1992

Integrated Symbolic-Numeric Computing in //ELLPACK: Experiences and Plans

Sanjiva Weerawarana

Ann C. Catlin

Elias N. Houstis
Purdue University, enh@cs.purdue.edu

John R. Rice
Purdue University, jrr@cs.purdue.edu

Report Number:
92-092

INTEGRATED SYMBOLIC-NUMBERIC COMPUTING IN //ELLPACK: EXPERIENCES AND PLANS

Sanjiva Weerawarana
Ann C. Catlin
Elias N. Houstis
John R. Rice

CSD-TR-92-092
March 10, 1992
(Revised 11/92)

Integrated Symbolic–Numeric Computing in //ELLPACK: Experiences and Plans

Sanjiva Weerawarana, Ann C. Catlin, Elias N. Houstis and John R. Rice *
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907.

March 10, 1992

Abstract

In this paper we describe the use of integrated symbolic–numeric computing techniques in the evolving //ELLPACK[1][HRC+90] Problem Solving Environment. //ELLPACK is a problem solving environment (PSE) for solving partial differential equations (PDEs) using parallel architectures. It was originally developed to use the ELLPACK[RB85] system as the numerical computing engine. The domain of applicability of ELLPACK is restricted to second order linear elliptic boundary value problems in two and three dimensions. We apply hybrid symbolic–numeric techniques to extend the domain of applicability of the //ELLPACK PSE using both ELLPACK and other numerical PDE solving systems as numerical engines.

These techniques have been implemented as an interactive tool using the X Window System[SG86]. Once the PDE problem is specified, it is symbolically manipulated using the MAXIMA[2] computer algebra system to generate a //ELLPACK program to solve the problem using either ELLPACK or FIDISOL[SSM85], a finite–difference method PDE solver. The GENCRAY[WW89] code generation system is used to generate the //ELLPACK program.

Several examples of symbolic processing of PDE problems are presented to illustrate the approach. Finally, we consider the shortfalls of these techniques and discuss our plans for solving some of the problems.

1 Introduction

The //ELLPACK problem solving environment consists of several specialized editors and a programming environment that coordinates these editors.[3] Figure 1 shows the hierarchy of editors in //ELLPACK. The editors assist the user in developing a solution to a PDE problem in a graphical manner, solving it on a parallel computer, and visualizing the solution and performance data. Each editor either defines a component of the problem (for example, the domain), or specifies a component of the solution process. The editors coordinate their activities by communicating their results to the programming environment which maintains them textually in the //ELLPACK language.[4] The program thus developed is translated by the //ELLPACK language processor into FORTRAN code for solving the PDE on the target architecture, and compiled executed (possibly in parallel). An example //ELLPACK program is shown in Figure 2.

*This research was supported in part by AFOSR 88-0234, ARO grant DAAG29-83-K-0026, NSF grant CCF-8619817 and ESPRIT project GENESIS.

[1]read "parallel ELLPACK"

[2]MAXIMA is the Common Lisp port of MACSYMA[TMG77] by Bill Schelter of the University of Texas, Austin.

[3]A problem solving environment is a software system that is specific to a certain problem domain and assists the user in a "domain-intelligent" manner. PSEs are usually interactive and contain many software tools to assist the user during all phases of the problem specification and solution process. For example, a PSE for solving PDEs would include geometry editors to specify the domain, capabilities to visualize and edit the discrete domain, and visualize and analyze the solution.

[4]The //ELLPACK language is a superset of the ELLPACK PDE solving language; the extensions are in the direction of parallel solution of PDEs.

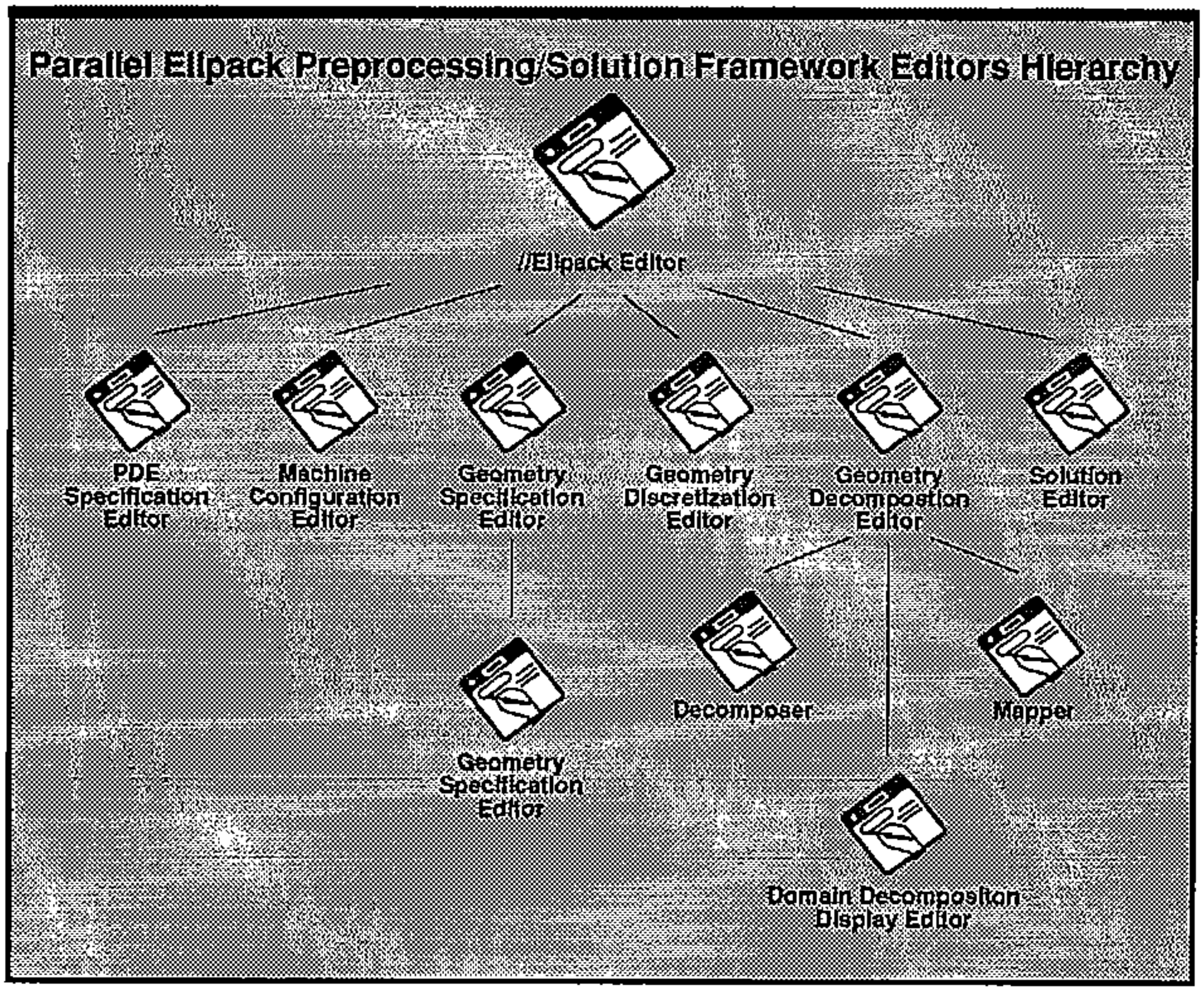

Parallel Ellpack Preprocessing/Solution Framework Editors Hierarchy
//Ellpack Editor
PDE Specification Editor
Machine Configuration Editor
Geometry Specification Editor
Geometry Discretization Editor
Geometry Decompostion Editor
Solution Editor
Geometry Specification Editor
Decomposer
Mapper
Domain Decompositon Display Editor

Figure 1: Hierarchy of composition editors in //ELLPACK.

```
options.
     xplot3d

equation.
     uxx + uyy = 0

boundary.
     u = 1.0 on x = 0
     u = 1.0 on y = 1
     u = 1.0 on x = 1
     u = 1.0 on y = 0

machine.
     machine name = ncube2
     number of pes = 8
     topology = hypercube

decomposition.
     read fem decomposition from '/u/u27/saw/tmp/decomp'

discretization.
     bi-linear fem

indexing.
     as is

solution.
     jacobi si

output.
     table (u,10,10)

end.
```

Figure 2: An Example //ELLPACK Program. Each component of the problem specification or solution specification is called a "segment." Each segments corresponds to either a problem component (for example, the `EQUATION` segment defining the PDE operator or the `BOUNDARY` segment defining the domain), a solution component (for example, the `DISCRETIZATION` segment defining the operator discretization technique) or output specification (`OUTPUT` segment). In the generated FORTRAN program, each segment corresponds to a sequence of FORTRAN statements. Therefore, the language also allows one to insert arbitrary FORTRAN code between modules by placing such code in `FORTRAN` segments.

The PDE solvers in //ELLPACK and ELLPACK are built out of an extensive library of modules that correspond to the various phases of the numerical processes for linear elliptic second order boundary value problems. A detailed description of the ELLPACK system for sequential problems is given in [RB85], while the //ELLPACK system is described in [HRC+90]. Since the ELLPACK language allows one to embed FORTRAN code between segments, one can extend the solution capabilities of ELLPACK by using constructs such as loops and conditionals to perform Newton iteration on nonlinear problems, and to solve time dependent problems via time stepping. Our first extensions are in this direction; we applied the symbolic processing capabilities of MAXIMA to symbolically linearize the PDE operator and then generate a //ELLPACK program that uses Newton iteration to solve the nonlinear problem. Similar actions are taken in the case of time dependent problems.

We have recently integrated the FIDISOL PDE solving package into //ELLPACK. FIDISOL solves nonlinear systems of two– or three–dimensional elliptic and parabolic PDEs using difference methods with variable order and step size. However, FIDISOL does not have its own PDE solving language- it is simply a collection of library routines for which "driver" program(s) must be written by the user. These driver routines provide information to FIDISOL, such as the Jacobian of the PDE operator, and of course need to be written in the form required by FIDISOL. We have greatly simplified this process by allowing the user to simply change to "FIDISOL mode" and then specify the system of PDE operators. Our tool then generates a FIDISOL mode //ELLPACK program that contains all the subroutines that FIDISOL requires. The //ELLPACK language processor (recognizing the "FIDISOL mode") then generates the correct FORTRAN program to invoke FIDISOL to solve the problem. Post processing activities, including output visualization and performance evaluation, is still performed in exactly the same manner as with vanilla ELLPACK.

In the rest of this document, we first describe our experiences with integrating symbolic computation to a numerical simulation environment. Then, we discuss some of the difficulties we've observed and elaborate on what our plans are in this direction.

2 Symbolic–Numeric Computing in //ELLPACK

We apply symbolic computation at the PDE problem level in //ELLPACK. Depending on the target numerical computing engine, different transformations are done. If the target is ELLPACK, we generate code to solve nonlinear problems via Newton iteration, and time dependent problems via time stepping. If the target is FIDISOL, we generate a FIDISOL mode //ELLPACK program along with FORTRAN functions for computing the Jacobian of the PDE operators, and the Jacobian of the boundary conditions. In this section, we will briefly describe the transformations being performed by the symbolic tool to generate the //ELLPACK program.

We are concerned with PDE problems of the form

$$\alpha u_t + \beta u_{tt} = F(t, x, y, z, u, u_x, u_y, u_z, u_{xx}, u_{yy}, u_{zz}, u_{xy}, u_{xz}, u_{yz}) \equiv Fu \tag{2.1}$$

including, for example, in a simpler case,

$$\begin{aligned} \alpha u_t + \beta u_{tt} &= c_0 + c_1 u + c_2 u_x + c_3 u_y + c_4 u_z + c_5 u_{xx} + c_6 u_{yy} + c_7 u_{zz} + \\ &\quad c_8 u_{xy} + c_9 u_{xz} + c_{10} u_{yz} \\ &\equiv Lu \end{aligned} \tag{2.2}$$

Problems (2.1) and (2.2) are defined on $(0, T] \times \Omega$ with $\Omega \subset \Re^3$ and are subject to boundary conditions

$$Gu \equiv G(t, x, y, z, u, u_x, u_y, u_z) = \psi(t) \tag{2.3}$$

or, again in the simpler case,

$$Bu \equiv d_0 + d_1 u + d_2 u_x + d_3 u_y = 0 \tag{2.4}$$

on the boundary of Ω, and initial conditions at $t = 0$

$$u(0, x, y, z) = \phi(x, y, z). \tag{2.5}$$

The coefficients c_i and d_i may depend on (x, y, z) and yet the problem remains in the class of linear PDEs that //ELLPACK currently assumes. The coefficients of L and B could be functions of the solution u, producing a semi–linear problem which is not in the class that //ELLPACK currently assumes. The parameters α and β are chosen to make the equation (2.1) elliptic ($\alpha = 0, \beta = 0$), parabolic ($\alpha = 1, \beta = 0$) and hyperbolic ($\beta = 1$).

2.1 Generating Code for ELLPACK

For nonlinear problems, the PDE is symbolically linearized to generate a //ELLPACK program that solves the problem using Newton's method. Consider the general PDE problem (2.1), (2.3) with $\alpha = 0 = \beta$. The idea of the method is to approximate $F(u) = 0$ and $G(u) = 0$ with their linear counterparts

$$\begin{aligned} F(u_0) + F'(u_0)(u_1 - u_0) &= 0 \\ G(u_0) + G'(u_0)(u_1 - u_0) &= 0 \end{aligned}$$

and then iteratively solve these linear problems. The linear counterparts are the Fréchet derivatives of the operators F and G with respect the the function u and its derivatives. While the mathematical foundations of such differentiation is complex, its mechanics are similar to ordinary differentiation. The corresponding symbolic/numeric process that implements Newton's method can be described as follows:

```
Compute Fr'echet derivatives $L(u), B(u)$ of $F(u)$ and $G(u)$
repeat
    Solve $L(u_0)u = -(F(u_0) - L(u_0)u_0)$, and $B(u_0)u = -(G(u_0) - B(u_0)u_0)$
    Set $u_0 := u$
until converged.
```

For time dependent problems, the time derivatives are replaced by difference quotients and then solved via time stepping. The procedure can be viewed as opposite to method of lines, since the time discretization is done first and the original problem is reduced to a sequence of linear or nonlinear time independent PDEs on the various time levels. In the case of a parabolic PDE ($\alpha = 1, \beta = 0$) and Crank–Nicholson discretization, equation (2.1) is reduced to

$$u(t) - u(t - \Delta t) = \frac{\Delta t}{2}\left\{F(u)\mid_{u=u(t)} + F(u)\mid_{u=u(t-\Delta t)}\right\}. \tag{2.6}$$

Note that we have suppressed the space variables and derivatives of u in the above equation. Assuming that the solution and its derivatives are known at the $t - \Delta t$ level, then the nonlinear PDE with respect to $u(t)$ is solved over the domain Ω with boundary conditions

$$G(u(t)) = \psi(t). \tag{2.7}$$

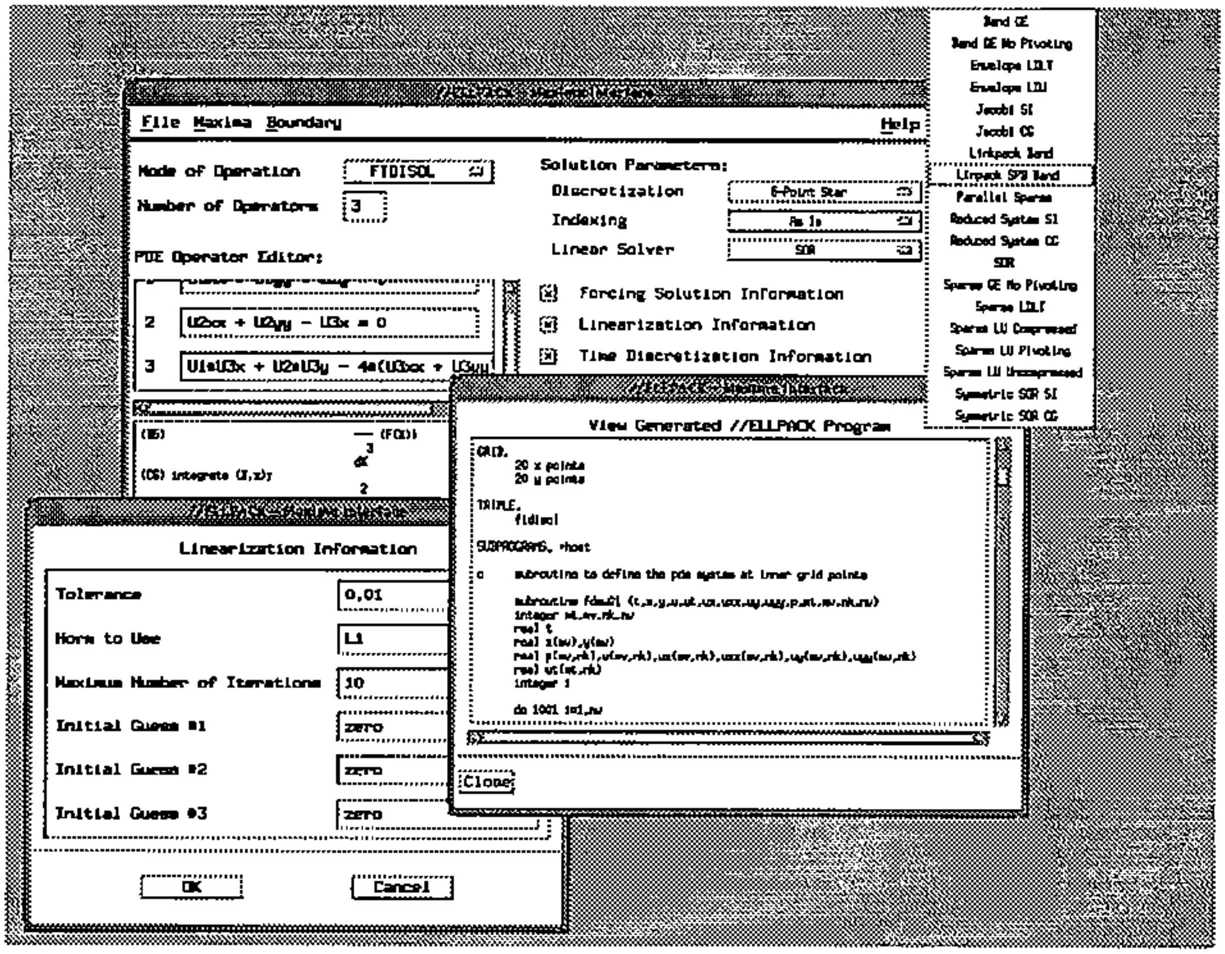
2.2 Generating Code for FIDISOL

FIDISOL's domain of applicability is nonlinear systems of two– or three–dimensional elliptic and parabolic PDEs on logically rectangular domains. That is, FIDISOL can solve PDEs of the form (2.1), (2.3) with $\beta = 0$. As input, it expects one to specify the PDEs, their Jacobians, the boundary conditions and their respective Jacobians in a discrete form (i.e., on the current grid), and many other parameter settings that affect the PDE solution process.

The Jacobian matrices of the PDE operators are generated by simple differentiation of each operator in u with respect to u, u_x, u_y, ... The other main task performed by the symbolic tool is that of deriving certain properties about u, uxx, and uyy. Such information is also recorded in the generated ELLPACK program as options.

Also, for time dependent problems and for nonlinear problems, it is necessary to provide an initial condition or initial guess, respectively, to start off the iteration. This information is also provided in the form of FORTRAN functions generated by the symbolic tool. The ability to force a certain solution on the PDE operators is still maintained and is achieved by generating the proper FORTRAN functions and also changing the PDE operator accordingly.

An example of a generated FIDISOL program is shown in Appendix A.



Mode of Operation: FIDISOL
Number of Operators: 3

PDE Operator Editor:

2 U2xx + U2yy − U3x = 0
3 U1*U3x + U2*U3y − 4*(U3xx + U3yy

Solution Parameters:
Discretization: 5-Point Star
Indexing: As Is
Linear Solver: SOR

[✓] Forcing Solution Information
[✓] Linearization Information
[✓] Time Discretization Information

```
(D5)                           — (FOO)
                               3
                               dX
(C6) integrate (Z,x);
                               2
```

Linearization Information

Tolerance	0.01
Norm to Use	L1
Maximum Number of Iterations	10
Initial Guess #1	zero
Initial Guess #2	zero
Initial Guess #3	zero

View Generated //ELLPACK Program

```
GRID.
      20 x points
      20 y points

TRIPLE.
      fidisol

SUBPROGRAMS, +host

c     subroutine to define the pde system at inner grid points

      subroutine fdmc1 (t,x,y,u,ut,ux,uxx,uy,uyy,p,mt,mv,nk,nv)
      integer mt,mv,nk,nv
      real t
      real x(mv),y(mv)
      real p(mv,nk),u(mv,nk),ux(mv,nk),uxx(mv,nk),uy(mv,nk),uyy(mv,nk)
      real ut(mt,nk)
      integer i

      do 1001 i=1,nv
```

Figure 3: The //ELLPACK Symbolic Tool

3 The //ELLPACK Symbolic Tool

The //ELLPACK symbolic tool is the PDE specification editor in the //ELLPACK PSE and implements the functionality described above. It is implemented as a separate X client that communicates to the //ELLPACK environment via sockets[Wan88] using a simple protocol. Figure 3 shows the a view of the symbolic tool. In this section, we will provide a brief description of its implementation; the first subsection discusses the connection with MAXIMA, the second discusses the implementation of the PDE transformations, and the third discusses //ELLPACK code generation.

3.1 The MAXIMA Connection

The symbolic tool currently uses MAXIMA as its computer algebra engine. MAXIMA runs as a separate process (possibly on another machine) and is also connected to the symbolic tool via sockets.

MAXIMA is a closed system. As such, it was not designed to be used by other *programs* as a compute engine; it is meant to be used as an end tool by a human user. Thus, when one attempts to connect MAXIMA to another program, one must decide how to handle input to and output from it. The most straightforward method would be to write a communication driver that would send input in MAXIMA syntax and understand the MAXIMA output syntax. However, this implies that one must parse the entire MAXIMA language, when what is typically needed is the ability to recognize a few key statements.

What we have done is the following. For input to MAXIMA, we use MAXIMA syntax. But, for MAXIMA output that concerns us, we use a back-door path to the symbolic tool. This is implemented as follows- before the symbolic tool executes MAXIMA, it sets up three input/output channels with MAXIMA. The first is for sending input to it, the second is for MAXIMA's output and error messages, and the third is for special messages coming from MAXIMA to the symbolic tool. For communication along this channel, we use the same protocol as the one between the //ELLPACK environment and the symbolic tool.

Therefore, when a MAXIMA operation that the symbolic tool has initiated needs to communicate with the symbolic tool, it sends a message via the special channel and then calls a function that waits for the reply from the symbolic tool. The symbolic tool responds to the message along the MAXIMA input channel. The `WaitFor` function called within MAXIMA does the following- if the argument predicate evaluates to false, then it executes a read–eval loop which loops until that predicate does evaluate to true. Thus, when the symbolic tool is responds to a query, it also signals that it has replied by assigning a true value to the predicate being checked.

This simple callback mechanism from MAXIMA to the symbolic tool suffices for our needs- it allows for decisions to be made within MAXIMA about when and where to request user assistance. For example, when a given PDE is nonlinear, this knowledge is only realized within MAXIMA. At this point, it is necessary to ask the user to specify additional parameters that are needed to generate a //ELLPACK program to solve the problem via Newton iteration. This is done by sending a message to the symbolic tool requesting additional information. The symbolic tool then opens a new window and asks the user to specify the additional information necessary. Thus, this callback mechanism allows us to avoid asking questions such as "Is this operator linear?", when such information is readily derivable within MAXIMA.

3.2 Linearization, Time Discretization, and FIDISOL Program Generation

The transformations described earlier are implemented as MAXIMA functions that are called by the symbolic tool. These are written in Lisp and in the MAXIMA language and are loaded in to MAXIMA upon startup. Information is sent from the symbolic tool to MAXIMA by calling MAXIMA top-level functions that define parts of the PDE problem and fill in some data structure.

For linearizing operators, the Fréchet derivative is computed by differentiating the PDE operator $F(u)$ with respect to its independent variable (u) and its derivatives $(u_x,\ u_y,\ \ldots)$. To obtain the linearized PDE, the original operator $F(u)$ is replaced by the following linear PDE (for the two–dimensional elliptic case):

$$\begin{aligned}(F_{u_{xx}}(u)|_{u=u_0})\,u_{xx} + (F_{u_{xy}}(u)|_{u=u_0})\,u_{xy} + (F_{u_{yy}}(u)|_{u=u_0})\,u_{yy} + (F_{u_x}(u)|_{u=u_0})\,u_x +\\ (F_{u_y}(u)|_{u=u_0})\,u_y + (F_u(u)|_{u=u_0})\,u =\\ -(F(u)|_{u=u_0} - (F_{u_{xx}}(u)|_{u=u_0}\,u_{0xx} + F_{u_{xy}}(u)|_{u=u_0}\,u_{0xy} + F_{u_{yy}}(u)|_{u=u_0}\,u_{0yy} +\\ F_{u_x}(u)|_{u=u_0}\,u_{0x} + F_{u_y}(u)|_{u=u_0}\,u_{0y} + F_u(u)|_{u=u_0}\,u_0))\end{aligned}$$

where u_0 represents the solution obtained during the previous iteration of Newton's method.

For time dependent (parabolic) operators, the time derivative u_t is replaced by the following difference quotient

$$\frac{u(t) - u(t - \Delta t)}{\Delta t}.$$

Then, the parabolic PDE operator $F(u) + u_t$ is replaced by (with the space variables suppressed for clarity):

$$\lambda\, F(u(t)) + \frac{u(t)}{\Delta t} = (\lambda - 1)\, F(u(t - \Delta t)) + \frac{u(t - \Delta t)}{\Delta t}$$

where λ is a parameter of the time discretization (for example, if $\lambda = \frac{1}{2}$, then this is the Crank–Nicholson discretization) and $u(t - \Delta t)$ is known. The problem is solved by starting with the solution at time t_{start} and time stepping in Δt size steps until t_{end} is reached.

3.3 //ELLPACK Code Generation

Once all the transformations are done within MAXIMA, we generate the //ELLPACK code using the GENCRAY code generation system.

GENCRAY has a Lisp–like input syntax and generates FORTRAN code as output. Although the //ELLPACK language is not the same as FORTRAN, we find that it is still convenient to use GENCRAY as the //ELLPACK–only part of the code (such as the discretization method selection) can easily be generated using GENCRAY's `(literal ...)` construct.

In typical code generation situations, the code generator is loaded into MAXIMA and then called with MAXIMA expressions as arguments. However, rather than load GENCRAY into the MAXIMA image, what we have done is link GENCRAY to the symbolic tool. In MAXIMA, we call a GENCRAY supplied routine called MACCRAY that converts the MAXIMA representation to GENCRAY syntax. This program is sent to the symbolic tool via the special channel and the symbolic tool calls GENCRAY to actually generate the //ELLPACK program.

We find that this two-stage translation process is very convenient for several reasons: firstly, the MACCRAY package is small (only a few hundred lines of Lisp); hence loads quickly and easily into MAXIMA. Secondly, having MAXIMA and GENCRAY code to manipulate within the MAXIMA Lisp environment is much easier than having the //ELLPACK code as the MAXIMA/GENCRAY code is really a sequence of Lisp expressions. This allows us to splice together pieces of generated code conveniently to generate the final program.

4 Limitations of the Current Approach

Although the symbolic tool is reasonably well integrated, we find that there are many difficulties with our approach. Most difficulties are due to lack of infrastructure in the area of integrated symbolic–numeric computing. Some are due to the un–object–oriented design of various components of the //ELLPACK environment. In this section, we will highlight some of the difficulties we experienced.

MAXIMA was designed as a program meant to be used by humans. Therefore, when another program attempts to use MAXIMA as a tool, it needs to pretend to be human in its behavior. For example, when one needs to differentiate some expression, one must send the string "`diff (expr, x);` " to MAXIMA. To obtain the result, one must parse the output produced by MAXIMA and then represent it in a usable manner (for example, as a syntax tree). One obvious difficulty of sending strings is what would happen if there were a simple syntax error in the input? Or, what if there were a run-time error? How is this information given to the symbolic tool, which is waiting for the result? We currently ignore the latter issue as its not a severe problem in our usage (excepting of course a lack of robustness). However, in other situations, this may be a problem. The section on future plans discusses an example where a tighter coupling between the symbolic processor (i.e., MAXIMA) and the programming environment is essential.

As described earlier, we have implemented a simple callback mechanism which allows MAXIMA to make requests from the symbolic tool. However, this mechanism is rather ad hoc and not robust. The main reason for it not being robust is that we do not have any control over how MAXIMA behaves when it needs some information. For example, MAXIMA sometimes asks questions such as "`Is A Positive, Zero, or Negative?`." In our current implementation, we have no mechanism to relay this query to the user through the symbolic tool. One option of course is to redirect all of MAXIMA's queries to our callback routines,[5] but this is not really feasible due to the changes that must be made to the internals of MAXIMA.

Another difficulty that we observed is with the placing of the symbolic tool in the //ELLPACK environment. Currently, the symbolic tool is an independent editor (containing the symbolic processor) that generates some data for the //ELLPACK environment. Thus, it does not take input from the environment nor is it used by other editors that may benefit from it. For example, the boundary tools need access to the symbolic tool as the boundary conditions may need symbolic manipulation (such as linearization, or Jacobian computation). However, with the symbolic processor being a part of the symbolic tool, the boundary tool cannot access it. (For the time being, we have implemented an ad hoc work-around for this problem.) What is needed is for the symbolic processor to be accessible from the environment directly, rather than only through the symbolic tool.

We have not concerned ourselves with another important issue related to differences in data representation. We mentioned earlier that we allow the symbolic tool to execute on a machine different from the one executing the //ELLPACK environment. However, this leads to another set of problems related to byte-order, datatype sizes etc.. To overcome this, we need an RPC[Man90]-like standard communication protocol for communicating mathematical objects.[6] Our use of ascii as a representation avoids some of

[5]During the implementation of SUI[DW90] at Kent State University, such redirection was done in order to have better control over MAXIMA.

[6]The MathLink protocol of Mathematica[Wol88] is an example of such a protocol.

these issues, but we introduce other problems related to the exact ascii representation of numbers, for example.

5 The Future

The //ELLPACK project is a proof–of–concept prototype of an integrated software system for solving elliptic PDEs. We have now embarked on the next phase of this research– the design and implementation of a truly integrated problem solving environment for solving PDE–based problems. In this section, we briefly discuss the proposed architecture of PSEs for PDE solving and then identify other situations where symbolic computing would be of great benefit.

5.1 Architecture of PSEs for PDE Solving

The concept of problem solving environments evolved due to the need for software systems that assist the computational scientist in all aspects of his/her work– during the formulation of the problem, the solution of the problem, and the analysis of the results obtained. Hence, a true PSE is necessarily all encompassing in the sense that it must support a very wide variety of seemingly unrelated activities. Building such a software environment is no simple task, and hence must be approached with the correct vision and careful object–oriented thinking.

The paradigm that we are attempting to provide in the PSE is that of a notebook and a calculator. When solving a simple arithmetic problem, one typically uses a notebook to write down some expressions and then uses the calculator to evaluate them. This can be viewed as a problem solving environment for solving simple arithmetic problems. Our vision is to develop a software environment that supports this notebook–calculator paradigm for PDE problem solving. Such an environment would have the software analog of a notebook and of course many specialized "calculators" that assist the user in his computations. The notebook would be an interactive environment which collaborates with the supporting calculators (editors) to provide an integrated environment where every editor would be aware of the existence and nature of the other editors.

The organization of the notebook is therefore key to the success of such an environment. We are developing the notebook concept on top of a computing kernel that supports many of the necessary abstractions. A separate report[HR92] discusses this kernel and other issues in the development of problem solving environments for PDE solving.

5.2 Symbolic Computing in a PDE Solving PSE

Symbolic processing is an integral part of a problem solving environment for PDE solving. In this section, we identify several situations where symbolic computing would be used.

We have applied symbolic computing so far in the pre–processing phase of the problem. Another application of symbolic computing in this stage would be in extracting qualitative characteristics (such as the smoothness of the input data) of the PDE problem. This information would be given as input to an expert system that provides advice on what method to use and what machine to use for solving this problem. We are currently developing a rule–based expert system called ATHENA[HRH+91] that does exactly this and also attempts to provide a priori information about the convergence properties of the problem.

Symbolic computing can also help in the important task of solving the PDE problem in parallel. For example, in the case of domain decomposition,[7] it is possible to attempt PDE operator–dependent domain decomposition where one would use the "degree of difficulty of solving a sub–problem" as a criteria in assigning the domains. For example, some problems have the property that they are nonlinear over a certain part of the domain and linear over another. Then, its possible to use a linear solver for the linear sub–problem and a nonlinear solver for the nonlinear sub–problem. Since the nonlinear solver is more expensive, the domain decomposer could then use more processors in that region than for other parts of the domain, thus speeding up the entire solution process.

[7]Domain decomposition is a technique used to solve a problem in parallel. The idea is to decompose the problem domain into several pieces and then solve the (sub–)problem on each piece in parallel. Of course, the amount of effective parallelism is dependent on the amount of communication that needs to take place between the processors. However, for the parallel solution of PDE problems, this technique is very effective; see [CHENH+91].

Another important use of symbolic computing is in the analysis of the computed solution. Identifying properties of the computed solution makes it possible to use that information when solving the problem again (for example, in the case of time dependent problems). Also, this information is useful for expert systems as they can apply the knowledge in making a priori predictions about similar problems.

Symbolic computing is also necessary at the global level in the proposed notebook–calculator paradigm PSE. That is, since the notebook manipulates symbolic statements and descriptions, it would have to be implemented as a symbolic computing system as well. We are currently considering the possibility of actually building this software on top of an existing symbolic computing system such as MAXIMA or MAPLE[Gro87]. However, there are certain limitations with this too; mainly due to the closed nature of these systems.

These applications of symbolic computing in a PDE solving PSE indicate that symbolic computing is indeed an integral and essential component of such a software system. However, the minimal infrastructure that currently exists for integrated symbolic–numeric computing is a severe limitation to this end.

6 Conclusions

We have described the use of integrated symbolic–numeric computing as it currently stands in the //ELLPACK environment. Also, we have indicated what our plans are for developing problem solving environments for solving PDE–based problems and the need for symbolic computing there.

We have found that the lack of infrastructure for integrated symbolic–numeric computing is a severe restriction when attempting to employ this form of computing. In the following paragraphs, we briefly identify some of the features that we would like to have in both symbolic and numeric systems.

One major difficulty in integrating symbolic and numeric computing is in the area of communicating data. We would like to see the development of some syntax and representation scheme that can easily be translated to/from symbolic and other systems. Such a language would allow computing systems to collaborate with each other and solve problems more effectively.

The closed–ness of symbolic computing systems is also a severe limitation. Unlike numerical computation facilities, symbolic computing systems have evolved into large user–level systems, rather than computational kernels. This is also a limitation with respect to integrated symbolic–numeric computing as this forces the client of a symbolic processor to pretend to be human. What we would like are software systems that have not only a user interface, but also a functionally equivalent programming interface.

Another difficulty with symbolic computing systems is their large size. It is unfortunate that one must load in an entire system when one only needs a small subset of its operations. It would be very useful if only the necessary operations could be loaded into an application program. Furthermore, this ability to limit what functionality is incorporated into an application could easily lead to the tighter coupling with the symbolic processor that we desperately need in order to implement some of the functionalities mentioned in the previous section.

The use of "foreign" systems as compute servers in numerical simulation environments is another essential feature of future PSEs. What we mean by a foreign system is a software system that exists outside of the PSE and which must be used as a "black box"; i.e., without having intricate understanding of its internals. The //ELLPACK environment now uses FIDISOL in this manner. We note that without the integrated symbolic–numeric computation capability, this would not have been possible due the need to compute Jacobians.

We patiently await the day when our wish list would be satisfied. But, in the meantime, we will attempt to build the next generation PDE solving PSE using ad hoc means to get around difficulties that arise due to the lack of supporting infrastructure.

7 Acknowledgements

Work on //ELLPACK is done by the members of the pellpack group within our CAPO (Computing About Physical Objects)[BHH+88] project. The following people have been or are also involved with the //ELLPACK project (in no particular order): Nikos P. Chrisochoides, H. C. Karathanasis, Panos N.

Papachiou, Meletis K. Samartzis, E. A. Vavalis, Ko-Yang Wang, Winnie Ng, Pelayia Varodoglu, Sang Bae Kim, Tunghsing Ku, Yue-Jinning Lian, and Po Ting Wu.

References

[BHH+88] C. Bajaj, C. Hoffmann, E. N. Houstis, J. T. Korb, and J. R. Rice. Computing about physical objects. In *Proceedings of the 12th World Congree on Scientific Computing*, volume 4, pages 642–684, New Brunswick, NJ, 1988. IMACS.

[CHENH+91] N. P. Chrisochoides, C. E. Houstis, P. N. Papachiou E. N. Houstis, S. K. Kortesis, and J. R. Rice. Domain decomposer: A software tool for mapping pde computations to parallel architectures. In R. Glowinski et al., editors, *Domain Decomposition Methods for Partial Differential Equations IV*, pages 341–357. SIAM Publications, 1991.

[Dew89] M. C. Dewar. IRENA: An integrated symbolic and numerical computation environment. In *Proceedings of the ACM-SIGSAM 1989 International Symposium on Symbolic and Algebraic Computation*, pages 171–180. ACM Press, 1989.

[DW90] Yaser Doleh and Paul S. Wang. SUI: A system independent user interface for an integrated scientific computing environment. In *Proceedings of the ACM-SIGSAM 1990 International Symposium on Symbolic and Algebraic Computation*, pages 88–95. Addison-Wesley, 1990.

[Gro87] Symbolic Computation Group. *Maple User's Manual.* University of Waterloo, Department of Computer Science, Waterloo, Canada, 1987.

[HR92] Elias N. Houstis and John R. Rice. Problem solving environments and methods for the development of PDE based applications on parallel machines. Technical report, Department of Computer Sciences, Purdue University, 1992. in preparation.

[HRC+90] E. N. Houstis, J. R. Rice, N. P. Chrisochoides, H. C. Karathanasis, P. N. Papachiou, M. K. Samartzis, E. A. Vavalis, Ko-Yang Wang, and S. Weerawarana. //ELLPACK: A numerical simulation programming environment for parallel MIMD machines. In J. Sopka, editor, *Proceedings of Supercomputing '90*, pages 97–107. ACM Press, 1990.

[HRH+91] E. N. Houstis, J. R. Rice, C. E. Houstis, T. S. Papatheodorou, and P. Varodoglu. ATHENA: A knowledge based system for parallel ELLPACK. In E. Diday and Y. Lechevallier, editors, *Symbolic – Numeric Data Analysis and Learning*, pages 459–467. Nova Science, 1991.

[Man90] SunOS Reference Manual. *Network Programming: Remote Procedure Call Programming Guide.* Sun Microsystems, Inc., Mountain View, CA, 1990.

[RB85] J. R. Rice and R. F. Boisvert. *Solving Elliptic Problems Using ELLPACK.* Springer–Verlag, New York, 1985.

[SG86] Robert W. Scheifler and Jim Gettys. The X window system. *ACM Transactions on Graphics*, 5(2):79–109, 1986.

[SSM85] W. Schönauer, E. Schnepf, and M. Müller. *The FIDISOL Program Package.* University of Karlsruhe, Karlsruhe, Germany, 1985.

[TMG77] Laboratory for Computer Science The MATHLAB Group. *MACSYMA Reference Manual, Version 9.* M.I.T., Cambridge, MA, 1977.

[Wan86] Paul S. Wang. FINGER: A symbolic system for automatic generation of numerical programs in finite element analysis. *Journal of Symbolic Computation*, 2:305–316, 1986.

[Wan88] Paul S. Wang. *An Introduction to Berkeley UNIX.* Wadsworth Publishing Co., Belmont, CA, 1988.

[Wan90] Paul S. Wang. Advances in integrating symbolic, numeric and graphics computing. In *IV-th International Conference on Computer Algebra and its Applications in Physical Research*, 1990.

[WHR92] S. Weerawarana, E. N. Houstis, and J. R. Rice. An interactive symbolic-numeric interface to parallel ellpack for building general pde solvers. to appear, 1992.

[Wol88] Stephen Wolfram. *Mathematica: A System for Doing Mathematics by Computer.* Addison–Wesley, 1988.

[WW89] S. Weerawarana and P. S. Wang. GENCRAY: A portable code generator for cray fortran. In *Proceedings of the ACM-SIGSAM 1989 International Symposium on Symbolic and Algebraic Computation*, pages 186–191. ACM Press, 1989.

[Zip90] Richard Zippel. Applying symbolic techniques to differential equations. Technical report, Department of Computer Science, Cornell University, 1990. (awaiting proper reference).

A Example FIDISOL Program Generated by the Symbolic Tool

```
OPTIONS.
      fidisol
      ilnsys=3
      llpara= .false.
      lltime= .true.
      xplot3d

EQUATION.
      fidisol

BOUNDARY.
      fidisol on x =  0.0
              on x =  1.0
              on y =  0.0
              on y =  2.0

GRID.
      20 x points
      20 y points

TRIPLE.
      fidisol

SUBPROGRAMS. +host

c     subroutine to define the pde system at inner grid points

      subroutine fdsu01 (t,x,y,u,ut,ux,uxx,uy,uyy,p,mt,mv,nk,nv)
      integer mt,mv,nk,nv
      real t
      real x(mv),y(mv)
      real p(mv,nk),u(mv,nk),ux(mv,nk),uxx(mv,nk),uy(mv,nk),uyy(mv,nk)
      real ut(mt,nk)
      integer i

      do 1001 i=1,nv
         p(i,1)=uxx(i,1)+uyy(i,1)+uy(i,3)
         p(i,2)=uxx(i,2)+uyy(i,2)-ux(i,3)
         p(i,3)=u(i,1)*ux(i,3)+u(i,2)*uy(i,3)-4*(uxx(i,3)+uyy(i,3))
 1001 continue

      return
      end

c     subroutine to define the boundary conditions

      subroutine fdsu02 (irand,t,x,y,u,ut,ux,uxx,uy,uyy,p,mt,mv,nk,nb)
      integer irand,mt,mv,nk,nb
      real t
      real x(mv),y(mv)
      real p(mv,nk),u(mv,nk),ux(mv,nk),uxx(mv,nk),uy(mv,nk),uyy(mv,nk)
      real ut(mt,nk)
      integer i

      goto (10,20,30,40) irand

 10   do 1002 i=1,nb
```

```
         p(i,1)=u(i,1)
         p(i,2)=u(i,2)
         p(i,3)=uy(i,1)-ux(i,2)+u(i,3)
 1002 continue
      return

 20   do 1003 i=1,nb
         p(i,1)=u(i,1)
         p(i,2)=u(i,2)
         p(i,3)=uy(i,1)-ux(i,2)+u(i,3)
 1003 continue
      return

 30   do 1004 i=1,nb
         p(i,1)=u(i,1)
         p(i,2)=u(i,2)
         p(i,3)=uy(i,1)-ux(i,2)+u(i,3)
 1004 continue
      return

 40   do 1005 i=1,nb
         p(i,1)=u(i,1)
         p(i,2)=u(i,2)
         p(i,3)=uy(i,1)-ux(i,2)+u(i,3)
 1005 continue
      return
      end

c     subroutine to define the jacobian matrices of the pde
c     system at inner grid points

      subroutine fdsu03 (iequ,icom,t,x,y,u,ut,ux,uxx,uy,uyy,pu,put,pux,p
     .uxx,puy,puyy,mt,mv,nk,nv)
      integer iequ,icom,mt,mv,nk,nv
      real t
      real x(mv),y(mv),pu(mv),put(mv),pux(mv),puxx(mv),puy(mv),puyy(mv)
      real u(mv,nk),ut(mv,nk),ux(mv,nk),uxx(mv,nk),uy(mv,nk),uyy(mv,nk)
      integer i
      goto (100,200,300) iequ
 100  goto (110,400,130) icom
 110  do 1006 i=1,nv
         puxx(i)=1
         puyy(i)=1
 1006 continue
      go to 400
 130  do 1007 i=1,nv
         puy(i)=1
 1007 continue

 200  goto (400,220,230) icom
 220  do 1008 i=1,nv
         puxx(i)=1
         puyy(i)=1
 1008 continue
      go to 400
 230  do 1009 i=1,nv
         pux(i)=-1
 1009 continue

 300  goto (310,320,330) icom
 310  do 1010 i=1,nv
```

```
        pu(i)=ux(i,3)
 1010 continue
      go to 400
 320  do 1011 i=1,nv
        pu(i)=uy(i,3)
 1011 continue
      go to 400
 330  do 1012 i=1,nv
        pux(i)=u(i,1)
        puxx(i)=-4
        puy(i)=u(i,2)
        puyy(i)=-4
 1012 continue

 400  continue
      return
      end

c     subroutine to define the jacobian matrices of the boundary
c     conditions at boundary grid points

      subroutine fdsu04 (irand,iequ,icom,t,x,y,u,ut,ux,uxx,uy,uyy,pu,put
     .,pux,puxx,puy,puyy,my,mv,nk,nb)
      integer irand,iequ,icom,mt,mv,nk,nb
      real t
      real x(mv),y(mv),pu(mv),put(mv),pux(mv),puxx(mv),puy(mv),puyy(mv)
      real u(mv,nk),ut(mv,nk),ux(mv,nk),uxx(mv,nk),uy(mv,nk),uyy(mv,nk)
      integer i
      goto (1000,2000,3000,4000) irand
 1000 goto (1100,1200,1300) iequ
 1100 goto (1110,5000,5000) icom
 1110 do 1013 i=1,nb
        pu(i)=1
 1013 continue
      go to 5000

 1200 goto (5000,1220,5000) icom
 1220 do 1014 i=1,nb
        pu(i)=1
 1014 continue
      go to 5000

 1300 goto (1310,1320,1330) icom
 1310 do 1015 i=1,nb
        puy(i)=1
 1015 continue
      go to 5000
 1320 do 1016 i=1,nb
        pux(i)=-1
 1016 continue
      go to 5000
 1330 do 1017 i=1,nb
        pu(i)=1
 1017 continue

      go to 5000

 2000 goto (2100,2200,2300) iequ
 2100 goto (2110,5000,5000) icom
 2110 do 1018 i=1,nb
        pu(i)=1
```

```
1018 continue
     go to 5000

2200 goto (5000,2220,5000) icom
2220 do 1019 i=1,nb
        pu(i)=1
1019 continue
     go to 5000

2300 goto (2310,2320,2330) icom
2310 do 1020 i=1,nb
        puy(i)=1
1020 continue
     go to 5000
2320 do 1021 i=1,nb
        pux(i)=-1
1021 continue
     go to 5000
2330 do 1022 i=1,nb
        pu(i)=1
1022 continue

     go to 5000

3000 goto (3100,3200,3300) iequ
3100 goto (3110,5000,5000) icom
3110 do 1023 i=1,nb
        pu(i)=1
1023 continue
     go to 5000

3200 goto (5000,3220,5000) icom
3220 do 1024 i=1,nb
        pu(i)=1
1024 continue
     go to 5000

3300 goto (3310,3320,3330) icom
3310 do 1025 i=1,nb
        puy(i)=1
1025 continue
     go to 5000
3320 do 1026 i=1,nb
        pux(i)=-1
1026 continue
     go to 5000
3330 do 1027 i=1,nb
        pu(i)=1
1027 continue

     go to 5000

4000 goto (4100,4200,4300) iequ
4100 goto (4110,5000,5000) icom
4110 do 1028 i=1,nb
        pu(i)=1
1028 continue
     go to 5000

4200 goto (5000,4220,5000) icom
4220 do 1029 i=1,nb
```

```
        pu(i)=1
 1029 continue
      go to 5000

 4300 goto (4310,4320,4330) icom
 4310 do 1030 i=1,nb
        puy(i)=1
 1030 continue
      go to 5000
 4320 do 1031 i=1,nb
        pux(i)=-1
 1031 continue
      go to 5000
 4330 do 1032 i=1,nb
        pu(i)=1
 1032 continue


 5000 continue
      return
      end

END.
```