
               1*˜ u2\ 2+˜ -1*˜ u22=˜0],
            RightOL=[1*˜ u1=˜0, 1*˜ u2=˜0]]],
      2=[
         regions=[
            LeftOL=[-1*˜ u11=˜0, -1*˜ u21=˜0], ... ]]]]
```

Figure 9. Formatted infix form, for direct input and pretty-printing.

```
equation_specs = [unknowns = [u1, u2, u11, u12, u21, u22],
domains = [
1 = [regions =
[Interior = [OpEqn(OpSum(OpSum(OpProd(-2*‘\nu‘+2,OpDiff(u11,1)),
    OpProd(1-2*‘\nu‘,OpDiff(u12,2))),OpProd(1,OpDiff(u21,2))),0),
    OpEqn(OpSum(OpSum(OpProd(1-2*‘\nu‘,OpDiff(u21,1)),
    OpProd(-2*‘\nu‘+2,OpDiff(u22,2))),OpProd(1,OpDiff(u11,2))),0),
    OpEqn(OpSum(OpProd(1,OpDiff(u1,1)),OpProd(-1,u11)),0),
    OpEqn(OpSum(OpProd(1,OpDiff(u1,2)),OpProd(-1,u12)),0),
    OpEqn(OpSum(OpProd(1,OpDiff(u2,1)),OpProd(-1,u21)),0),
    OpEqn(OpSum(OpProd(1,OpDiff(u2,2)),OpProd(-1,u22)),0)],  ...
 Right = [OpEqn(OpProd(1,u11),0),
    OpEqn(OpProd(1,u21),0),
    OpEqn(OpSum(OpProd(1,OpDiff(u1,1)),OpProd(-1,u11)),0),
    OpEqn(OpSum(OpProd(1,OpDiff(u1,2)),OpProd(-1,u12)),0),
    OpEqn(OpSum(OpProd(1,OpDiff(u2,1)),OpProd(-1,u21)),0),
    OpEqn(OpSum(OpProd(1,OpDiff(u2,2)),OpProd(-1,u22)),0)],
 RightOL = [OpEqn(OpProd(1,u1),0), OpEqn(OpProd(1,u2),0)]]],
2 = [regions = [
LeftOL = [OpEqn(OpProd(-1,u11),0), OpEqn(OpProd(-1,u21),0)],  ... ]]]]
```

Figure 10. Unformatted prefix (raw input) form, compatible with e.g. MAPLE.

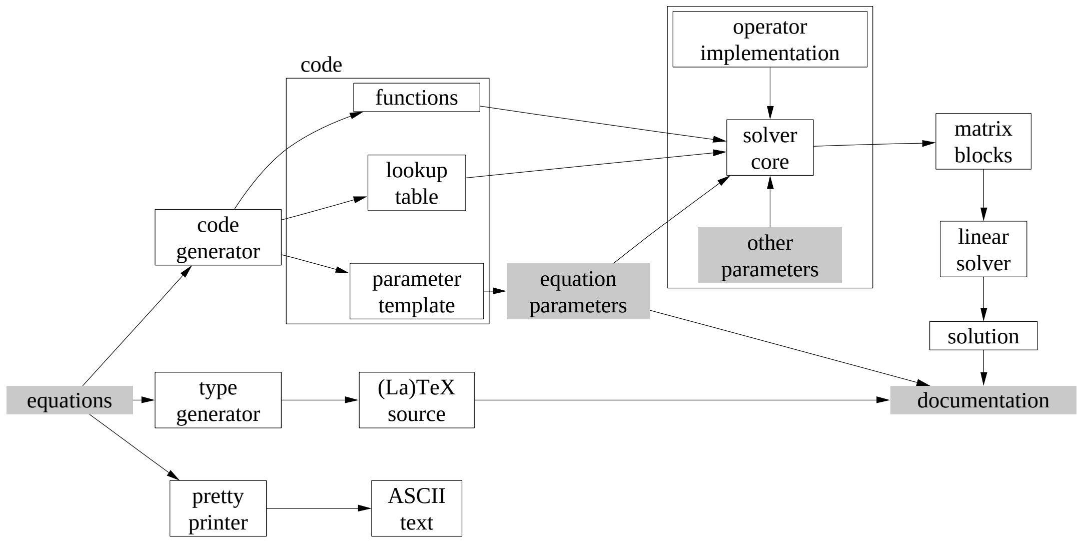

Figure 11. Modular differential equation solver.

with limited interaction between "modules". Specifically, the domain, region, equation, expr and plain-expr elements all affect different parts of the solver differently, so that properly chosen input allows every one of these component of the grammar to be tested independently of the others. Also, the interactions of specific combinations of these parts can be tested.

It should be noted that for any complex PDE, a wrong parameter value can result in a problem without solution, which would at the earliest show in the linear system solution; preceding solver components would not be affected. Of course, there is no way to test for all possible combination of parameter values, and restriction of values can only happen for a specific PDE, not for the class dealt with here. Thus, the present testing verifies the correctness of program modules and input structure, but not overall validity of the input. However, this absence of mathematical validity checks is exactly what is needed in the above hybrid testing approach, so it would seem to be advantageous.

Using the above ideas, the author's testing proceeded roughly as follows. The first test problem was the trivial equation

```
equation_specs=[
   unknowns=[u1],
   domains=[
      1=[
         regions=[
            Interior=[ u1 =~ 2]]]]]
```

which is of course not a PDE at all, but allows for easy testing of the operator implementation – all intermediate values are known. After verifying the correctness of the code generator, this produced an matrix block of incorrect size, but containing correct values. This problem was quickly traced to an incorrect parameter in a loop, and fixed. Having tested the matrix blocks' correctness for this input, testing was continued on the remainder of the solver – the assembly of the matrix blocks, solution of the system, and extraction of the solution. This was very simple, and worked the first time – a definite confidence boost.

Next, to test correct translation of coefficients, the marginally more interesting

```
... Interior=[ 5*x *~ u1 =~ 2] ...
```

was used, followed by some more complicated expressions also involving y, followed by a summation. This tested the trivial case with one unknown, one domain and one region, but all parts of an expression except the operator. For that, a trivial term was again used; and because the coefficients had already been tested, there was no need to repeat that test. With the expressions and equations tested, tests for other regions only involved verifying the proper region index in the output; for the domains, the proper index and proper scoping of variables were checked.

Returning to the prototype, all problems encountered there are now removed or sufficiently reduced. Problems 1 and 2 (special code and implicitly formed matrix blocks) were already eliminated through the use of a table of functions. Testing of individual components of the solver is now far more reliable than a table-only approach (problem 3); with a hand-produced table, there could always be a mistake in the table, while the likelyhood of overlooking a mistake in the input language (when looking at the typeset version) is very small. The structure of the input language now allows modular testing of the input equations (problem 4). Using the pretty printer (and the language itself), verification of the (structural) correctness of the equations is trivial (problem 5); and last, using the type generator, the documentation problem (number 6) is gone.

Some notes on using a table-only approach are in order. The idea for this orthogonal testing came from using and testing the language, and repetitions are handled by the table generator, making this style of testing practical. In a hand-written table, using such mutually independent constructs is hard, since the table has to be modified and extended for expression testing, and there are no clean ways of dealing with the needed repetitions for regions and domains[††]. Also, the language clearly identifies, to the implementor, the pieces that need testing. The main use of the mathematical equations, e.g. modeling of a physical system, is irrelevant in the implementation of a solver[‡‡] or a pretty-printer, and in fact can be a distraction here. This is a problem because there are many ways to write a given set of equations, usually tailored to the needs of the physical problem, not the needs of implementation. Use of the language eliminates these distractions by restricting – hence focusing – one's thoughts on the structure of the equations.

A note on the mathematical method should be made here. Although the solver described above was originally written for one particular numerical method (the sinc method, see [2]), the modularity

[††]Using a macro expander is a possibility, but this gets very ugly; it is, after all, equivalent to using a part of the little language without the rigorous grammar.

[‡‡]Of course the characteristics of the equations and their solutions affect the choice of numerical method and other such details, but these choices are made before the solver is implemented.

imposed by this little language also makes it suitable for other numerical methods; it took the author only one day to replace the operator implementation with another (finite differences), including testing! The second day, running the whole solver on the present sample problem produced expected results. Thus, method independence turns out to be another excellent way of testing individual modules, and the solver as a whole.

ACKNOWLEDGEMENTS

The implementation of the language translators was greatly simplified through the use of the ocaml ([5] and [4]) language, especially its tree pattern matching facility and its very comprehensive library.

The design of the original solver greatly benefitted from the automatic graph layout provided by dot [1], which also generated figure 11.

REFERENCES

1. GANSNER, E. R., AND NORTH, S. C. An open graph visualization system and its applications to software engineering. *Software — Practice and Experience 30*, 11 (Sept. 2000), 1203–1233.
2. HOHN, M. *On the Solution of Mixed Boundary Value Problems in Elasticity*. Ph.D. thesis, Department of Mathematics, University of Utah, Salt Lake City, UT, USA, Dec. 2001. Available at `http://www.math.utah.edu/~hohn/thesis-final.pdf`.
3. KAUFMAN, L. TTGU — a package for solving time varying partial differential equations on a union of rectangles. Tech. Rep. 154, AT&T Bell Labs Computing Sciences, December 1990. Available at `http://cm.bell-labs.com/cm/cs/cstr/154.ps.gz`.
4. LEROY, X. Le système caml special light: modules et compilation efficace en caml. Tech. rep., INRIA, 1995. This report describes the precursor to ocaml. Available at `http://pauillac.inria.fr/~xleroy/publi/caml-special-light-rr.ps.gz`.
5. LEROY, X. Objective caml. Online, `http://pauillac.inria.fr/ocaml`, 1997. Most recent version at `ftp://ftp.inria.fr/lang/caml-light/ocaml-3.00.tar.gz`.
6. RICE, J. R., AND BOISVERT, R. F., Eds. *Solving Elliptic Problems Using ELLPACK*. Springer-Verlag, Berlin, Heidelberg, New York, Tokyo, 1985.
7. ROITZSCH, R., ERDMANN, B., AND LANG, J. Kaskade manual. Tech. rep., Konrad-Zuse-Zentrum für Informationstechnik Berlin, 1993.