Purdue University

Purdue e-Pubs

Department of Computer Science Technical Reports

Department of Computer Science

1983

Software Parts for Elliptic PPE Software

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

Report Number:
83-448

Rice, John R., "Software Parts for Elliptic PPE Software" (1983). *Department of Computer Science Technical Reports.* Paper 367.
https://docs.lib.purdue.edu/cstech/367

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries.
Please contact epubs@purdue.edu for additional information.

SOFTWARE PARTS FOR ELLIPTIC PDE SOFTWARE

John R. Rice
Computer Science
Purdue University

(For the IFIP WG2.5 Conference in Söderkoping, Sweden - August 1983)
PDE SOFTWARE: Modules, Interfaces and Systems

CSD-TR 448
July 1, 1983

ABSTRACT

We examine the question of whether very high level elliptic problem solvers can be built (in theory or in practice) from a collection of software parts. In theory the answer is yes, in current practice the answer is no, because most of the required software parts are missing. The "levels" of software parts needed are identified and their contents outlined. There are isolated collections of parts currently available (e.g. the high level problem solving modules in ELLPACK, the low level vector processing routines in the BLAS), but their percentage of those needed is low, perhaps 10-20 percent. A set of priorities for creating the needed software parts is given and sets of software parts in the area domain processing are considered in detail.

CONTENTS

1. Introduction
2. The Ideal
3. The Virtual Machine Parts
4. Levels of Software Parts
    A. Basic operations
    B. Intermediate facilities
    C. High level modules
5. Prospects and Priorities
6. Software Parts for Domain Processing

This work supported in part by DOE contract DE-AC01-ER81-01997 and ARO contract DAAG29-83-K-0026.

SOFTWARE PARTS FOR ELLIPTIC PDE SOFTWARE

John R. Rice
Computer Science
Purdue University
W. Lafayette, Indiana 47907, USA

1. INTRODUCTION

The establishment of a software parts technology has been proposed for some time now, see [Rice, 1979], [Comer et al, 1980] and [Wasserman and Gutz, 1982]. A major effort incorporating this concept is the US Department of Defense STARS program, see [Druffel and Riddle, 1983] and [Batz et al 1983] for more details. The long term goal of this technology is to (a) dramatically reduce the cost of software development by reusing high quality software parts and (b) to have the software parts "standardized" for particular subdisciplines so as to form an informal, but tacitly standardized, "linqua franca" for software construction.

There are three steps to the creation of a set of good software parts. The first is to design the set; it requires considerable experience and good judgement to define a set of parts which is general enough to be useful, natural in its functions, and consistent internally and externally. The second step is to implement the set of parts and test the design. This step is more than an order of magnitude move work than the first. The third step is to refine the design based on the testing and to make sure that all the parts are of high quality (reliable, robust, efficient, well documented, accurate, etc.) The third step is again an order of magnitude more effort than the second. This ever escalating amount of effort required is the reason that many attempts at software parts have failed; not enough effort and talent was invested.

The purpose of this paper is to discuss the progress and potential for software parts needed for the construction of elliptic PDE software. Most of the required sets do not exist now and we give our view of the priorities and difficulties in creating them. We finally conclude with somewhat more detailed analysis of the parts sets for domain processing.

2. THE IDEAL

We first present an ideal hierarchy for software construction, see Figure 1. The goal is to use parts from each level as the major components in building parts at the next higher level. There would, of course, be some general purpose algorithmic code used as well. In an ideal world, one merely replaces the bottom level of parts as one moves software from machine to machine. The levels of parts defined are:

Virtual Machine Parts. We include a standard algorithmic language here (e.g., Fortran or Ada) plus parts for operations that normally must be implemented in a machine dependent manner. This set of parts must contain all the functionality needed for the higher levels.

Basic Operations. These parts perform standard, relatively simple operations. Examples are: matrix multiplication, matching two character strings, evaluating a function for an array of arguments, finding the maximum element of an array.

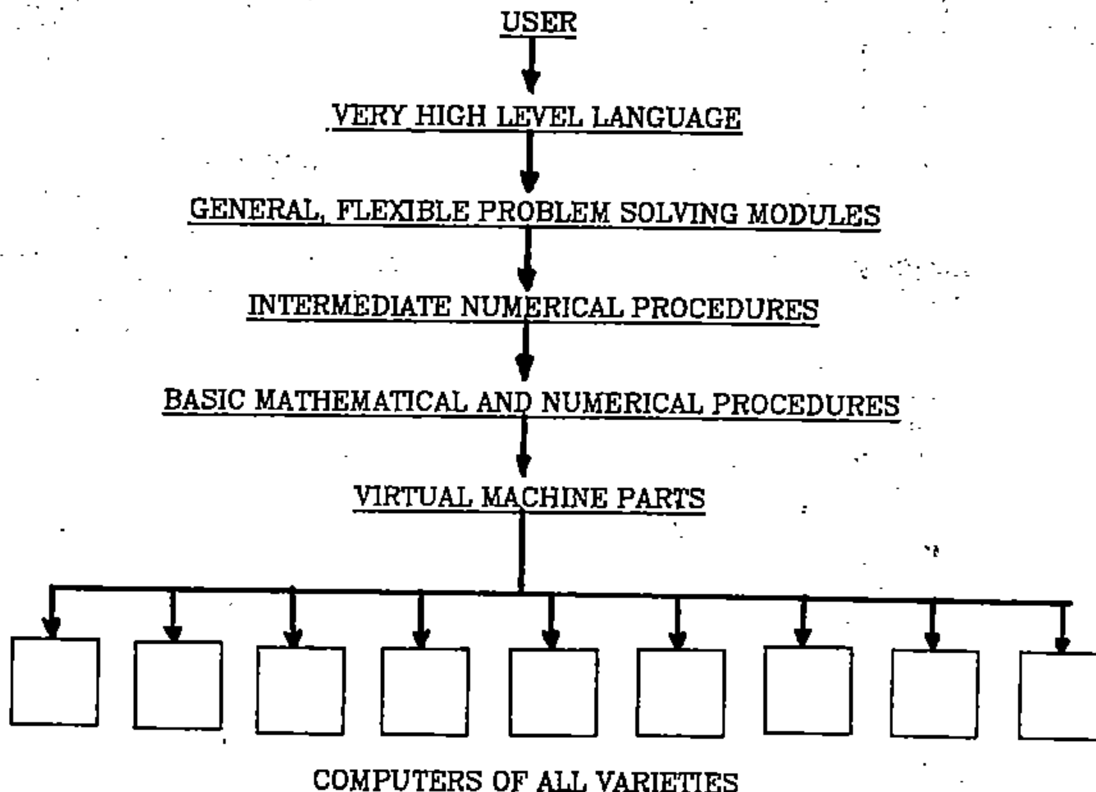


Figure 1. The ideal hierarchy in creating very high level PDE solving systems upon layers of increasingly powerful software parts.

Intermediate Procedures. These parts perform somewhat more complicated, but still fairly standard, tasks. Examples are: tridiagonal linear system solver, intersection of a straight line and parameterized curve, evaluation of a B-spline and its derivatives on a standard domain, numerical integration on standard domains.

General Problem Solvers. These parts include complex algorithms for a general class of problems. Examples are: sparse matrix method for a linear system, integration over a general two dimensional domain, Rayleigh-Ritz discretization of an elliptic operator on a partition of a domain, computing all eigenvalues of a band matrix.

Very High Level Languages. These are languages (systems) which allows one to state a problem and obtain its solution with very minimal effort. Examples are: ELLPACK, MATLAB, SAS, PROTRAN.

3. THE VIRTUAL MACHINE PARTS

We list 7 categories of parts and facilities that define a virtual machine for PDE software. Table 1 lists them along their status and the extent that Fortran provides the facilities. The categories listed in Table 1 are nearly self explanatory. By data structures we mean simple things like stacks, and records (a la Pascal) and by algorithm construction we mean things like arithmetic on numerical variables, control (IF, looping, etc.), subprograms with separate compilation

and declarations.

The conclusion to be drawn from Table 1 is that the algorithmic language and software parts needed to define the virtual machine are either already available or rather well identified. However, to my knowledge, no one has presented a detailed specification of the entire set of parts. This is a task that needs to be done; one might view the current efforts of the Fortran standards committee, X3J3, as an attempt to incorporate all these facilities within Fortran.

There is a danger in having the virtual machine parts incorporated within a complex language. One becomes dependent upon the compiler writer for the implementation of the facilities. The perennieal inefficiency of Fortran I/O illustrates the difficulty; it is inconvenient to extract the facility from the language and almost impossible to improve it within the language. One would probably prefer a leaner language that allowed parts to be easly incorporated, exchanged and rewritten.

Table 1. The 7 categories that define the virtual machine. The column Fortran contains a brief comment on the extent to which the Fortran language provides the required facilities.

Parts Category	Status	Fortran
Array Operations	Well identified, some sets of parts already exist	Nil
Functions	Usually part of the algorithmic language	OK
Graphics	Becoming standard, parts well identified	Nil
Character Operations	Often part of the algorithmic language, parts well identified	Poor (Fortran 66)
Text and Data I/O	Often part of the algorithmic language, often grossly inefficient	Inefficient
Data Structures	Often part of the algorithmic language, parts well identified	Poor, only arrays
Algorithm construction	The essence of the algorithmic language, well identified facilities	OK

4. LEVELS OF SOFTWARE PARTS

We list additional sets of parts that are required to build elliptic PDE software. Three "levels" are identified: *basic*, *intermediate* and *high level*. The dividing line between these levels is not sharply defined, but the categorization is useful.

4.A. Basic Operations

These parts are somewhat of the nature of utilities that can be built out of virtual machine parts. We list seven sets of such parts along with examples, six of these sets also contain parts at the virtual machine level:

Array Operations	Matrix multiply, Transpose, Array multiply, Norms, Elementary elimination steps, Permutations, Evaluation of array expressions.
Functions	Array arguments, Arrays of functions, Tensor products of arrays of functions, Specific sets of polynomials in 1, 2 and 3 variables.
Graphics	Plots of $y=f(x)$, Contour plots of $f(x,y)$, Contour sections of $f(x,y,z)$, Domain plots.
Characters	Matching word from list, Conversion of expressions from infix to Polish and back, Aliasing in tables and lists, Arrays of messages.
I/O	Tables of arrays, Global control of output (put switches on generation and destination of output), Tables of functions. Zero structure of arrays.
Data Structures	Sparse matrix representation of matrices e.g., (coef, idcoef) vectors, (a_{ij}, i, j) vectors, (A, row_mark, col_mark) vectors, Transformations between array representations, Basic symbol table (name, id, value), Storage allocation stack
Differentation	Local univariate derivative estimates of functions, Creation and application of finite difference stencils.

4.B. Intermediate Facilities

These parts are more complex and more specific to PDEs than the basic operation, the four sets include some of the simpler numerical solution methods. As above, we list the sets of parts along with examples.

Array Operations	Tridiagonal solvers, Gauss elimination for standard representations, FFTs, Inverses.
Domain Processing	Definition facilities, Gridding, Triangulation, Inside/outside determination, Point location (relative to a grid, triangulation, etc.), Line and curve itersection, Normal to a curve or surface, Mappings of standard domains.
Integration	Over standard domains, Along parametric curves, Handling of standard singularities.
Basis Functions	Splines, Piecewise polynomials, Triangular elements, Compositions and transformations.

4.C. High Level Modules

These parts solve particular elliptic problems or carry out a major step in the solution. The ELLPACK system [Rice and Boisvert, 1983], for example, divides these into categories such as *Discretization* (of an elliptic problem), *Solution* (of a linear system), *Indexing* (transformation of a linear system) and *Triples* (complete solution of an elliptic problem). At this time, ELLPACK has 58 modules identified at the user level; 51 of these are high level, 6 are intermediate facilities and one is a dummy. ELLPACK also contains numerous intermediate level facilities in its I/O and domain processing programs. We list the names

of some of the ELLPACK modules to illustrate the variety of software parts at this level:

5 POINT STAR	LINPACK BAND	FFT 9 POINT
HODIE HELMHOLTZ	JACOBI CG	DYAKANOV CG
NESTED DISSECTION	SPARSE LU PIVOTING	MULTIGRID MG00
SET U BY BLENDING	REMOVE BICUBIC BC	FISHPAK HELMHOLTZ

It is easy to identify another 20-30 modules that would be appropriate to include in ELLPACK; this suggests that there are at least 100 high level modules for elliptic PDEs that are interesting and important.

5. PROSPECTS AND PRIORITY

A start has been made on a set of software parts adequate for elliptic PDE software, but it is a small one. The number of parts required in even a very narrow area is surprisingly large; recall that there are 38 BLAS and 52 programs in EISPACK. There are literally hundreds of software parts yet to be written which are relevant to elliptic PDE software.

All of the functionality needed in the sets of software parts is in existing programs, it just has not been isolated, organized, parameterized and made robust. The amount of effort needed to design and implement a set of software parts has always been surprising. The design of the BLAS took over four years and while this was a low level activity, this does show that one cannot sit down and make a finished design in an afternoon or two. It takes considerable experimentation and reflection to arrive at natural names, a set of parts that is natural to use, with neither too few or too many parts.

The past experience shows that useful sets of software parts can be constructed and that they are expensive. The return on the investment far exceeds the cost, so the creation of software parts is economically the right thing to do. One economic difficulty is that there is no straight forward mechanism for the beneficiaries of software parts to defray the initial investment in creating the parts.

A software parts technology will not arrive full blown; this technology can be adopted piecemeal. Indeed, the current ideas about modular programming, etc. are naturally condusive to a software parts technology. Given that one can proceed piecemeal, the natural question is:

What are the priorities in developing software parts for PDE software?

Most people will single out array and vector operations at all levels, as the area with the highest priority. I agree with this assessment; the primary computational bottleneck is in manipulating and solving linear systems of equations. Furthermore, people already are trained to think in terms of vectors, matrices, etc., so there is an existing natural framework within which to develope these parts.

I group the following sets of parts as next most important:

Functions (Basic operations)
Basis Functions
Domain Processing
Differentiation

Their importance stems from the following: (a) the evaluation and manipulation of functions is often the second most computationally expensive part of solving

elliptic problems (b) the algorithmic language facilities for functions, domain processing and differentiations are usually primitive so that a lot of the obscurity in PDE software comes from these sources. Right behind these sets of parts I place

I/O Facilities
Graphics

These are not particular to PDE software, so one can hope that other groups will develope most of the software parts for these two areas.

I do not discuss further software parts for array and vector processing because there is aleady so much activity in this area. I note that basis functions is an area where there is a lot of natural scientific notation and background and thus one already has a framework within which to work. This appears to be an "easy" set of software parts to create. Domain processing is an area where one draws pictures easily and writes corresponding programs with great difficulty. There is no standard framework here and thus this area provides a test of our capability to create a useful set of parts in an "unstructured" or "novel" area. Furthermore, some of the basic processes are computationally complex, so this appears to be a "hard" set of software parts to create. I believe that creating parts for differentiation is somewhere between basis functions and domain processing in difficulty. Generality in sets of parts for basis functions or differentiation requires domain processing capabilities, but useful sets of such parts could be built for specific classes of domains and partitions without explicit domain processing facilities. I discuss in some detail sets of software parts for domain processing.

6. SOFTWARE PARTS FOR DOMAIN PROCESSING

The first task in creating a set of software parts is to define the conceptual framework. For a specific set, this means precise formal definitions of a number of terms, objects, procedures, etc. However, there is also a need for closely related specific sets to be in a common, but more intuitive, framework. Consider one set of parts for the triangularization of general two dimensional domains and another set for rectangular grids in three dimensional boxes. There might be no overlap in specific software or facilities, yet a software parts technology needs to have these two sets closely related at the conceptual level.

The principal objects in the conceptual framework are:

1. Domain. This is a general geometric entity such as a disk in the plane or a box in 3-space. Domains have boundaries, interiors and exteriors.
2. Elements. This is one of a small set of standard domains such as triangles, boxes, boxes with one curved side, etc.
3. Partition. This is a collection of elements which covers a domain and which overlaps only on element boundaries.

To simplify the discussion, we assume that there are just two domains of interest: The **problem domain**, associated with the elliptic PDE, and the **frame**, a domain that contains the problem domain. All processing is done within the frame, for some parts sets the frame and problem domain may be the same. There is a natural hierarchy of subsets of domains, namely:

interior/exterior
boundary = faces + edges + vertices

916

The elements are related to standard representations, one for each element type in the partition. A standard representation may be parameterized such as the case of the unit square with the (1,1) vertex cut off by a curve. There is a "simple" mapping between each actual element and its standard representation. Thus, a "simple" mapping will preserve geometric features (e.g. edges and vertices) and not unduly distort shape or area. The mapping may be more than just linear; one of the features of a specific set of parts is how the relevant mappings are made.

The general functionality of domain processing parts sets consists of

Creation:	Objects are initially defined
Information:	Itemized or collective information is provided about one or a group of objects.
Operations:	Manipulate the objects (e.g. split or merge two elements), provide new information (e.g. where is the inside of a domain or what is the map between a given element and its standard representation).

Many of these functions can be defined across all sets of parts for domain processing, most of them can be defined with natural analogy across all sets of parts.

The partition has certain rules about the nature of the elements (it might require uniform size or that no more than five elements meet at a point), and about neighbors (it is common to assume that a point which is a vertex of one element is a vertex of all elements containing it). Specific sets of parts will incorporate specific sets of rules to define the class of partitions involved.

The conceptual framework for domain processing should encompass the following provisions and specific instance.

<u>Domains</u>:

(a) 1, 2 and 3 dimensional.

(b) Defined by a set of boundary pieces or as unions/intersections from a small catalog of shapes or by the truth of a logical function of position.

(c) Interior defined by orientation or connectivity to a specified point.

<u>Boundaries</u>:

(a) Defined by a set of points (with tacit linear interpolation) or by an ordered set of pieces (faces, edges or vertices).

(b) Boundary pieces may be defined parametrically or by an implicit standard (e.g. linear interpolation).

<u>Elements</u>:

(a) May be based on rectangles (or boxes), on triangles (or simplices), or on stencils (as in finite differences).

(b) A wide variety of irregular elements are included to accommodate general domains.

<u>Partitions</u>:

(a) Includes completely uniform partitions or somewhat irregular sizes.

(b) Rules for neighbors give some flexibility in the partition.

(c) Allow multiple partitions of a given domain.

<u>Creation:</u>

(a) Can define large numbers of objects at once or add a new object to a given collection.

(b) All types of objects can be created and named.

<u>Information:</u>

(a) Provide various levels of information about a single element, a group of element (or subdomain) or the whole problem.

(b) Provide functions or notations for specific information (e.g. location, vertices, type, neighbors) that can be used freely in composition.

(c) Provide for a reference grid within the frame.

<u>Operations:</u>

(a) Provide considerable power in splitting, merging, discarding, mapping, etc.

Within this conceptual framework, there will be several, even many, sets of parts for particular choices of element shape, domain structure, etc. Figure 2 shows element data structures for quadrilateral, point and triangular elements which fit into this framework. For each case, the "basic" standard domain is shown along with one "variation" standard domain needed to handle boundary/domain interaction. The compass point labeling is useful for visualization but we propose a more uniform labeling as follows. Each element has a name (normally a numerical index), then each component of an element (face, edge or vertex) is identified by the elements to which it belongs. For the grid stencil element this just gives a label to each point; for the other element types one has (the element label itself is not repeated in the components). Figure 3 shows pieces of two domains with partitions; the names of the objects are listed:

Element	16	(basic quadrilateral element)
Edges	17,8,15,24	
Vertices	17+9+8, 8+7+15, 15+23+24, 24+25+17	

The plus sign + is actually a set intersection operator. A standard ordering is used for the components of an object; here we chose starting with EAST and proceeding clockwise.

Element	92	(one vertex cut off a triangle)
Edges	107,91,93, Null	
Vertices	107+106+105+90+91, 91+78+79+80+93, 93,107	

The second example assumes that the element is on the boundary of the domain and there are no outside elements. We assume that element components can also have individual names, thus vertex 87 might equal 107+106+105+90+91+92.

A set of parts thus is based on specific choices of

1. Domain representations allowed

2. Standard element domains
3. Operations on elements and domains

The third group of choices defines both the general nature of the partition as well as its iteraction with the domain. Each set consists of a large number of parts; we give a parts list (see Table 2) with about 30 generic items. We expect actual sets to have more parts. For example, the GRIDPACK system [Brandt and Ophir, 1983] has 56 "parts" that are identified for general uses; presumably there are many more used to build GRIDPACK which that system does not provide to the user. On the other hand, some GRIDPACK parts are for basis function rather than pure geometry as discussed here.

REFERENCES

Batz, J., Cohen, P., Redwine, W. (1983), The STARS program, IEEE Computer, Vol. 16, to appear.

Brandt, A. and Ophir, D. (1983), GRIDPACK, these proceedings.

Comer, D., Rice, J.R., Schwetman, H. and Snyder, L. (1980), Project Quanta, CSD-TR 300, Computer Science, Purdue University.

Druffel, L. and Riddle, W. (1983), The STARS program, IEEE Computer, Vol. 16, to appear.

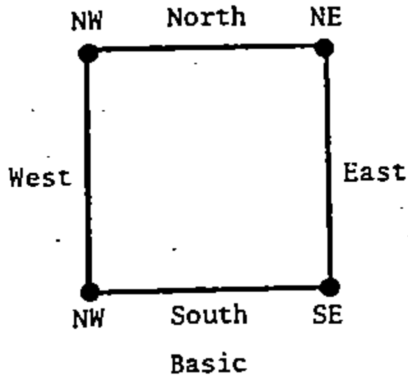
Rice, J.R. (1979), Software for Numerical Computation, in *Research Direction in Software Technology* (P. Wegner, ed.) M.I.T. Press, pp. 688-708.

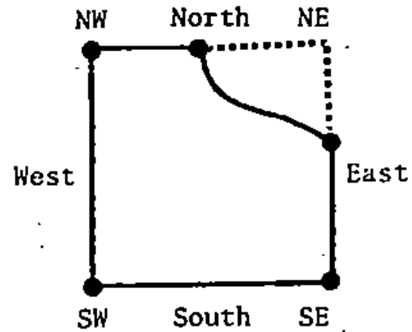
Rice, J.R. (1982), Numerical computation with general two dimensional domains, CSD-TR 416, Computer Science, Purdue University.

Rice, J.R., and Boisvert, R.F. (1984), *Solving Elliptic Problems Using ELLPACK*, Springer-Verlag, New York.

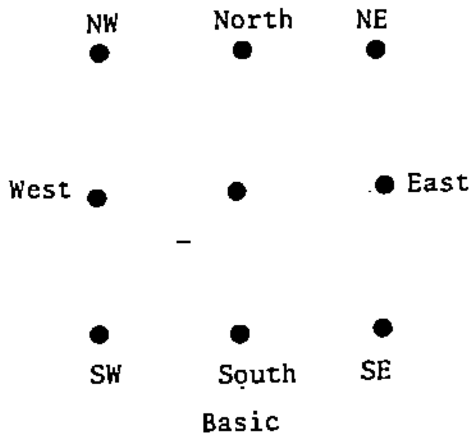
Rice, J.R., Gear, C.W., Ortega, J.M., Parlett, B., Schulz, M., Shampine, L.F. and Wolfe P. (1980), Numerical Computation, in *What Can be Automated (COSERS)*, (B. Arden, ed.), M.I.T. Press, pp. 51-136.

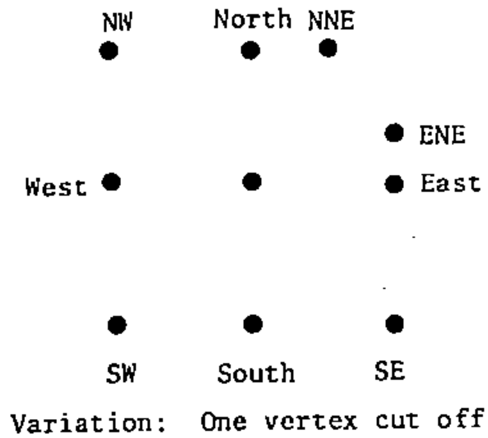
Wasserman, A.I. and Gutz, S. (1982), The future of programming, Comm. ACM., Vol. 25, pp. 196-206.



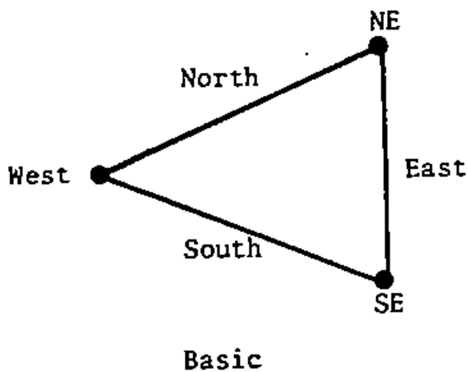


STANDARD QUADRILATERAL ELEMENTS





STANDARD GRID STENCIL ELEMENTS



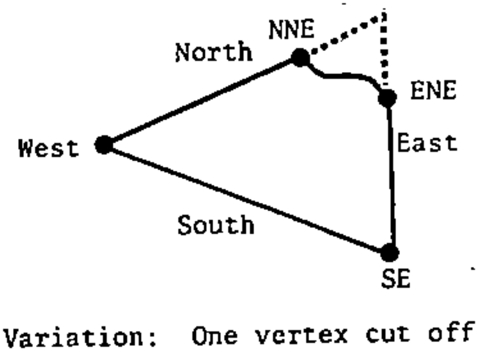


Figure 2. Example standard domains for three element types. The compass point labeling is for visualization purposes.

Table 2. Generic parts list for domain processing. Most of these software parts should be present in any set for domain processing.

NAME	BRIEF DESCRIPTION
Informational	
DISPLAY ELEMENT(i)	Basic information about the element
DISPLAY FACE(i)	Basic information about the face
DISPLAY EDGE(i)	Basic information about the edge
DISPLAY VERTEX(i)	Basic information about the vertex
DISPLAY BOUNDARY(i)	Basic information about the boundary piece
ID OF (x,y)	Locates elements containing the boundary piece
Analogs of the following functions are needed for faces, edges and vertices also	
INOUT(i)	True if i is inside domain
BOUNDARY(i)	True if i is adjacent to the boundary
TYPE(i)	ID of corresponding standard element domain
FACES(i)	Produces list of faces
EDGES(i)	Produces list of edges
VERTICES(i)	Produces list of vertices
LOCATION(i)	Produces standard (x,y) point in element
SIZE(i)	Approximate volume/area of element
DIAMETER(i)	Approximate diameter of element
FRAME(i)	Location relative to frame
The following functions are for boundaries	
START(i)	(x,y) coordinates of initial point
STOP(i)	(x,y) coordinates of final point
PIECES(i)	Number of pieces of boundary
Creative	
The following parts have complex input not specified in detail here.	
DOMAIN	Define a domain
PARTITION	Create a set of elements that form a partition
BOUNDARY	Create a boundary
STANDARD ELEMENT(t)	Create a instance of a standard element domain of type t
ELEMENT	Create an actual element as prescribed.
Operational	
ADD ELEMENT	Add element to a partition
DISCARD ELEMENT	Discard element from partition
ADD PIECE	Add piece to a boundary
DISCARD PIECE	Discard piece from a boundary
MERGE ELEMENT	Create one element out of two
MERGE PIECE	Create one piece out of two
SPLIT ELEMENT	Create new elements from old using boundary intersections
INSERT BOUNDARY	Compute all boundary/partition intersection information
SET INSIDE	Specify interior of domain
MAP INTO	Create mapping of standard element to actual element
MAP OUTOF	Create mapping of actual element to standard element

23 24 25
15 16 17
7 8 9

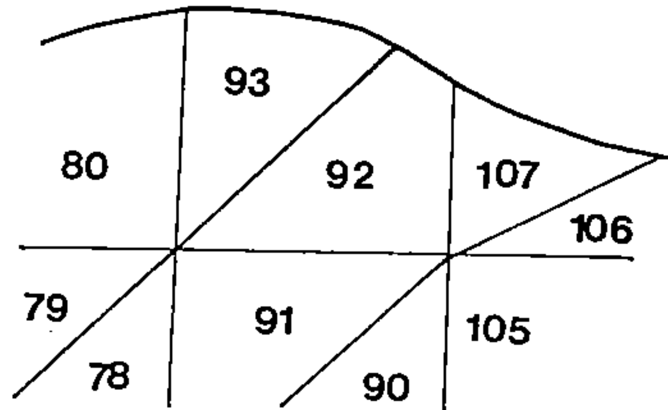


Figure 3.. Two examples of domains partitioned into elements. The numbers are the names of the elements.