Title
A NETWORK DEFINITION AND SOLUTION OF SIMULATION PROBLEMS

Author
Anderson, J.L.

Publication Date
1987-09-01

LBL-21522
UC-32

Lawrence Berkeley Laboratory
UNIVERSITY OF CALIFORNIA

APPLIED SCIENCE DIVISION

A Network Definition and Solution of Simulation Problems

J.L. Anderson

September 1987

APPLIED SCIENCE DIVISION

Prepared for the U.S. Department of Energy under Contract DE-AC03-76SF00098

DISCLAIMER

LBL - 21522

A NETWORK DEFINITION AND SOLUTION OF SIMULATION PROBLEMS

Jeffrey L. Anderson

Simulation Research Group
Applied Science Division
Lawrence Berkeley Laboratory
University of California
Berkeley, California 94720

September 1986

ABSTRACT

Most software tools currently used to solve simulation problems are monolithic FORTRAN programs. Such programs retard advances in modeling by being inflexible, hard to modify, and hard to use. Simulation programs can be defined in terms of a network of physical objects and links. The network concept allows modeling algorithms to be defined as abstract data types called objects. Objects separate the details of a model from the interface needed to use it in a simulation. This allows a network simulation program to be more flexible and more easily modified than traditional simulation programs. A network simulation program kernel that provides basic tools for simulations is described. The kernel provides a language designed to define simulation networks and a solution technique for simulation problems that works well with network definitions. This method is analyzed and shown to be faster than traditional solution methods by a significant factor. A network simulation problem kernel providing data abstraction should accelerate advances in modeling by providing facilities to create flexible, easily modified simulation programs.

This work was supported by the Assistant Secretary for Conservation and Renewable Energy, Office of Building and Community Systems, Building Systems Division of the U. S. Department of Energy under Contract No. DE-AC03-76SF00098.

INTRODUCTION

While simulation was one of the first computer applications to be explored, it has been among the last to advance with improvements in software technology. Most current large simulation systems are written in FORTRAN; these simulation programs often involve more than 100,000 lines of FORTRAN tied together in a rigid structure of subroutines and common blocks. The systems are inflexible; often even simple changes can require expert programmers and several months of work.

The immense difficulty associated with modifying large FORTRAN programs has a number of adverse effects on the simulation field. Algorithms modeling a phenomenon can not be easily varied or updated as models improve. It is exceedingly difficult to add models of new phenomena that were not considered when the original program was written. Communications between modelers at different sites can be severely restricted. A subroutine written to model a phenomenon for one FORTRAN model is rarely easily inserted into a different FORTRAN model. Even worse, local updates to a model can make versions of the same original FORTRAN program incompatible.

Many of these problems can be solved by introducing an object-oriented kernel system for simulations. A *simulation kernel* provides basic tools for defining and solving simulations without placing many restrictions on the algorithms used in the simulation [PLA85]. This paper discusses the Simulation Problem Analysis Kernel, SPANK. SPANK was designed as a prototype to guide the development of more complete and robust kernel systems.

The SPANK system provides tools to define and solve a simulation problem that is defined in terms of a *network.* A network is a group of objects that model individual phenomena linked together to model some larger problem. SPANK builds upon earlier simulation network software developed by Levy and Low [LEV83] and later by Sowell, Taghavi, Levy and Low [SOW84]. Other groups have also built simulation programs that use approaches similar to networks to define problems [KLE76]. SPANK gains many of its advantages over earlier simulation programs by being an object-oriented system that provides data abstraction [STR84] for objects in the network.

Data abstraction is the concept of separating the implementation details of a subprogram from its interface. In a network simulation, objects in a network can be linked together to form a problem without knowing any details of the method used by the objects to model a particular phenomenon. Conversely, the internal details of the object can be changed without affecting the way in which the object is linked into a network to define a problem.

By providing data abstraction, SPANK accrues many advantages over traditional programs. The details of the implementation of an object are independent of the problem specification (but, unfortunately, not independent of the solution

methods in the current implementation). Therefore, objects can be modified and new objects introduced without concern for any other part of the system. Different sites can more easily share developments; all that need be done is to share definitions for new types of objects. These abilities will allow simulation techniques to evolve more quickly than they have in the past.

SPANK also examines a relatively new solution technique for simulation problems that are defined in terms of networks. The solution technique is found to be considerably faster than more traditional methods that could be applied to simulation problems.

This report begins by providing a general overview of the idea of defining a simulation problem in terms of a network structure in section 2. Section 3 takes a closer look at the implementation of a simple language that can define simulation networks. Sections 4 and 5 are concerned with the numerical solution techniques implemented in SPANK. Instead of using more traditional solution techniques for numerical problems, SPANK implements a new technique that is naturally applied to simulations defined as networks. The relative merits of this new technique, compared to more traditional solution methods, are examined in section 6. Section 7 provides a complete example of a simulation problem implemented with SPANK. Finally, section 8 examines some shortcomings of SPANK and further developments that could enhance the SPANK system.

The SPANK kernel has been successfully implemented and used to define several complex simulations in the building energy analysis field. It appears to be much easier to use than traditional simulation programs while providing greater flexibility for development of new simulation techniques. Hopefully, this project can serve as the basis for more advanced simulation kernels that will find widespread use in all simulation communities.

2. OVERVIEW OF NETWORK APPROACH TO SIMULATION

All problems in SPANK are viewed as networks of physical objects linked together by physical connections. This makes it simple to define many physical problems in SPANK. This section examines the idea of a network definition of a simulation problem. Certain solution techniques for simulation problems are most naturally implemented on simulation problems defined as networks. Such a solution technique is used in SPANK; an introduction to this solution technique is also provided here.

2.1. Network Terminology and Depiction

For the purpose of this paper, a network is a hypergraph consisting of a set N of nodes $\{n_1, n_2, ..., n_n\}$, a set E of hyperedges $\{e_1, e_2, ..., e_n\}$, and a function ϕ from the set E to the set of all non-null subsets of N. The nodes represent physical objects in a simulation problem and are also referred to as objects. A hyperedge, also called a link, is a structure that connects one or more nodes together [BER73]. The hyperedges may be either directed or undirected; a directed

hyperedge e distinguishes a single element of the set $\phi(e)$ as the source node. All other nodes in $\phi(e)$ are destination nodes of the directed hyperedge. Throughout the paper, depictions of hypergraphs will have directed hyperedges marked with an arrow pointing to the destination nodes; a hyperedge without an arrow is assumed to be undirected.

2.2. Objects

A physical object is modeled by one or more objects in the simulation network. An object is composed of two related parts, an equation and a set of interface variables. The interface variables represent the set of unknowns in the object's equation. The equation defines a functional equality that must be satisfied by the interface variables.

A simple example of an object from the building energy simulation field is a mass collector, a device in a ventilation system that has two input ducts which merge into a single output duct. The mass of the combined output air flow from a mass collector must equal the sum of the masses of the input flows. An object modeling a collector has three interface variables: *mass_out*, *mass_in*1, and *mass_in*2. The equation defined between these interfaces is:

$$mass\_out = mass\_in1 + mass\_in2$$

An object can express only a single, albeit arbitrarily complex, equation between its interfaces. A physical object normally requires several equations to be satisfied by some physical parameters. Several network objects, or a macro object, must be used to model physical objects of this type. Macro objects, introduced in section 2.5, can be used to group related network objects into a single entity.

2.3. Links

Links allow independent objects to be connected to form a simulation problem. As a simple example, consider an air duct system in which four ducts are combined into one, first by combining two pairs of ducts and then by combining the two resulting ducts, fig. 2.1a. The network for this problem is represented in fig. 2.1b. It is interesting to notice that the simulation network can be drawn to bear some resemblance to the physical situation being modeled. Defining network objects that correspond to physical objects causes this to be the case in many network simulations. Note that there is no concept of direction defined for the links in the network.

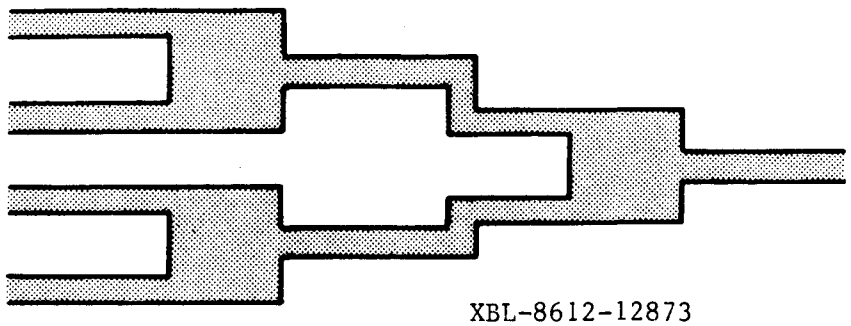

Fig. 2.1a: The Physical System of Ducts Being Modeled

An alternative perspective of networks comes from viewing each object as defining an equation, and each link as defining a variable shared by the equations. A network can then be viewed as a set of simultaneous equations. An example is the set of equations $xy = a$ and $x - y = 0$.

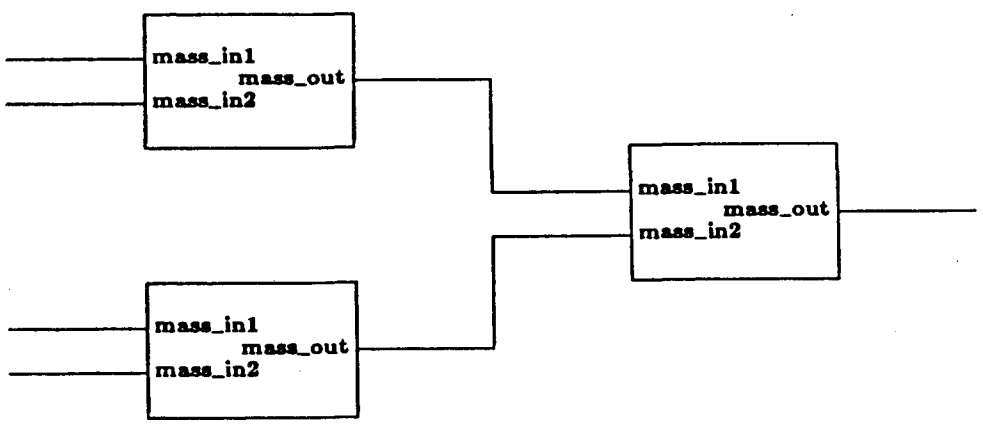

Fig. 2.1b: Network representation of three mass collectors which combine four air flows into one. Boxes represent objects, lines are links and interfaces are labeled.

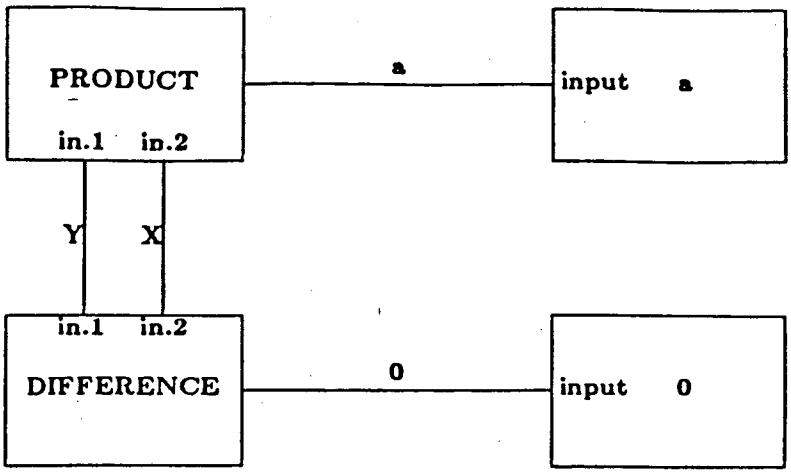

Fig. 2.2: A network definition of the simultaneous equations: $xy = a$ and $x - y = 0$. The boxes on the right define the parameters a and 0 for the problem.

These equations are solved when $x = y = \pm\sqrt{a}$. A network definition of this problem is shown in fig. 2.2. An important point follows from this alternate view of networks. Any simulation problem that can be defined by a set of simultaneous equations can be immediately translated into network form. Using a network to define a simulation problem in no way limits the problems that may be defined.

2.4. Network Based Solution Technique

There are a variety of numerical methods that can be used to solve simulation problems. Since SPANK problems are defined in terms of a network, it seems reasonable to employ a method that is naturally defined in terms of a network. The solution method chosen is a relatively new one originally developed by Levy and Low [SOW84].

The Levy-Low method attempts to pick a dependent variable for each object's equation. For instance, in the mass collector object defined earlier in this section, each of the three interface variables can be defined in terms of the other two:

$$\begin{aligned} mass\_out &= mass\_in\,1 + mass\_in\,2 \\ mass\_in\,1 &= mass\_out - mass\_in\,2 \\ mass\_in\,2 &= mass\_out - mass\_in\,1 \end{aligned}$$

Selecting one of these equations for the object selects a dependent variable. The dependent variable is the output variable for the object. The other interface variables are input variables, and the object's function becomes the computation of the output variable from the input variables. In some cases, it is possible that

only some subset S of the interface variables can actually be computed from the remaining interface variables. Only members of S are eligible to be chosen as the dependent variable.

The Levy-Low technique is only viable when output variables are assigned so that each link is connected to only one output variable. The link can then be thought of as a directed hyperedge, carrying data produced from one object's output to the input interfaces of some other objects. This directed network is a data flow graph; details on this structure can be found in section 4. Nodes in the hypergraph produce results from inputs carried on incoming hyperedges and send the results to other nodes along outgoing hyperedges. The network simulation problem is solved by introducing input data and then letting the results of this data *flow* through the graph. Complications, such as the need for iterative solutions, nearly always arise in a simulation problem, but the general solution technique remains the same. Details of the solution technique are discussed in section 5.

2.5. Macro Objects

In order for the solution method to function, each object must define a single equation. In general, physical objects are more complicated than this; they require a number of equations among various physical quantities. It makes sense to group all the equations associated with a physical object into a single entity called a *macro object.* A macro object has a set of interface variables and may be linked into simulation networks like an ordinary object. Unlike an ordinary object, the macro object may contain an arbitrarily complex set of other objects and links that define many equations among its interface variables. Macro objects may contain other macro objects nested to any depth. The solution technique *expands* instances of macro object types into their constituent set of simple objects and links.

2.6. Advantages of Networks

Data abstraction can be defined as "to separate the incidental details of the implementation of a subprogram from the properties essential to the correct use of it. Such a separation can be expressed by channeling all use of the subprogram through a specific interface." [STR84] The network technique accrues many of the advantages associated with object-oriented languages that provide data abstraction. Objects and macro objects in a network are instances of abstract data types. The equation(s) enforced between the interface variables need not be known to specify a simulation. All that is needed is to link the interface variables together to model a physical situation. It is no longer a problem to introduce a model of a physical situation that was not anticipated when the network software was written. Exchange of ideas between members of a modeling community is facilitated; to share new modeling techniques, different groups need only provide definitions of new object types. The benefits accrued from data abstraction are the greatest advantage of the network technique over previous simulation methods.

3. NETWORK SPECIFICATION LANGUAGE

Many present day simulation programs are hampered by poor user interfaces. Interfaces are often either so rigid as to be entirely inflexible [DOE81], or so complex as to be incomprehensible [ACS75, CON77]. This section describes a network specification language, NSL, that provides a simple set of commands to allow unsophisticated users to link together simulation problems from a set of predefined object types. More sophisticated users can define their own object and macro object types. This section begins by describing NSL which is used for description of simulation networks in SPANK. The section then discusses higher level interfaces NSL's relationship to more traditional object-oriented languages.

3.1. Details of the Network Specification Language

The network specification language must provide constructs that allow users to create objects and to link interfaces of these objects together. Methods for defining new object and macro object types, and for specifying the input of data are also needed. This leads to five commands in the basic language implementation: *define*, *declare*, *link*, *input* and *macro*. Each command is described below.

1. DEFINE — The define statement allows an object type with a particular equation and interface variables to be defined. An example definition is shown in fig. 3.1. After the key word *define* is a type name *mass_coll*. The definition of the interface variables for a *mass_coll* is enclosed in braces. The name of each interface is found on the left hand side of an equal sign. The function on the right side defines the formula relating the interface being defined and the other interfaces. For instance, the interface *mass_out* is defined as $\sum$(*mass_in1*, *mass_in2*). The functions for the *mass_in*1 and *mass_in*2 interfaces should be different forms of the same equation. The functions used to define the interfaces are standard C functions that receive their arguments as an array of double precision values. The C code for function *sum* can be seen in fig. 3.2. If no function is defined for an interface, this interface will not be selected as the dependent interface by the solution technique.

```
define mass_coll {
                        mass_in1 = difference(mass_out, mass_in2);
                        mass_in2 = difference(mass_out, mass_in1);
                        mass_out = sum(mass_in1, mass_in2);
}
```

Fig. 3.1: Definition of object type mass_coll.

```
double sum(args)
double args[];
{
return(args[0] + args[1]);
}                                    /* End of sum */
```

Fig. 3.2: Function sum.f.c.

2. DECLARE Declarations create particular instances of an object type. The syntax is *declare* followed by a type name from a previous definition. The type name is followed by a list of names of object instances of this type. For example, fig. 3.3 shows the program to specify the four-way mass collector that was used in a previous example. Three mass collectors, *mc*1, *mc*2, and *mc*3 are declared.

```
declare mass_coll mc1, mc2, mc3;

link intern_mass1(mc1.mass_out, mc3.mass_in1)
link intern_mass2(mc2.mass_out, mc3.mass_in2)
link output_mass(mc3.mass_out)

input mass_in1(mc1.mass_in1)
input mass_in2(mc1.mass_in2)
input mass_in3(mc2.mass_in1)
input mass_in4(mc2.mass_in2)
```

Fig. 3.3: Specification of the Four-Way Mass Collector Problem

3. LINK The link command creates a link between one or more interface variables. The syntax can be seen in fig. 3.3. The keyword *link* is followed by a link name and an interface list enclosed in parentheses. The interface list specifies each interface connected by the link in the form *object_name.interface_name*. In the figure, link *intern_mass*1 connects the *mass_out* interface of object *mc*1 to the *mass_in*1 interface of object *mc*3.

4. INPUT The input command allows the user to indicate that an input data value will be available to one or more interfaces of the network. An input value is a parameter of the simulation that the user wishes to vary at the time the solution program is executed. The syntax is similar to that for links; the keyword *input* is

followed by an input name. This precedes an interface list in the same form as those used for links. In fig. 3.3, there are four inputs declared, one to each of the mass input interfaces of objects *mc* 1 and *mc* 2.

5. MACRO — The macro command defines a macro object type. The network specification language expects the definition of a macro type to contain a set of linked objects. The interfaces of a macro object type are a subset of the links defined in its definition. Section 7 provides an example of the use of the macro command.

3.2. Higher Level Interfaces

While the language presented in the previous pages is probably reasonable for use by SPANK system designers, it may be too complex for the casual simulation user. In general, such users will not want to be troubled with defining object types or C functions to implement interfaces. Instead, they will want access to a library of predefined objects and macro objects which can be linked to define a simulation problem. SPANK allows libraries of object types to be defined and accessed by default if no object definitions are provided in a simulation specification program. While even higher level interfaces, graphical interfaces for example, are obviously of interest, they were beyond the scope of this work.

3.3. Relation to Traditional Object Oriented Languages

The network specification language is closely related to more traditional *object-oriented* languages such as SIMULA, SMALLTALK, or C++ [STR84]. Fig. 3.4 shows equivalences between concepts in C++ and the network specification language.

C++	Network Specification Language
class	object type
class instance	object
internal data (private)	equation
member or friend functions	links
inheritance, derived classes	macro objects

Fig. 3.4: A comparison of features in the Network Specification Language and C++.

An object is an instance of an object type which can be mapped to the C++ concept of a class. The object *hides* its internal details, i.e. its equation, just as a class instance hides its private parts. Links cause a certain order of execution to be imposed on the object's internals, just as a set of calls to member or friend functions act in a C++ program. There is also a notion of inheritance; a macro object contains the internal data of its various component objects. Since macro

objects can contain many types of objects, this is more closely related to the multiple inheritance available in SMALLTALK. Because of this close relation between C++ and the network specification language, many of the benefits accrued by object-oriented languages also exist in the network specification language. In particular, it is easy to change the internals of an object as long as the interface remains the same. Errors tend to be localized to the object in which they exist. Inheritance makes it easy to define more complex objects that are still comprehensible and easy to maintain. The network specification language is essentially an object-oriented language specifically designed for defining network problems. Unlike true object-oriented languages, however, NSL is processed by a solver that produces a traditional C program.

4. NUMERICAL SOLUTION METHODS

A simulation problem defined as a network can be mapped directly to a set of simultaneous equations. A traditional method for solving such systems is Newton-Raphson iteration. SPANK applies graph theory technique to reduce the size of the system of equations before applying Newton-Raphson iteration. This section describes the solution technique and the constraints that it places on network defined simulation problems.

4.1. Traditional Solution Techniques

Any network specification language problem in which every interface variable has been linked can be mapped directly to a set of simultaneous equations. Each link is mapped to an unknown and each object is mapped to the equation it defines. Throughout this section it is assumed that every interface has been linked and that macro objects have already been expanded into their constituent objects and links.

A network with n objects is mapped to a set of n simultaneous equations; an object with k interfaces defines an equation of the form:

$$interface_1 = F(interface_2, interface_3, \ldots, interface_k)$$

where F is an arbitrarily complex function. While certain simulations may involve only linear equations, most will be non-linear. A general purpose technique for network simulation problems must, therefore, be equipped to handle the solution of systems of non-linear equations.

The traditional technique for solving a system of non-linear equations is Newton's method [ATK78]. Details on Newton's method can be found in section 5.3.

4.2. Overview of SPANK Solution Method

The SPANK solution method attempts to use techniques based on graph theory to reduce the number of simultaneous equations that must be solved by Newton's method. As mentioned in section 2.4, a matching of links to objects can produce a data flow graph from a network. If the data flow graph is cyclic, it can

not be solved directly; iteration is required. The traditional Newton implementation treats each variable in the problem as an iteration variable. However, a vertex cycle cut set of the data flow graph can be used to define a smaller set of equations that characterize the network. A vertex cycle cut set of size c leads to a set of c simultaneous equations to be solved by traditional methods. Since "c" is smaller than the original number of equations, this new system can be solved more rapidly.

4.3. Defining a Data Flow Graph

The first step in the SPANK solution method is to create a data flow graph from the network specification. A data flow graph is a graph that represents a computation. Such graphs have been used to define computations for so-called data flow machines [DAV82]. The data flow graph is represented as a set of nodes and a set of directed hyperedges, one emanating from each node. Each node represents a computation that is performed on its incoming hyperedges to produce a result on its outgoing hyperedge. A node can execute its function, called *firing*, when all of its inputs are present. When a node fires, it consumes its input data and sends its output value to all nodes that are destinations of its outgoing hyperedge. The term *data flow* is natural; data flows along hyperedges from one node to another, causing certain nodes to fire and thus producing more data.

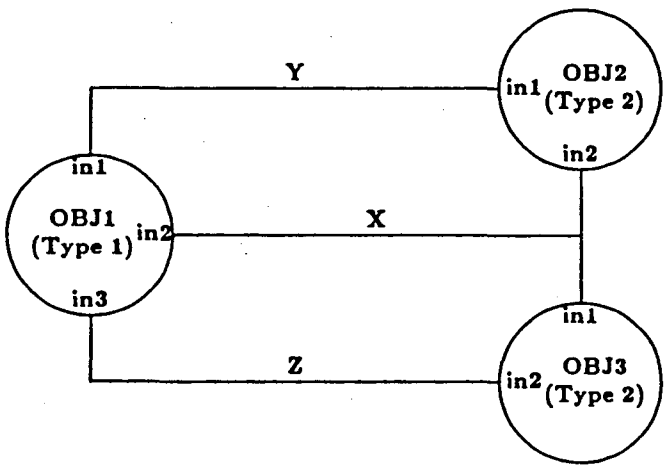

Fig. 4.1a: Network Problem Specification

A data flow graph can by obtained from a network by finding a one-to-one matching of objects to links. Such a matching can be viewed as selecting a dependent variable for each object. This means that the object will define a value for the link variable to which it is matched in terms of all the other links that are incident upon the object. Once such a matching is made, a data flow graph follows quite naturally from the original network definition. Each link becomes a data flow path between objects. A link points from the object to which it was matched to all other objects to which it is linked. An example will clarify this procedure. Fig. 4.1a-e shows a sample network problem, the matching between objects and links, and the resulting data flow graph.

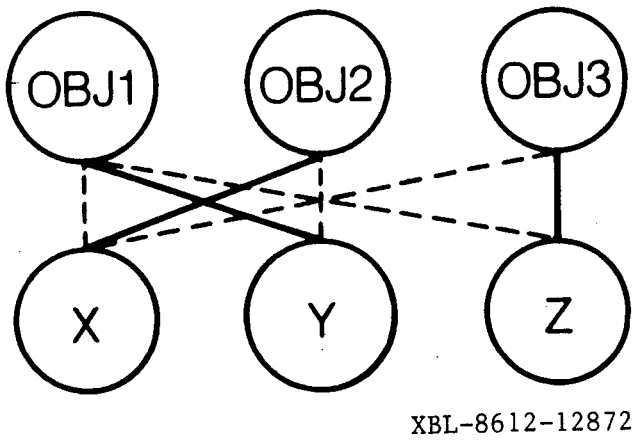

Fig. 4.1b: Corresponding Matching Problem

```
define type1 {
        in1 = f1(in2, in3);
        in2 = f2(in1, in3);
        in3 = f3(in1, in2);
}


define type2 {
        in1 = r1(in2);
        in2 = r2(in1);
}


declare type1 obj1;
declare type2 obj2, obj3;


link x(obj1.in2, obj2.in2, obj3.in1)
link y(obj1.in1, obj2.in1)
link z(obj1.in3, obj3.in2)
```

Fig. 4.1c: Network Specification Language for the Network

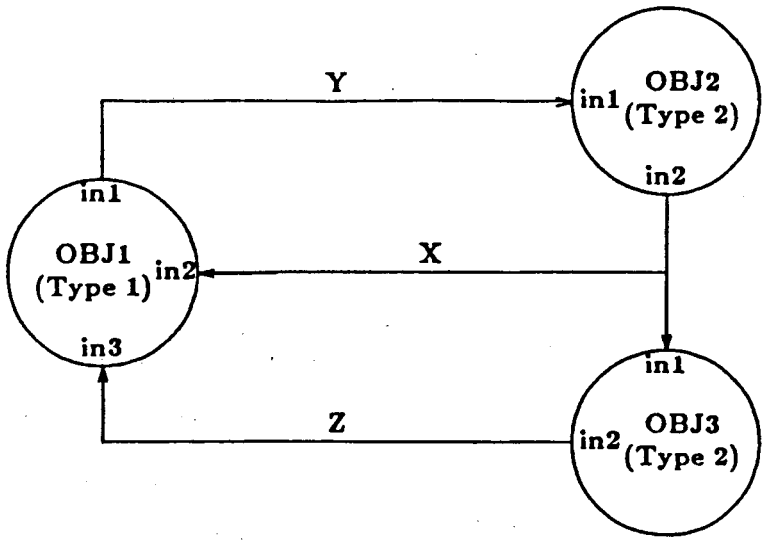

Fig. 4.1d: Data Flow Graph

obj1 defines y = f2(x, z) obj2 defines x = r2(y) obj3 defines z = r1(x)
Fig. 4.1e: Resulting relations.

4.4. Simulation Problem Networks Have Perfect Matchings

To define a data flow graph from a network, a one-to-one matching between the variables and equations must exist in the network. It can be shown that simulations of physical problems lead to networks that have one-to-one matchings, provided three conditions are met.

1. A simulation of a physical situation should always have a unique solution, i.e. a set of values for the links such that the equations of all objects are satisfied. Any problem that has no such solution is not considered a valid simulation; impossible physical situations do not exist. Problems that have many solutions are not specified strongly enough. Additional constraints can be added to create a problem with a unique solution.
2. There should be only a single link for each physical quantity. Providing more than one link allows the same physical quantity to be simultaneously assigned different values in a simulation. This does not correspond to any real physical situation.
3. There must be no redundant modeling of a physical constraint. There should not be two sets of equations that enforce the same physical constraints on the same physical quantities. Any redundant constraints must be removed from

a simulation.

A series of Lemmas proves that a perfect matching between objects and links must exist in a simulation satisfying these conditions.

Lemma 1:

A network that satisfies the three conditions above has the same number of equations and variables (objects and links).

Proof:

By condition 3, the equations in the network are independent and by condition 2, the variables are unique. Assume there are n equations. If there are $k<n$ variables, some of the equations must be redundant which contradicts condition 3. If there are $k>n$ variables, there can be no unique solution which contradicts condition 1.

Lemma 2:

Any set S of n equations from a network that satisfies conditions 1 to 3 must involve at least n variables.

Proof:

If a set of n equations has fewer than n variables, this set of equations is overdetermined and some set of equations is redundant in violation of condition 3.

Theorem 1: (Hall's Theorem)

Let X and Y be two bipartite sets that are connected by a set of edges E. Then there exists a matching in which all elements of X are matched if and only if for all subsets S of X, $|N(S)| \geq |S|$ where $N(S)$ is the set of all elements of Y that have edges to elements of S [ROB84].

Lemma 3:

In a problem satisfying conditions 1 to 3, there exists a matching in which all equations are matched.

Proof:

Lemma 2 showed that every subset of j equations has edges to at least j variables. By Hall's Theorem, there exists a matching in which all equations are matched.

Lemma 4:

In a problem satisfying conditions 1 to 3, there exists a perfect matching of equations to links.

Proof:

By Lemma 1, there are equal numbers of equations and links. By Lemma 3, there exists a matching in which all equations are matched. In this matching, all objects must also be matched.

As long as the three conditions are met, a perfect matching exists and the solution techniques can be applied. The conditions do not appear to unreasonably limit simulation problems; a number of Building Energy Simulation problems have been defined in SPANK and all have had perfect matchings of objects to links.

4.5. Producing Results from Data Flow Graphs

Once a matching has been used to find a data flow graph, the graph can be used to compute solutions to the simulation problem. Fig. 4.2 shows an acyclic data flow graph; it is clear what to do to produce numerical results from the data flow graph. The input value is introduced to the system. Upon receiving the value, obj_1 can fire producing variable x. Variable x allows obj_2 to fire producing y. y in turn allows both obj_3 and obj_4 to fire producing output results.

Any acyclic data flow graph can sequentially fire all its nodes whenever all of its external inputs are available. A cyclic graph, on the other hand, represents a problem that involves iteration and cannot be solved immediately [DAV82].

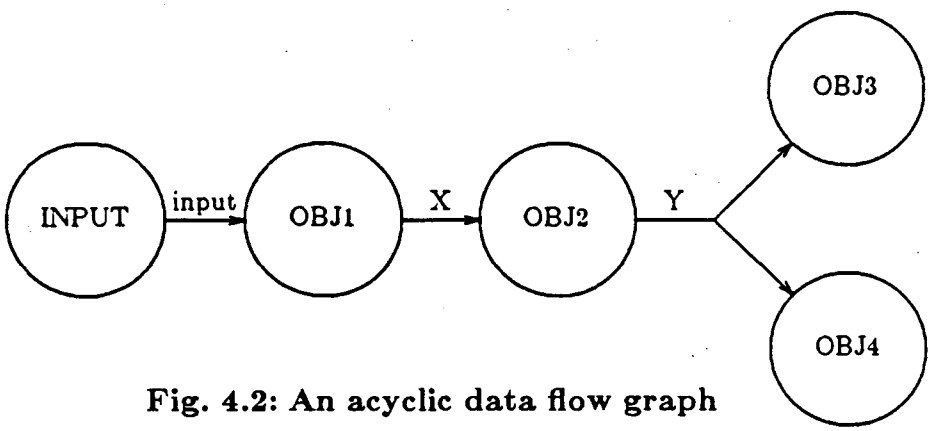

Fig. 4.2: An acyclic data flow graph

4.6. Iterative Solution for Cyclic Flow Graphs

Guessing the values of some of the hyperedges in a cyclic data flow graph can allow the rest of the graph to execute. Fig. 4.3 depicts making a guess for the value of obj_1's output in the data flow graph from fig. 4.1. Guessing a value for y, y_{guess}, allows the rest of the graph to execute. For a randomly selected value of y_{guess}, however, the solutions resulting from the flow graph execution are not related to the answers of the original network in any predictable fashion.

In the original network of fig. 4.1d, y was constrained to be equal to a function $f_2(x,z)$. This constraint has been removed when y_{guess} is introduced in

fig. 4.3. The values of x and z that result only satisfy the original network problem if the output y of obj_1 is equivalent to the output y_{guess} of obj_1', an extremely unlikely occurrence for a randomly chosen y_{guess}.

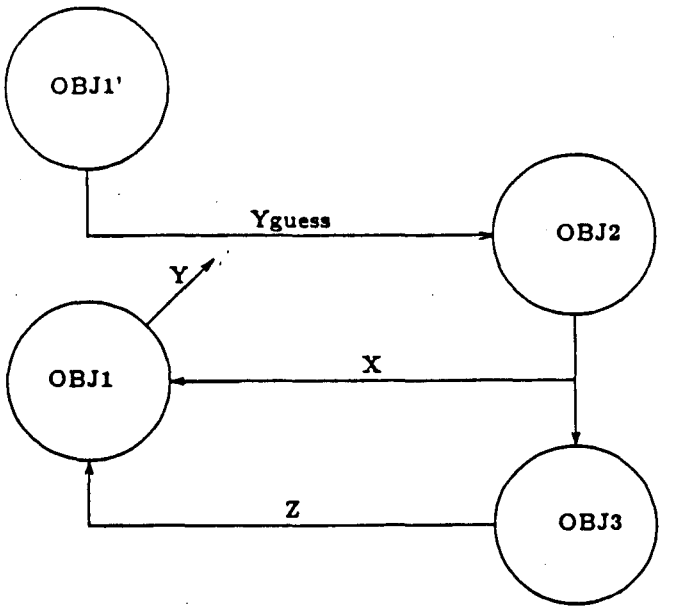

Fig. 4.3: Guessing value of y

It is interesting to notice, however, that y, the output of obj_1, is a function of y_{guess}, $y = G(y_{guess})$. In order to satisfy the original network definition, the condition $y = y_{guess}$ must be fulfilled. This implies that:
$y - y_{guess} = G(y_{guess}) - y_{guess} = F(y_{guess}) = 0$. But, $F(y_{guess}) = 0$ is simply a system of simultaneous equations, in this case a single equation in a single variable. The classical techniques for solving non-linear systems of equations, outlined in section 4.1, can now be employed to solve for y_{guess}. Each time a function evaluation of $F(\alpha)$ is called for in the iterative solution method, α is introduced to the data flow graph in 4.3b as $y_{guess} = \alpha$, and $G(\alpha)$ is the resulting value of y produced by the graph. A guess α for which $F(\alpha)$ is equal to zero is a solution of the network simulation problem.

Notice that in the example above, the original network problem was three equations in three variables. By using the SPANK solution method as a preprocessing step, the number of unknowns that must be solved by a traditional iterative method has been reduced from three to one.

This method can easily be extended to apply to arbitrarily complex data flow graphs. The procedure is to first find a set of objects (obj_1, obj_2, ..., obj_n) that forms a vertex cycle cut set in the flow graph. A vertex cycle cut set is a set of nodes that, *if* removed from a graph along with all edges incident to them, results in an acyclic graph. A new node, obj_i', is introduced to the graph for each member of the cut set. A directed edge is introduced from obj_i' to all nodes to which obj_i has a directed edge. The directed edges from obj_i are then deleted

from the graph.

The n output values $(x_1, x_2, \ldots, x_n)$ of the cycle cut set objects $(obj_1, obj_2, \ldots, obj_n)$ can be viewed as functions of the n guessed variables $(x_1', x_2', \ldots, x_n')$ produced by the introduced objects $(obj_1', obj_2', \ldots, obj_n')$.

$$\begin{aligned} x_1 &= G_1(x_1', x_2', \ldots, x_n') \\ x_2 &= G_2(x_1', x_2', \ldots, x_n') \\ &\vdots \\ &\vdots \\ x_n &= G_n(x_1', x_2', \ldots, x_n') \end{aligned}$$

The solution of the original network occurs when the guessed value x_i' is equivalent to the resulting value x_i for $i = 1, 2, \ldots, n$. This can be expressed as

$$\begin{aligned} x_1 - x_1' &= G_1(x_1', x_2', \ldots, x_n') - x_1' = F_1(x_1', x_2', \ldots, x_n') = 0 \\ x_2 - x_2' &= G_2(x_1', x_2', \ldots, x_n') - x_2' = F_2(x_1', x_2', \ldots, x_n') = 0 \\ &\vdots \qquad\qquad\qquad\qquad \vdots \\ &\vdots \qquad\qquad\qquad\qquad \vdots \\ x_n - x_n' &= G_n(x_1', x_2', \ldots, x_n') - x_n' = F_n(x_1', x_2', \ldots, x_n') = 0 \end{aligned}$$

This is a set of n equations in n unknowns that can be solved iteratively using Newton or a similar method. The functions:

$$F_1(\alpha_1, \alpha_2, \ldots, \alpha_n), F_2(\alpha_1, \alpha_2, \ldots, \alpha_n), \ldots, F_n(\alpha_1, \alpha_2, \ldots, \alpha_n)$$

can be evaluated for particular guesses $\alpha_1, \alpha_2, \ldots, \alpha_n$ by introducing these values as the outputs of the guess objects $obj_1', obj_2', \ldots, obj_n'$ in the flow graph. The values of

$$G_1(\alpha_1, \alpha_2, \ldots, \alpha_n), G_2(\alpha_1, \alpha_2, \ldots, \alpha_n), \ldots, G_n(\alpha_1, \alpha_2, \ldots, \alpha_n)$$

are the outputs of the objects $obj_1, obj_2, \ldots, obj_n$ after the flow graph has executed with these inputs.

Since the number of dimensions of the iterative solution problem is the same as the size of the cycle cut set, it is advantageous to have as small a cut set as possible. The algorithms employed for determining the cut set are discussed in section 5. An analysis of the complexity of the solution technique can be found in section 6.

4.7. Summary of the Solution Technique

A simulation problem defined in network form can be viewed as a system of simultaneous equations. One option for solving this system would be to immediately apply traditional Newton or quasi-Newton methods to the system. Since the

problem is defined in network form, a solution technique that depends on network representation of the simulation was examined instead. The data flow graph representation allows a smaller set of simultaneous equations to be iteratively solved. There will be one equation for each member of a cycle cut set of the data flow graph. The entire solution method should be viewed as an attempt to reduce the number of equations before applying traditional iterative techniques. The value of this technique will be discussed in section 6.

5. IMPLEMENTATION OF NUMERICAL METHOD

5.1. Matching

An efficient algorithm for matching on a bipartite graph, Dinic's algorithm, was implemented. The matching will fail and inform the user that an invalid simulation has been defined if there exists no perfect matching between the objects and their adjacent links.

Dinic's algorithm is actually designed to determine a maximum flow on a network. The bipartite matching can be converted to a unit network flow problem by introducing source and sink nodes s and t to the matching graph. Each node in one of the bipartite sets is attached to the sink; all nodes in the other bipartite set are attached to the source. The set of edges between the bipartite set in a maximum network flow from s to t are also edges in a matching. For details on this algorithm, see the work by Tarjan [TAR83].

5.2. Vertex Cycle Cut Set

A vertex cycle cut set of a hypergraph must be found for every simulation problem that results in a cyclic data flow graph. The cycle cut set represents the nodes whose values must be guessed and solved for iteratively. It is always desirable to have as few simultaneous equations to solve as possible, therefore, a minimum cycle cut set of the data flow graph is sought.

Unfortunately, the problem of finding a minimum cycle cut set for an arbitrary directed graph has been proven to be an NP-complete problem by Karp [KAR81]. An algorithm that produces a small cycle cut set, but not necessarily the smallest such set, is the best alternative. Fortunately, algorithms producing small cut sets are known; a modified version of an algorithm due to Levy and Low [LEV83] is implemented in the SPANK system.

The Levy-Low algorithm is designed to be applied to directed graphs. The data flow graphs under consideration are directed hypergraphs. Before the algorithm can be applied, the hypergraph is converted to a simple graph. Each directed hyperedge that goes from a node n to a set of destination nodes $\{d_1, d_2, \ldots, d_k\}$ is replaced by k edges, from n to d_i, $i = 1, 2, \ldots, k$.

5.3. Newton's Method

A version of Newton's iteration method for solving systems of non-linear equations is implemented in SPANK. Newton's method solves systems of non-

linear equations:

$$f_1(x_1, x_2, \ldots, x_n) = 0$$
$$f_2(x_1, x_2, \ldots, x_n) = 0$$
$$\vdots$$
$$\vdots$$
$$f_n(x_1, x_2, \ldots, x_n) = 0$$

The general iteration involves repeatedly computing a new guess for the vector $\bar{x}$ using the old value of both $\bar{x}$ and the vector function $\bar{f}$. In particular,

$$\bar{x}^{i+1} = \bar{x}^i - F^{-1}(\bar{x}^i)\bar{f}(\bar{x}^i)$$

where $\bar{x}^i$ is the current value of the vector $\bar{x}$, $\bar{x}^{i+1}$ is the new guess for $\bar{x}$, and $\bar{f}(\bar{x}^i)$ is the current value of the vector function $\bar{f}$. $F(\bar{x}^i)$ is the Jacobian matrix of the vector function $\bar{f}$ evaluated at $\bar{x}^i$. In the implementation, the computation of the inverse of the matrix $F(\bar{x}^i)$ is avoided because of its cost. Instead, the vector equation

$$F(\bar{x}^i)\bar{\delta}^{i+1} = -\bar{f}(\bar{x}^i)$$

is solved for the correction term $\bar{\delta}^{i+1}$ using a Gaussian elimination routine. The new guess for $\bar{x}$ is computed as

$$\bar{x}^{i+1} = \bar{x}^i + \bar{\delta}^{i+1} \quad .$$

The Jacobian matrix is computed using numerical forward difference derivative approximations. Each Jacobian matrix requires n executions of the data flow graph. Introducing $(x_1 + \delta, x_2, \ldots, x_n)$ as the inputs of the cut set nodes allows the computation of the n partial derivatives with respect to x_1 from the outputs. Adding δ to the other inputs allows computation of all the other partial derivatives in turn.

The iteration is continued until all the vector functions, $f_i(x_1, x_2, \ldots, x_n)$ differ from zero by less than a given error tolerance e. The number of iterations is limited; problems that fail to converge after a fixed number of iterations will terminate with an error message.

5.4. Comparison of SPANK and Explicit Equation and Derivative Evaluators

There are software systems that develop explicit solution formulas for equations and their derivatives. MACSYMA, MIT's symbolic algebraic manipulation system, is one well-known example; given a set of equations, it can explicitly solve for a particular variable in terms of all others. For some class of objects, MACSYMA could be applied to a SPANK data flow graph to solve explicitly for each of the cut set variables in terms of the guessed values.

Derivatives can also be explicitly solved by programs like MACSYMA. Each node in the network could be required to provide partial derivatives of its equation with respect to each of its interface variables. Making use of the chain rule and explicit differentiation formulas, the partial derivatives of the output variable with respect to each cut set variable could be computed each time a data flow node fired. Data flow hyperedges would carry both the value of the hyperedge and the partial derivatives of that hyperedge with respect to each guessed cut set variable. The partial derivatives needed for the Jacobian would be available at the output arcs of the original cut set nodes.

Explicit methods like these were not implemented in SPANK for a number of reasons. First, programs to explicitly solve for equations are extremely complex; adding the ability to explicitly differentiate formulas would further increase this complexity. Second, the abilities of programs to explicitly solve equations and derivatives is limited to a standard set of equation types. The SPANK program allows much more generality by allowing equations to be anything that can be expressed as a C function. As an example, a C function that iteratively solves a heat equation could define the equation for a SPANK object. Explicit solving programs are simply not prepared to handle such general equation types. Finally, programmers would have to provide partial derivative functions for each interface variable of each object type. This would greatly increase the difficulty involved with using SPANK. In order to decrease program complexity, increase capabilities, and decrease programming effort, SPANK does not implement explicit solution techniques.

MACSYMA-like techniques could be used to enhance SPANK and SPANK techniques could be used to enhance MACSYMA. For example, SPANK objects require explicit formulas for each interface variable. Each formula is derived from the underlying mathematical model of the object. MACSYMA-like techniques could be used to automate these derivations which are presently done manually. Conversely, the SPANK techniques for matching and cut set determination could be used to allow MACSYMA to develop iterative solutions for systems without closed form solutions. These avenues are yet to be explored.

6. ANALYSIS OF SOLUTION METHOD

This section performs a non-rigorous analysis of the run time behavior of the solution algorithm implemented in SPANK. The run time efficiency of the matching, cycle cut set, and solution algorithms are examined. Also, the relation of the implemented solution technique to an immediate application of a traditional Newton's method is examined in terms of speed and stability. The fact that the matching and cut set solutions can be non-unique is also discussed.

6.1. Matching

A simulation problem with n objects results in a matching of two bipartite sets with n members each. Tarjan [TAR83] provides an analysis of the time taken by Dinic's matching algorithm. On a unit network such as the one

produced in a matching problem, Dinic's algorithm finds a maximum flow in time $O(\sqrt{n}\,m)$ where n is the number of nodes and m is the number of edges.

In the matching problem, there is an edge from a node representing an object to the nodes for each link that touches the object. The number of links per object is bounded above by the number of interfaces per object. One expects the number of interfaces per object in a problem to remain essentially constant independent of the total number of objects in a simulation. Another way of phrasing this is, in a simulation problem, one expects an object to be directly related only to a few other objects that are physically *close* to it. One does not expect an object to be directly related to all other physically remote objects in the simulation. If this is the case, the number of edges m is then $O(n)$, where n is the number of objects in the simulation, and the total run time for Dinic's algorithm to find a matching is $O(n^{1.5})$.

6.2. Cycle Cut Set

Levy-Low provide an analysis of their cycle cut set algorithm [LEV83]. For a graph with m edges and n nodes, the algorithm produces a cycle cut set in time $O(m \log n)$. Using the same assumptions as were used in the matching analysis, the number of edges can be assumed to be $O(n)$. This gives a run time complexity of $O(n \log n)$ for the cycle cut set with n the number of objects in the simulation problem.

6.3. Solution of Data Flow Graph

In a simulation problem that results in an acyclic data flow graph, the solution algorithm will have to solve n equations, one for each object, and ship n results. If the assumption of the last two subsections that the number of interfaces per object is O(1) is valid, each equation evaluation represents an amount of work that is O(1) in terms of the number of objects n. Hence, the solution of an acyclic graph simulation requires time $O(n)$.

The analysis rapidly becomes more complex if a cyclic data flow graph results from the simulation definition. An analysis of Newton's method and quasi-Newton methods is needed first.

When using divided difference methods to compute derivatives, each iteration of Newton's method requires n^2+n function evaluations, n to compute the original value of the function and n^2 to compute n partial derivatives for each of the n equations [ATK78]. The overriding cost comes from solving the linear equation

$$F(\overline{x}^n)\overline{\delta}^{n+1} = -\overline{f}(\overline{x}^n)$$

where F is the Jacobian, $\overline{\delta}$ is the correction vector, and $\overline{f}(\overline{x}^n)$ is the current function vector. In general, solving an n x n system like this requires $O(n^3)$ operations, so the cost for each iteration of a standard Newton's method is $O(n^3)$. Quasi-Newton methods are faster but only by a constant factor if they are general enough to solve an arbitrarily complex system of equations [ATK78].

Suppose the same system of equations is solved using the SPANK solution technique. A cyclic graph will result in the selection of some number c of cut set objects whose values will be guessed. As noted in section 5, this results in a new set of c equations in c unknowns to be solved by a standard Newton method.

Computing the Jacobian for these c equations is performed in the same fashion as for a general Newton's method solution. The current value of the c functions is computed by pushing the current guesses through the data flow graph in time $O(n)$. The c partial derivatives for a given variable are obtained by adding a delta to that variable and pushing the guesses through the graph at cost $O(n)$ for each cut set node. Thus the Jacobian can be computed in time $O(cn+n)$.

In this cycle cut set case, the Jacobian is a $c \times c$ matrix $C(\overline{c}_n)$; solving $C(\overline{c}_n)\overline{\delta}^{(n+1)} = \overline{-f}(\overline{c}_n)$ for $\overline{\delta}^{n+1}$ requires time $O(c^3)$. The total time needed for each iteration of the solution is $O(c^3+cn+n)$. Since the size, c, of a cycle cut set satisfies $1 \leq c \leq n$, where n is the number of objects, the cost per iteration of the SPANK solution technique is $O(c^3 + cn)$. The implemented solution method's speed relative to Newton's method is dependent on the size of the cycle cut set, c.

The expected size of the cycle cut set can be determined by the analysis of pseudo-random networks. Such problems are notoriously difficult to attack. When legitimate probabilistic analysis proved to be too difficult, empirical studies were used to determine the expected size of the cycle cut set. The results of these studies for several types of pseudo-random graphs are shown in tables 6.1 through 6.3.

The first table shows the size of the cycle cut set obtained from the Levy-Low algorithm for random graphs with no locality. Each node in these graphs was given a set of N directed edges to neighbors selected with equal probability from amongst the entire node set.

The results in tables 6.2 and 6.3 are probably closer to the behavior of actual simulation problem networks. In this case, a locality condition is placed on the pseudo-random edges. The nodes are ordered, and each node is only assigned edges to a set of nodes within L behind or ahead of itself in the ordering. One can argue that simulation problems should display such locality characteristics. A physical object is generally affected by some set of physically adjacent objects, but

Average Number of Interfaces	Number of Nodes in Network									
	2	4	8	16	32	64	128	256	512	1024
2	1	1	4	2	3	3	3	2	2	4
4	1	3	4	2	6	12	18	36	67	141
8	1	3	3	7	15	27	58	105	225	441
16	1	3	5	11	22	41	91	174	354	702

Table 6.1: Size of Levy-Low algorithm cycle cut set for entirely random edge selection.

An examination of some common object types defined for building energy simulation models indicates that the average number of inputs for an object ranges somewhere between four and eight. Assuming the worst case, an average of eight interfaces per object, the cut size c is still seen to be less than $n/2$. This would result in a speedup by a factor of 8. In the four interfaces per object case, $c \leq n/4$ and the speedup is 64.

Average Number of Interfaces	Number of Nodes in Network									
	2	4	8	16	32	64	128	256	512	1024
2	1	1	1	2	2	5	9	19	29	53
4	1	1	3	3	5	11	26	48	90	187
8	1	2	4	6	13	29	59	115	230	452
16	1	3	6	11	22	49	108	223	442	875

Table 6.2: Size of Levy-Low cycle cut set for random selection from a set of 10 nodes ahead of and behind the current node.

Average Number of Interfaces	Number of Nodes in Network									
	2	4	8	16	32	64	128	256	512	1024
2	1	1	2	1	3	5	11	22	47	99
4	1	1	1	2	7	14	28	58	114	235
8	1	2	3	6	15	28	57	113	224	435
16	1	3	6	12	24	51	100	204	398	802

Table 6.3: Size of Levy-Low cycle cut set for random selection from a set of 5 nodes ahead of and behind the current node.

The speedup will be the same for quasi-Newton methods of solution. These methods solve a set of n equations in time rT where T is the time needed for solving n equations with Newton's method and $r < 1$ is a constant. The time for solving the cycle cut set equations will now be rT_c, where T_c is the time to solve the cycle cut set with Newton's method. The speedup, $rT/rT_c = T/T_c$ is the same as with Newton's method.

The conclusion is that the cost per iteration of the solution method implemented in SPANK is significantly less than that of traditional methods. As long as the number of iterations required for Newton's method to converge does not increase in the cut set method, this new method is preferable.

6.4. Convergence of Solution Methods

The analysis of convergence for iterative methods is another complex problem. A brief argument that the convergence behavior of the new method should be no worse that that of a traditional Newton's method follows.

In the general n variables, n equations case, Newton's method begins with a guess of the solution of the equations. At this point, the values of the equations and their derivatives are computed. These derivatives are then used to find a point where all functions would become zero, if the slope of all functions were constant (this is the process of solving the Jacobian for δ).

In the cut set case, only $c < n$ variables and equations remain. For the remaining variables, an initial guess is made as above. The partial derivatives of the c equations can be derived by applications of the chain rule from the partials used in the general case. Essentially, $n - c$ of the original equations are already solved. Since the remaining equations and their derivatives are derived from the original set, no new problems of bad behavior around the zero are introduced. On the average one would expect the c equations problem to converge at least as rapidly as the n equation problem.

Given that the number of iterations, k, for the cycle cut set equations is not greater than that for the full Newton's method, a final comparison of the run times can be made. The cost for k iterations of the full Newton's method is $O(kn^3)$. For localized pseudo-random graphs expected in simulation problems, the cost of the SPANK method is $O(kc^3 + n^{1.5} + n \log n)$ including the matching and cycle cut set times. The performance of the SPANK solution method is clearly superior since c is only a fractional part of n. If $c < n/2$, the new method is at least eight times faster than the traditional Newton's method. In the case $c < n/4$, the speed of the new method is at least 64 times faster than that of the traditional methods.

6.5. Non-Uniqueness of Solution Method

In general, neither the matching problem solution for a given simulation, nor the cycle cut set for a given matching, are unique. Different choices of matching may result in different sizes for the cut set and different convergence behavior for the solutions. The current implementation of SPANK does nothing to attempt to pick a matching that has the smallest minimum cycle cut set.

Lemma 5:

Different matchings can result in different sizes for minimum cycle cut sets.

Proof by example:

Fig. 6.4a shows a data flow graph with 3 cycles. Node A is matched to hyperedge x and node B to hyperedge y. The minimum cut set for this graph is {A, B}. In fig. 6.4b, the data flow graph that results when A is matched to y and B to x is shown. In this case, a cut set is {A}.

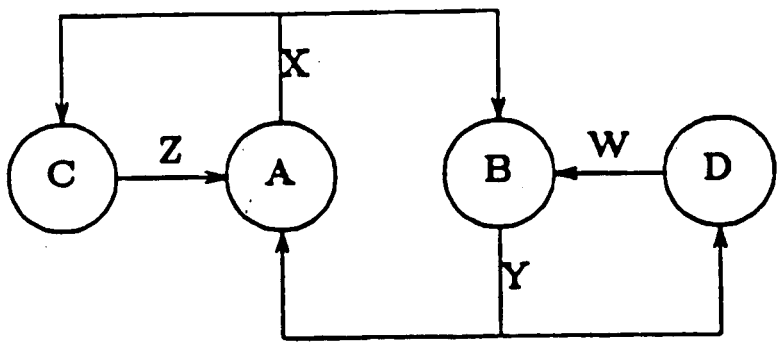

Fig. 6.4a: A data flow graph with cycle cut set {A,B}

The theory of hypergraphs is complex and limited [BER73]. It does not seem likely that one could select a matching that results in a data flow graph with the smallest cycle cut set. Local optimizations to decrease the size of the cut set may be possible. It will probably take considerable theoretical work to say anything stronger about the relation between a particular matching and the size of the resulting cycle cut set.

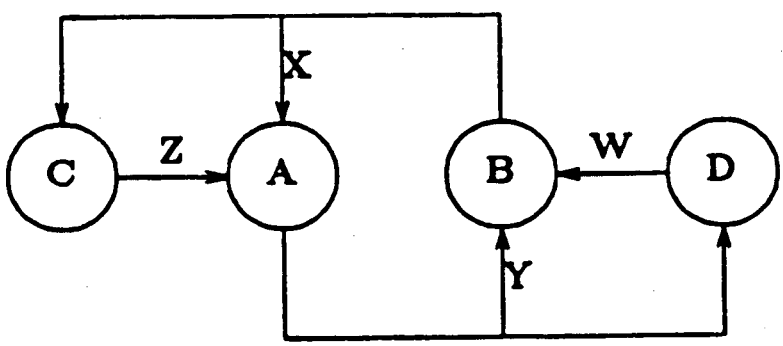

Fig. 6.4b: A data flow graph for the same problem with a cycle cut set consisting of {A}

7. A SAMPLE PROBLEM

A short sample problem is described from specification through solution in this section. The problem involves determining the control on an outside inlet air damper in order to keep recirculating air in a room at a constant predefined flow rate and enthalpy. A depiction of the physical problem is found in fig. 7.1. The problem models the mass and enthalpy of air in a ventilation system; these two quantities are assumed to define the complete state of air for this problem.

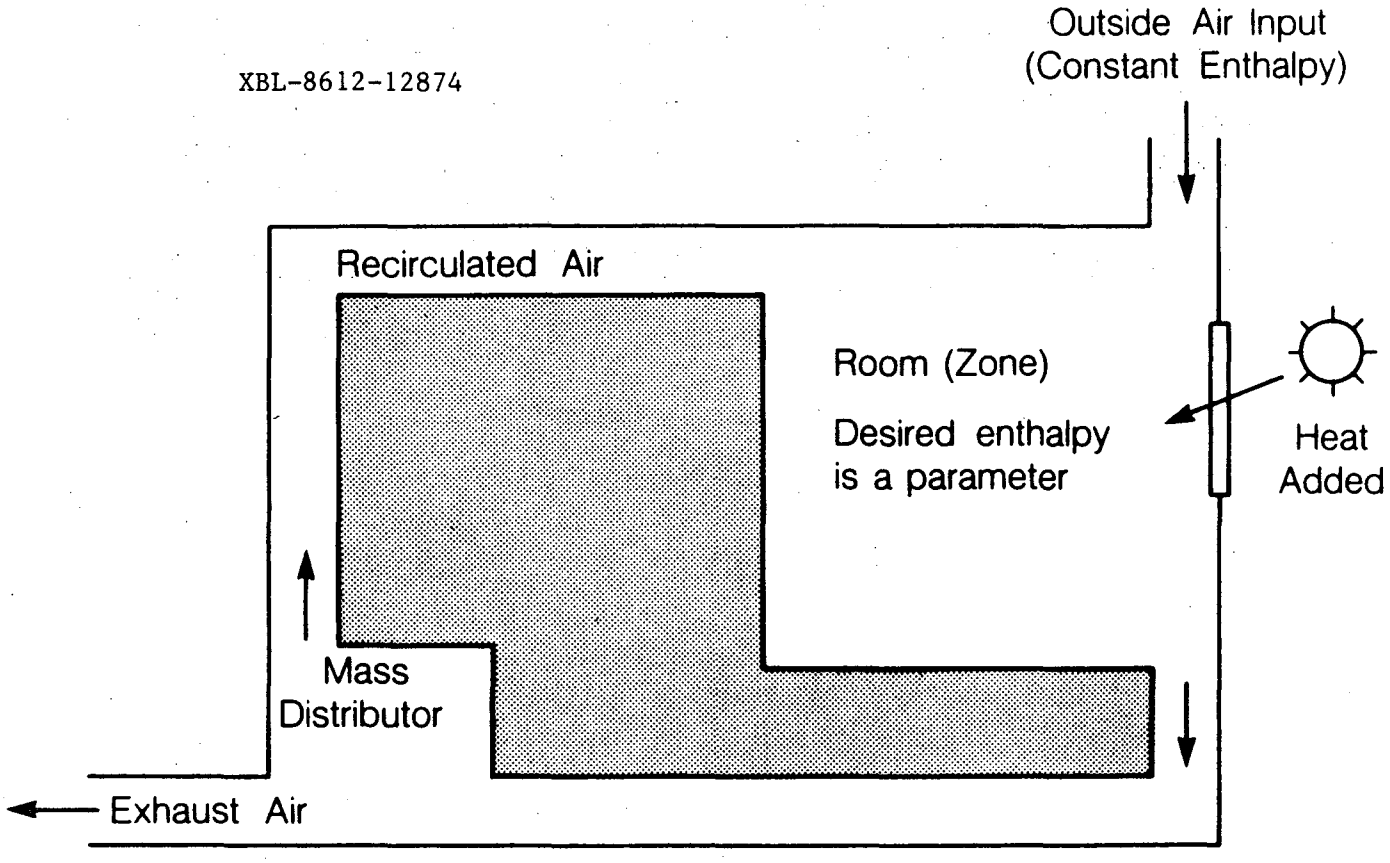

Fig. 7.1: Physical layout of recirculation problem

7.1. Specification

Two types of macro objects, a zone and a distributor, composed of three types of objects are needed for this simulation. The zone, a room with a window to admit sunlight, is a macro object composed of two objects: a mass collector which defines the mass balance and an enthalpy zone object which defines the

energy balance for the zone. The distributor is also composed of two objects, one for mass and one for energy balance. The mass balance object is a mass collector type, here used for its inverse function, while the energy distributor is an enthalpy distribution object. The definitions of the three object types and the two macro object types are listed in fig. 7.2. Fig. 7.3 shows the definitions of the functions used by the three object types. In fig. 7.2, notice that interface *q* has a null interface definition. This demonstrates the null interface feature; if, for some reason, the modeler knows that *q* should never be the determined variable for a zone, it is not even placed in the matching computation. The code specifying the simulation problem is shown in fig. 7.4.

```
define mass_coll {
                        mass_in1 = difference(mass_out, mass_in2);
                        mass_in2 = difference(mass_out, mass_in1);
                        mass_out = sum(mass_in2, mass_in1);
}
```

Fig. 7.2a: Definition of mass collector object.

```
define enth_zone {
        enth_in1 = zone_dif(mass_out, enth_out, heat_in, mass_in2,
        enth_in2, mass_in1);
        enth_in2 = zone_dif(mass_out, enth_out, heat_in, mass_in1,
        enth_in1, mass_in2);
        heat_in; /*real definition is = heat_flux(m3, h3, m1, h1, m2, h2);*/
        enth_out = zone_sum(mass_in1, enth_in1, mass_in2, enth_in2,
        heat_in, mass_out);
        mass_in1 = zone_dif(mass_out, enth_out, heat_in, mass_in2,
        enth_in2, enth_in1);
        mass_in2 = zone_dif(mass_out, enth_out, heat_in, mass_in1,
        enth_in1, enth_in2);
        mass_out = zone_sum(mass_in1, enth_in1, mass_in2, enth_in2,
        heat_in, enth_out);
}
```

Fig. 7.2b: Definition of enthalpy zone object.

7.2. Solution

Actual solution computations for the problem produced the output in fig. 7.5. Notice how rapidly Newton's method converges for well behaved problems like this. This small example hopefully gives the reader an idea of the way in which much larger, more complex simulations could be specified and solved.

```
define enth_dist {
                                    enth_in = equiv(enth_out);
                                    enth_out = equiv(enth_in);
}
```

Fig. 7.2c: Definition of enthalpy distributor object.

```
macro zone

{
declare enth_zone e;
declare mass_coll m;

link mass_out(m.mass_out, e.mass_out)
link mass_in1(m.mass_in1, e.mass_in1)
link mass_in2(m.mass_in2, e.mass_in2)

link heat_in(e.heat_in)
link enth_out(e.enth_out)
link enth_in1(e.enth_in1)
link enth_in2(e.enth_in2)
}
```

Fig. 7.2d: Definition of zone macro object.

```
macro dist

{
declare mass_coll m;
declare enth_dist e;

link mass_in(m.mass_out)
link mass_out1(m.mass_in1)
link mass_out2(m.mass_in2)

link enth_in(e.enth_in)
link enth_out(e.enth_out)
}
```

Fig. 7.2e: Definition of distributor macro object.

```
double sum(args)
double args[];
{
return(args[0] + args[1]);
}                                    /* End of sum */
```

Fig. 7.3a: Function sum.

```
double difference(args)
double args[];
{
return(args[0] - args[1]);
}                                    /* End of difference */
```

Fig. 7.3b: Function difference.

```
double equiv(args)
double *args;
{
return(*args);
}
```

Fig. 7.3c: Function equiv.

```
double zone_dif(args)
double args[];
{
return((args[0] * args[1] - args[2] - args[3] * args[4])/ args[5]);
}
```

Fig. 7.3d: Function zone_dif.

```
double zone_sum(args)
double args[];
{
return((args[0] * args[1] + args[2] * args[3] + args[4])/ args[5]);
}
```

Fig. 7.3e: Function zone_sum.

```
double heat_flux(args)
double args[];
{
return(args[0] * args[1] - args[2] * args[3] - args[4] * args[5]);
}
```

Fig. 7.3f: Function heat_flux.

```
    /* Problem is composed of a zone and a distributor */
declare dist d;
declare zone z;

input heat_flux (z.heat_in)
input outside_enth (z.enth_in2)
input desired_enth(z.enth_out, d.enth_in)
input recirc_mass(z.mass_in1, d.mass_out1)

link recirc_enth(d.enth_out, z.enth_in1)
link zone_exit_mass(z.mass_out, d.mass_in)
```

Fig. 7.4: Problem specification for recirculation problem.

Inputs		
	Run 1	Run 2
recirc_mass	500	1000
desired_enth	20	13
outside_enth	14	12
heat_flux	1000	2500
Successive Values of Cut Set Variable outside_mass		
	1.00	1.00
	166.66	2499.74
	166.67	2500.000050
		2500.00
Outputs		
outside_mass	166.67	2500.00
exiting_mass	166.67	2500.00
zone_exit_mass	666.67	3500.00
recirc_enth	20.00	13.00

Fig. 7.5: Results of two runs of recirculation problem.

8. FURTHER DEVELOPMENTS

A few suggestions for further development of SPANK are presented in this section. The current version has been written with many of these enhancements in mind, so adding them should be relatively simple.

8.1. Dynamic Problems

Perhaps the most interesting problem for further development is the introduction of dynamic problems. The current system solves only static problems unless great efforts are taken to introduce time dependent equations. Adding the ability for problems to involve derivatives with respect to time would greatly increase the usefulness of SPANK. To do this, a variety of changes would have to be made. First, object interfaces would have to be modified to allow time derivatives to be specified. Second, the ability to solve time derivatives would have to be added. One approach would be to encase the solution program in an iterative solver which would reevaluate the static solution repeatedly, altering the time by some time step Δt. Time derivatives can be treated as normal variables and solved for at each time step. Past values of variables with time derivatives and the values of the derivatives can be input to numerical integration routines to find new values for the variables. This will require introducing a database to store past values of all links, as well as some *intelligence* to control the value of the

time step. The time step value will be critical for controlling the stability of the results.

8.2. Complex Links

A natural extension of links is to allow one macro link to carry a related group of single links. For instance, it might often be natural to link the mass and enthalpy interfaces of one object as a pair to the mass and enthalpy interfaces of another object. The idea of allowing types other than double precision floating points on links has also been considered. The interface of an object would have to be modified to specify the type of the interface. The need to do computations with and solve for varied link types makes this a complex change to implement.

8.3. Testing and Analysis of Networks

A more extensive analysis of the types of networks that would result from simulation problems is needed. There is no currently available set of network defined simulations to analyze so this will probably have to wait until some users have developed problems for SPANK. The nature of networks produced can lead to a better idea of the usefulness of the solution technique.

8.4. Problems with SPANK

SPANK has numerous shortcomings in the current implementation; a few of the more critical are discussed here.

First, each interface of an object requires a functional relation to all other interfaces in the form of a C function. For simple functions, it would be reasonable to have the user input an object's equation only once. A program like MACSYMA could be used to derive the proper form of the equation once a dependent interface has been selected.

Second, there has been a great deal of concern about the data abstraction provided by the objects. As soon as a network specification language program is interpreted, all objects and links are combined into a large network structure. More generality in future solution techniques might be possible if the objects were maintained as abstract data types throughout the specification and solution programs.

8.5. Higher Level Interfaces

The network specification language is not the most elegant interface to specify network simulation problems. It is not difficult to imagine a number of enhancements that would increase the power and decrease the complexity of the language.

Graphical interfaces hold great potential for specifying networks. Since networks are a graphical structure, it is natural to define them using graph drawing packages, many of which are already available. Furthermore, in many problems the network can be drawn to closely resemble the physical situation being

modeled. In the future, this holds the hope for a direct translation from engineer's drawings to a network simulation language program.

9. CONCLUSION

The SPANK system implements a number of new techniques for simulation problems. The idea of defining and solving simulation problems in the form of networks appears to have significant advantages over traditional monolithic programs.

Specifying simulation problems in terms of a network allows data abstraction to be introduced. SPANK objects are instances of abstract data types that hide the implementation of an algorithm. SPANK links allow objects to be joined to form a simulation problem without knowing details of the objects' implementations. The equation abstracted by an object can be modified without changing the way the object is linked into programs. This allows models to be more easily developed and updated than in traditional simulation programs. Sharing of models between different model development sites is also easier; only the definition of a new object type needs to be shared. SPANK allows algorithms to be developed, updated and shared without any changes to any other part of the simulation program unlike traditional programs that often require massive modifications to achieve changes.

The solution method implemented in SPANK, naturally defined in terms of a network, is also an important improvement. Analysis of the solution technique indicates that simulation problems can be solved from eight to sixty-four times faster than with traditional techniques. This increase in speed should make a number of larger simulation problems practical to run.

A number of problems remain in SPANK. While data abstraction holds many advantages, it is not clear that SPANK's choice of physical objects as the unit of abstraction is the most appropriate. Also, the boundaries of this abstraction are violated by the solution program. This may impede future development of more advanced solution techniques. The SPANK network simulation language is primitive; more sophisticated interfaces are needed. Finally, a number of theoretical results are needed to fully analyze the solution method. In particular, the relation between a particular matching and the size of the resulting cycle cut set needs to be examined. A better understanding of this relation could lead to a further reduction in the size of cycle cut sets. Solving these problems in SPANK should lead to even more powerful simulation programs.

It is hoped that SPANK can serve as the foundation of a new generation of simulation systems. These new systems should both simplify and accelerate the solution of simulation problems. Improved simulation systems should in turn lead to improvements in engineering and other fields that are heavily dependent on simulations.

ACKNOWLEDGEMENTS

Thanks are due to Edward F. Sowell, California State University, Fullerton and David P. Anderson, University of California, Berkeley, for their help and guidance. Some of the ideas developed here are based on research done in the Simulation Research Group at Lawrence Berkeley Laboratory, and earlier at the IBM Los Angeles Scientific Center. This material is based upon work supported under a National Science Foundation Graduate Fellowship.

REFERENCES

[ACS75] **ACSL, Advanced Continuous Language, User Guide and Reference Manual,** Mitchell and Gauthier Associates, Inc., Concord, Massachusetts, 1975

[AHO77] Aho, A. V., and J. D. Ullman, **Principles of Compiler Design,** Addison Wesley Publishing Co., Reading, Massachusetts, 1977

[AHO83] Aho, A. V., J. E. Hopcroft, and J. D. Ullman, **Data Structures and Algorithms,** Addison Wesley Publishing Co., Reading, Massachusetts, 1983

[ATK78] Atkinson, K. E., **An Introduction to Numerical Analysis,** John Wiley and Sons, New York, 1978.

[BER73] Berge, C., **Graphs and Hypergraphs,** North Holland Mathematical Library, New York, 1973.

[CLA85.1] Clark, D. R., **HVACSIM+ Building Systems and Equipment Simulation Program Reference Manual,** U.S. Department of Commerce, National Bureau of Standards, Gaithersburg, Maryland, January, 1985.

[CLA85.2] Clark, D. R. and W. B. May, Jr., **HVACSIM+ Building Systems and Equipment Simulation Program Users Guide,** U.S. Department of Commerce, National Bureau of Standards, Gaithersburg, Maryland, October, 1985.

[CLA85.3] Clarke, J. A., **Energy Simulation in Building Design,** Adam Hilger, Ltd., Bristol, U. K., 1985.

[CON77] **Continuous System Simulation Language Version IV User's Guide and Reference Manual,** Nilsen Associates, 1977.

[DAV82] Davis, A. L., and R. M. Keller, *Data Flow Program Graphs,* **IEEE Computer,** February, 1982.

[DOE81] **DOE-2 REFERENCE MANUAL,** LBL-8706 Rev. 2, Simulation Research Group and Group WX-4, Los Alamos National Laboratory, May 1981.

[GON84] Gondran, M., and M. Minoux, **Graphs And Algorithms,** John Wiley and Sons, New York, 1984.

[KAR81] Karp, R. M., *Reducibility Among Combinatorial Problems,* in R.E. Miller and J.W. Thatcher, **Complexity of Computer Computations,** Plenum Press, New York, 1981.

[KLE76] Klein, S. A., W. A. Beckman and J. A. Duffie, *TRNSYS - A Transient Simulation Program,* **ASHRAE Trans.** vol. 82, 1976.

[LEV83] Levy, H., and D. W. Low, **A New Algorithm for Finding Small Cycle Cut Sets,** IBM Scientific Center, Los Angeles, June, 1983.

[McG82] McGraw, J. R., *The VAL Language: Description and Analysis,* **ACM Transactions on Programming Languages and Systems, Vol. 4, No. 1,** January, 1982.

[PLA85] **Plan for the Development of the Next Generation Building Energy Analysis Computer Program (Working Document),** Simulation Research Group, Lawrence Berkeley Laboratory, Berkeley, California, 1985.

[PRE73] Preparata, F. P., and R. T. Yeh, **Introduction to Discrete Structures,** Addison Wesley Publishing Co., Inc., Reading, Massachusetts, 1973.

[ROB84] Roberts, F. S., **Applied Combinatorics,** Prentice Hall Inc., Englewood Cliffs, New Jersey, 1984.

[SIL81] Silverman, G. J., et al., **Modeling and Optimization of HVAC Systems Using Network Concepts,** ASHRAE Annual Meeting, Cincinnati, Ohio, June, 1981.

[SOW84] Sowell, E. F., et al., *Generation of Building Energy System Models,* **ASHRAE Transactions, AT-84-11,** November, 1984.

[STA79] Stakgold, I., **Green's Functions and Boundary Value Problems,** John Wiley and Sons, New York, 1979.

[STR84] Stroustrup, B., *Data Abstraction in C,* **A.T.&T. Bell Laboratories Technical Journal, Vol. 63, #8,** October, 1984.

[TAR83] Tarjan, R. E., **Data Structures and Network Algorithms,** Society for Industrial and Applied Mathematics, Philadelphia, Pennsylvania, 1983.

LAWRENCE BERKELEY LABORATORY
TECHNICAL INFORMATION DEPARTMENT
UNIVERSITY OF CALIFORNIA
BERKELEY, CALIFORNIA 94720