COMPUTER AIDED DESIGN OF
STRUCTURAL FLOORING SYSTEMS

by

Albert Marshall Bleakley Jr.
B.S., United States Military Academy
(1979)

SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS OF THE
DEGREE OF
CIVIL ENGINEER

at the

MASSACHUSETTS INSTIITUTE OF TECHNOLOGY

December 1984

© Massachusetts Institute of Technology 1984

Signature of Author

Department of Civil Engineering
December 20, 1984

Certified by

Robert D. Logher
Thesis Supervisor

Accepted by

Francois M. M. Morel
Chairman, Departmental Committee on Graduate Students

Vol. 4
MASSACHUSETTS INSTITUTE
OF TECHNOLOGY

JUN 17 1985
UPEP RIES Archives
COMPUTER AIDED DESIGN OF
STRUCTURAL FLOORING SYSTEMS

by

Albert Marshall Bleakley Jr.

Submitted to the Department of Civil Engineering on December 26, 1984 in partial fulfillment of the requirements for the Degree of Civil Engineer

ABSTRACT

This thesis develops techniques and computer programs for the analysis of oneway or twoway structural flooring systems consisting predominantly of simply supported members. While indeterminant frame and finite element analyses software packages are widely available, these packages are not efficient for floor framing systems or easy to use. Member selection and code checking modules also exist in these packages, however, hand calculations must be done, and the data input into the member selection routine: a process more time consuming than the design itself. For these reasons, this class of problem is not well suited to solution by standard analysis programs.

This thesis describes a set of programs to efficiently perform the analysis of oneway and twoway floor framing systems. The scope of this thesis is limited solely to the load takedown and analysis of the flooring system. Before these programs can conveniently be applied to actual design, two more modules, a graphical data input editor, and a member selection and code checking module are needed.

Thesis Supervisor: Robert D. Logcher, ScD

Title: Professor of Civil Engineering
ACKNOWLEDGEMENT

The author would like to express his appreciation to Professor Robert D. Logcher for making this work possible, and to Mr. D. Fraser Sinclair for his helpful insight on design practice.

"I can do all things through Christ which strengtheneth me."

Philippians 4:13 (KJV)
CONTENTS

Title page 1
Abstract 2
Acknowledgement 3
Table of Contents 4
CHAPTER I Introduction 6
CHAPTER II Data Management 11
  2.1 Data Structures in C 11
  2.1 Data Structures in Flooring Systems 16
CHAPTER III Graphical Data Manipulation 22
  3.1 Graph Representation of a Flooring System 22
  3.2 Adjacency Determination 25
  3.3 Area Bounding 29
  3.4 Path Testing 32
  3.5 Path Shortening 33
  3.6 Geometric Algorithms 34
CHAPTER IV Analysis of a Floor 41
  4.1 Analysis Control 41
  4.2 Load Areas for Oneway Slabs 49
  4.3 Load Areas for Twoway Slabs 53
  4.4 Intersection of Floor Loads and Member Areas 55
  4.5 Member Load Generation for Tributary Areas 61
  4.6 Evaluation of Behavioral Assumptions 66
CHAPTER V File Operations 69
  5.1 Saving a Floor 69
  5.2 Retrieving a Floor 71
  5.3 Output of Results 73
CHAPTER VI Sample Problem 75
  6.1 Description and Design of the Problem 75
  6.2 Input 77
  6.3 Processing Commands 77
  6.4 Validity Checks on Results 77
CHAPTER VII FUTURE DEVELOPMENTS

7.1 Graphical User Interface
7.2 Additional Analysis Modules
7.3 Member selection
7.4 Current limitations

Bibliography

APPENDIX A User's Manual

A.1 Input File Formats
A.2 Command Menu Structure

APPENDIX B Program Documentation and Internal Logic
APPENDIX C Program Listings
APPENDIX D Data Structure Documentation
APPENDIX E Data Structure Listings
CHAPTER I

INTRODUCTION

Computers have been used extensively in Civil Engineering for the last three decades. The computer is a particularly well suited tool for the engineer into two categories of applications. The first category includes numerical techniques, such as finite element analysis, which demand a very large number of calculations and exceedingly high accuracy, possibly 15 digits or more for chained matrix operations. Such techniques would literally be impossible without the assistance of a computer to perform the calculations. A second category of problems involves routine, repetitive, simple, but nonetheless time consuming calculation tasks such as member selection. The computer's capacity for fast retrieval of data from member tables, and ability to rapidly check alternatives permits the engineer to concentrate more on the exceptional or unique aspects of the structure, such as configuration.

The trend in the use of computers over the past twenty years has been towards less and less centralization of computing resources. Even small engineering firms which in the past have leased computer time are now able to afford to purchase their own computers at modest prices. The emerging generation of micro and supermicro computers will encourage and accelerate this trend. Because the computations involved in analyzing a floor are not computer intensive, floor analysis and graphical display can easily be implemented on
mini or micro computers.

This trend toward decentralization will mean that individual engineers will have engineering workstations capable of high resolution graphics and sophisticated calculations, just as engineers now use personal calculators to assist them. The typical individual workstation will probably have a 16 or 32 bit architecture, high resolution graphics on the order of 1000x800 pixel resolution, one to three megabytes of core memory, access to hard disk storage, and floating point arithmetic in the hardware. These engineering workstations have been available from CAD firms for some years, but at a price in the $40,000 range. It is now possible to obtain a microcomputer with these capabilities for well under $10,000, and this favorable price/performance trend shows no sign of reversing.

With these trends and capabilities in mind, a wider range of useful engineering tools can be developed. The state of the art in the first category of computer applications mentioned earlier has progressed to the point that nearly any indeterminate analysis problem may now be modelled with the computer. In the second category of application, however, there is one area of design where the computer has made little impact in general practice: the design of the members in beam/girder structural flooring systems. The design of flooring systems clearly fits into the second category of computer applications described earlier in that it is both highly routine and time consuming, but each step in the
process is quite simple. Because most flooring systems are comprised of members assumed to be simply supported, the stiffness methods used for frame analysis are not efficient for such a simple problem. Although member selection programs are commonly available in Civil Engineering software such as STRUDL(7), the engineer still must first perform a load takedown manually because of the analysis inefficiency. Data input into the member design for automated selection, however, requires more effort than the design itself, thus making the use of the computer solely for member selection impractical. Computerization of this problem is effective only if the entire process can be integrated.

The goal of this thesis was to investigate the possibility of writing an integrated package of programs to efficiently perform the analysis of oneway or twoway planar, predominantly simply connected (cantilevered members are allowed with some restrictions) structural flooring systems. The major objectives involved in this goal are: 1) the identification of member tributary areas, 2) the transfer of floor loads on these areas to the supporting members, and 3) load takedown through the flooring system to the supporting columns. Load takedown in either a oneway or twoway flooring system includes automatically solving for the member end reactions, and transferring these loads to supporting members to find the design moments and shears. Ideally, such a program should also include a graphical interface which would allow the engineer to input and alter a flooring system by altering the drawing, just as would currently be done on paper. Although
it was originally hoped that at least part of the graphical
interface could be implemented as a part of this thesis,
delays in the availability of an appropriate graphics package,
and the size of the analysis code have precluded any develop-
ment in this area. The other area that has not been included
here is the use of member selection routines. These routines
have been used extensively for some time in commercially
available products; the interested reader should consult the
STRUDL User’s Manual (7) or similar software for a description.

This thesis was implemented using the C programming lan-
guage for three reasons: first, C is a modern structured lan-
guage that allows the development of a powerful data structure
and modular, easily modified programs; second, C allows for
dynamic memory allocation and recursive functions which are
essential to the implementation of some of the algorithms used
in this thesis; third, programs written in C language are more
easily portable to a variety of operating environments than
programs written in most other languages.

Chapter II introduces the data management capabilities of
the C language and discusses of the use of these capabilities
in the modelling and analysis of a flooring system. Chapter
III introduces the mathematics of graph theory and the appli-
cation of the theory to a flooring system. This introduction
is followed by a discussion of algorithms that use the graph
model to associate members with the areas of floor which they
support. Chapter III concludes with a discussion of two dim-
ensional geometric algorithms. Chapter IV covers the analysis
of a floor, beginning with the algorithms used for load
takedown and member analysis, and for finding member tributary
areas in either one-way or two-way flooring systems. Next comes
a discussion of the intersection of floor loads and member
tributary areas to determine the loaded area for a member, and
the algorithms to translate this loaded area into loads on the
member. The chapter concludes with a discussion of the
assumptions made during the load takedown. Chapter V explains
the file operations that the program performs. Chapter VI
contains a sample two-way flooring analysis. Chapter VII is a
discussion of the current limitations of the program and
recommendations for future developments. Appendix A, the
user’s manual, details the input file format and command
structure of the program. Appendix B documents internal logic
for individual routines, and includes block diagrams for the
major routines. Program listings are contained in Appendix C.
The data structures used in this program are diagrammed in
Appendix D. The final appendix, Appendix E, contains listings
of the data structures.

As indicated in the outline above, we will now move to
the next chapter, Data Management, to examine the data man-
agement capabilities of the C language in the context of
flooring system analysis.
CHAPTER II
DATA MANAGEMENT

The design of the data management system used to represent a flooring system is influenced by the algorithms used for analysis, and in turn, impacts the ease of writing and modifying the programs which implement these algorithms. As was indicated in the introduction, one of the reasons that this thesis is implemented in the C language is that C provides powerful tools for data abstraction. In this chapter, the C language data structure will be explained, and the application of these C data structures in this program will be examined.

2.1 Data Structures in C

Maintaining the information that defines a floor requires an internal storage strategy. In FORTRAN, a typical structure would use parallel arrays to keep the various data about the nodes and members. The C language provides a more flexible facility for logically grouping information, a user definable data structure (28,30). A data structure may be comprised of any arbitrary combination elements of other data types, e.g. integers, characters, real numbers, or other data structures. Within the definition of a data structure, each individual element is given a name. Once the data structure has been defined, it acts as a template for use in the program.

As an example, we will consider the data structure used to represent a node. The minimum information need to define a node is its number and its coordinates.
struct node {
    int node_number;
    double x, y, z;
},

The C keyword "struct" indicates that what follows is a data structure declaration. The name of this particular structure is "node". Declaration of a structure does not allocate any memory, it merely defines the structure as a data template. Once the template has been defined, an instance of a node may be created by defining a variable of the type node structure:

    struct node a;

The above line will cause memory to be allocated for a variable named "a" which is a node data structure. Individual elements (members) of the data structure may be accessed by using the operator ".". The following C statement sets the x coordinate of node a to 5.5:

    a.x = 5.5;

Single nodes are useful, but a floor is a collection of nodes. Arrays of data structures can be created to keep track of the nodes, but as in FORTRAN, the size of arrays must be determined at compile time. However, as was stated in the introduction, C also provides dynamic memory allocation, that is, memory can be allocated as needed by a program while it is running. In order to reference this dynamically allocated memory, C provides a pointer data type, and two unary address operators, "&" which finds the memory address of an object, and "*" which indicates that its object is a memory address (or pointer). Pointers are variables which contain the address in memory of another variable, instance of a data
structure, or function. The declaration of a pointer variable is similar to the declaration of a normal variable, except for the unary operator "*":

```c
int *node_pointer;
```

declares a variable named "node_pointer" which is a pointer to (address of) an integer variable.

By combining pointer variables with the ability to create instances of the node data structure "on the fly", we introduce a new method of managing data: the linked list (20). Linked lists function somewhat like arrays in that they are composed of a sequence of like data structures, however, unlike arrays, arbitrary sized linked lists can be created while a program is executing. If a floor has 40 nodes, then a linked list of 40 elements can be created. If the floor has 400 nodes, then a linked list with 400 nodes can be created. The following diagram represents a linked list with three members. The arrows represent the pointers discussed above.

```
   A
    next   B
          next   C
             NULL
```

In order to implement the linked list concept, we will make a minor change to the nodes data structure introduced earlier, by adding a pointer to the next node structure:

```c
struct node {
    int node_number;
    double x, y, z;
    struct node *next;
};
```

It may appear that the node data structure is recursively
used within itself, but the element *next is seen to define a
pointer to another instance of the node data structure, not
the node data structure itself. With this data structure, we
can make a linked list by setting the "next" pointer in the
last node as new nodes are added. The end of the list is
marked with a NULL pointer (value 0) for the "next" pointer.

There are two reasons for using linked lists to represent
data: first, it is not necessary for the program to know before
hand how long the list will be, and second a list may be rear-
ranged by simply changing the pointers. For example, one
application of the linked list is the representation of a
member's tributary area as a linked list of the vertices of the
area. As an example, consider the following polygon:

```
A

E  D  C

B
```

In the figure, A, B, C, D, and E, the vertices of the
area, are nodes. The linked list representation for this
area would look like this:

```
A  B  C  D  E

next  next  next  next  null
```

If a new node, F, is added, only the pointers need to be
readjusted:
The new linked list would look like this:

```
A ----> B ----> C ----> D ----> E
   ^     |     |     |     |
ext  next next next next
```

Removing a node from the list is also a simple operation. To remove node D, the pointers are simply rearranged so that node C points to node E rather than to D.

```
A ----> B ----> C ----> E
   ^     |     |     |
ext  next next
```

The linked list now looks like this:

```
A ----> B ----> F ----> C ----> D ----> E
   ^     |     |     |     |     |
ext  next next next next
```

Notice that while node D still points to node E, there is no way to arrive at node D in the linked list. If node D's address is not kept somewhere else, it has become "lost", and cannot be found by the program. If node D is not needed for
any other reason, its memory could be freed for reallocation later in the program.

2.2 Data Structures in Flooring Systems

Now that data structures in the C programming language have been introduced, their implementation in this program's data storage strategy will be discussed in more detail (see Appendix D for diagrams of the data structures used in this program, and Appendix E for listings). Any explanation of data representation for this program must begin with a discussion of what items of data are needed to describe a flooring system. The flooring system is assumed to be planar, and members to be straight, so that a floor may be described as a set of members and nodes (connections). Coordinates are a part of the node data structure and a member's geometry is completely defined by its beginning and end nodes. In addition, members may "own" nodes that represent connections where other members frame into the owner member at points between its ends. This connectivity data is maintained in a linked list of nodes on the member. The pointer to (address in memory of) the beginning of this list is a part of the member data structure. While it would be sufficient to keep the connectivity information only in the member data structure, this would require searching through the list of members to find nodes. To make searches through the floor more efficient, the node data structure also includes a pointer to a linked list of members that end or begin at the node.

The geometry and connectivity data for a floor is accessed through two linked lists: a linked list of nodes in
the structure, and a linked list of members. As indicated above, the node data structure (see Appendix D) includes connectivity information as well as geometric information. The list of members that frame into a node is accessed by the pointer to the beginning of a linked list of frame_in_list data structures which in turn contain pointers to the appropriate members. The node data structure also contains an integer node number to identify the node, and pointers to the member that the node is on (if it is not a column), to a list of nodal loads, and to the next and last nodes in the node list. The node also contains a structure named "flags" which is a series of one bit flags to indicate the support conditions of the node.

The members are also maintained a linked list. Like the node data structure, the member structure contains an integer member number and pointers to the next and last members in the list. The member structure also contains bit flags to indicate whether it has been analyzed yet, whether it is a real member or a fictional member inserted to bound an area during load takedown (see section 3.1), and whether the member is a cantilever. The member data structure, also contains a pointer to a linked list of node_list structures to indicate which nodes are on the member, and pointers to the member's beginning and end nodes. Member load data is accessed through a pointer to a linked list of load_case data structures, and member tributary areas are accessed through a pointer to a linked list of areas. Finally, the member data structure
contains a pointer to a section data structure that contains data about the member properties (not currently used).

The load_case data structure contains an integer load_case number, a reaction data structure for each end of the member, a pointer to a linked list of member load data structures, and a pointer to the next load_case in the list. Member loads are represented in the load data structure by a beginning and ending local coordinate (0 at the beginning of the beam, 1 at the end), and a beginning and ending magnitude. Administrative information, the date assigned, and the user ID number of the person that assigned the load are also a part of the structure. A coordinate flag is included in the structure, but not currently used. Finally, the load structure contains a pointer to the next load in the list.

Member tributary areas are represented in the area data structure. This data structure contains a pointer to a circular linked list (explained below) of corner data structures that bound the area, the magnitude of the area (not currently implemented - see Chapter VII), and a pointer to the next area in the list. Each corner contains the coordinates \((x, y, z)\) of one point, and a pointer to the next corner in the list. The last corner points back to the first corner in the area, making the list circular.

During load take down (Chapter 4), the member's loads are summed for each load case. The results of this summation are maintained in the reaction data structures mentioned earlier. Each reaction structure contains a moment, a shear, and an axial force. Since each member has two ends, two reaction
data structures are used in each load_case structure.

The final data structure necessary to describe a flooring system is the floor_load data structure. The floor_load structure is similar to the member load data structure in that it contains data on who assigned the load, the date that it was assigned, and a coordinate flag. Floor loads must be uniform, so only one magnitude is included in the structure. The area over which the load acts is accessed by a pointer to a linked list of h_area data structures define the area. The corner points of a floor load must be defined in a clockwise direction, and the floor load must be convex in order for the member tributary area - floor load intersection algorithm (section 3.6) to function correctly.

Due to the incorporation of homogeneous coordinates (section 3.6), the h_area (homogeneous area) data structure differs from the area data structure used in the member data structure. Each h_area contains pointers to an h_point (homogeneous point) data structure, and to the next h_area. Like the corner representation above, the h_area representation is a circular list. Each h_point data structure contains four homogeneous coordinates (x, y, z, w), where x, y, and w are the homogeneous coordinates of a point in the x-y plane.

Using the data structures enumerated above, a floor data structure is defined which contains pointers to the member list, node list, floor load list, and to the opening list (floor openings use the same data structure as floor loads). The relationship of the data structures is shown in figure 2.1.
Indicates a linked list of data structures.

FIGURE 2.1 DATA STRUCTURE DESIGN
Additional data structures, pathnode, qnode, listnode, and the adjacency list model of a floor, are used to perform graphical data manipulation. These data structures, and their uses will be examined in Chapter III.
CHAPTER III

GRAPHICAL DATA MANIPULATION

In this program, a floor is represented as a linked series of physical members and nodes (connections). This arrangement allows member load take down to be performed quickly, once the loads upon the member are known. In flooring system design, however, the majority of the structural loads are not expressed as member loads, but rather as floor loads. The program must then be able to take floor loads distributed over an area, and distribute them as member loads on the appropriate members. This requires that the program first create a model of the flooring system that efficiently represents its connectivity data, then the connectivity model must be used to identify parts of the floor which are bounded by members, and finally the program must divide the bounded areas, assigning parts of the floor area to each of the bounding members in accordance with the framing type. The details of how these steps are implemented will now be discussed.

3.1 Graph Representation of a Flooring System

Describing a member's tributary area is relatively simple if we consider an intuitive approach to a oneway flooring system. A human designer would look at the drawing, decide which members lie on the left and right of the area to be supported, and assume that the floor acts as a simple beam. For uniformly loaded sections each end member would support 1/2 of the floor load. Twoway floors are usually assumed to
carry loads to the closest member to a point on the floor. In practice, this means drawing bisectors at each corner of the section, intersecting the bisectors to determine the closest member for a given part of the floor, and assuming that each bounding member carries the floor load in its area.

In order to create a general algorithmic approach to this problem, it is necessary to formalize the test for which members support which part of the floor. In essence, the computer must determine what floor area a given member supports. This reduces to finding sets of members, or member segments, which bound an area and has no other members inside it. In this model, some members, such as the member between nodes 1 and 14 in figure 3.1, have nodes on them which they support. The parts of the member between these interior nodes are represented separately as segments in the floor model. In figure 3.1, the segment of member 1-14 between nodes 2 and 3 bounds area E, as do members 2-7, 3-8, and member segment 7-8. These members must directly support that section of floor. By examining both sides of all of the members, the entire floor area will be allocated.

![Figure 3.1 Circuits in a Floor](image-url)
The mathematics of graph theory (18,19) may be applied to identify these areas. The solution starts with finding all of the closed loops (circuits in graph theory) that include any segments of a member. Figure 3.1 shows a section of floor that is divided into five areas, A through E. The nodes are numbered 1 through 16; the members are not numbered, but will be referred to by their begin and end node pairs, e.g. member 1-14 contains internal nodes 2, 3, and 4. If the flooring system is one-way in the horizontal direction, member 1-14 does not directly support any of the areas. In a two-way system, member 1-14 would directly support part of each of floor areas A-E. Notice that each of the closed loops representing areas A-E include a part, though not necessarily all of the physical member 1-14.

In order to model the floor’s area-support relationships, the floor is represented as a graph. In graph terminology, the floor is represented as a set of vertices (nodes), edges (members), and connectivity information. Since the closed paths that represent areas that must be supported may not include an entire physical member, the graph’s edges will represent member segments – the shortest path between nodes. If part of the floor slab is cantilevered so that there is no member bounding it, a fictitious edge beam(s) must be created to bound the area. A fictitious member is treated as if it were a real member to distribute loads during the analysis of the floor. Nodes, on the other hand, remain unique, so the node numbers already assigned may be used to identify them.
3.2 Adjacency Determination

A graph may be represented as an adjacency list (20). In this representation, each vertex is assigned a linked list of other vertices that are connected to it. This is done via a one-dimensional array of N pointers to linked lists. Element 1 of the array points to the linked list of vertices connected to vertex number 1. Lookup is very fast because finding the neighbors of vertex \( j \) simply involves looking up the pointer at element \( j \) of the array and traversing the linked list. In the adjacency list model below, the ampersand indicates that the members of the array are memory addresses of (pointers to) the linked lists \( L1...Ln \). Similarly, the ampersand within the linked list indicates that the elements of the list are pointers to node 1 (\&1) and to node 5 (\&5).

Array adjlist[N]

\[
\begin{align*}
& \&L1 & \&L2 & \&L3 & \&L4 & \&L5 & \&Lj & \&LN
\end{align*}
\]

List 1  \[ \begin{array}{c}
\&1 \\
\text{next} \\
\Rightarrow \\
\&5 \\
\text{NULL}
\end{array}\]

Before this adjacency list representation can be used to examine the floor, it must be created from the connectivity information supplied during user definition of topology and geometry. The user input consists of identifying the locations of nodes, the member that the node is on (if it is not a column), and the members that connect the nodes together. It is possible to derive the adjacency list using only this information. Consider the following floor:
For the purposes of this example, assume that the nodelist for the above floor is:

1->2->3->4->5->6->7->8->null

Beginning at the head of the linked list of nodes, the first step is to find the nodes that are neighbors of node 1. There are two possible types of neighbors: nodes that are on members that frame into node 1, and, if node 1 is on a member, nodes that are also on, or at the end of that member. The order in which these two types of neighbors are searched is arbitrary; we will begin our search by looking for the first type. To determine which members frame in to node 1, we loop through node 1's frame_in_list. The first member in the list is member 1. Member 1 is then examined to see whether it contains any intermediate nodes, which could be neighbors to node 1. Because there are no intermediate nodes on member 1,
its node_list is NULL. A check is then made to see whether node 1 is the beginning or ending node for member 1. Since node 1 is member 1's beginning node, its ending node, node 3, must be node 1's neighbor.

When a neighbor is found, an edge is added to the adjacency list. This addition involves looking up node 1 in the adjacency list and traversing its linked list of edges. If node 3 is not in the list, a new element is linked to the end of the list. Node 1 is also added to node 3's adjacency list. The adjacency list now looks like this:

Array adjlist[N]

\[0\] \[1\] \[2\] \[3\] \[4\] \[5\] \ldots \[j\] \ldots \[N\]
&L1 &L2 &L3 &L4 &L5 &Lj &LN

List 1

\[ NULL \]

List 3

\[ &1 \]

The next member in node 1's frame_in_list is member 4. Unlike member 1, member 4, node_list is not NULL, indicating that member 4 contains at least one intermediate node. By traversing the list of nodes that are on member 4 (the list contains only one node in this case), node 5 is found to be the next neighbor. Again an edge is added to the adjacency list, giving:

Array adjlist[N]

\[0\] \[1\] \[2\] \[3\] \[4\] \[5\] \ldots \[j\] \ldots \[N\]
&L1 &L2 &L3 &L4 &L5 &Lj &LN
The procedure continues until all of the nodes have been examined, resulting in the following adjacency list:

Array adjlist[N]

[0] [1] [2] [3] [4] [5] ... [j] ... [N]
&L1 &L2 &L3 &L4 &L5 &Lj &LN

List 1

&3 \(\rightarrow\) &5
next \(\rightarrow\) NULL

List 2

&5 \(\rightarrow\) &7
next \(\rightarrow\) NULL

List 3

&1 \(\rightarrow\) &8
next \(\rightarrow\) NULL

List 4

&7 \(\rightarrow\) &8
next \(\rightarrow\) NULL

List 5

&1 \(\rightarrow\) &2
next \(\rightarrow\) &6
next \(\rightarrow\) NULL

List 6

&3 \(\rightarrow\) &8
next \(\rightarrow\) &7
next \(\rightarrow\) NULL

List 7

&2 \(\rightarrow\) &4
next \(\rightarrow\) &6
next \(\rightarrow\) NULL

List 8

&3 \(\rightarrow\) &4
next \(\rightarrow\) &6
next \(\rightarrow\) NULL
3.3 Area Bounding

The function PATH uses a breadth first search algorithm (20) to find the two shortest (defined as the number of graph members traversed) paths on opposite sides to get from the beginning node to the end node of a graph member. These two paths correspond to the areas which the member supports in the floor framing system. During the search, an array is maintained to indicate whether a node has been seen (visited) in the current search. Nodes which have not yet been visited are added to a queue to await examination. By adding the most recently seen nodes to the end of the queue, the search always resumes with the oldest, or least recently encountered node. This queueing method ensures that all equally distant nodes will be examined before moving to nodes that are farther away, e.g. all nodes that can be reached by traversing one graph member will be examined before examining any nodes that require two graph members to be traversed.

As an example, consider the functioning of the algorithm in finding the tributary area for graph member 4-5 in figure 3.3.

![Figure 3.3 Bounded Areas](image-url)
The algorithm first temporarily deletes the graph member joining the two nodes (member 4-5). Next the search begins at node 4 and looks at the neighbors of node 4. Neighbors are found by looking node 4 up in the adjacency list and traversing the linked list of neighbors until the end of the list (next = NULL) is found. If the neighbor has not been visited yet (visited[i] = 0), then the pointer for that node is added to the end of the priority queue, and the queue size is incremented. Each new node that is visited has its parent set to the node which caused the new queue entry (node 4 in this case). After finishing at node 4, the priority queue looks like this:

* qsize = 2
[0] [1] [2] [3]
node # 1 7
\* qposition

note: actually the queue contains pointers to nodes & parents

When all neighbors of node 4 have been examined, the algorithm looks at the node at the current qposition (node 1). Examining node 1's neighbors, node 4 is found to have been visited, so it is not added to the queue. Node 2 has not been visited, so qsize is incremented, node2 is marked as visited, and added to the queue. Since there are no other neighbors, the algorithm increments the qposition and goes on to the node at that position.

* qsize = 3
[0] [1] [2] [3]
1 7 2
\* qposition
Examining node 7’s neighbors, node 4 is found to have
been visited, so it is not added to the queue. Node 8 has not
been visited, so qsize is incremented, node 8 is marked as
visited, and added to the queue. Since there are no other
neighbors, the algorithm increments the qposition and goes on
to the node at that position.

* qsize = 3
[0] [1] [2] [3]
 1 7 2 8  * qposition

Examining node 2’s neighbors, node 1 is found to have
been visited, so it is not added to the queue. Node 5 is the
desired end of the search, so a path has been found. The
algorithm pauses to create a path: the length of the path
(number of graph members) followed by a list of pointers to
nodes that must be followed to travel in a circuit. In this
case, the path would be:

(4) 4 1 2 5 4
(pointers to nodes)

If the member being examined has two sides in the
structure, as member 4-5 does, then the search will be
continued until node 5 is found a second time. Notice that
node 5 was not marked as visited after being found the first
time. Since there the desired second path must be on the
opposite side of the member, no other neighbors of node 2 need
to be examined. In another configuration, node 3 could be on
the physical member that graph member 4-5 is a component of,
so a legal second path might have to pass through it. Node 3
is therefore is left unvisited. The algorithm increments the
qposition and goes on to the node at that position.

* qsize = 4

[0] [1] [2] [3]

1 7 2 8

qposition = 4

After visiting node 8:

* qsize = 5

[0] [1] [2] [3] [4] [5]

1 7 2 8 6

qposition = 5

On visiting node 6, node 5 is found a second time. The second path would be:

(5) 4 7 8 6 5 4

To ensure that the two paths are on opposite sides of the member, a test, described in section 3.4, is performed. If the test passes, the two paths are accepted and passed back for processing. Should the test fail (indicating that the two shortest closed circuits lie on the same side of the member), the search resumes until a legal second path is found.

3.4 Path Testing

As discussed in section 3.3, once the first path is found for a member, all subsequent paths found must be checked to ensure that they do not lie on the same side of the member. This testing is done by the TESTPATH function which evaluates a pair of paths to determine whether they lie on opposite sides of a member. Because PATH begins all searches at the same point, paths on one side of the member must be traversed clockwise, while those on the other side must be traversed.
counterclockwise. The first and last nodes in any path must be the end nodes of the graph member. The actual test is performed by calling the RIGHT_OF function (see section 3.6) to evaluate whether the second point in the path is to the right of, left of, or colinear with the ends of the graph member. If the second point is colinear, subsequent points are tested until a point is found to the right or left. In order for the second path to be legal, it must turn the opposite direction from the first.

3.5 Path Shortening

The function FILTER processes paths found by the function PATH to remove any nodes that are not at corners, i.e. nodes which lie on, but do not connect the sides of the area found by PATH. There are two reasons for removing these extra nodes: first, they must be removed for the ONEWAY or TWOWAY functions which determine the tributary area for the member since corner to corner distances are used; and second, they must be removed for the UNIQUE function which checks for duplicate paths.

FILTER works by examining path nodes in sets of three. The function CORNER is called to take node pointers for the three nodes, translate them into homogeneous coordinates (see section 3.6), and call the RIGHT_OF function to determine the position of the third point with regard to the first two. Since the first two points define a line, the third point must be to the left of, to the right of, or colinear with the other two points. If the three points are colinear, the middle
point (node) is deleted from the list, all of the following nodes are moved up, and the test is repeated until the first three points are not colinear (indicating that a corner has been reached).

Once a corner has been found, the algorithm repeats the process with the second, third and fourth nodes. The algorithm continues until the n-2th, n-1th, and nth nodes are tested.

3.4 Geometric Algorithms

Data analysis for a floor framing system requires several algorithms for solving two dimensional geometric problems. These include line intersection, determination of colinearity, and orientation checking (23). In order to implement these algorithms easily and efficiently, homogeneous coordinates are used (21). In two dimensions, three values are required to describe a point. In addition to the familiar x, and y cartesian coordinate, there is also a scaling coordinate, w. When w is one, the first two homogeneous coordinates map directly to cartesian coordinates, and vice versa. If w is not one, then the x and y values must first be divided by w in order to map into cartesian coordinates.

There are several reasons for using homogeneous coordinates. First, they are convenient for performing transformations on coordinates. For example, a 2D rotation may be performed in cartesian coordinates by multiplying the vector which represents the desired point by a transformation matrix containing the direction cosines of the desired rotation. Any series of rotations may be combined by multiplying a series of
transformation matrices together. If a translation is desired, however, matrix multiplication will not suffice.

Instead scalars must be added to the elements of the vector being transformed. Scaling requires the multiplication of each of the vector's elements by a scalar. There are now three different operations must be performed on a vector (point) to translate, scale, and rotate it. If an image consisting of 500 nodes is to be translated, scaled, and rotated, 500 identical multiplications, additions, and matrix multiplications will be required.

If homogeneous coordinates are used, translation and scaling can be represented as matrix multiplications, permitting the scaling, translation and rotation to be combined into a single transformation matrix. Each point may then be multiplied by this single matrix, saving 500 additions and multiplications over the cartesian representation.

Figure 3.4 shows several examples of transformations of the point (5, 2) \((5, 2, 1)\) using both cartesian and homogeneous coordinates.

It might seem from the above examples that homogeneous coordinates only complicate calculations. For single transformations, this would appear to be true, however if we return to the earlier problem of simultaneously translating, scaling,
cartesian

Scale by a factor of S

\[ \begin{bmatrix}
1 & 0 & 0 & ; & 5 \\
0 & 0 & 1 & ; & 2 \\
0 & 1 & 0 & ; & 1 \\
\end{bmatrix} \]

Translate by \((dy, dx)\)

\[ \begin{bmatrix}
1 & 0 & dy & ; & 5 \\
0 & 1 & dx & ; & 2 \\
0 & 0 & 1 & ; & 1 \\
\end{bmatrix} \]

Rotate 45° clockwise about origin

\[ \begin{bmatrix}
cos 45 & -sin 45 & ; & 5 \\
\end{bmatrix} \]

\[ \begin{bmatrix}
sin 45 & cos 45 & ; & 2 \\
\end{bmatrix} \]

\[ \begin{bmatrix}
0 & 0 & 1 & ; & 1 \\
\end{bmatrix} \]

**FIGURE 3.4 COORDINATE TRANSFORMATIONS**

and rotating an object, the advantage of homogeneous coordinates becomes clearer. All three operations can be achieved by a single matrix multiplication:

\[ \begin{bmatrix}
cos 45 & -sin 45 & dy & ; & 5 \\
\end{bmatrix} \]

\[ \begin{bmatrix}
sin 45 & cos 45 & dx & ; & 2 \\
\end{bmatrix} \]

\[ \begin{bmatrix}
0 & 0 & 1 & /S & 1 \\
\end{bmatrix} \]

Although additional calculations will be involved in initially creating the transformation matrix, these calculations need be performed only once regardless of the number of points being transformed (500 nodes in a floor system, for example).
Aside from the simplicity of performing coordinate transformations, homogeneous coordinates in two dimensions exhibit some very useful properties in determining the equation of a line through two points, and in finding the intersection of two lines. The equation of a line written in general form is:

$$Ax + By + C = 0$$

Notice that three quantities are required to describe a line, just as three quantities are required to describe a homogeneous point. The general form of the equation for a line has the added advantage of representing both horizontal and vertical lines without the numerical difficulties that occur with a zero or infinite slope in other forms of the line equation.

In order to find the intersection of two lines:

$$A_1x + B_1y + C_1 = 0$$
$$A_2x + B_2y + C_2 = 0$$

an arbitrary third line through the intersection \((x, y, w)\) of the first two is created:

$$Ax + By + C = 0$$

and the following matrix can be written:

$$\begin{bmatrix}
A_1 & B_1 & C_1 & | & x \\
A_2 & B_2 & C_2 & | & y \\
A & B & C & | & w
\end{bmatrix}$$

If the third line passes through the intersection, it must be linearly dependent on the first two, so the determi-
nant of the matrix must be zero. Taking the determinant by expanding on the last row yields:

\[ A(B1C2 - C1B2) - B(A1C2 - C1A2) + C(A1B2 - B1A2) = 0 \]

Comparing this equation to the general form of the equation of a line:

\[ Ax + By + Cw = 0 \]

we see that the coordinates of the point of intersection are:

\[ x = B1C2 - C1B2 \]
\[ y = C1A2 - C2A1 \]
\[ w = A1B2 - B1A2 \]

The equation of the line defined by two points may be found by similar reasoning.

Consider two known homogeneous points:

\[ (x_1, y_1, w_1) \]
\[ (x_2, y_2, w_2) \]

and an arbitrary third point:

\[ (x, y, w) \]

The three points may be grouped in the following matrix:

\[
\begin{bmatrix}
1 & x_1 & y_1 & w_1 \\
1 & x_2 & y_2 & w_2 \\
1 & x & y & w
\end{bmatrix}
\]

If the third point is collinear with the first two, its coordinates will be linearly dependent on the first two points. In order for the all three points to be on one line, the determinant of the matrix must be zero. Taking the
determinant by expanding on the last row yields:

\[ x(y_1 w_2 - w_1 y_2) - y(x_1 w_2 - w_1 x_2) + w(x_1 y_2 - y_1 x_2) = 0 \]

Comparing this equation to the general equation of a line:

\[ Ax + By + Cw = 0 \]

we see that the coefficients of the line defined by \((x_1, y_1, w_1)\) and \((x_2, y_2, w_2)\) are:

\[
\begin{align*}
A &= y_1 w_2 - w_1 y_2 \\
B &= w_1 x_2 - w_2 x_1 \\
C &= x_1 y_2 - y_1 x_2
\end{align*}
\]

The duality between points and lines in 2D is a useful property not only for intersections and line definition, but for determining whether a point is to the right of, left of, or colinear with a line.

Consider three homogeneous points:

\((x_1, y_1, 1)\)

\((x_2, y_2, 1)\)

\((x_3, y_3, 1)\)

The determinant of the matrix

\[
\begin{vmatrix}
x_1 & y_1 & 1 \\
x_2 & y_2 & 1 \\
x_3 & y_3 & 1
\end{vmatrix}
\]

gives twice the area of the triangle defined by the three points \((23)\). The only way for the area to be zero is for the three points to colinear. If the above product is positive, the triangle \((x_1, y_1, 1), (x_2, y_2, 1), (x_3, y_3, 1)\) is being traversed counterclockwise. Conversely, a negative determi-
nant indicates that the triangle is being traversed clockwise. These relationships are used in the RIGHT_OF function to determine whether point p3 lies to the right of the line defined by points p1 and p2.
CHAPTER IV
ANALYSIS OF A FLOORING SYSTEM

4.1 Analysis Control

A human engineer designing a floor beam would first look at the beam to see if any other members are supported by the member. If other members frame in, each of these must in turn be examined until members are found which support no other members. The engineer would start by analyzing all of these primary members, then continue carrying the end reactions to the supporting members until all of the loads on the desired beam are known and the beam's reactions can be found.

The approach of the human engineer in this problem is to conceptually break a floor into a collection of like objects (members) which are analyzed in an identical fashion. What the engineer does is select a recursive approach as the natural method of solving the problem. This same recursive approach is employed in the floor analysis algorithm in this program.

The analysis, or load takedown, involves three functions: ANALYZE, which traverses the linked list of members; RECURSE, which performs the actual takedown of member loads; and GET_REACTIONS, a function to resolve all of a member's loads into end shears and moments. As may be inferred from the name, the algorithm in RECURSE works through a series of recursive calls.

The controlling analysis routine, ANALYZE, traverses the linked list of members until a member is found that has not
been analyzed (indicated by the solved flag in the member data structure). When an unsolved member is found, ANALYZE calls RECURSE, passing the pointer (address) to the member.

In RECURSE, actual analysis of a member is done by the function GET_REACTIONS, which increments the member's beginning end reactions as if it were a cantilever. The recursive routine looks at the member's frame in list to determine whether the member supports any intermediate nodes, then checks to see what members frame into the nodes. Each member framing in is then examined to determine whether it has been solved. If a member framing in has not yet been solved, RECURSE calls itself to solve that member before proceeding with the load takedown.

This recursion will continue until a member is reached which has only known loads. When this happens, program flow falls through the recursive portion of the algorithm, and GET_REACTIONS is called. GET_REACTIONS increments the beginning shear and moment for each of the member loads. At this point, the member is examined to see whether it actually is a cantilever. Through statics, the member's end reactions are calculated by balancing the end moment to zero (for a simply supported beam), or by adjusting the end moment is the member is actually fixed at the opposite end from that assumed in GET_REACTIONS.

The last step before returning the solved member is to check to see if the member is opposite from a cantilever. If so, then the cantilever will introduce a carryover moment at one end of the member being analyzed. If the cantilever
has not yet been analyzed, RECURSE will be called for the cantilever. Once all of the above checks have been made, RECURSE marks the member as solved and returns. The return may be to the next higher level of the recursion, or, at the top level, to the calling function, ANALYZE.

When RECURSE returns control to ANALYZE, ANALYZE continues to traverse the linked list of floor members, skipping those already solved, and calling RECURSE for the unsolved members.

The floor in figure 4.1 will be used as an example.

![Diagram of a floor with labeled nodes and members.]

# indicates node number    ---#-- indicates member number

FIGURE 4.1 EXAMPLE FLOOR

For this example, assume that the linked list of members is assembled as shown in figure 4.2. Before analysis is performed, the function RESET is called to ensure that all of the members are marked as unsolved before starting a new analysis. It is not possible to perform a partial reanalysis. The new analysis begins with a call to ANALYZE, passing the pointer
FIGURE 4.2 MEMBER LIST

to member 1 (the head of the list). ANALYZE checks member 1's solved flag, and finds that it is unsolved, so it calls RECURSE, passing the pointer to member 1. RECURSE checks member 1's frame_in_list, and determines that member 1 supports no intermediate nodes. Member 1 thus falls through the recursive part of the algorithm, and GET_REACTIONS is called, again with the pointer to member 1. GET_REACTIONS checks to see that there are loads on the member, and performs the analysis. In this case, the only loads on the member are its own self weight, which may be estimated for initial design, and the floor loads. GET_REACTIONS solved the reactions at the beginning end of the member, assuming that it is a cantilever. RECURSE checks member 1's data structure to determine its actual end conditions, then uses a moment balance to adjust the end shears to their proper values. Since both ends of member 1 are connected to columns, there is assumed to be no carryover moment from opposing cantilevers (there are none in this case), RECURSE returns to ANALYZE.

ANALYZE looks at the next member in the member list, member 4, finds that it is not solved, and calls RECURSE for member 4. RECURSE examines the frame_in_list for member 4 and
sees that it supports node 5. RECURSE then examines node 5's frame_in_list and finds that member 2 frames into it. Member 2 has not been solved, so RECURSE calls itself for member 2.

In the second level of recursion, RECURSE examines the frame_in_list for member 2, which shows that it supports node 6. Node 6's frame_in_list shows that member 5 frames in, so RECURSE examines member 5, finding that it is not solved. RECURSE calls itself again to solve member 5. Even on this small section of floor the recursion is now three levels deep.

Like the first member analyzed, member 1, member 5 supports no intermediate nodes, and so falls through the recursive part of the algorithm. GET_REACTIONS is called, and performs the analysis. Since there are no members framing in, the only loads on the member are its self weight (if known) and the floor load. GET_REACTIONS returns to RECURSE, and RECURSE checks member 5's data structure to determine that it is a simply supported beam. A moment balance is used to adjust the end shears to their proper values. Unlike the first time when RECURSE returned to ANALYZE, RECURSE now returns to itself, one level shallower in the recursion.

Now that the third level of recursion has finished, the second level finds that member 5 is now solved, so its end reaction is carried down to member 2. Since member 5 is simply supported, only shear is transmitted to member 2. There are no other members framing in, so member 2 falls through the recursive load takedown. GET_REACTIONS is called, as before,
incrementing the end reactions. RECURSE then treats member 2 as simply supported, and performs a moment balance.

The last step in RECURSE is to check for opposing cantilevers. This time, the member (member 2) does not begin or end at columns, so moments can be carried over. At the beginning node, node 5, there is no opposing cantilever, so no action is taken. At node 8, however, there is an opposing cantilever, member 7, so some end moment will be carried over. Since RECURSE finds that member 7 is not solved, it again drops to a third level in the recursion, calling itself to solve member 7.

RECURSE now examines member 7. Because member 7 is a cantilever, it must support any loads applied at its free end. Node 9 is a special case, it is marked as a cantilever node, so although it is the end node for member 7, member 7 also supports it, just as if it were an intermediate node on a simply supported member. RECURSE finds that member 9 frames in to member 7, and that member 9 is also unsolved, so RECURSE calls itself once again, this time four levels deep in the recursion.

Member 9 supports no intermediate nodes, so it falls through to the second part of the RECURSE function. GET_REACTIONS increments member 9's end reactions, and returns. RECURSE finds that member 9 is a simply supported member, and adjusts its end reactions. Member 9 has no opposing cantilevers, so it returns one level higher in the recursion.

Now that the end reactions of member 9 are known, they
are carried down to member 7. Member 7 then falls through the recursion and GET_REACTIONS is called to increment its end reactions. In this case, member 7 is a cantilever fixed at its beginning end, so RECURSE does not adjust its end reactions. Because member 7 is a cantilever, it cannot have any opposing cantilevers, so member 7 is marked as solved, and RECURSE returns to the second level of recursion.

Now that the end moment of member 7 is now known, a carry over moment of equal magnitude but opposite sign is added to the end of member 2 at node 8 (any torsional resistance in member 6 is neglected). The end shears are then adjusted, member 2 is marked as solved, and RECURSE returns to its top level.

At the top level, RECURSE finds that member 2 is now solved, so member 4 falls through to GET_REACTIONS. RECURSE takes the cumulative returned shear and moment from the member loads and the carry down load from member 2, and adjusts the end reactions appropriately for the actual end conditions. Member 4 ends at columns, so there can be no moment carry over from opposing cantilevers (there are none), so RECURSE marks member 4 as solved and returns. RECURSE is now at the top level so it returns, not to itself, but to ANALYZE.

ANALYZE continues to traverse the member list, finding member 3. Member 3 has not been solved, so ANALYZE calls RECURSE. RECURSE finds that member 3 supports node 7, which in turn supports member 5. Member 5 has already been solved, however, so RECURSE does not need to call itself. Instead,
member 3 falls through the recursive load takedown, and is passed to GET_REACTIONS. Member 3 is then solved using the results from the load takedown and GET_REACTIONS. Member 3 begins and ends at columns so there can be no carry over moment from the cantilever at node 4 (member 8). It is assumed that all of the moment from the cantilever is taken by the column. A frame analysis is required to determine what actually happens at this point. RECURSE returns to ANALYZE.

The next member in the list is member 2, but member 2 has already been solved, and is skipped. The next member, member 6, has not been solved, so RECURSE is called. As was the case with member 3, member 6 has members framing in (members 2 and 7), but these have been solved. Member 6 falls through the load take down portion of RECURSE, and GET_REACTIONS is called. The end reactions are then adjusted as before. Member 6 begins and ends at columns, so there can be no carry over moment, so RECURSE returns.

Member 7, the next member in the list, has been solved, so it is skipped. Member 8 has not yet been solved, so RECURSE is called. RECURSE finds that member 9 frames into member 8, but has been solved already, so the end shear is carried down to member 8. GET_REACTIONS is then called, and since member 8 is a cantilever, the end reactions are left unchanged. As was the case with member 7, member 8 is a cantilever, so there can be no carry over moments from other cantilevers. RECURSE returns to ANALYZE.

ANALYZE continues in the member list, finding member 9, which has been solved. Member 9 is the last entry in the list.
(marked by a NULL next pointer in the data structure), so
ANALYZE returns to main.

4.2 Load Areas for Oneway Slabs

The ONEWAY function uses the closed circuits found by the
path algorithm to create tributary area polygons for members.
The tributary areas are stored as circular lists of points
that define the corners of the polygon. Like TWOWAY, ONEWAY's
algorithm is only valid for polygons of five or less sides.

Oneway slabs are, to a degree, simply a convenient
approximation, the most common form being the steel deck
reinforced concrete slab. Once the concrete slab is in place,
two factors prevent this slab from truly acting in only one
direction: first, the concrete slab has significant strength
along any axis; and second, the decking is normally continuous
over several supports. These factors affect the final
strength and load distribution characteristics of the slab,
but are not considered in design because the most critical
loading is usually the construction load when the steel deck
alone must support the wet concrete as it is poured. Because
the steel deck has practically no strength perpendicular to
its strong axis, the assumption that the floor can only carry
load in one direction is justified.

The ONEWAY algorithm makes this assumption about the
directionality of the load carrying capacity based on user
input of a strong direction. The first step in the algorithm
is to construct a homogeneous line (11 in fig. 4.3) defined by
the member being analysed. Once the line is found, a compar-
ison is made between the floor's direction (also in homogeneous line format), and the member line. If the two are parallel, the member does not directly support any floor area, so it is skipped.

![FIGURE 4.3 ONEWAY STEP ONE](image)

Otherwise, a line (12 in fig. 4.4) is constructed through the first end point of the member being analyzed, parallel to the floor's direction.

![FIGURE 4.4 STEP 2](image)

Next, a line (13 in fig. 4.5) is constructed through the second and third points in the area, and the two lines are intersected using the INTERSECT function, giving point p1.

After the intersection is found, a test is performed using the RIGHT_OF function to see which side of line a1-a2 point p1 is on. A similar test is performed to determine
which side of a1-a2 point a3 lies on. If p1 and a3 are on the same side of the line (more precisely, not on opposite sides to allow for p1 and a2 being coincident), then the next point in the tributary area will be the midpoint of the segment a1-p1. In the current example, a3 and p1 are on opposite sides, so a line (12' in fig. 4.6) is constructed through a2 parallel to the strong axis, and intersected with the original line, 11, which is defined by the member to give p2. The next point in the tributary area is the midpoint (m1) of a2-p2.

A similar procedure (fig. 4.7) is followed at the end node of the member to obtain the last two points in the tributary area. In this case, p1 is between a3 and a2, so it was
not necessary to construct $12'$. The tributary area is defined by $a_4$, and the midpoint ($m_2$) of the segment $a_4-p_1$.

![Diagram of a tributary area](image)

**FIGURE 4.7 STEP 5**

The final tributary area polygon is shown in figure 4.8.

![Diagram of the final tributary area](image)

**FIGURE 4.8 FINAL TRIBUTARY AREA**

The internal representation of the tributary area is the circular list of points:

```
a_1 ---> m_1 ---> m_2 ---> a_4
    |
    v
---<---?
```

For some geometries (e.g. triangular areas), there may be concurrent points in the tributary area, however these will not affect subsequent analysis. Although it has not been explicitly stated, the algorithm detailed above effectively distributes floor loads at oblique connections by bisecting the corners. This method assumes that the floor acts as a simply supported beam, which is reasonable for steel decking.
4.3 Load Areas for Twoway Slabs

The TWOWAY function uses the closed circuits found by the PATH function, to create tributary area polygons for members. These tributary area polygons are stored as circular lists of points that define the corners of the polygon. The algorithm employed is accurate only for polygons of five or less sides. This is not a significant limitation, as multi-sided floor sections are not common.

A twoway slab is actually indeterminate; however, in practice assumptions are made concerning the slab's behavior to simplify calculations. For a uniformly loaded twoway slab, the normal assumption is that the load will be carried by the nearest supporting member. Based upon this assumption, the slab is divided into tributary areas by bisecting the slab's corners. The shape of the resulting area will depend upon the location of the intersection of the bisectors from adjacent corners.

For the four sided case (fig. 4.9), three steps are needed to determine the shape of the tributary area. First, find the distance from the end node of the graphical member to the intersection of the bisectors at node 1 and node 2. Second, find the distance to the intersection of the bisectors at node 1 and node 4. The last step is to compare the two distances. The shortest distance determines which intersection will be a corner of the tributary area.
The same three steps are applied at node 4 for intersections 4-1 and 3-4 (fig. 4.10). If the intersection 1-4 is closer than intersection 1-2, the area is a triangle, and the tributary area list contains the coordinates of node 1, the intersection, and node 4. Otherwise, the area is determined by node 1, the nearest intersections at node 1 and node 4, and node 4.

In the above example, the final tributary area for member 1-4 would be the circular list:

\[
1 \rightarrow a \rightarrow d \rightarrow 4
\]

The arrows represent pointers to the next item in the list.
In the program's internal representation, the points are structures containing the coordinates of the points.

For member 1-2, the tributary area would be a triangle described by the circular list:

1 --> b --> 4

If the path has four or five sides, the steps above will correctly determine the tributary area. If there are more than five sides, the same procedure may be applied, however there may be an area in the middle of the slab which is not assigned to any of the members. This area could be found by taking the area of the path polygon and subtracting the areas of the tributary area polygons. Once the residual was found, it could be distributed among the areas. Due to the problems inherent in altering the linked lists to increase their area, this approach is not implemented.

4.4 Intersection of Floor Loads and Member Areas

Thus far, we have discussed the algorithms used to create an adjacency list representation of floor connectivity, and the algorithm used to search for closed circuits which are then converted into either oneway or twoway tributary areas for each member. A polygon clipping algorithm (23) in the CLIPPOLY function is then used to determine whether or not these areas are loaded by one or more floor loads, and if loaded, a polygon representing the intersection of the tributary area and the load is generated.
Floor loads, like tributary areas, are represented as circular linked lists of points. A linked list is circular if its last element points back to the head of the list. This is a convenient representation because it is easy to travel around an area: processing continues until we arrive at our starting point once again. Using this convention, it is not necessary to maintain any information about how many points are in the area, making it a simple matter to handle N sided areas.

The only limitations on the shape of floor loads are: 1) floor load polygons must be described by travelling clockwise around the boundary; and 2) only convex polygons are permitted (a floor load can have no reentrant corners). The first limitation is simply a convention so that the clipping algorithms will be able to determine whether a point lies inside of a floor load polygon; the second limitation is imposed primarily for the sake of simplicity in clipping. Any non-convex polygon may be broken into a set of convex polygons, so arbitrarily complex loadings may still be placed on the floor.

The actual clipping of the tributary area against the floor area is accomplished by traversing the floor area and testing for intersections with each of the sides of the tributary area. In the example shown in figure 4.11, both the floor load area, area F1...F4, and the member tributary area, A1...A4 have four sides. As stated earlier, the floor area's vertices must be in clockwise sequence. The tributary area's vertices may be in either clockwise or counterclockwise order.
FIGURE 4.11 FLOOR LOAD MEMBER AREA CLIPPING

The linked list representing the floor area would look like this:

F1 --> F2 --> F3 --> F4

'--------<--------'

The linked list for the tributary area would be:

A1 --> A2 --> A3 --> A4

'--------<--------'

The clipping procedure begins (fig. 4.12) with the first side of the floor area, F1-F2, and the first side of the tributary area, A1-A2. First a homogeneous line is constructed passing through points F1 and F2. Next point A1 is checked to see if it lies to the right of, left of, or on line F1-F2. Because of the convention that all floor areas are defined in clockwise order, a point to the left of F1-F2 cannot be inside the floor area. In this case, A1 is to the right of F1-F2, so it may be inside the area. Point A2 is checked in similar fashion. If A1 and A2 are both to the right of F1-F2, they may both be inside of the floor area, so no change is made in the tributary area polygon. In the same way, sides A2-A3, A3-A4,
and A4-A1 are checked against side F1-F2. In this case, all of the vertices of area A lie to the right of F1-F2, so no change is made in the loaded area.

The next step is to go to the next side in the floor area, F2-F3, and repeat the above tests. Again all of the vertices of area A lie to the right of line F2-F3, so no changes are made.

Continuing the process, side F3-F4 is used to test (fig. 4.12). This time when side A1-A2 is tested, A1 lies to the right of F3-F4, but point A2 lies to the left, indicating that it must be outside of the floor load polygon. Since the two points, A1 and A2 are on opposite sides of the line described by F3-F4, side A1-A2 must intersect this line. To find the intersection, a homogeneous line is constructed through A1 and A2 and intersected with F3-F4 using the INTERSECT algorithm (section 3.6).

![Figure 4.12 STEP 3](image)

This intersection gives the point A1', which is inserted before point A2 in the area list:
A1---->A1'---->A2---->A3---->A4

Notice that while the area list has been altered, the shape of the clipped area has not yet been changed. Clipping against line F3-F4 continues as before (fig. 4.13), except that the altered area is used for subsequent tests.

![Diagram](image)

**FIGURE 4.13 STEP 4**

When side A2-A3 is checked, another intersection point, A2' is added to the temporary area, giving the new linked list:

A1---->A1'---->A2---->A2'---->A3---->A4

The next test (fig. 4.14) checks points A3 and A4 against line F3-F4. Because both points are to the right of line F3-F4, no change is made to the area list. This part of the algorithm concludes with a check of the last pair of points, A4 and A1, against line F3-F4, which results in no change to the area list. The final area looks like this:
Before going on to the next side of the floor area, the algorithm must discard the points in the area list that lie on the exterior of the floor load, giving the final loaded area:

A1—>A1'—>A2—>A2'—>A3—>A4

The final step in the algorithm is to clip the altered area list against side F4-F1. Both A1 and A1' lie to the left of line F4-F1, so the area list is not changed. The next pair of points, A1'-A2', lie on opposite sides of F4-F1, so an intersection must be found (fig 4.14).

![Diagram](image)

**FIGURE 4.14 INTERSECTION OF A1-A4 AND F4-F1**

The intersection point, A1'', is found and inserted into the list, giving:

A1—>A1'—>A1''—>A2'—>A3—>A4

In the next test, neither A2' or A3 lie to the left of line F4-F1, so the area list is not changed. The last pair
of points, A4-A1 lie on opposite sides of F4-F1, so an intersection must be found:

![Diagram of F2, F3, F4, and A1-A4 showing intersection]

**FIGURE 4.15 INTERSECTION OF A1-A4 AND F4-F1**

The final result of the clipping is the linked list:


All that remains is for the algorithm to discard the points that lie on the exterior of the floor load, giving the final loaded area:

![Diagram of F2, F3, F4, and A1-A4 showing final loaded area]

**FIGURE 4.14 FINAL LOADED AREA**

The final linked list is:

A1'' --- > A2' --- > A3 --- > A4 --- > A4' --- > A2' --- > A1''

4.5 Member Load Generation for Tributary Areas

Now that a loaded area has been found, it is necessary to convert it into member loads which may be used in
the analysis routines. Because the loaded areas are bounded by straight lines, the member loads will be linear with a change in rate only at points on the member perpendicular to the corners of the loaded area. For both oneway and twoway floors, the magnitude of the member load at a point is the floor load magnitude times the distance across the load perpendicular to the member. The LINE_DISTANCE function is used to determine the perpendicular distance to each of the successive points in the loaded area. A single member load is generated by each pair of vertices in the loaded area.

Consider member A in figure 4.16 which supports a section of two-way flooring and has a loaded area defined by corners a1...a4, which have the following coordinates:

\[
\begin{align*}
ap1 & (40.0, 2.5) 
nap3 & (25.0, 12.5) 
nap2 & (45.0, 12.5) 
nap4 & (20.0, 2.5)
\end{align*}
\]

![Diagram of member A with loaded area](image)

**FIGURE 4.16 MEMBER WITH A LOADED AREA**

Member A begins at point (0,0). The shaded area represents a floor load of 50 psf which has already been intersected with member A's tributary area, resulting in loaded area a1-a4.

The CARRYDOWN algorithm first considers vertices a1 and a2. The distance from member A to point a1 is 2.5 feet, giving the first member load, p1, a beginning magnitude of:
2.5 \times 50\text{psf} = 125\text{ pounds/linear foot}

In the load data structure, the location of the load is specified in local (member) coordinates. The local coordinate at the beginning of the member is 0, the coordinate at the end is 1. The local coordinate is found by finding the distance of the vertex from the member's beginning node, d, and the line distance from member A, line_distance. By the Pythagorean theorem,

\[
\text{local_x} = \sqrt{d^2 - \text{line_distance}^2}/\text{length}
\]

The local coordinate of a1 is found as follows:

\[
d = \sqrt{(40-0)^2 + (2.5-0)^2} = 40.078
\]

\[
\text{local_x} = \sqrt{(40.078)^2 - 2.5^2}/50 = .80
\]

The second point, a2, is 12.5 feet from member A, so the ending magnitude for p1 is 12.5 \times 50\text{ psf} = 625\text{ pounds/foot}. The local end coordinate for a2 is .90. Load p1 (fig. 4.17) is added to member A's load list.

![Figure 4.17 Member Load p1](image)

During the analysis portion of this program, the GET_REACTIONS function determines a member load's sign by calculating the difference (begin_x - end_x). Because load
pl's end y is greater that its beginning y, pl will be a negative load.

Next, the algorithm looks at vertices a2 and a3. The magnitude and location of the load at a2 is already known, giving the second member load, p2, a beginning magnitude of 625 and a beginning y of 0.90. The second point, a3, is also 12.5 feet from member A, so the ending magnitude for p2 is also 625 pounds/linear foot. The ending y is 0.50. Load p2 (fig 14.18) is added to member A's load list.

Since member load p2's end y is less than its end y, load p2 is a positive load.

The load, p3 (fig. 4.19), defined by the last two vertices, a3 and a4, has a beginning magnitude of 625 lb/ft, and a beginning y of .50. The ending magnitude is 125 lb/ft. This load is exactly the same in magnitude as load p1, except that load p3's end y is less than its begin y, so that it will be a positive load.
The final member load to be assigned is load p4 (fig. 4.20), which begins at point a4 and ends at point a1. The begin and end magnitude and local x coordinates are already known for this load: begin magnitude 125 lb/ft, end magnitude 125 lb/ft, begin x .40, end x .80.

Because the end x is greater than the begin x, this load will be negative.

\[ \begin{array}{c}
1 \\
4 
\end{array} \]

FIGURE 4.20 MEMBER LOAD p4

The final result of the process, as shown in figure 4.21, is the superposition of four loads.

\[ \begin{array}{c}
\text{FIGURE 4.21 MEMBER LOAD SUPERPOSITION} \\
\end{array} \]

This superposition remains valid for loads on the other side of the member as well because member tributary areas, which are derived from the paths found by the PATH function, are traversed in opposite directions on the two sides of the member.
When the end of the loaded area is reached, the CARRYDOWN function will return to the CLIP_POLYGON function which will continue looping over the floor loads to find more loaded areas to be processed.

4.6 Evaluation of Behavioral Assumptions

In the discussion of the twoway floor algorithm, it was assumed that the load was uniform over the entire floor area in order to simplify calculations. The same assumption applies to the algorithm used in the oneway floor algorithm. During the foregoing discussion of clipping tributary areas, it should have become apparent that once a tributary area is clipped, that the uniform loading assumption is no longer valid. To properly account for this situation, a moment balance should be performed to distribute the load properly to the sides of the floor area that provide support. In a one-way floor system, this entails treating the floor slab as a simply supported beam and solving for the reactions at each end. It would be possible to account for this possibility by calculating the area enclosed by the circuit found in PATH, keeping this information for each member, clipping the paths against each floor load, and then comparing the clipped and unclipped areas to see if they are the same. If they are the same, then the assumption of uniform load over the floor area is valid, and the ONEWAY and TWOWAY routines may be used without modification. If the clipped and unclipped areas are different, a moment balance is necessary.

Although the moment balance would not be difficult for
the one-way case, it is not clear how one should treat a two-way floor with a nonuniform load. Because of these difficulties, the program treats nonuniform loads as if the uniform loading assumption were valid. The worst case error in this assumption occurs when a floor section is loaded by a narrow strip load offset slightly from the centerline of the slab as shown in figure 4.22.

![Diagram](image)

**FIGURE 4.22 WORST CASE NON UNIFORM LOAD**

Theoretically, as the width, $w$, of the strip load approaches zero, the reactions at both ends approach $pw/2$. The CARRYDOWN algorithm, however, would assign the entire load to side A. The largest possible error is then 50%.

In real buildings, however, the error will be substantially less. Partitions in a building would be the closest real approximation of a line load, but a partition that was continuous across the floor section would also be supported by the dashed and members which are assumed not to carry any load directly. A more typical example would be a floor load which covered only one half of the floor area, as shown in figure 4.23.
In this case, the reaction at A would be $3pw/4$, and the reaction at B would be $pw/4$. The CARRYDOWN algorithm would assign the entire load of $pw$ to side A.
CHAPTER V

FILE OPERATIONS

Files are used in this set of programs for three purposes: inputting floor data, output of floor data for later analysis, and output of results for the analysis of the current floor. The program uses the following files:

* memlist.dat  member data
* nodelist.dat  node data
* fil_list.dat  node connectivity data
* nl_list.dat  member connectivity data
* floorload.dat  floor load data
* opening.dat  floor opening (hole) data
* loadlist.dat  member load data
* results.out  member end reactions and moments

The files above marked with an asterisk must exist prior to attempting to retrieve a floor from files, even if they contain no data e.g. if the floor has no openings, an empty file named "opening.dat" must still be present. The file loadlist.dat is optional. The file results.out is written by the program following a successful analysis of a flooring system. The specific use of these files will be discussed in detail below.

5.1 Saving a Floor

In order to save a floor for future analysis, it is necessary to create disk files containing the data to describe the flooring system and its loads. Because all of the attributes for a floor can be generated from the floor's node and member connectivity data, member loads, and floor loads, only this primary data about a floor needs to be saved in files.
Primary data are those pieces of information which the program cannot generate; the rest of the data used in the program may be calculated from primary data. Calculated data is not saved, both to save space, and to ensure accuracy of calculated values after changes. Since recalculation of the analysis is not lengthy, forcing reanalysis of the flooring system after a save/retrieve cycle imposes only a slight time penalty while ensuring the accuracy of the solution.

The SAVE_FLOOR function performs the storage operation for a floor. It consists of three major loops, one over the list of nodes in the flooring system, and a second over the list of members in the flooring system, and the third over the list of floor loads. During the node loop, two files are written: the first, nodelist.dat, contains the node number, coordinates, and flags. In the internal list of nodes, connectivity data is maintained by a list of pointers to the members that frame in to the node. These pointers, however, are only addresses in memory which may change each time that the program is run, and thus would be meaningless if saved in a file. To solve this difficulty, node numbers and member numbers are saved, to be converted back to pointers when the floor is recreated. The second file, fila_list.dat, is created by looping through each node's frame_in_list, storing connectivity data for the node in the form of pairs of node and member numbers.

Once the loop through the nodes has completed, the SAVE_FLOOR function loops through the member list, similarly writing two files: memblist.dat, and n1_list.dat. The first
file, memblist.dat, contains the member number, followed by the begin and end node numbers, and flags for the member. The second file is similar to the file_list.dat file discussed above except that it is created by looping through the member's node_list, storing connectivity data as pairs of member numbers and node numbers.

After saving the floor's geometric and connectivity information, the routine writes the load information to two files: loadlist.dat, and floadlist.dat. The former is written while looping through each member's loads, the latter while looping through the floor load list. Only member loads that are directly applied are stored. Member loads that result from a load takedown, whether from a floor load, or from another member which frames in are not saved. The file loadlist.dat consists of the loadcase number, followed by administrative information, e.g. which user assigned this load, and the date it was assigned, the begin and end magnitude, and the begin and end local coordinates.

5.2 Retrieving a Floor

In order to retrieve a floor for further analysis, it is necessary to recreate the internal representation of the floor from disk files containing the data necessary to describe the flooring system and its loads. Because only the floor's node and member connectivity data, member loads, and floor loads are saved, recalculation of the analysis must be performed after a save/retrieve.
The GET_FLOOR function performs the retrieval operation for a floor. It consists of three loops, one over the list of nodes in the flooring system, the second over the list of members in the flooring system, and the third over the floor loads. During the node loop, two input files are read: nodelist.dat, which contains the node coordinates and flags; and fil_list.dat, which contains connectivity data for the node's frame_in_list. When a node number or member number is read from the file, it is converted into a pointer to the proper data structure by the TO_MPTR or TO_NPTR function. These functions check to see if the node or member already exists, if not, memory for the structure is allocated. In either case a valid pointer will be supplied to place in the recreated node structure.

Once the loop through the nodes has completed, the SAVE_FLOOR function loops through the member list, similarly reading two files: memblist.dat, and nl_list.dat. The first file, memblist.dat, contains the member number, followed by the begin and end node numbers, and flags for the member. The second file is similar to the fil_list.dat file discussed earlier. By the time that the member file is read, all of the nodes and members will have already been created since every member must be connected to at least one node. Memory will be allocated for the member's node_list by calls to the ADD_NL function. Once all of the member and node definition and connectivity data have been read, GET_FLOOR will call the READ_MEMBERLOAD function to read the file loadlist.dat (if present). This file contains data to represent directly
applied member loads (as opposed to member loads derived from floor loads). The data for each load consists of a load number which contains both the member number and the load case number to which the load is applied, the date that the load was assigned, the user ID of the person who assigned it, a coordinate flag, the beginning and ending load magnitude, and the beginning and ending (local) x coordinate. As each new member load is read, READ_MEMBERLOAD checks finds the appropriate load case by calling the TO_LCPTR function (if the load case does not already exist, it will be created). Then memory is allocated for the loadlist by a call to the ADD_LOAD function.

The last step in the retrieval process is to call the READ_FLOORLOAD function to read the floorload.dat file, and then the opening.dat file which are both in the same format. Floor loads contain the same administrative information as member loads discussed above. As the floor load is read, the coordinates of its corners (in clockwise order) are converted into homogeneous coordinates and placed in a homogeneous area (h_area) format. Openings are represented in exactly the same format as floor loads, except that they have a magnitude of zero.

5.3 Output of Results

Although the floor has been retrieved, none of the calculated data will be available until a new analysis is performed. This reanalysis is not automatic, and thus may be preceded by editing of the floor or altering the loads (when
the floor editor becomes operational). When analysis is performed, the program will generate the file results.out which contains the end shears and moments for each floor member broken down by load case.
CHAPTER VI
SAMPLE PROBLEM

6.1 Description and Design of the Problem

The example problem in this chapter performs the analysis of a two way flooring system of moderate size and complexity (see fig. 6.1). The floor has 49 members, one hole bounded by members 1, 5, 6, and 7, two cantilevers, one of which (member 11) is cantilevered from a column, the other, member 12, is cantilevered from member 15 so that its end moment must be balanced by member 31 on the other side of the joint. A live load (LL) of 75 psf and a dead load (DL) of 40 psf are applied over the entire structure. In the bay bounded by nodes 10, 47, 29, and 12, there is an additional 25 psf LL to bring the total bay load to 100 psf, and in the bay bounded by nodes 12, 29, 33, and 14, there is an additional load of 50 psf LL that covers part of the floor. Two members, 5 and 6, have member loads of 60 plf. Member 5's load only acts between nodes 6 and 16 (local coordinates .50 and 0 respectively), while the load on member 6 covers the entire length. As is the case with most floors in practice, most of the members are in a basic grid layout, but a few members, 45, 46, and 47, intersect at oblique angles.
FIGURE 6.1 SAMPLE FLOOR
6.2 Input

The problem was input by creating the following files:

- memblist.dat  member data
- nodelist.dat  node data
- fil_list.dat  node connectivity data
- nl_list.dat  member connectivity data
- floorload.dat  floor load data
- opening.dat  floor opening (hole) data
- loadlist.dat  member load data

The format for the contents of these files is discussed in Appendix A. For a discussion of the process of reading the files, see section 4.2.

6.3 Processing Commands Results

Three commands were necessary to run this example: From the main menu, the File menu was selected by typing the letter "f" (upper or lower case) followed by a carriage return "<CR>". The File menu option to retrieve a floor was invoked by typing an "r". The main menu reappears, and the option Analyze by typing "a" <CR>.

The results file results.out is listed in table 6.2.

6.4 Validity Checks on Results

The results for each member were individually checked by hand calculations, but several quick checks for reasonableness can be done without calculations. Member 1's loads are not symmetric, which indicates that no loads were assigned for the opening. Members 2, 3, and 4 are symmetrically loaded as is expected. Members 5 and 6's LL and DL are not in the proportion 75:45 as are the majority of the member's loads in the floor. This indicates that the member DL was carried down. Member 7's loads are not symmetrical, again indicating the
<table>
<thead>
<tr>
<th>Member</th>
<th>load case 1</th>
<th>load case 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>#</td>
<td>Vbag</td>
<td>Mbag</td>
</tr>
<tr>
<td>1</td>
<td>4843.13</td>
<td>0.00</td>
</tr>
<tr>
<td>4</td>
<td>2953.12</td>
<td>0.00</td>
</tr>
<tr>
<td>5</td>
<td>3948.75</td>
<td>0.00</td>
</tr>
<tr>
<td>6</td>
<td>1181.25</td>
<td>0.00</td>
</tr>
<tr>
<td>7</td>
<td>6412.50</td>
<td>0.00</td>
</tr>
<tr>
<td>13</td>
<td>17223.75</td>
<td>0.00</td>
</tr>
<tr>
<td>2</td>
<td>4303.12</td>
<td>0.00</td>
</tr>
<tr>
<td>3</td>
<td>2871.09</td>
<td>0.00</td>
</tr>
<tr>
<td>8</td>
<td>8606.25</td>
<td>0.00</td>
</tr>
<tr>
<td>9</td>
<td>1518.75</td>
<td>0.00</td>
</tr>
<tr>
<td>10</td>
<td>1518.75</td>
<td>0.00</td>
</tr>
<tr>
<td>11</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>12</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>14</td>
<td>14111.72</td>
<td>0.00</td>
</tr>
<tr>
<td>15</td>
<td>16877.34</td>
<td>0.00</td>
</tr>
<tr>
<td>17</td>
<td>2471.87</td>
<td>0.00</td>
</tr>
<tr>
<td>18</td>
<td>1406.25</td>
<td>0.00</td>
</tr>
<tr>
<td>19</td>
<td>1406.25</td>
<td>0.00</td>
</tr>
<tr>
<td>20</td>
<td>5343.75</td>
<td>0.00</td>
</tr>
<tr>
<td>21</td>
<td>5343.75</td>
<td>0.00</td>
</tr>
<tr>
<td>22</td>
<td>1406.25</td>
<td>0.00</td>
</tr>
<tr>
<td>23</td>
<td>5343.75</td>
<td>0.00</td>
</tr>
<tr>
<td>24</td>
<td>1406.25</td>
<td>0.00</td>
</tr>
<tr>
<td>25</td>
<td>4938.28</td>
<td>0.00</td>
</tr>
<tr>
<td>26</td>
<td>5132.81</td>
<td>0.00</td>
</tr>
<tr>
<td>27</td>
<td>5132.81</td>
<td>0.00</td>
</tr>
<tr>
<td>28</td>
<td>5132.81</td>
<td>0.00</td>
</tr>
<tr>
<td>29</td>
<td>5828.91</td>
<td>0.00</td>
</tr>
<tr>
<td>30</td>
<td>4587.50</td>
<td>0.00</td>
</tr>
<tr>
<td>31</td>
<td>9618.75</td>
<td>0.00</td>
</tr>
<tr>
<td>32</td>
<td>2003.91</td>
<td>0.00</td>
</tr>
<tr>
<td>33</td>
<td>4007.81</td>
<td>0.00</td>
</tr>
<tr>
<td>34</td>
<td>4007.81</td>
<td>0.00</td>
</tr>
<tr>
<td>35</td>
<td>4007.81</td>
<td>0.00</td>
</tr>
<tr>
<td>36</td>
<td>4007.81</td>
<td>0.00</td>
</tr>
<tr>
<td>37</td>
<td>4007.81</td>
<td>0.00</td>
</tr>
<tr>
<td>38</td>
<td>4007.81</td>
<td>0.00</td>
</tr>
<tr>
<td>39</td>
<td>4007.81</td>
<td>0.00</td>
</tr>
<tr>
<td>40</td>
<td>4534.86</td>
<td>0.00</td>
</tr>
<tr>
<td>41</td>
<td>2648.56</td>
<td>0.00</td>
</tr>
<tr>
<td>42</td>
<td>685.89</td>
<td>0.00</td>
</tr>
<tr>
<td>44</td>
<td>7659.49</td>
<td>0.00</td>
</tr>
<tr>
<td>45</td>
<td>8121.09</td>
<td>0.00</td>
</tr>
<tr>
<td>46</td>
<td>8121.09</td>
<td>0.00</td>
</tr>
<tr>
<td>47</td>
<td>5512.70</td>
<td>0.00</td>
</tr>
<tr>
<td>48</td>
<td>18949.22</td>
<td>0.00</td>
</tr>
<tr>
<td>49</td>
<td>17929.69</td>
<td>0.00</td>
</tr>
</tbody>
</table>

**Table 6.2 Results of Analysis**
effect of the opening. Members 8, 9, and 10 are symmetrical as expected. Member 11 has an end moment, but member 29 does not, indicating that the program recognized that there is no moment carried over at the column. Member 12 has an end moment, and this moment is carried over to member 31. Member 31's end shear is negative, indicating an uplift due to the moment from member 31. Members 18's LL and DL are in the proportion 100:40, indicating that the additional bay load was considered. Members 19, 22, and 24 all have the same reactions, and these are in the correct proportion. Members 20, 21, and 23 similarly have the same reactions. Members 26-28 have the same symmetric reactions, but their LL:DL ratio is not 75:40, or 125:40, indicating that the additional LL in that bay was carried down. The reactions of members 33-39 are symmetric, and equal, and their LL:DL ratio is 75:40. Member 41's total reactions are approximately equal to 1/4 of the area bounded by nodes 33, 52, 45, and 44, plus 1/4 of the area bounded by nodes 52, 53, 46, and 45, times the LL and DL. The reactions are not symmetric, indicating that the oblique corners were considered. Members 42 and 44, have reasonable reactions by similar reasoning.
CHAPTER VII
FUTURE DEVELOPMENTS

7.1 Graphical Interface

This program accomplishes the goal of providing an analysis tool for solving planar, determinate flooring systems. In order to be truly useful, however, two more modules need to be added: a graphical user interface/editor, and a member selection and code checking module.

A graphical editor interface should, at a minimum, allow the user to add, delete, or move nodes (and columns), add or delete members, and add or delete floor loads while viewing the results. Other extensions that would be desirable would be the ability to view tributary areas for the members as a way of insuring that the program is doing a "reasonable" load takedown. These features would greatly enhance the utility of the program as a whole by allowing the engineer to deal with the flooring system as a whole, rather than as a list of numbers. Seeing changes in the floor as they are input would greatly reduce the chance of an input error because a misplaced node would be immediately apparent.

A second benefit of a graphical interface would be the improved conceptual appeal of manipulating a drawing of the floor. This image manipulation is what engineers actually do when analyzing a floor by hand, so it would be a superior way to present the large amount of data that describes even a relatively simple flooring system.

The final benefit of a graphical interface is that it should reduce the "black box" nature of the program by allow-
ing the user to see what is being done to the floor. Making the operation of the program more transparent should also increase the user's confidence in the results.

7.2 Member Selection

A graphical user interface would improve the utility of the analysis program, but would do nothing to assist in translating the results into the design of the members for the flooring system. As was explained in the introduction to this thesis, the computer is quite useful as a tool for performing the routine aspects of design, freeing the designer to concentrate on the exceptions. To assist in this phase of the design, a member selection module could be added to automatically take the output from the analysis module and perform the appropriate code checks. This step should not preclude the results of the analysis from being available should a user desire to choose the members.

Although members could be selected automatically, it is still necessary that the user be able to obtain useful information about the loads on every member. The best way to present this information would be to have the program draw the shear and moment diagrams for any member on demand.

7.3 Additional Analysis Modules

A final major addition that could be made to this program is the addition of additional modules for different flooring systems. At present, the program is geared towards analyzing one and two way slabs on simply supported steel beams. Other reasonable systems would be composite slab/beam, or all
concrete systems. Because of the modular nature of this program, it would not be difficult to introduce these new modules. The editor and load takedown routines need no modification, nor would the sections of the code that find the closed circuits associated with members. Calls to the new framing type modules would simply need to be added at the end of the MEMBERLP function. One qualification for new routines, however, is that this program will only work in a flooring system in which load takedown is performed with supporting beams and girders. For this reason the program is of little use in analyzing a flat plate or waffle slab where no beams are present.

7.4 Present Limitations

This program currently has several limitations that the user must be aware of. First, the program currently cannot analyze floors with fixed connections (with the exception of allowing some cantilevers). Although cantilevers are allowed, there are two restrictions on their placement: a cantilever may frame into a column, in which case the cantilever moment is assumed to be taken by the column, or a cantilever may frame into another member where there is another collinear member framing in on the opposite side, in which case the member on the other side is assumed to have a fixed connection and to take all of the cantilever moment (the torsional effects on the member that the cantilever frames into are ignored). The effect of another fixed connection on the opposite side of a cantilever - column system is not modelled by the program. Additionally, the program does not make any
checks to see whether the cantilever frames into the strong or the weak axis of the column. This case falls under the category of exceptions which must be dealt with by the designer since a frame analysis is required.

Although multiple load cases are implemented, they could be made more flexible by including a text string identifier in the LOAD_CASE data structure, rather than simply a load case number. There is no automatic calculation of self weight of the members in the analysis. The purpose of allowing multiple load cases is to have the ability to differentiate between load classes, e.g. live load and dead load, for which the building code allows different factors of safety or load reduction strategies. The LOAD_CASE data structure includes an entry for the size of that load case's tributary area, but this information is not currently maintained. In order to make full use of the codes provision for load reduction mentioned above, this tributary area information must be included during load takedown. Finally, for load cases to be useful, some sort of protocol needs to be established for which load case represents which type of load in the codes. As an example, the first load case could always be reserved for the member's self weight. Although this will not be known at the time of the initial analysis, a space would be reserved by initially setting the self weight to zero and filling it in later following member selection. Similarly, load case two could be assigned for dead load, and load case 3 for live load. Several other strategies are possible, including tables
or tags in the load case structure itself.

In Chapter IV, when the algorithm that carries down floor loads to create member loads was discussed, it was pointed out that the algorithm is not wholly applicable if a floor load is not distributed evenly over the entire bay. For suggestions on dealing with this situation in a more correct manner, see section 4.5.

During the course of the development of this program, some inconsistencies arose in the data structures representing an area. In order to make use of efficient algorithms for geometric manipulations (e.g. intersection and line definition) the h_point (homogeneous point) and h_area (homogeneous area) data structures were introduced. As a result of these inconsistencies, several functions, such as ONEWAY and TWOWAY, perform internal calculations in one format, but return a result in the other format. These translations of data format could be eliminated by consistently using the h_area format in the member data structure, and by using the h_point format to represent nodal coordinates in the node data structure. For a more complete discussion of the two representations now used, see the documentation for the CAREA_TO_HAREA, HAREA_TO_CAREA, CORNER_TO_HPOINT, and HPOINT_TO_CORNER, functions in Appendix B, and the hpoint, corner, area, and harea data structure documentation in Appendices D and E.

Because memory for some of the data structures in this program is allocated dynamically, memory should be freed when no longer needed so that the space may be reallocated later by the program. Currently, the HP_DELETE function deletes a
vertex from an area represented as a linked list of homogenous points, but does not free the memory of the released point. This is not a problem when working on a machine with large amounts of core memory, but should be changed to prevent memory from being cluttered with unneeded data. The path function was written to reuse the same array for each member, so no "garbage" is created by this function. This insurance against needlessly allocating memory comes at the cost of always allocating the same initial amount of memory for the array. A more general approach would be to use dynamically allocated linked lists for the path function.

The program currently cannot save/retrieve multiple floors because the same names are always given to the files created by a save. This limitation may be overcome by either including the floor number (currently not implemented, but allowed for in the data structure) in the file names, or by asking the user for a name for the floor to be saved.

7.5 Conclusion

This thesis has demonstrated that computer aided analysis of structural flooring systems is feasible. Additional work needs to be done to make the analysis package easy to use, but the potential for improving the speed and ease of flooring system design clearly warrants this effort.
1. Appleton, J.H., "Increased Design Productivity Using
Computers," 7th Conference on Electronic Computation,

2. Blum, Frederic L., "Verification of CAD Programs, The User's
Viewpoint," 7th Conference on Electronic Computation,
ASCE, 1979, p.767-772.

3. Conaway, John H., "The Economics of Structural Analysis
on Superminis. 7th Conference on Electronic Compu-
tation, ASCE 1979, p.374-385.


5. Ketchum, Milo S., "Structural Analysis Programs For Home
Computers," 7th Conference on Electronic Computation,
ASCE, 1979, p.38-43.

6. Lamberti, H. V., "The Algorithm is a Necessary Engineering
Aid," 7th Conference on Electronic Computation, ASCE,
1979, p.773-780.

7. Logcher, Robert D., et. al., ICES STRUDL-II, Engineering
User's Manual, Volume II. Department of Civil
Engineering, Massachusetts Institute of Technology,

System," 6th Conference on Electronic Computation,
ASCE, 1974.

9. Louie, Peter, and Kenneth Hatai, "Microcomputer Applications
in Small Offices," 7th Conference on Electronic

10. Louie, Peter, "Microcomputer Graphics," 7th Conference on

6th Conference on Electronic Computation, ASCE,
1974.

12. Noland, Jimmy L. and Chuan C. Feng, "The ACI 318-71
Building Code in Decision Logic Table Format," 6th

Model for Design Specifications," 6th Conference on


**GRAPH THEORY**


**GRAPHICS**


**ANALYSIS**


APPENDIX A

User's Manual

Currently, flooring systems to be analyzed must be entered through the input files mentioned in Chapter V (File Operations). The format of each of these files will be discussed in the first section of this appendix. As was indicated in Chapter VII (Future Developments), the first priority for future development of this program should be a graphical interface. The menu structure for this interface has been partially implemented, although the choice of items in the menu is quite restricted. A description of the menu, and the use of commands currently available in the menu structure comprise the latter half of this appendix.

A.1 Input File Formats

To illustrate the file format for the problem input, input files will be created for the section of flooring shown in figure A.1.

-6-

![Diagram of floor segment]

# Indicates node number  -#- Indicates member number

FIGURE A.1 EXAMPLE FLOOR SEGMENT
Nodes 10 and 11 are included to indicate that the floor actually continues to the right of the section shown. In a complete floor, there would be no reason to include nodes which have no member framing into them. Creation of this floor will begin by creating the nodelist.dat and memblist.dat files which describe the geometry of the floor. Once these files have been created, the fil_list (frame in list) and ni_list (node list) files will be created to represent connectivity information. The file opening.dat will then be created to represent the opening in this floor. Finally, loads will be assigned by creating the floorload.dat and loadlist.dat (member load) files. All of the input files are standard ASCII text files, so they may be created with any editor. Individual data items may be separated by any combination of spaces, tabs, or newline characters.

In the nodelist.dat file, a node is represented by an integer node number followed by three real nodal coordinates (x, y, z). The next 8 items are the node's flags, and must be either 1 (yes or true) or 0 (no or false). The first flag is the node_type which may be 1 or 0 (not currently used), the next three flags indicate whether the node resists translation in the x, y, and z directions respectively. The next set of three flags indicate whether the node resists rotation about the x, y, and z axes. The last flag is the cantilever flag which indicates that the node is at the free end of a cantilever. The last two items of data are integers to indicate the member that the node is on (0 if the node is on a column),
and the column that the node is on (0 if the node is on a member). Column numbers are not currently used, so any non-zero number will have the same effect. The complete file for the floor in figure A.1 is shown in figure A.2.

1 0.000000 0.000000 0.000000 1 1 1 1 0 0 0 0 0 1
2 100.000000 0.000000 0.000000 1 1 1 1 0 0 0 0 0 1
3 0.000000 50.000000 0.000000 1 1 1 1 0 0 0 0 1 7 0
4 100.000000 50.000000 0.000000 1 1 1 1 0 0 0 0 0 1
5 50.000000 0.000000 0.000000 1 1 1 1 0 0 0 0 4 0
6 50.000000 25.000000 0.000000 1 1 1 1 0 0 0 0 2 0
7 100.000000 25.000000 0.000000 1 1 1 1 0 0 0 0 3 0
8 50.000000 50.000000 0.000000 1 1 1 1 0 0 0 0 6 0
9 0.000000 25.000000 0.000000 1 1 1 1 0 0 0 0 1
10 100.000000 9.000000 0.000000 1 1 1 1 0 0 0 0 3 0
11 100.000000 18.000000 0.000000 1 1 1 1 0 0 0 0 3 0

FIGURE A.2 FILE NODELIST.DAT

The memblst.dat file consists of a list of member records, each of which contains: an integer member number, flags (0 or 1) to indicate whether the member is solved (this is always reset before analysis so its value has no effect on the program), whether it is a fictitious edge beam inserted to perform path tests (section 3.1), and whether it is a cantilever. The last two data items are the integer node numbers of the member’s begin and end nodes respectively. The complete memblst.dat file for the floor would is shown in figure A.3.

1 0 0 0 1 9
2 0 0 0 5 6
3 0 0 0 2 4
4 0 0 0 9 6
5 0 0 0 6 7
6 0 0 0 3 4
7 0 0 1 9 3

FIGURE A.3 FILE MEMBLIST.DAT
Although both the nodelist.dat and memblist.dat files have been listed in node and member number sequence, the program is insensitive to the order of entry.

Connectivity information for the floor is contained in the next two files: fil_list.dat and ni_list.dat. The fil_list.dat file contains pairs of a node number and a member that frames into that node. The number of entries for each node will vary, however all nodes normally have at least one member that frames into it. The fil_list.dat file for the sample floor is shown in figure A.4.

```
1 1
1 4
2 3
2 4
3 7
3 6
4 6
4 3
5 2
6 5
6 4
7 5
8 2
9 1
9 7
```

**FIGURE A.4 FILE FIL_LIST.DAT**

The ni_list.dat file contains analogous information about the nodes which are on each member. Again the file consists of pairs of integers, this time the member number comes first, followed by the node number. Unlike the other input files, the order of entry is important for this file. Members may be in any order, but for each member, nodes must be given in order from those nearest to the beginning of the member to those nearest to the end. Table A.5 gives the ni_list.dat
file for this example floor.

4 5
3 7
3 11
3 10
2 6
6 8
7 3

FIGURE A.5 FILE NL_LIST.DAT

Notice that node 3 shown in figure A.5 is defined as being on member 7. Because it is a cantilever node, the member that supports it must be identified even though it is also the end node of member 7. Nodes 11, 10 and 7 are in the correct order since member 3 begins at node 2.

The final step in defining the floor's geometry is to create the opening.dat file. The opening.dat file consists of an integer name, date assigned, the ID number of the user who assigned the opening, a coordinate flag (currently not used), and a magnitude, which is always 0. Next comes a list of the corners (points) that bound the hole. The list must be traversed in clockwise order around the perimeter, and the opening must be convex (no reentrant corners). Non convex openings may be entered by superimposing a series of convex openings. The list itself consists of a non zero integer tag, followed by four homogeneous coordinates \((x, y, z, w)\). The \(w\) coordinate is always 1 if \(x, y, \) and \(z\) are cartesian coordinates. There may be an arbitrary number of points. The end of the list of points is marked by a tag of 0. If there is more than one opening in the floor (see the floorload.dat file listed in FIGURE A.7), this set of data may be repeated in the
file. For the opening in the above floor, the opening.dat file is given in figure A.6.

1
841220
101
1
0
1 50.000 0.000 0.000 1.0
2 50.000 25.000 0.000 1.0
3 100.000 25.000 0.000 1.0
4 100.000 0.000 0.000 1.0
0

FIGURE A.6 FILE OPENING.DAT

Next, floor loads will be assigned by creating a floorload.dat file. The format of this file is the same as that of the opening.dat file discussed above. For floor loads, the name corresponds to the load_case number (1 live load(LL), 2 dead load(DL)). The magnitude is the magnitude of the load. Three loads will be assigned to the example floor: 75 psf LL and 40 psf DL over the entire floor, and an additional 50 psf LL over the upper right 1/4 of the floor. The same restrictions on direction of traversal and convexity that apply to openings apply to floor loads. At least one floor load must be applied for each load case, and the first instances of each load case must be in order, e.g. at least one load for load case 1 must be applied before any loads are applied for load case 2. The floorload.dat file for the example floor is listed in figure A.7.

Loadlist.dat, the final file required to describe the floor, is list of directly applied member loads for the floor. The format of this member load file is similar to that of the floorload.dat and opening.dat files, except that each load has
<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>841220</td>
<td>101</td>
<td>1</td>
<td>75</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>1.0</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0.000</td>
<td>50.000</td>
<td>0.000</td>
<td>1.0</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>100.000</td>
<td>50.000</td>
<td>0.000</td>
<td>1.0</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>100.000</td>
<td>0.000</td>
<td>0.000</td>
<td>1.0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>841220</td>
<td>101</td>
<td>1</td>
<td>40</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>1.0</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0.000</td>
<td>50.000</td>
<td>0.000</td>
<td>1.0</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>100.000</td>
<td>50.000</td>
<td>0.000</td>
<td>1.0</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>100.000</td>
<td>0.000</td>
<td>0.000</td>
<td>1.0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>841220</td>
<td>101</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>50.000</td>
<td>25.000</td>
<td>0.000</td>
<td>1.0</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>50.000</td>
<td>50.000</td>
<td>0.000</td>
<td>1.0</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>100.000</td>
<td>50.000</td>
<td>0.000</td>
<td>1.0</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>100.000</td>
<td>25.000</td>
<td>0.000</td>
<td>1.0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>

**FIGURE A.7 FILE FLOORLOAD.DAT**

two magnitudes, one at its beginning, and one at its end, and only two coordinates, for its beginning and end. Member load coordinates are given in local coordinates (0 at the beginning, 1 at the end of the member). The name of the load contains both the member number and the load case number for the load. The least significant digit is the load case number, the other digits give the member number to which the load is applied e.g. load number 231 would belong to load case 1 of member 23. Two member loads will be applied to the example floor, a LL of 25 plf on the entire length of member 5, and a
DL of 30 plf on the part of member 3 between nodes 7 and 4. The same restriction about assigning floor loads for loadcase 1 before assigning floor loads for load case 2 also applies to member loads. For this reason, a member LL of 0 is assigned to member 3 prior to assigning the DL. The complete file is listed in figure A.8.

```
51
  841220
  101
  1
  25.0 25.0
  0.0 1.0

31
  841220
  101
  1
  0.0 0.0
  0.0 1.0

32
  841220
  101
  1
  30.0 30.0
  00.5 1.0
```

FIGURE A.8 FILE LOADLIST.DAT

A.2 Command Menu Structure

Once the necessary file have been created, the program is started by typing "floor". The program is menu driven (fig. A.9), although only a few of the menu choices are currently implemented.

A choice is made by typing the choice (or the first letter of the choice) in upper or lower case followed by a carriage return. Each of the choices will cause execution of a command, or lead to a sub menu of commands for the functional areas. Each of these sub menus shall be examined in
the order that they appear above.

The first choice in the main menu, EDIT, invokes the edit menu (fig. A.10). None of the EDIT menu choices are currently active.

The second choice on the main menu, FILE, invokes the FILE menu (fig. A.11), which has three options. Because a floor must be loaded from files, and it is not possible to edit the floor, only the RETRIEVE option is currently active.

The third choice on the main menu invokes the LOADS menu (fig. A.12), which allows the user to assign either member or floor loads. None of these options are currently active.

The fourth choice on the main menu, ANALYZE, invokes the analysis routines for the floor, after checking to see that loads have been assigned.

The last choice on the main menu, RESULTS, invokes the RESULTS menu (fig A.13). The selections on this menu (when they are implemented) allow the user to specify whether the results of the analysis should be displayed on the screen, sent to a file, or both. This sub menu also includes a choice to display shear and moment diagrams for any member. None of these selections is currently implemented.

To run the example above, the letter "f" or word "file" (upper or lower case) is typed from the main menu, followed by a carriage return (\textasciicircum CR) to invoke the file menu. In the FILE menu, "r"\textasciicircum CR or "retrieve"\textasciicircum CR is typed to retrieve the floor from data files. Once the floor retrieval is complete, the main menu reappears. The ANALYZE option is selected by typing
"a"(CR) or "analyze"(CR). The analysis will begin, and will write the file results.out (table 6.2). Program execution is terminated by typing "quit" (CR) or "q" (CR).

---

**Figure A.9 Main Menu**
FIGURE A.10 EDIT MENU

FIGURE A.11 FILE MENU
FIGURE A.12 LOAD MENU

FIGURE A.12 RESULTS MENU
APPENDIX 3

Program Documentation and Internal Logic
NAME

    add_column -- attach a new column to the floor

SYNOPSIS

    struct column *add_column();

DESCRIPTION

    This function allocates memory for a new column.

RETURNS

    column1 -- pointer to new column structure if successful
    NULL    -- if unsuccessful

CAUTIONS

    Function does not currently have a way of checking whether
    floor loads have been set.
NAME

add_edge -- attach a new node to the floor

SYNOPSIS

add_edge(n1, n2, adjlist);
struct node   *n1, *n2;
adjnode       *adjlist[];
typedef listnode {
    struct node   *node_ptr;
    struct listnode   *next;
} adjnode;

DESCRIPTION

This function adds an edge connecting n1 and n2 in an adjacency list. Two new nodes are added to the adjacency list, one adds n2 to n1's list, the other adds n1 to n2's list. Space for the new nodes is added by a call to the add_adj() function.

RETURNS

Nothing

CAUTIONS

Function does not currently return a status to indicate success or failure of the procedure.
NAME

add_fil -- attach a new node to the floor

SYNOPSIS

struct frame_in_list *add_fil(node_ptr);
struct node   *node_ptr;
struct frame_in_list {
    struct member    *in_member;
    struct frame_in_list   *next;
}

DESCRIPTION

This function allocates memory for a new frame_in_list for a
node. If the node already has a frame_in_list(s), the new
structure is added to the end of the linked list.

RETURNS

fill -- pointer to new frame_in_list structure if successful

CAUTIONS

Function does not currently return an error status to
indicate failure to allocate space.
NAME
add_lc -- attach a new member to the floor

SYNOPSIS

struct load_case *add_lc(member_ptr);

struct member *member_ptr;

struct load_case {
    int lc_number;
    struct reaction begin_reac;
    struct reaction end_reac;
    struct load *loads;
    struct load_case *next_lc;
}

DESCRIPTION

This function allocates memory for a new load_case for a member. If the member already has a load_case(s), the new structure is added to the end of the linked list.

RETURNS

lc1 -- pointer to new load_case structure if successful

CAUTIONS

Function does not currently return an error status to indicate failure to allocate space.
NAME

add_load -- attach a new load to the floor

SYNOPSIS

struct load *add_load(ic_ptr);

struct load_case *ic_ptr;

struct load_case {
    int load_number;
    struct reaction begin_reac;
    struct reaction end_reac;
    struct load *loads;
    struct load_case *next_load;
}

struct load {
    int date_assigned;
    int who_assigned;
    int coord_flag;
    double begin_x, end_y;
    double begin_mag, end_y;
    struct load *next_load;
}

DESCRIPTION

This function allocates memory for a new load in a load_case. If the load_case already has a load(s), the new structure is added to the end of the linked list.

RETURNS

load1 -- pointer to new load_case structure if successful

CAUTIONS

Function does not currently return an error status to indicate failure to allocate space.
NAME

    add_member -- attach a new member to the floor

SYNOPSIS

    struct member *add_member();

DESCRIPTION

    This function allocates memory for a new member. The new member is attached to the end of the linked list of members for the current floor.

RETURNS

    member1 -- pointer to new node structure if successful
    NULL -- if unsuccessful

CAUTIONS

    Function does not currently have a way of checking whether floor loads have been set.
NAME

add_nl -- attach a new node_list to a member

SYNOPSIS

struct node_list *add_nl(member_ptr);

struct member *member_ptr;

struct node_list {
    struct node *in_node;
    struct node_list *next;
}

DESCRIPTION

This function allocates memory for a new node_list for a member. If the member already has a node_list(s), the new structure is added to the end of the linked list.

RETURNS

nll -- pointer to new node_list structure if successful

CAUTIONS

Function does not currently return an error status to indicate failure to allocate space.
NAME

add_node -- attach a new node to the floor

SYNOPSIS

struct node *add_node();

DESCRIPTION

This function allocates memory for a new node.

RETURNS

node1 -- pointer to new node structure if successful
NULL -- if unsuccessful

CAUTIONS

Function does not currently have a way of checking whether floor loads have been set.
NAME

adj_list -- create an adjacency list representation of a floor

SYNOPSIS

adjnode *adj_list(adjlist)
adjnode *adjlist[];

typedef listnode {
    struct node *node_ptr;
    struct listnode *next;
} adjnode;

DESCRIPTION

This function creates an adjacency list representation for a floor. This function traverses the linked list of floor members, calling the find_neighbors function to determine the neighbors for each node. A completed adjacency list consists of an array of pointers to adjsnode with one pointer for each node. Each of these adjsnode is the beginning of a linked list of neighbors which represents the connectivity of the floor.

RETURNS

adjlist -- pointer to the adjacency list if successful
NULL -- if unsuccessful

CAUTIONS

A warning about incompatible pointer types is generated by the return of the pointer adlist.
NAME

adjust_for_cantilever -- perform recursive analysis of a member

SYNOPSIS

adjust_for_cantilever(member_ptr);

struct member *member_ptr;

DESCRIPTION

This function is the last to be called before recurse() returns from analysis of a member. Adjust_for_cantilever calls the function find_carry_over() to determine whether there is a cantilever opposite from the end of the member being analyzed. If there is a cantilever, the program requires the member to have a fixed connection and a carry over moment at that end to balance the cantilever.

The carry over moment is applied to the member in the form of a couple by adjusting the member's end shears.

RETURNS

nothing

CAUTIONS

Function does not currently return an error status if it fails.
NAME

analyze -- perform the analysis of a floor

SYNOPSIS

analyze();

DESCRIPTION

This function loops through the members in a floor, calling the recurse() function to solve for their end reactions.

RETURNS

nothing

CAUTIONS

Function does not currently have a way of checking whether floor loads have been set.
NAME

carea_to_harea -- converts the data format of an area using corner data structure into an area using the h_area data structure.

SYNOPSIS

h_area *carea_to_harea(ca_ptr);

struct area *ca_ptr;

struct area {
    double  *area;
    struct corner *corner;
    struct area  *next;
};

typedef struct harea {
    h_point  *point;
    h_area   *next;
} h_area;

DESCRIPTION

This function performs a conversion of the data format for representing an area as a linked list. It is needed because of a change in the representation during the course of program development. The area representation using the corner data structure was the original, and is the one used in the member structure. The h_area data structure was implemented to facilitate the use of homogeneous coordinates when the geometric algorithms were written later.

RETURNS

h_area *h_head -- pointer to the clipped area

CAUTIONS

Function does not currently return an error status if it fails.
NAME

carry_down - transform floor loads into member loads

SYNOPSIS

carry_down(area_ptr, member_ptr, magnitude)

h_area *area_ptr;
struct member *member_ptr;
double magnitude;

typedef harea(
    h_point *point;
    struct harea *next;
) h_area;

DESCRIPTION

This function takes a clipped floor area found in the clip_polygon function and converts the area load into a member load which is then attached to the member.

RETURNS

Nothing is returned.

CAUTIONS

Because no error indication is returned, there is no way to tell if the function was successful.
NAME

clip -- clips a polygon against a line defined by two corners of a floor load polygon.

SYNOPSIS

h_area *clip(a1, a2, area_ptr);

h_area *a1, *a2, *area_ptr;

typedef struct harea {
    h_point *point;
    h_area *next;
} h_area;

DESCRIPTION

This function clips the area polygon pointed to by area_ptr against the directed line from corner a1 to corner a2 of a floor load. This procedure clips the area, retaining only the portion that lies to the right of the directed line.

RETURNS

h_area *area_ptr -- pointer to the clipped area

CAUTIONS

Function does not currently return an error status if it fails.
NAME
  clip_openings -- perform an intersection of floor opening polygons with floor load polygon.

SYNOPSIS
  clip_openings();

DESCRIPTION

  This function performs a polygon intersection of each of the floor opening polygons with each of the floor load polygons. Clip_openings() loops through the opening areas, starting at opening_head, an external pointer to the head of the opening list. The function then loops over the floor loads, using the clip() and filter_ha() functions to clip the opening against each side of the floor load.

  If an area of intersection is found, the function creates a new floor load of equal but opposite magnitude so that the superposition of the real and fictitious load will be 0 over the hole area.

RETURNS

  Nothing

CAUTIONS

  None
NAME

clip_polygon -- finds the intersection of a floor load polygon and a member area polygon.

SYNOPSIS

clip_polygon(load, member_ptr);

struct f_load *load;
struct member *member_ptr;

struct f_load {  
  int date_assigned;
  int who_assigned;
  int coord_flag;
  double magnitude;
  h_area *area;
  struct f_load *next;
}

DESCRIPTION

This function clips the tributary area polygon of the member pointed to by member_ptr successively against lines through the sides of the floor load polygon by calling clip(). The result of this procedure is the polygon of intersection, if any, of the floor load with the member tributary area. Once the loaded area (intersection) has been found, the carry_down() function is called to transform the floor load into member loads, which are attached the member.

RETURNS

nothing

CAUTIONS

Function does not currently return an error status if it fails.
NAME

    copy_harea -- copy an h_area data structure

SYNOPSIS

    h_area *copy_harea(a_ptr);

    h_area   *a_ptr;

    typedef struct harea {
        h_point   *point;
        h_area    *next;
    } h_area;

DESCRIPTION

    This function creates a new instance of an h_area data structure and copies an existing h_area data structure into the new h_area. It is needed because some versions of C compilers will not allow assignment of data structures.

RETURNS

    struct area *ca_ptr -- pointer to the converted area.

CAUTIONS

    Function does not currently return an error status if it fails.
NAME

corner -- check three nodes to see if they form a corner

SYNOPSIS

corner(n1, n2, n3);

struct node *n1, *n2, *n3;

DESCRIPTION

This function determines whether n1, n2, n3, form a corner by converting the nodes' coordinates into homogeneous coordinates and calling the right_of() function. If right_of() returns zero, then the three points are colinear and cannot be a corner. Otherwise, if right_of returns a non zero result, the three nodes form a corner.

RETURNS

0 -- nodes are not a corner (nodes are colinear)
1 -- nodes are a corner (nodes are not colinear)

CAUTIONS

None
NAME

corner_to_hpoint -- converts the data format of an area
using corner data structure into an
area using the h_area data structure.

SYNOPSIS

h_point *corner_to_hpoint(c1, h1);

struct corner *corner;

h_point *point;

struct corner {
    double x, y, z;
    struct corner *next;
};

typedef struct hcoord {
    double x, y, z, w;
} h_point;

DESCRIPTION

This function performs a conversion of the data format
for representing a corner of an area. It is needed because
of a change in the representation during the course of program
development. The area representation using the corner data
structure was the original, and is the one used in the member
data structure. The h_point data structure was implemented to
facilitate the use of homogeneous coordinates when the geometric
algorithms were written later.

RETURNS

h_point *h1 -- pointer to the converted point

CAUTIONS

Function does not currently return an error status if it fails.
NAME
    edit_node -- change node data

SYNOPSIS
    edit_node(node_ptr);
    struct node *node_ptr;

DESCRIPTION
    This function permits the user to alter node data interactively.

RETURNS
    nothing

CAUTIONS
    None
NAME

filter -- removes non corner nodes from a path

SYNOPSIS

    filter(path1);

union pathnode  *path1;

union pathnode {
    int        *length;
    struct node  *node_ptr;
}

DESCRIPTION

This function takes a circuit created by the path() function and removes any nodes that are not corners of the path. The resulting filtered path contains only the corners of the area represented by the path.

RETURNS

Nothing

CAUTIONS

Currently does not return an error status if the algorithm fails.
NAME

filter_ha -- removes points from a member tributary area polygon that lie outside of a clipping line.

SYNOPSIS

h_area *filter_ha(p1, p2, ha_ptr);

h_point *p1, *p2;

h_area *ha_ptr;

typedef struct hcoord {
    double x, y, z, w;
} h_point;

typedef struct harea {
    h_point *point;
    h_area *next;
} h_area;

DESCRIPTION

This function takes an area created by clip which may contain points to the right of, left of or on the directed line p1-p2, and discards the points to the left of the line. The result is a clipped polygon that lies to the right (inside) of the line.

RETURNS

h_area *area_head -- pointer to the filtered list.

CAUTIONS

Currently does not return an error status if the algorithm fails.
NAME

find_carry_over -- perform recursive analysis of a member

SYNOPSIS

double find_carry_over(ic_number, member_ptr, node_ptr);

int ic_number;
struct member *member_ptr;
struct node *node_ptr;

DESCRIPTION

Find_carry_over() calls the function find_opposite() to
determine whether there is a cantilever at node_ptr opposite
from the member being analyzed. If there is a cantilever,
find_carry_over() checks to see if it has been solved. If
not, the function calls recurse() to solve the cantilever.
Once the cantilever is solved, the cantilever end moment is
reversed and returned to adjust_for_cantilever.

RETURNS

carry_over_moment - if there is one
0 - otherwise

CAUTIONS

Function does not currently return an error status if it fails.
NAME

find_neighbors -- find the nodes that are adjacent to a given node.

SYNOPSIS

struct node *find_neighbors(node_ptr, neighbors);

struct node, neighbors[];

DESCRIPTION

This function locates all of a node's neighbors in order to create an adjacency list for the floor. The function examines the connectivity information stored in the node and member data structures to locate neighbors.

RETURNS

neighbors -- pointer to the list of neighbors

CAUTIONS

Function makes the assumption that the list of nodes in a member's node_list are in sequence.
NAME

find_opposite -- perform recursive analysis of a member

SYNOPSIS

find_opposite(member_ptr, node_ptr);

struct member *member_ptr;
struct node *node_ptr;

DESCRIPTION

Find_opposite() loops through *node_ptr's frame_in_list, checking to see if any cantilevers frame into the node. If a cantilever is found, it checked using the right_of() function to determine whether the cantilever is opposite from the member being analyzed. The definition of opposite for test purposes is that right_of() returns 0, indicating that the cantilever and the member being tested are collinear.

RETURNS

struct member *opposite -- pointer to opposite cantilever if there is one

NULL -- otherwise

CAUTIONS

Function does not currently return an error status if it fails.
NAME

get_area -- calculate the area of a polygon defined as a circular linked list of points

SYNOPSIS

double *get_area(area_ptr);

h_area *area_ptr;

typedef struct harea {
    h_point *point;
    h_area *next;
} h_area;

DESCRIPTION

This function finds the area of a polygon.

RETURNS

double area -- magnitude of the area.

CAUTIONS

Function does not currently return an error status to indicate failure to allocate space.
NAME

get_fload -- read floor loads from user input

SYNOPSIS

struct f_load *get_fload();

struct f_load {
    int date_assigned;
    int who_assigned;
    int coord_flag;
    double magnitude;
    h_area *area;
    struct f_load *next;
};

DESCRIPTION

This function reads user input for floor loads for a flooring system.

RETURNS

fli -- pointer to new f_load structure if successful

CAUTIONS

Function does not currently return an error status to indicate failure to allocate space.
NAME

get_floor -- read floor data from disk files

SYNOPSIS

get_floor(f_number);

int f_number;

DESCRIPTION

This function reads data files as input to create a floor.

RETURNS

Nothing

CAUTIONS

The function does not return an error status to indicate failure to create a floor
NAME

get_load -- find next load on a member

SYNOPSIS

struct load *get_load(member_ptr);

struct member *member_ptr;

DESCRIPTION

This function finds the next load on a member. It keeps track of where the analysis has progressed to so that subsequent calls for the same member will return pointers to subsequent loads.

RETURNS

*current_load -- pointer to the next load if successful
NULL if unsuccessful

CAUTIONS

Function does not currently have a way of distinguishing between running out of loads and indicating that loads have not been set.
NAME

get_reactions -- solve for beam end reactions

SYNOPSIS

get_reactions(ic_ptr, p1, p2, p3, p4);

struct load_case *ic_ptr;
double p1, p2, p3, p4;

DESCRIPTION

This function solves for the end reactions on a member for a single load, represented by p1 and p2 (begin and end magnitudes), and p3 and p4 (begin and end position on the member in local coordinates). Reactions are found assuming that the member is a cantilever fixed at its begin_node. The member's end reactions for the load_case are incremented. Adjustments for actual end conditions are made in the recurse() function.

RETURNS

Nothing

CAUTIONS

Function does not currently return an error status to
indicate failure to solve reactions.

NAME

get_reaction -- find end reactions with loads known

SYNOPSIS

get_reaction(member_ptr);

struct member *member_ptr;

DESCRIPTION

This function finds the end reactions on a member with known loads. The function will employ recursion to find needed end reactions of members that frame into the selected member. If distributed (floor) loads have not been set, the function will return an error (not currently functionaly)uc.

RETURNS

0 if successful
-1 if unsuccessful

CAUTIONS

Function does not currently have a way of checking whether floor loads have been set.
NAME

harea_to_carea -- converts the data format of an area using the h_area data structure into an area using corner data structure.

SYNOPSIS

struct area *harea_to_carea(a_ptr);

h_area    *a_ptr;

struct area {
    double   *area;
    struct corner *corner;
    struct area   *next;
};

typedef struct harea {
    h_point   *point;
    h_area    *next;
} h_area;

DESCRIPTION

This function performs a conversion of the data format for representing an area as a linked list. It is needed because of a change in the representation during the course of program development. The area representation using the corner data structure was the original, and is the one used in the member structure. The h_area data structure was implemented to facilitate the use of homogeneous coordinates when the geometric algorithms were written later.

RETURNS

struct area *ca_ptr -- pointer to the converted area.

CAUTIONS

Function does not currently return an error status.
NAME

`hpoint_to_corner` -- converts the data format of a point using the `h_area` data structure into a corner data structure.

SYNOPSIS

```c
struct corner *hpoint_to_corner(h1, c1);

h_point *point;
struct corner *corner;

typedef struct hcoord {
double x, y, z, w;
} h_point;

struct corner {
double x, y, z;
struct corner *next;
};
```

DESCRIPTION

This function performs a conversion of the data format for representing a corner of an area. It is needed because of a change in the representation during the course of program development. The area representation using the corner data structure was the original, and is the one used in the member data structure. The `h_point` data structure was implemented to facilitate the use of homogeneous coordinates when the geometric algorithms were written later.

RETURNS

`struct corner *c1` -- pointer to the converted point

CAUTIONS

Function does not currently return an error status.
NAME

hp_delete -- removes a corner from an h_area list

SYNOPSIS

hp_delete(ha);

h_area *ha;

typedef struct harea {
    h_point *point;
    h_area *next;
} h_area;

DESCRIPTION

This function deletes the corner after the one pointed to by ha, readjusting the pointers linking the list.

RETURNS

Nothing

CAUTIONS

Currently does not free the memory used by the deleted harea.
NAME

hp_equate -- equates two homogeneous points

SYNOPSIS

hp_equate(p1, p2);

h_point *hp1, *hp2;

typedef struct hcoord {
    double x, y, z, w;
} h_point;

DESCRIPTION

This function sets the coordinates of p1 equal to the coordinates of p2. This function is necessary because some versions of the C language do not allow data structures to be equated.

RETURNS

Nothing

CAUTIONS

Currently does not return an error status if it fails.
NAME

hp_insert -- inserts a corner in an h_area list

SYNOPSIS

hp_insert(ha, p2);

h_area    *ha;

h_point   *hp;

typedef struct harea {
    h_point   *point;
    h_area    *next;
} h_area;

typedef struct hcoord {
    double    x, y, z, w;
} h_point;

DESCRIPTION

This function inserts a new point after the hp in the linked list ha, readjusting the pointers linking the list.

RETURNS

Nothing

CAUTIONS

Does not return error status if it fails.
NAME
input_node -- change node data

SYNOPSIS
input_node();

DESCRIPTION
This function permits the user to input node data interactively.

RETURNS
nothing

CAUTIONS
None
NAME
    intersect -- change node data

SYNOPSIS

    intersect(L1, L2, point);
    h_line     *L1, *L2;
    h_point    *point;

typedef struct h_coord {
    double   x, y, z, w;
} h_line, h_point;

DESCRIPTION

    This function finds the coordinates of the intersection
    between two homogeneous lines.

RETURNS

    point -- pointer to the homogeneous point of intersection

CAUTIONS

    This function does not return an error status if it fails.
NAME

lc_init -- initialize a new load_case

SYNOPSIS

lc_init(lc_ptr);

struct load_case *lc_ptr;

struct load_case {
    int lc_number;
    struct reaction begin_reac;
    struct reaction end_reac;
    struct load *loads;
    struct load_case *next_lc;
}

DESCRIPTION

This function initializes all of the members of a load_case to zero.

RETURNS

Nothing

CAUTIONS

Function does not currently return an error status.
NAME
length -- find the length of a member

SYNOPSIS

double length(member_ptr);
struct member *member_ptr;

DESCRIPTION

This function finds the length of a member by finding the
distance between its begin and end nodes.

RETURNS

length -- length of the member

CAUTIONS
This function does not return an error status if it fails.

NAME

length -- find length of a member

SYNOPSIS

double length(member_ptr);

struct member *member_ptr;

DESCRIPTION

This function finds the length of a member by taking the end
coordinates and applying the Pythagorean Theorem.

RETURNS

length if successful
-1 if unsuccessful

CAUTIONS

Function currently considers the length in the Z direction
although members are planar and should not have any Z length.
NAME

line -- change node data

SYNOPSIS

line(p1, p2, line);

h_point   *p1, *p2;

h_line    *line;

typedef struct h_coord {
    double   x, y, z, w;
} h_line, h_point;

DESCRIPTION

This function finds equation of the homogeneous line defined by two homogeneous points.

RETURNS

line -- pointer to the homogeneous line defined by p1, p2.

CAUTIONS

This function does not return an error status if it fails.
NAME

line_distance -- change node data

SYNOPSIS

double line_distance(p1, L1);

h_line *L1;

h_point *p1;

typedef struct h_coord {
    double x, y, z, w;
} h_line, h_point;

DESCRIPTION

This function finds the shortest distance between a line and a point.

RETURNS

distance -- distance between the line and the point

CAUTIONS

This function does not return an error status if it fails.
NAME

list_members -- list all members in the structure

SYNOPSIS

list_members()

DESCRIPTION

This function lists all of the members in a flooring system.

RETURNS

Nothing

CAUTIONS

This function does not return an error status if it fails.
NAME
list_areas -- list all areas in the structure

SYNOPSIS
list_areas() 

DESCRIPTION
This function lists all of the member areas in a flooring system.

RETURNS
Nothing

CAUTIONS
This function does not return an error status if it fails.
NAME
    list_fil -- list all frame_in_lists in the structure

SYNOPSIS
    list_fil()

DESCRIPTION
    This function traverses the floor's nodelist starting at
    the external pointer node_head. The members in each node's
    frame_in_list are listed to the stdout in tabular form.

RETURNS
    Nothing

CAUTIONS
    Function does not currently return an error status if it
    fails.
NAME
list_nl -- list all node_lists in the structure

SYNOPSIS
list_nl()

DESCRIPTION
This function traverses the floor's member list starting at the external pointer member_head. The nodes in each member's node_list are listed to the stdout in tabular form.

RETURNS
Nothing

CAUTIONS
Function does not currently return an error status if it fails.
NAME
  list_reactions -- list all reactions in the structure

SYNOPSIS
  list_reactions()

DESCRIPTION
  This function lists all of the member end reactions in a flooring system.

RETURNS
  Nothing

CAUTIONS
  This function does not return an error status if it fails.
NAME

load_init -- initialize a new load_case

SYNOPSIS

load_init(date, who, cf, bx, ex, bm, em, load_ptr);

int            date;
int            who;
int            cf;
double         bx, ex;
double         bm, em;
struct load    *load_ptr;

struct load {
    int            date_assigned;
    int            who_assigned;
    int            coord_flag;
    double         begin_x, end_x;
    double         begin_mag, end_mag;
    struct load    *next_load;
}

DESCRIPTION

This function sets all of the members of a load except for the pointer, next_load, which is updated in add_load.

RETURNS

Nothing

CAUTIONS

Function does not currently return an error status.
NAME

main -- overall controller routine

SYNOPSIS

main();

DESCRIPTION

This function is the main controller that calls all of the other routines. It is organized into a menu structure (see Appendix A).

RETURNS

Nothing

CAUTIONS

None
NAME

member_lp -- loops over the physical members in a floor, finding the member's tributary areas

SYNOPSIS

member_lp(adjlist, allpaths);
adjnode *adjlist[];
union pathnode allpaths[][MAXPATH];
typedef listnode {
    struct node *node_ptr;
    struct listnode *next;
} adjnode;
union pathnode {
    int *length;
    struct node *node_ptr;
}

DESCRIPTION

This function loops over the physical members in a floor, finding the eir tributary areas. The first part of the function identifies the graphical members that comprise the physical member, then calls the path() function to find the closed circuits (areas) associated with the member. The paths are checked for duplications by the unique() function. Once all of the graphical members' paths have been found, the appropriate one-way() or two-way() function is called to translate the paths into tributary areas. The areas are added to the physical member's data structure.

RETURNS

Nothing

CAUTIONS

Does not return an error status if it fails.
NAME
memb_init -- initialize a new member

SYNOPSIS
memb_init(memb_ptr);
struct member *memb_ptr;

DESCRIPTION
This function initializes all of the members of a member
to zero.

RETURNS
Nothing

CAUTIONS
Function does not currently return an error status.
NAME

m_to_hlin -- finds the equation of a homogenous line between the end points of a member.

SYNOPSIS

h_line *m_to_hlin(member_ptr, line_ptr)

struct member *member_ptr;

h_line *line_ptr;

typedef h_coord {
    double x, y, z, w;
} h_line;

DESCRIPTION

This function constructs a homogeneous line through the end points of a member. The homogeneous line representation is used to perform efficient intersections and distance calculations.

RETURNS

line_ptr -- pointer to the homogeneous line if successful

NULL -- if unsuccessful

CAUTIONS

This algorithm is only valid for two dimensional geometry.
NAME
  nodelist -- list all nodes in the structure

SYNOPSIS
  nodelist()

DESCRIPTION
  This function lists all of the nodes in a flooring system.

RETURNS
  Nothing

CAUTIONS
  This function does not return an error status if it fails.
NAME

node_init -- initialize a new node

SYNOPSIS

node_init(node_ptr);
struct node *node_ptr;

DESCRIPTION

This function initializes all but the members of a node data
structure to zero except that the node's coordinates are all set
to -1.0, the node type flag is set to indicate that it is on a
girder, and the translation flags are set to 1 to indicate that
the node is fixed against rotation in the x, y, and z directions.
The pointers to the next and last nodes in the node data
structure are set in the add_node() function.

RETURNS

Nothing

CAUTIONS

Function does not currently return an error status.
NAME

node_to_hpoint -- creates a homogeneous point with the coordinates of a given node

SYNOPSIS

h_point *node_to_hpoint(n1, h1);

struct node *n1;

h_point *h1;

typedef struct hcoord {
    double x, y, z, w;
} h_point;

DESCRIPTION

This function performs a conversion of the data format for representing a coordinate. The h_point data structure was implemented to facilitate the use of homogeneous coordinates in geometric algorithms.

RETURNS

h_point *h1 -- pointer to the converted point

CAUTIONS

Function does not currently return an error status.
NAME

one_way -- converts a member's list of circuits to tributary areas

SYNOPSIS

one_way(member_ptr, all_paths, direction);

struct member *member_ptr;

union pathnode *all_paths[MAXPATH];

h_line *direction;

union pathnode {
    int *length;
    struct node *node_ptr;
}

typedef struct hcoord {
    double x,y,z,w;
} h_line;

DESCRIPTION

This function converts a member's paths (areas) found by the path() function into tributary areas for a one way flooring system. The strong axis is indicated by the line equation direction.

RETURNS

struct area *ca_ptr -- pointer to the tributary area.

CAUTIONS

Does not return an error status if it fails.
NAME

path -- list all members in the structure

SYNOPSIS

path(n1, n2, adj1, paths)

struct node *n1, *n2;
adjinode *adj1[];
union pathnode *paths[][MAXPATH];

typedef listnode {
    struct node *node_ptr;
    struct listnode *next;
} adjinode;

union pathnode {
    int *length;
    struct node *node_ptr;
}

DESCRIPTION

This function attempts to find the two shortest legal closed paths between nodes n1 and n2 by employing a breadth first search traversing the adjacent list of the flooring system. In order to be a legal path, a new path must be on the opposite side of the member defined by n1 and n2.

RETURNS

2 -- two legal paths found
1 -- one legal path found
0 -- no legal path found

CAUTIONS

The shortest path is considered to be the path requiring the least number of edges to be traversed, physical length of the path is never calculated
NAME
print_loads -- list all loads on a member

SYNOPSIS

print_loads(member_ptr)

struct member *member_ptr;

DESCRIPTION

This function lists loads on a member.

RETURNS

Nothing

CAUTIONS

This function does not return an error status if it fails.
NAME

read_floorload -- read floor load (or floor opening) data from disk files

SYNOPSIS

read_floorload(file_ptr);

FILE *file_ptr;

DESCRIPTION

This function is called from get_flor() to read the floor load data file, and the opening data file.

RETURNS

struct fload *fload_head - pointer to head of load list

CAUTIONS

If the function is passed a pointer to an empty file, it will still return fload_head, even though there will be no loads in the list.
NAME
read_memberload -- read member load data from disk file

SYNOPSIS
read_memberload(file_ptr);

FILE *file_ptr;

DESCRIPTION
This function is called from get_flor() to read the member load data file.

RETURNS
Nothing

CAUTIONS
Function does not currently return an error status if it fails.
NAME

recurse -- perform recursive analysis of a member

SYNOPSIS

recurse(member_ptr);

struct member *member_ptr;

DESCRIPTION

This function loops through the nodes on a member, checking to see whether the end reactions of any members framing into the member are known (indicated by each member's solved flag). If a needed member has not been solved, recurse will call itself to solve that member's end reactions. Once the tributary member's end reaction is known, its loads are carried down by calling the get_reactions() function.

RETURNS

nothing

CAUTIONS

Function does not currently return an error status if it fails.
NAME

reset -- reset solved flag to NO (0) in all members

SYNOPSIS

reset()

DESCRIPTION

This function resets solved flags of all of the members in a floor to NO (0) to indicate that the floor will have to be reanalyzed. This routine is called any time that floor data is changed using the editor.

RETURNS

Nothing

CAUTIONS

This function does not return an error status if it fails.
NAME

reverse -- reverse the loads for a member which was assumed to be cantilevered from its beginning node, but is actually cantilevered from its end node.

SYNOPSIS

reverse(member_ptr, lc_ptr);

struct member *member_ptr;
struct load_case *lc_ptr;

DESCRIPTION

When recurse() performs load takedown, all loads are summed assuming that the member is a cantilever from its beginning node. This function is called by recurse if the member being analyzed is actually cantilevered from the other end. Reverse() adjusts the end shears and moments to the correct values.

RETURNS

Nothing

CAUTIONS

Function does not currently return an error status to indicate failure to solve reactions.
NAME

right_of -- determine the orientation of a point with respect to two other points

SYNOPSIS

double right_of(p1, p2, p3);

h_point *p1, *p2, *p3;

typedef struct h_coord {
    double x, y, z, w;
} h_line, h_point;

DESCRIPTION

This function determines the orientation of point p3 with respect to the directed line defined by points p1 and p2. The right_of() function calculates a checksum that is proportional to the area of the triangle formed by the three points. If the area is negative, p3 is to the right of line p1-p2, if the area is positive p3 is to the left of line p1-p2, if the area is zero, the three points are collinear.

RETURNS

checksum -- the calculated checksum

CAUTIONS

None
NAME

save_floor -- write floor data to disk files

SYNOPSIS

save_floor(f_number);
int f_number;

DESCRIPTION

This function writes files to save the data necessary to represent a floor.

RETURNS

Nothing

CAUTIONS

The function does not return an error status to indicate failure to save a floor. The floor number is currently ignored.
NAME

set_corner -- locate one corner of a tributary area

SYNOPSIS

set_corner(c1,c2,c3,c4,direction);

h_point *c1,*c2,*c3,*c4;
h_line *direction;

typedef struct hcoord {
    double x,y,z,w;
} h_line, h_point;

DESCRIPTION

This function locates the corner of a one way flooring system by making calls to intersect() and right_of().

RETURNS

h_point *p1 -- pointer to the desired corner

CAUTIONS

Function does not currently return an error status.
NAME

testadjl -- controller routine to manipulate adjacency list

SYNOPSIS

testadjl();

DESCRIPTION

This function is used as a controller to call get_adjlist() to read an adjacency list from a file for debugging, and to call memberlp() to perform operations on the adjacency list.

RETURNS

Nothing

CAUTIONS

This function does not return an error status if it fails.
NAME

test -- test a pair of paths found by path() to see if they are a legal pair

SYNOPSIS

test(paths);

union pathnode paths[2][MAXPATH];

union pathnode {
    int length;
    struct node *node_ptr;
}

MAXPATH -- compile time constant defined in header file "strall2.h"

DESCRIPTION

This function checks to see if a path found by the path() function is legal by comparing it to the first path and determining whether the two paths lie on opposite sides of the member. If the paths lie on opposite sides of the same member, one must traverse clockwise, the other counterclockwise. The actual checking is performed by calls to the right_of() function.

RETURNS

1 -- paths are legal (on opposite sides of the member)
0 -- paths are not legal

CAUTIONS

This function does not check to see whether a path crosses the axis of the member. This would result in an error, however such a situation is not likely to be encountered in a real building.
NAME
  to_global -- convert local (member) coordinates to global coordinates

SYNOPSIS

  double *to_global(member_ptr, local);

  struct member *member_ptr;

  double local[];

DESCRIPTION

  This function takes the local coordinates of a point on a member and converts them into global coordinates for the same point. Member coordinates start at zero at the beginning of the member and increase to 1 at the end of the member.

RETURNS

  local -- pointer to the array containing the converted coordinates

CAUTIONS

  This function does not return an error status if it fails.
NAME
to_lcptr -- convert a loadcase number to a loadcase pointer

SYNOPSIS

struct load_case *to_lcptr(lc_number, member_ptr)

int lc_number;
struct member *member_ptr;

struct load_case {
    int lc_number;
    struct reaction begin_reac;
    struct reaction end_reac;
    struct load *loads;
    struct load_case *next_lc;
}

DESCRIPTION

This function finds the pointer to the load_case structure with the desired load_case number. If the desired load_case is not found, one is created.

RETURNS

this lc -- pointer to the desired load_case

CAUTIONS

This function does not return an error status if it fails.
NAME

to_local -- convert global (member) coordinates to local coordinates

SYNOPSIS

double *to_local(member_ptr, global);

struct member *member_ptr;

double global[];

DESCRIPTION

This function takes the global coordinates of a point on a member and converts them into local (member) coordinates for the same point. Member coordinates start at zero at the beginning of the member and increase to 1 at the end of the member.

RETURNS

local -- pointer to the array containing the converted coordinates

CAUTIONS

This function does not return an error status if it fails.
NAME
  to_mnum -- convert a member pointer to a member number

SYNOPSIS

  struct member *to_mnum(member_pointer)

  struct member *member_pointer;

DESCRIPTION

  This function finds the member number of the member structure pointed to by member_ptr.

RETURNS

  -- number to the desired member if found

  NULL -- if the member was not found

CAUTIONS

  None
NAME
to_mptr -- convert a member number to a member pointer

SYNOPSIS

struct member *to_mptr(member_number);

int member_number;

DESCRIPTION

This function finds the pointer to the member structure with the desired member number.

RETURNS

this_member -- pointer to the desired member if found
NULL -- if the member was not found

CAUTIONS

None
NAME
  to_nnum -- convert a node pointer to a node number

SYNOPSIS

  struct node *to_nnum(node_number)
  struct node *node_number;

DESCRIPTION

  This function finds the node number of the node structure
  pointed to by node_ptr.

RETURNS

  -- number to the desired node if found
  NULL   -- if the node was not found

CAUTIONS

  None
NAME
to_nptr -- convert a node number to a node pointer

SYNOPSIS

struct node *to_nptr(node_number)

int node_number;

DESCRIPTION

This function finds the pointer to the node structure with the desired node number.

RETURNS

this_node -- pointer to the desired node if found
NULL -- if the node was not found

CAUTIONS

None
NAME

two_way -- converts a member's list of circuits to tributary areas

SYNOPSIS

two_way(member_ptr, all_paths);
struct member *member_ptr;
union pathnode *all_paths[MAXPATH];
union pathnode {
    int *length;
    struct node *node_ptr;
}

DESCRIPTION

This function converts a member's paths (areas) found by the path() function into tributary areas for a two way flooring system. The strong axis is indicated by the line equation direction.

RETURNS

struct area *area_head -- pointer to the tributary area.

CAUTIONS

Does not return an error status if it fails.
NAME

unique -- test a path found by path() for duplication of a previously found path.

SYNOPSIS

unique(all_paths, i);

union pathnode all_paths[MAXPATH];

int i;

union pathnode {
    int length;
    struct node *node_ptr;
}

MAXPATH -- compile time constant defined in header file "strail2.h"

DESCRIPTION

In the process of finding paths for physical members which are composed of more than one graph member, some paths will be duplicated. This duplication is eliminated by comparing the newest path(s) returned by path() with the paths that have already been returned and eliminating any duplications.

RETURNS

i -- index to the last valid path in allpaths

CAUTIONS

This function does not return an error status if it fails.
PROGRAM FLOW

adjlist()

loop over nodes

call find_neighbor()

loop through neighbors

call add_edge()

return
PROGRAM FLOW

analyze()

loop over members

has member been solved?  no  call recurse()
yes

return
PROGRAM FLOW

- carry_down()

- call m_to_hline()

- call add_ic()

- call add_load()

- if more than one point in area? then
  - yes
    - call length()
    - loop through corners of area
    - calculate member load
  - no
- return
PROGRAM FLOW

clip

loop on sides of area

call right_of()

intersection? no

yes

call intersect()

alter area

continue

return
PROGRAM FLOW

clip_openings

loop on
floor openings

loop on floor
load areas

call copy_harea() for clipping
copy of area

loop on floor
load corners

call clip()
call
filter_ha()

return
PROGRAM FLOW

clip_polygon

loop on member areas

loop on floor areas

call area format conversion
carea_to_harea

loop on corners

call clip()
call filter_ha()
call carry_down()

return
PROGRAM FLOW

find_opposite()

loop on members at node

is the member a cantilever?

yes

call node_to_coordinate hpoint()

conversion

call right_of()

no

is member colinear with the member being analyzed

yes

cantilever is opposite

no

return pointer to opposite

return NULL if not found

get_floor()

open files
call open()

read nodes

read members

read node_list file

read frame_in_list

read connectivity data

write

close files
call fclose()

return
PROGRAM FLOW

memberlp()

loop on members

are there nodes on this member? yes
no call path()

loop on pairs of nodes

call path()

check for duplicate call unique()
is twoway?

no

yes

call twoway()

is oneway?

call oneway()

return
PROGRAM FLOW

oneway()

---

call line() construct line along member axis

---

loop over paths

---

call alloc()

call node_to_hpoint()

call set_corner() set one corner of loaded area

call alloc()

call node_to_hpoint()

call harea_to_carea() Convert data format

return
was a path found?  

- yes
  - call filter()

- no break

construct path from que

- loop while searching for 2nd path

- back at start node?  
  - yes
  - no

- has node been visited?  
  - yes
  - no

- add node to que
was a path found?

yes

construct path from queue

call filter()

no

is the path legal?

no

return legal paths
PROGRAM FLOW

: recurse()

: loop on nodes
  on member

: loop on members
  at node

: member solved?

: call recurse()

: call
  get_reactions()
1

loop on member loads

call get_reactions()

balance moments

call adjust_for_cantilever()

return
PROGRAM FLOW

save_floor()

---
call open
fopen()

---

loop on nodes

---
write nodes

---
write connectivity
frame_in_list : data

---

loop on members

---
write members

---
write connectivity
node_list_file : data

---
call close files
fclose()

---
return
PROGRAM FLOW

<table>
<thead>
<tr>
<th>twoway()</th>
</tr>
</thead>
<tbody>
<tr>
<td>loop over paths</td>
</tr>
<tr>
<td>construct vectors along sides of path</td>
</tr>
<tr>
<td>construct bisectors at corners</td>
</tr>
</tbody>
</table>

| call allocate space malloc() : for a new corner |
| call find corners intersect() : of loaded area |

| is area a triangle? no |
| add new corner to area |

| call allocate space malloc() : for a new corner |

| return |
COMPUTER AIDED DESIGN OF
STRUCTURAL FLOORING SYSTEMS

Vol. 2

by

Albert Marshall Bleakley Jr.
B.S., United States Military Academy
(1977)

SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS OF THE
DEGREE OF
CIVIL ENGINEER

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

December 1984

© Massachusetts Institute of Technology 1984

Signature of Author: Albert Marshall Bleakley

Robert D. Logcher
Thesis Supervisor

Accepted by: Francois M. M. Morel
Chairman, Departmental Committee on Graduate Students

JUN 17 1985 Archives
APPENDIX C

Program Listings
/* this function adds an adjnode to an adjlist */

adjnode *add_adj()

{
    static int adj_count;
    adjnode *adj;
    static adjnode *adj_head, *adj_tail, *next_adj;
    char *calloc();

    adj = (adjnode*) calloc(1, sizeof(adjnode));  /* correct pointer cast */
    calloc(1, sizeof(adjnode));  /* & allocate memory */
    adj->next = NULL;

    return(adj);
}

/* The new adj is initialized to have NULL pointers */
/* This function is called add_edge. It adds edges to the adjacency list for a floor */

#include "strall2.h" /* separate compilation */

add_edge(n1, n2, adjlist)

struct node *n1, *n2;
adjnode *adjlist[1];

{
adjnode *adjnode1, *adjnode2, *current, *last;
adjnode *add_adj();

if(adjlist[n1->node_number] == NULL) {
    adjnode1 = adjlist[n1->node_number] = add_adj();
    adjnode1->node_ptr = n2;
}
if(adjlist[n2->node_number] == NULL) {
    adjnode2 = adjlist[n2->node_number] = add_adj();
    adjnode2->node_ptr = n1;
}

current = last = adjnode1 = adjlist[n1->node_number]; /* get to end of list */
while((current != NULL) && (current->node_ptr != n2)) {
    last = current;
    current = current->next;
}

if(current == NULL) {
    current = add_adj();
    last->next = current;
    current->node_ptr = n2;
}
current = last = adjnode2 = adjlist[n2->node_number]; /* get to end of list */
while((current != NULL) && (current->node_ptr != n1)) {
    last = current;
    current = current->next;
}
if(current == NULL) {
    current = add_adj();
    last->next = current;
    current->node_ptr = n1;
}

return;

/* This function assumes that if the node belonging to n2 is not in */
/* the n1 list, that the node belonging to n1 is not in n2's list. */
/* This is reasonable because this routine always creates new adjacency */
/* list entries in pairs: one for each end of the edge. */
#include "strall2.h"
#include <stdio.h>

struct frame_in_list *add_file(node_ptr) /* returns pointer to */
struct node *node_ptr; /* space for new file */
{
    static int fill_count;
    struct frame_in_list *fill;
    static struct frame_in_list *fil_head,*fil_tail;
    static struct node *current;
    char *malloc();

    fill = (struct frame_in_list *) /* correct pointer cast */
        malloc(sizeof(struct frame_in_list)); /* & allocate memory */
    fill_count++;
        /* increment # of files */

    if(node_ptr->frame_in == NULL) {
        node_ptr->frame_in = fill;
    }

    if(node_ptr != current :: fill_count == 1) { /* first for node? */
        fill_head = fill; /* beginning of list */
        fill_tail = fill; /* end of file list */
    } else {
        fill_tail->next = fill; /* set next fill ptr */
        fill_tail = fill; /* prev fill; update */
        /* end of file list */
    }

    current = node_ptr;
    fill->in_member = NULL; /* initialize: no member */
    fill->next = NULL; /* end of list */

    return(fill);
}
This function attaches a new load_case for a member

```c
#include "strall2.h" /* include only for */
#include <stdio.h> /* separate compilation */

struct load_case *add_lc(member_ptr) struct member *member_ptr;
{ static int lc_count;
  struct load_case *lc1; /* ptr to struct load_case */
  static struct load_case *this_lc, *lc_head, *lc_tail;
  static struct member *current;
  char *malloc();

  if(lc_count == 0)
    current = NULL;

  lc1 = (struct load_case*) malloc(sizeof(struct load_case)); /* & allocate memory */
  lc_count++; /* increment # of lcs */

  if (member_ptr != current) { /* different member? */
    this_lc = member_ptr->loads;
    if(this_lc == NULL) /* no loads yet */
      member_ptr->loads = lc1;
    else { /* find end of list */
      while(this_lc->next_lc != NULL)
        this_lc = this_lc->next_lc;
      this_lc->next_lc = lc1;
    }
  }
  lc_tail = lc1; /* end of lc list */

  else { /* add to list */
    lc_count = 0;
    lc_tail->next_lc = lc1;
    lc_tail = lc1; /* prev lc; update */
  }
  current = member_ptr;
  lc1->next_lc = NULL; /* end of list */

  return(lc1);
}
```

/* lc_head, lc_tail, are static, count is static */
/* The new lc is initialized to have NULL pointers */
/ * This function attaches a new load for a member

#include "strall2.h" /* include only for */
#include <stdio.h> /* separate compilation */

struct load *add_load(1c_ptr)
struct load_case *1c_ptr;
{
    static int load_count;
    struct load *load1;
    static struct load *this_load, *load_head,*load_tail;
    static struct load_case *current;
    char *calloc();

    if(load_count == 0)
        current = NULL;

    load1 = (struct load*)
calloc(1, sizeof(struct load)); /* & allocate memory */
    load_count++;

    if (lc_ptr != current) ( /* different load_case?*/
        this_load = lc_ptr->loads;
        if(this_load == NULL) {
        lc_ptr->loads = load1;
    }
    else { /* find list end*/
        while(this_load->next_load != NULL) /* list end*/
            this_load = this_load->next_load;
        this_load->next_load = load1;
    }
    load_tail = load1; /* end of load list */
}
else { /* add to list */
    load_count = 2;
    load_tail->next_load = load1; /* set next_load ptr of */
    load_tail = load1; /* prev load; update */
    current = lc_ptr;
    load1->next_load = NULL; /* end of list */
}

return(load1);

/* load_head, load_tail, are static, count is static */
/* The new load is initialised to have NULL pointers */
This function attaches a new member to the floor

#include "strail2.h"

struct member *add_member() { /* returns pointer to */
    /* space for new member */

    struct member *member1;
    extern struct member *member_head,*member_tail;
    static int member_count;
    char *calloc();

    member1 = (struct member *)               /* allocate memory */
        calloc(1,sizeof(struct member));      /* increment # of members */
    member_count++;

    if (member_count == 1) { /* first member? */
        member_head = member1;            /* beginning of list */
        member_tail = member1;            /* end of member list */
        member1->last = NULL;
    }
    else {
        member_tail->next = member1;       /* set forward and back */
        member1->last = member_tail;       /* pointers in the list */
        member_tail = member1;             /* update */
        member_tail->member_number = member_count;
        member_tail->next = NULL;          /* mark end of list */
        return(member1);
    }

    /* member_head, member_tail, and count must be static */
    /* function memb_init initializes the member */
    /* member numbers may be changed at any time, linkage of the */
    /* list will not be affected */
/* This function attaches a new node_list for a nl */

#include "strl12.h" /* standard header for functions*/

struct node_list *add_nl(member_ptr) struct member *member_ptr;
{
    static int nl_count;
    struct node_list *nl1;
    static struct node_list *nl_head,*nl_tail, *next_nl;
    static struct member *current;
    char *malloc();

    nl1 = (struct node_list*) malloc(sizeof(struct node_list)); /* allocate memory */
    nl_count++;

    if(member_ptr->frame_in == NULL) {
        member_ptr->frame_in = nl1;
    }
    if (member_ptr != current) ( /* first nl for memb? */
        nl_head = nl1; /* beginning of list */
        nl_tail = nl1; /* end of nl list */
    }
    else {
        nl_tail->next = nl1; /* set next_nl ptr */
        nl_tail = nl1; /* prev nl; update */
    }
    current = member_ptr;
    nl1->in_node = NULL; /* initialize: no member */
    nl1->next  = NULL; /* end of list */

    return(nl1);
}

/* nl_head, nl_tail, are extern, count is static
/* The new nl is initialized to have NULL pointers */
// This function attaches a new node to the floor

#include "strail2.h"
#include <stdio.h>

struct node *add_node() {
    /* returns pointer to */
    /* space for new node */

    struct node *node1;
    extern struct node *node_head, *node_tail;
    static int node_count;
    char *malloc();

    node1 = (struct node *)
    malloc(sizeof(struct node)); /* alloc memory */
    node_count++;
    /* increment # of nodes */

    if (node_count == 1) {
        /* first node? */
        node_head = node1;
        /* beginning of list */
        node_tail = node1;
        /* end of node list */
        node1->last_node = NULL; /* no last node */
    }
    else {
        node_tail->next_node = node1; /* set next_node ptr */
        node1->last_node = node_tail; /* set prev node */
        node1->node_number = node_count;
        /* update */
        node1->next_node = NULL; /* end of node list */
    }

    return(node1);
}

/*/ node_head, node_tail, are extern, count is static. */
/*/ function nodeinit() initializes the node */
/* This file is named adjlist.c. It creates an adjacency list for a */
/* floor. */

#include <stdio.h>
#include "strail2.h" /* must add local.h to strail2.h*/

struct node *neighbors[8];

adjnode *adj_list(adjlist) /* may need to allow for multiple*/
/* floors */
adjnode *adjlist[];

(

extern struct node *node_head;
extern struct member *member_head;
struct node *find_neighbors();
struct node
int i, j, e, v, x, y;

/* 0th element of the array is not used. This allows a correspondence */
/* between node numbers and array elements, e.g. node 15 will be found */
/* at adjlist[15]. */

current_node = node_head;

while(current_node != NULL) {

for(i=0; i<8; i++) /* array must be reset each */
    neighbors[i] = NULL; /* it is used again */

find_neighbors(current_node, neighbors);

i = 0;
while(neighbors[i] != NULL) {

    add_edge(current_node, neighbors[i], adjlist);
    i++;

}

current_node = current_node->next_node;
}

return(adjlist); /* returns pointer to adjlist*/
This function checks whether a cantilever frame is opposite to either end of the member being analyzed, adjusting the members end reactions as necessary.

```c
#include "strall2.h"
#include "stdio.h"

struct member *adjust_for_cantilever(member_ptr)
{
    struct member *member_ptr;
    int recurse();
    double carry_over, find_carry_over(), length();
    struct member *opposite;
    struct load_case *lc_ptr;
    if(member_ptr == NULL) {
        /* check for valid input */
    }
    lc_ptr = member_ptr->loads;

    while(lc_ptr != NULL) {
        /* do this twice: once for begin_node, once for end node */
        carry_over = find_carry_over(lc_ptr->lc_number, member_ptr, member_ptr->begin_node);

        if(carry_over != 0) {
            /* there is a cantilever */
            lc_ptr->begin_reac.moment = carry_over;
            lc_ptr->begin_reac.shear -=
                (carry_over / length(member_ptr));
            lc_ptr->end_reac.shear +=
                (carry_over / length(member_ptr));
            /* cw cantilever mom increases shear at begin, decreases at end*/
        }
        carry_over = 0;
        /* initialization */

        /* Repeat process for the end node */
        carry_over = find_carry_over(lc_ptr->lc_number, member_ptr, member_ptr->end_node);

        if(carry_over != 0) {
            /* there is a cantilever */
            lc_ptr->end_reac.moment = carry_over;
            lc_ptr->end_reac.shear -=
                (carry_over / length(member_ptr));
            lc_ptr->end_reac.shear +=
                (carry_over / length(member_ptr));
            /* cantilever increases shear at end, decreases at begin*/
        }
        lc_ptr = lc_ptr->next_lc;
    }
}
/* This function is named analyze(). It performs analysis of */
/* a floor using the recurse() function */

#include "stral12.h"
#include <stdio.h>

#define floor_ptr

analyze() /*(floor_ptr)*/ /* not implemented yet */

/* struct floor *floor_ptr */

{

extern struct member *member_head, *member_tail;
struct member *member_ptr;

member_ptr = member_head;
while(member_ptr != NULL) {
    if(member_ptr->flags.solved == NO) /* Has been solved?*/
        recurse(member_ptr);
    }
    member_ptr = member_ptr->next;
}
return;
}
/* This function is called carry_down() it takes the clipped */
/* intersection polygon found in clippoly() and projects the load onto */
/* the member to form member loads. */

#include <math.h>
#include "strall12.h"

carry_down(area_ptr, load_case, member_ptr, magnitude)

h_area *area_ptr;
int load_case;
struct member *member_ptr;
double magnitude;

{
    extern int excludge;
    int count = 0;
    double distance, l_dist, m_length, p_dist, x_dist, y_dist,
    fabs(), length(), line_distance();
    h_area *area_head, *closest_point, *next_closest;
    h_line *l_ptr, *m_to_hlin(), *line();
    struct load *last_load, *load_head, *load_ptr, *add_load();
    struct load_case *add lc(), *lc_ptr, *to lcptr();

    /* carrydown() must know whether this is a different loadcase. This */
    /* may require a change in floor loads to indicate which loadcase they */
    /* belong to. The loadcase information will have to be passed from */
    /* clip_polygon. The to lcptr() function requires the head of the */
    /* load cases for the proper member to be passed */

    /* Pre passing of load_case code, should no longer be necessary */
    if(member_ptr->loadcase == NULL) {
        lc_ptr = to lcptr(load_case, member_ptr);
    } else {
        lc_ptr = member_ptr->load;
    }

    /* New steps to ensure that the load gets put in the proper lc */
    lc_ptr = to lcptr(load_case, member_ptr);

    /* If a load case with the proper number did not exist before, one was */
    /* created by to lcptr() calling add lc(). From here on, there are no */
    /* no changes in carrydown() since it deals with lc_ptr. */
    load_ptr = load_head = add_load(lc_ptr);
    last_load = NULL;
if(area_ptr->next == area_ptr) {
    /* check for NULL area */
    load_ptr->begin_x = 0;
    load_ptr->end_x = 0;
    load_ptr->begin_mag = 0;
    load_ptr->end_mag = 0;
    return; /* return 0 load */
}

area_head = closest_point = next_closest = area_ptr;

m_length = length(member_ptr);

l_ptr = m_to_hlin(member_ptr);

/* insure that the carrydown starts at point nearest member */

while(area_ptr != area_head :: count++ (1) {
    if(fabs(line_distance(area_ptr->point, l_ptr)) (fabs(line_distance(closest_point->point, l_ptr))) {
        next_closest = closest_point;
        closest_point = area_ptr;
    } else if(fabs(line_distance(area_ptr->point, l_ptr)) (fabs(line_distance(next_closest->point, l_ptr))) {
        next_closest = area_ptr;
    }
    area_ptr = area_ptr->next;
})

/* Check for direction of traversal of the area to make sure */
/* that the edge nearest to the member is done last. */

if(next_closest == closest_point->next) {
    area_ptr = area_head = next_closest;
} else {
    area_ptr = area_head = closest_point;
}

count = 0;

/* stop after processing the last h_point in the area */
while(area_ptr != area_head :: count++ (1) {
    if(last_load == NULL) { /* first point processed*/
        l_dist = line_distance(area_ptr->point, l_ptr);
        x_dist = member_ptr->begin_node->x - area_ptr->point->x;
        y_dist = member_ptr->begin_node->y - area_ptr->point->y;
        p_dist = hypot(x_dist, y_dist);
        load_ptr->begin_mag = l_dist * magnitude;
        load_ptr->begin_x = sqrt(p_dist*p_dist - l_dist*l_dist) / m_length;
        /* local coord = 0 at beginning of member, */
        /* 1 at end of member */
    }
}
else {
    /* set new begin = last end */
    load_ptr->begin_mag = last_load->end_mag;
    load_ptr->begin_x = last_load->end_x;
}

area_ptr = area_ptr->next;

l_dist = line_distance(area_ptr->point, l_ptr);
x_dist = member_ptr->begin_node->x - area_ptr->point->x;
y_dist = member_ptr->begin_node->y - area_ptr->point->y;
p_dist = hypot(x_dist, y_dist);
load_ptr->end_mag = l_dist * magnitude;
load_ptr->end_x = sqrt(p_dist * p_dist - l_dist * l_dist) / m_length;

if(load_ptr->begin_x == load_ptr->end_x) {
    load_ptr = load_ptr;
}

if(area_ptr != area_head) {
    last_load = load_ptr;
    load_ptr = add_load(lc_ptr);
}

/* adjust_signs(load_head); */
/* make sure that superposition */
/* of loads is correct */

count = 0;

/* return nothing */

/* This function takes a pointer to a member and converts it into an */
/* h_line for use in the carry down routines. */

*/
*/
*
*/

/*h_point */
point1, point2; /* declared extern for passing */

h_line *m_to_hlin(member_ptr)

struct member  *member_ptr;
{
    char       *calloc();
    h_line     *line_ptr, *line();
    h_point    *p1, *p2, *node_to_hpoint();
    extern int  exkludge;

    line_ptr = (h_line *) calloc(1, sizeof(h_line));
p1 = (h_point *) calloc(1, sizeof(h_point));
p2 = (h_point *) calloc(1, sizeof(h_point));
p1 = node_to_hpoint(member_ptr->begin_node, p1);
p2 = node_to_hpoint(member_ptr->end_node, p2);

line_ptr = line(p1, p2, line_ptr);
return(line_ptr);

/* This function checks the load just created to make sure that the */
/* superposition of loads is done correctly. */

adjust_signs(load_head)

struct load *load_head;
{
  int flag = 0;
  double dx, lastdx;
  struct load *last_load, *load_ptr;
  extern int exkludge;

  load_ptr = load_head;
  last_load = NULL;
  lastdx = 0;

  while(load_ptr != NULL) {
    dx = load_ptr->end_x - load_ptr->begin_x;
    /* This test must fail except on the last time through */
    if(load_ptr->next_load == NULL)
      if(load_ptr->end_mag) /* reverse sign */
        load_ptr->end_mag *= -1;
    if(lastdx * dx > 0 && last_load != NULL) {
      last_load->begin_mag *= -1; /* reverse sign */
      last_load->end_mag *= -1; /* reverse sign */
    }
  }

  /* the first time through the loop, this test must fail */

  else if(lastdx * dx < 0 && flag == 0) { /* sign changed*/
    last_load->begin_mag *= -1; /* reverse sign */
    last_load->end_mag *= -1; /* reverse sign */
    flag++;
  }

  last_load = load_ptr;
  lastdx = dx;
  load_ptr = load_ptr->next_load;
}

/* return (nothing) */
}
/* Check to be sure that cantilevers are legal: they either */
/* end at a column, or there is an opposite member to take */
/* the moment at the connection. */

#include "strall.h"

check_cantilevers() {

extern struct member
struct member
struct node

*member_head, *find_opposite();
*member_ptr;
*node_ptr;

member_ptr = member_head;
while(member_ptr != NULL) {
    if(member_ptr->flags.cantilever) {
        node_ptr = member_ptr->begin_node;
        if(node_ptr->flags.cantilever) {
            node_ptr = member_ptr->end_node;
            if(node_ptr->on_column != 0) {
                find_opposite(member_ptr, node_ptr) != NULL);
                else {
                    printf("\ttError: No member opposite\n");
                    printf("\tt cantilever, member \%d\n", member_ptr->member_number);
                }
            }
            node_ptr = member_ptr->end_node;
            if(node_ptr->flags.cantilever) {
                node_ptr = member_ptr->begin_node;
                if(node_ptr->on_column != 0) {
                    find_opposite(member_ptr, node_ptr) != NULL);
                    else {
                        printf("\ttError: No member opposite\n");
                        printf("\tt cantilever, member \%d\n", member_ptr->member_number);
                    }
                }
            }
        }
    }
    member_ptr = member_ptr->next;
}
return;
/* This function clips an area against a directed line */

#include "strall2.h"

h_area *clip(a1, a2, area_ptr) /* a_ptr is the members area */
   /* a1 a2 are the floor load */
   /* side being clipped against */

   h_area *a1, *a2, *area_ptr;

   {
      /* should point1, point2 be extern? */

      int count; /* test quantities */
      double u1, u2, right_of();
      h_area *ma1, *ma2;
      h_point intersection, point1, point2, *i1, *p1, *p2, *p_head,
         *s1, *s2, *intersect();
      h_line line1, line2, *l1, *l2, *line();

      s1 = a1->point;
      s2 = a2->point;

      count = 0;
      ma1 = area_ptr;
      p1 = p_head = area_ptr->point;
      u1 = right_of(s1, s2, p1);
      area_ptr = area_ptr->next;
      p2 = area_ptr->point;

      while(p1 != p_head) : count++ ( 1 ) { /* stop after 1st time */
         u2 = right_of(s1, s2, p2);

         if(u1 > 0 && u2 > 0) {
            u1 = u1; /* do nothing */
            /* if both u1 and u2 are positive, do nothing */
            /* Need to consider case where the whole area */
            /* is outside of the floor load. */
         }
         else if(u1 <= 0 && u2 <= 0) {
            u1 = u1; /* do nothing */
            /* keep both vertices in temporary list */
         }
         else if(u1 <= 0 && u2 <= 0) {
            if(u1 < 0 && u2 > 0) {
               /* clip line intersects this side */
               l1 = line(p1, p2, &line1);
               l2 = line(s1, s2, &line2);
               i1 = intersect(l1, l2, &intersection);
               /* keep p1, insert i1 in the */
               /* temporary liet */
               hp_insert(ma1, i1);
            }
            else if(u1 > 0 && u2 < 0) {
               /* clip line intersects this side */
            }
         }
      }
l1 = line(p1, p2, &line1);
l2 = line(s1, s2, &line2);
i1 = intersect(l1, l2, &intersection);
/* keep p2, set p1 equal to i1 in the temporary list */
hp_insert(ma1, i1);
}
else if(u1 == 0) {
    u1 = 0;       /* do nothing */
    /* keep p1 and p2 in the temporary list */
}
else if(u2 == 0) {
    u2 = 0;       /* do nothing */
    /* keep p1 and p2 in the temporary list */
}
else {
    printf("\tError in clipping\n");
}

/*

if(il->w == 0) {
    printf("\tError in clip_poly: il->w == 0\n");
    return(NULL);
}

*/

u1 = u2;       /* swap p1 and p2 for next pass. */
p1 = p2;
ma1 = area_ptr;    /* remember last area */
area_ptr = area_ptr->next;    /* next corner in area */
p2 = area_ptr->point;

return(area_ptr);     /* since *area_ptr is a copy, it */
/* be manipulated by clip. */

/* There should be a clean up routine to free area_ptr s after they have */
/* been used. */
* This file is named clip_openings.c. The function clip_openings clips
* floor load polygons against any openings in the floor. Areas of
* intersection are assigned a magnitude equal to but negative of the
* floor load. This way, the superposition of the new load and the
* original will be zero load.
* See Pavlidis p.348. This function is used to clip an arbitrary
* (floor opening) polygon against a convex (floor load) polygon.
* This is a version of algorithm 15.3, however, this application
* clips using the end points of the sides of the floor loads and
* applying right_of(). Floor loads are defined to be in clockwise
* order so that a point that is inside of the loaded area will be to
* the right of the side of the floor load polygon.

#include "strall2.h"

clip_openings()
{
extern struct f_load *fload, *opening_head;

struct f_load
  *fload1, *new_fload, *opening;

h_area
  *f1, *f2, *fl_head, *o_harea,
  *clip(), *copy_harea(), *filter_ha();

h_point
  intersection, *corner1, *corner2,
  point1, point2, *i1, *p1, *p2, *p_head,
  *corner_head, *node_to_hpoint();

h_line
  line1, line2, *i1, *i2;

char
  *calloc();

int
  count, fl1_count, to_mnum();

opening = opening_head;

while(opening != NULL) { /* loop over members areas */
  fload1 = fload;
  while(fload1 != NULL) { /* loop over floor loads */

    /* make a new copy of the area */
    o_harea = copy_harea(opening->area);
    fl = fl_head = fload1->area; /* one area per floor load */

    while(fl != NULL) { /* set to NULL at end of while */
      corner1 = corner_head = fl->point;
      f2 = fl->next;
      count = 0;
      while((fl->point != corner_head) || count++) { /*
        clip(fl1, f2, o_harea);
      */
        o_harea =
        filter_ha(f1->point, f2->point, o_harea);
        /* if filter returns an area with one */
        /* or two points, there is no need to */
        /* continue the clipping. The memory */
        /* used by o_harea should be freed */
      }
    }
  }
}
}
if(o_harea->next == o_harea :
    o_harea->next->next == o_harea) {
    free(o_harea); /* should be cast */
    o_harea = NULL;
    break; /* single element list */
} /* only one point is */
/* on boundary will be */
f1 = f2; /* set to 0 in filter_ha*/
f2 = f2->next;
}
f1 = NULL; /* should be f1->next? */

/* figure area of clipped polygon and assign load to */
/* member here. Floor loads must be defined counter- */
/* clockwise so that their areas will be subtracted */
/* for calculation of reduction factors */

if(o_harea != NULL) {
    new_fload = (struct f_load *)
        calloc(1, sizeof(struct f_load));
    new_fload->name = fload1->name;
    new_fload->area = o_harea;
    new_fload->who_assigned = opening->who_assigned;
    new_fload->date_assigned = opening->date_assigned;
    new_fload->magnitude = -(fload1->magnitude);
    new_fload->next = fload1->next;
    fload1->next = new_fload;
    fload1 = new_fload->next;
} else {
    if(fload1 == NULL) {
        break;
    }
    fload1 = fload1->next;
}

opening = opening->next;

return;
This file is named clippoly.c. The function clip_polygon clips a polygon by a line defined in homogenous form: Ax - By + C = 0. See Pavlidis p. 340. This function is used to clip an arbitrary (member area) polygon against a convex (floor load) polygon. This is a version of algorithm 15.3, however, this application clips using the end points of the sides of the floor loads and applying right_of(). Floor loads are defined to be in clockwise order so that a point that is inside of the loaded area will be to the right of the side of the floor load polygon.

#include "strl12.h"

clip_polygon(loads, member_ptr)
{
    struct f_load *loads;
    struct member *member_ptr;

    struct f_load
    struct area
    h_area
    *m_area;
    *clip(), *filter_ha();
    h_point
    intersection, *load_corner1, *load_corner2,
    point1, point2, *fl1, *p1, *p2, *p_head,
    *load_corner_head, *node_to_hpoint();
    h_line
    line1, line2, *fl1, *fl2;
    int
    count, fl1_count, to_mnum();

    m_area = member_ptr->area;

    while(m_area != NULL) { /* loop over members areas */

        fl1 = loads;
        while(fl1 != NULL) { /* loop over floor loads */

            /* make a new copy, convert to h_area format*/
            haresa = carea_to_harea(m_area);
            fl1 = fl1->head = fl1->area;

            while(fl1 != NULL) { /* set to NULL at end of while */
                load_corner_head = fl1->point;
                f2 = fl1->next;
                count = 0;
                while((fl1->point != load_corner_head) || count++(1) {
                    clip(fl1, f2, haresa);

                    haresa = filter_ha((fl1->point, f2->point, haresa);
                    /* if filter returns an area with one */
                    /* or two points, there is no need to */
                    /* continue the clipping. */

                    if(harea->next == haresa) {
                        haresa->next->next == haresa) {
                            break; /* single element list */
/ * only one point is * /
/ * on boundary will be * /
    f1 = f2;       /* set to 0 in filter_ha */
    f2 = f2->next;

    / * Changed to pass a load name (loadcase name) */
    carry_down(harea, fload1->name, member_ptr,
               fload1->magnitude);
    f1 = NULL;      /* should be f1->next? */

    / * figure area of clipped polygon and assign load to */
    / * member here. Have not decided on a way of passing */
    / * the clipped area yet */

    fload1 = fload1->next;
    if(fload1 == NULL) {
        break;
    }

    m_area = m_area->next;
}
return;

/ * This function converts a corner based area representation into an */
/ * h_area/h_point based representation. */

#undef DEBUG

h_area *carea_to_harea(ca_ptr)
struct area    *ca_ptr;
{
    int          count = 0;
    char         *calloc();
    h_area       *harea_ptr, *h_head, *last_harea;
    h_point      *corner_to_hpoint(), *hp;
    struct corner  *corner_ptr, *c_head;

    corner_ptr = c_head = ca_ptr->corner;
    harea_ptr = h_head = (h_area *) calloc(1, sizeof(h_area));
    hp = (h_point *) calloc(1, sizeof(h_point));

    while(corner_ptr != c_head) count++ {
        harea_ptr->point = corner_to_hpoint(corner_ptr, hp);
    }
hp = (h_point *) calloc(1, sizeof(h_point));
harea_ptr->next = (h_area *) calloc(1, sizeof(h_area));
last_harea = harea_ptr;
harea_ptr = harea_ptr->next;
corner_ptr = corner_ptr->next;
}
last_harea->next = h_head;
return(h_head);

/* This function converts a corner representation of a point into an
   * h_point representation (homogeneous point)

h_point *corner_to_hpoint(c1, h1)

struct corner *c1;
struct h_point *h1;
{
    h1->x = c1->x;
    h1->y = c1->y;
    h1->z = c1->z;
    h1->w = 1;
    return(h1);
}
/* Function to duplicate an h_area data structure. */

#include "strall2.h"
#undef DEBUG

h_area *copy_harea(ha_ptr)

h_area *ha_ptr;
{
    int i = 0;
    char *calloc();
    h_area *current, *harea_head, *new_harea;

    current = ha_ptr;

    new_harea = harea_head = (h_area *) calloc(1, sizeof(h_area));
    new_harea->point = (h_point *) calloc(1, sizeof(h_point));

    while(current != ha_ptr; i++; { i++) {
        hp_equate(new_harea->point, current->point);
        current = current->next;
        if(current == ha_ptr) {
            new_harea->next = harea_head;
        } else {
            new_harea->next = (h_area *)
                calloc(1, sizeof(h_area));
            new_harea = new_harea->next;
            new_harea->point = (h_point *)
                calloc(1, sizeof(h_point));
        }
    }

    return(harea_head);
}
/* This function takes three node pointers and determines whether they */
/* form a corner. Used in checking paths for graph manipulation. */

#include <math.h>
#include <stdio.h>
#include "string.h"

int exclude; /* used for debugging */

corner(n1, n2, n3)
{
    struct node *n1, *n2, *n3;

    extern int exclude;

double fabs(), flag, right_of();

h_point p1, p2, p3;

    if(n1 == NULL || n2 == NULL || n3 == NULL) {
        return(-1);
    }

    p1.x = n1->x;
    p1.y = n1->y;
    p1.w = 1;

    p2.x = n2->x;
    p2.y = n2->y;
    p2.w = 1;

    p3.x = n3->x;
    p3.y = n3->y;
    p3.w = 1;

    flag = right_of(&p1, &p2, &p3);

    if(fabs(flag) < .5) {
        /* the points are colinear */
        /* allow tolerance for round off */
        return(0);
    }

    else {
        /* the points are not colinear */
        /* so they must be a corner */
        return(1);
    }
}
/* This function edits nodes in a linked list of nodes. */
#include "strall2.h" /* standard header for functions*/
edit_node(node_ptr)
struct node *node_ptr
{
    char coordinates[5];
    char answer[5];
    int stop = 0;
    int member_number, how_many;
    int to_nnum(), to_mnum();
    struct node *node_ptr;
    struct node *add_node(), *to_nptr();
    struct member *memb_ptr;
    struct member *add_member(), *to_mptr();
    struct frame_in_list *add_fill(), *fill_ptr;

    while(stop != 1) {
        node_ptr = add_node();
        node_init(node_ptr);
        printf("Do you want to change the member this node is on?
(yes or no)?\n\n");
        scanf("%s", &more);
        if(tolower(more[0]) == 'y') {
            printf("Enter the member that this node is on\n");
            printf("member\% or 0 if a column\n");
            scanf("%d", &member_number);
            if(member_number != 0)
                node_ptr->on_member = to_mptr(member_number);
            else
                node_ptr->on_member = NULL;
        }
        printf("Do you want to change the node coordinates (yes or no)?\n\n");
        scanf("%s", &more);
        if(tolower(more[0]) == 'y') {
            printf("Enter 'g' for global or 'l' for local coordinates\n");
            scanf("%is", &coordinates); /* does nothing */

            printf("Enter coordinates x,y,z\n");
            scanf("%if %lf", &node_ptr->x, &node_ptr->y, &node_ptr->z);
        }
        printf("Do you want to change members framing in (yes or no)?\n\n");
        scanf("%s", &more);
        if(tolower(more[0]) == 'y') {
            printf("How many members frame into this node?\n");
            scanf("%d", &how_many);
            while(how_many-- > 0) {
                fill_ptr = add_fill(node_ptr);
                printf("Enter the member number that frames in\n");
                scanf("%d", &member_number);
fil_ptr->in_member = to_mptr(member_number);
}

printf("Enter 1 to stop, any other number to continue\n");
scanf("%d", &stop);
printf("scanf read response as %d\n", stop);

return;
/* This file is named filter.c. The function filter() removes non */
/* corner nodes from a path found by the path() search algorithm */

#include "strail2.h"

#define np node_ptr

filter(path1)

union pathnode path1[];
{
   int flag, i, k, length;
   i = k = length = 0;

   length = path1[0].length;

   while(++i <= length - 2) {
      flag = corner(path1[i].np, path1[i+1].np, path1[i+2].np);
      while(flag == 0) {
         for(k=i+1; k<=length; k++) {
            path1[k].np = path1[k+1].np;
         }
         length--;
         if(i > length - 2)
            break;
         flag = corner(path1[i].np, path1[i+1].np, path1[i+2].np);
      }
      if(length < 3) {
         length = 0;
         break;
      }
   }
   if(length < 3) {
      length = 0;
   }
   else {
      flag = corner(path1[1].np, path1[2].np, path1[length].np);
      while(flag == 0 && length > 3) {
         for(k=1; k<length; k++) {
            path1[k].np = path1[k+1].np;
         }
         length--;
         /* "rotate" list if colinear */
         flag = corner(path1[1].np, path1[2].np, path1[length].np);
      }
      if(length < 3) {
         length = 0;
      }
   }
}


```c
length = 0;
else {
    flag = corner(pathi[1].np, pathi[length-1].np, pathi[length].np);
    while(flag == 0 && length > 3) {
        length--; /* shorten path if colinear */
        flag = corner(pathi[1].np, pathi[length-1].np, pathi[length].np);
    }
}
if(length < 3) {
    length = 0;
}
pathi[0].length = length;
return;
```
/** This function filters a homogeneous area after clipping to remove */
/** points that are outside of the loaded area */

#include "strail2.h"

h_area *filter_ha(p1, p2, ha_ptr)

h_point *p1, *p2;

h_area *ha_ptr; /* side used for clipping */

h_area *area_head; /* loaded area being clipped */

{
    int count = 0;
    double right_of();

    area_head = ha_ptr;

    while(count++ < 1 : ha_ptr != area_head) {
        while(right_of(p1, p2, ha_ptr->next->point) > 0
              && ha_ptr != ha_ptr->next) {

            hp_delete(ha_ptr); /* deletes ha_ptr->next */
            area_head = ha_ptr; /* update head of list */
            count = 0; /* new anchor for search*/
        }
    ha_ptr = ha_ptr->next;
}

/* Check to see if the list has only two points. If so, */
/* remove one of them and return a list with a single point. */
/* This one element list will be treated as a special case in */
/* carrydown() so that a member load will not be created. */
/* A one or two element list can occur when a members tributary */
/* area lies outside of the floor load, but shares a common */
/* boundary. */

if(ha_ptr->next != ha_ptr && ha_ptr->next->next == ha_ptr) {
  hp_delete(ha_ptr);
}

return(area_head);
}
/* This function checks whether a cantilever frame is in opposite to either end of the member being analyzed. If so, the cantilever is checked to see if it has been solved. If not, recurse() is called to solve it, then the cantilever end moment is returned. */

#include "strall2.h"
#include (stdio.h)

double find_carry_over(ic_number, member_ptr, node_ptr)
{
int ic_number;
struct member *member_ptr;
struct node *node_ptr;

int recurse();
struct member *find_opposite(), *opposite;
struct load_case *lc_ptr, *to_lcptr();

if(member_ptr == NULL || node_ptr == NULL) {
  /* check for valid input */
  return(NULL);
}

opposite = NULL; /* initialization */

opposite = find_opposite(member_ptr, node_ptr);

if(opposite != NULL) {
  if(opposite->flags.solved == NO) {
    recurse(opposite);
  }
  lc_ptr = to_lcptr(ic_number, opposite);
  if(opposite->begin_node == node_ptr) {
    return(-lc_ptr->begin_reac.moment);
  }
  else {
    return(-lc_ptr->end_reac.moment);
  }
}

else {
  return(0);
}
}
/* This function finds the next neighbor for a node */
/* At present, I still need a way of defining "next neighbor". */
/* Perhaps what this function should do is return an array of neighbors */
/* and thus be called only once per node. */

#include "strall2.h" /* standard header for functions*/

struct node *find_neighbors(node_ptr, neighbors)

struct node *node_ptr;
struct node *neighbors[];

{
struct frame_in_list *fil_ptr;
struct node_list *nl_ptr;
struct member *current_member;
struct node *node1, *node2;
int i, j;

/* this portion of the function treats the case of nodes on a member */
/* that terminates or begins at *node_ptr. */

j = 0;

for(i=0; i<8; i++)
    neighbors[i] = NULL;
fil_ptr = node_ptr->frame_in;
while(fil_ptr != NULL) {
    /* The following routines treat the case where *node_ptr*/
    /* is one end of a member that has only 2 end nodes. */
    current_member = fil_ptr->in_member;
    if(current_member->frame_in == NULL) {
        if(node_ptr == current_member->begin_node) {
            /* neighbors[j++] places number in next */
            /* location of array neighbor. */
            neighbors[j++] = current_member->end_node;
        }
        else {
            neighbors[j++] = current_member->begin_node;
        }
    }
    /* next section covers case of members that frame in */
    /* to the node but have nodes on them. These routines */
    /* require that the nodelist belonging to a mem- */
    /* ber be in order from nearest to beginning of the */
    /* member to nearest the end of the member. */
    else if(node_ptr == current_member->begin_node) {
        neighbors[j++] = current_member->frame_in->in_node;
    }
    else if(node_ptr == current_member->end_node) {
        nl_ptr = current_member->frame_in;
        while(nl_ptr->next != NULL) {
            nl_ptr = nl_ptr->next;
        }
        neighbors[j++] = nl_ptr->in_node;
    }
}
fill_ptr = fill_ptr->next;

/* next set of routines required must check the member that the */
/* node is on to find its neighbors along the member. */
if(node_ptr->on_member != NULL) {  /* is it on a member */
  nl_ptr = node_ptr->on_member->frame_in;
  node1 = nl_ptr->in_node;
  node2 = NULL;
  while(nl_ptr != NULL && node1 != node_ptr) {
    node2 = node1;
    node1 = nl_ptr->in_node;
    nl_ptr = nl_ptr->next;
  }
  if(node2 != NULL)
    neighbors[j++] = node2;
  if(nl_ptr != NULL & nl_ptr->next != NULL) {
    nl_ptr = nl_ptr->next;
    neighbors[j++] = nl_ptr->in_node;
  }
}
return(neighbors);  /* returns pointer to array of */
/* neighbor node pointers. */

print_neighbors(neighbors)

struct node *neighbors[];

{
  int i;

  printf("i\tnumerals[i] to nnum(neighbors[i])\n");

  for(i=0; i<9; i++) {
    printf("%d\t", i, neighbors[i]);
    if(neighbors[i] != NULL)
      printf("%d", to_nnum(neighbors[i]));
    printf("\n");
  }
}
/* This function checks whether a cantilever frame is on the opposite */
/* side of a node from the member being analyzed. */

#include "strall2.h"
#include (stdio.h)

struct member *find_opposite(member_ptr, node_ptr)

struct member *member_ptr;
struct node *node_ptr;

{

double right_of();
struct member *memb_ptr, *opposite;
struct node *n1, *n2;
struct frame_in_list *fil_ptr;
h_point p1, p2, p3, p4, *begin, *end, *tb, *te,
*node_to_hpoint();

if(member_ptr == NULL && node_ptr == NULL) {
    return(NULL); /* check for valid input */
}

if(node_ptr->on_member == NULL) {
    return(NULL); /* if this is a column */
    /* there is no carry- */
    /* over */

    opposite = NULL; /* initialization */
    n1 = member_ptr->begin_node; /* check for opposite member */
    n2 = member_ptr->end_node;

    begin = node_to_hpoint(n1, &p1);
    end = node_to_hpoint(n2, &p2);

    fil_ptr = node_ptr->frame_in;
    while(fil_ptr != NULL && opposite == NULL) {
        memb_ptr = fil_ptr->in_member;

        if(memb_ptr != member_ptr) { /* a different member? */

            if(memb_ptr->flags.cantilever == YES) {
                tb = node_to_hpoint(memb_ptr->begin_node,&p3);
                te = node_to_hpoint(memb_ptr->end_node,&p4);

                if((right_of(tb, begin, end) == 0)
                    && (right_of(te, begin, end) == 0)) {
                    opposite = memb_ptr;
                }
            }
            fil_ptr = fil_ptr->next;
        }
    }

    return(opposite);
}
} /* The find_opposite() function will return a NULL pointer if an */ /* opposite member is not found at the specified node. To increase */ /* the generality of the algorithm, a tolerance for colinearity */ /* could be added. */
/* This function prints the loads in a results file. */

#include "strall12.h"
#include <stdio.h>

fprint_loads(member_ptr)

struct member *member_ptr;
{
    FILE *res_file, *fopen();
    int count = 0;
    struct load_case *loadcase;
    struct load *load;

    res_file = fopen("mloads.out","w");
    if(res_file == NULL) {
        printf("Could not open \mloads.out\n");
    }

    fprintf(res_file,"loads for member %d\n", member_ptr->member_number);
    loadcase = member_ptr->loads;
    while(loadcase != NULL) {
        fprintf(res_file,"\tloadcase %d %d\n", ++count, loadcase);
        fprintf(res_file,"\t\tbegint_x\tend_x\tbegint_mag\tend_mag\n" );
        load = loadcase->loads;
        while(load != NULL) {
            fprintf(res_file,"\t\u", load);
            fprintf(res_file,"\t%6.2f\t%6.2f\t",
                load->begin_x, load->end_x);
            fprintf(res_file,"%6.2f\t%6.2f\n",
                load->begin_mag, load->end_mag);
            load = load->next_load;
        }
        loadcase = loadcase->next_lc;
    }

    return;
}
/* This function is called list_reactions(). It prints out a list */
/* of all member reactions by load_case. */

flist_reactions()
{

FILE *reac_file, *fopen();
extern struct member *member_head;
struct member *member_ptr;
struct load_case *lc_ptr;

reac_file = fopen("results.out", "w");
if(reac_file == NULL)
{
    printf("Could not open (reactions.out)\n");
}

fprintf(reac_file, "Member 0
load case 1
load case 2\n");
fprintf(reac_file, "Vbeg Mbeg Vend Mend Vbeg Mbeg Vend\n");

member_ptr = member_head;
while(member_ptr != NULL) {
    lc_ptr = member_ptr->loads;
    fprintf(reac_file, "%d ", member_ptr->member_number);
    while(lc_ptr != NULL) {
        fprintf(reac_file, "%.2lf %.2lf %.2lf %.2lf ",
            lc_ptr->begin_reac.shear, lc_ptr->begin_reac.moment,
            lc_ptr->end_reac.shear, lc_ptr->end_reac.moment);
        lc_ptr = lc_ptr->next_lc;
    }
    fprintf(reac_file, "\n");
    member_ptr = member_ptr->next;
}
}
/* This function finds the area of a polygon defined as a circular list of h_points. See CRC p.169 for algorithm. */

#include "stat12.h"
#include <math.h>

double get_area(h_area *area_ptr)
{
    int count = 0;
    double area = 0;
    h_point *p1, *p2, *p_head;

    p1 = p_head = area_ptr->point;
    area_ptr = area_ptr->next;
    p2 = area_ptr->point;

    while(p1 != p_head || count++ < 1)
    {
        area = p1->x * p2->y - p2->x * p1->y;
        p1 = p2;
        area_ptr = area_ptr->next;
        p2 = area_ptr->point;
    }

    area /= 2; /* divide in half for right answer */

    return(area);
}
/* This file is named (GET_FLOOR.C) The get_floor() function gets a floor structure from disk files. */

#include "strall2.h"
#include "stdio.h"

#define READFLOORLOADS
#define READMEMBERLOADS

get_floor(f_number)

int f_number;
{
    char c;
    int
    begin, begin_node, cantilever, count, end, end_node,
    fiction, last, lc_number, member_number,
    next, node_number, on_member, on_column,
    rx, ry, rz, solved, type, x, y, z;
    shear, moment, axial;
    position;
    *load, *opening_head;
    *node_head, *node_ptr;
    *member_head;
    *last_node, *next_node, *node_ptr;
    *member_ptr, *memb_init();
    *fil_ptr;
    *nl_ptr;
    *lc_ptr;  /* * * */
    *load_ptr;
    FILE *filf_ptr, *filf_ptr, *lcf_ptr, /* files for input */
    *lodf_ptr, *memf_ptr, *nlf_ptr,
    *nodf_ptr, *opf_ptr;
    FILE
    *fopen();  /* func returns file ptr*/
    int
    to_nnum();  /* convert ptr to memb*/
    int
    to_nnum();  /* convert ptr to node*/
    struct node
    *to_mpnt(), *add_node(); /* memory allocate func */
    struct member
    *add_member(), *to_mpnt(); /* */
    struct frame_in_list
    *add_fil();  /* */
    struct node_list
    *add_nl();  /* */
    struct load
    *add_load();  /* */
    struct load_case
    *add_ic(), *to_lcptr();  /* */
    struct f_load
    *read_floorload();  /* files for output */

    nodf_ptr = fopen("node_list.dat","r");  /* files for output */
    memf_ptr = fopen("mem_list.dat","r");
    filf_ptr = fopen("fil_list.dat","r");
    nlf_ptr = fopen("nl_list.dat","r");
    lodf_ptr = fopen("load_list.dat","r");

    /* lcf_ptr = fopen("lc_list.dat","r"); */
    filf_ptr = fopen("floorload.dat","r");
    opf_ptr = fopen("opening.dat","r");
if (nodf_ptr == NULL) { /* were files read? */
    printf("Could not read \<NODELIST DAT\>\n");
    exit(nodf_ptr);
}
if (memf_ptr == NULL) {
    printf("Could not read \<MEMBLIST.DAT\>\n");
    exit(memf_ptr);
}
if (filf_ptr == NULL) {
    printf("Could not read \<FIL_LIST.DAT\>\n");
    exit(filf_ptr);
}
if (nlf_ptr == NULL) {
    printf("Could not read \<NL_LIST.DAT\>\n");
    exit(nlf_ptr);
}
if (lodf_ptr == NULL) {
    printf("Could not read \<LOADLIST.DAT\>\n");
    exit(lodf_ptr);
}
/*
if (lcf_ptr == NULL) {
    printf("Could not read \<LOADCASE.DAT\>\n");
    exit(lcf_ptr);
}
*/
if (filf_ptr == NULL) {
    printf("Could not read \<FLOORLOAD.DAT\>\n");  /*exit(filf_ptr);*/
}
if (opf_ptr == NULL) {
    printf("Could not read \<OPENING.DAT\>\n");  /*exit(opf_ptr);*/
}
/*
First pass -- set up linked lists to receive data
*/
end = count = 0;
while (end != EOF) { /* Create node list */
    if (count++ == 0)
        fscanf(nodf_ptr, "%d", &node_number);
    node_ptr = to_nptr(node_number);
    node_ptr->node_number = node_number;
    fscanf(nodf_ptr, "%*d %*d %*d %*d %*d %*d %*d %*d");
    fscanf(nodf_ptr, "%*d %*d %*d %*d %*d %*d %*d %*d");
    end = fscanf(nodf_ptr, "%d", &node_number);
}
end = count = 0;

while (end != EOF) { /* create member list */
    if (count++ == 0)
        fscanf(memf_ptr, "%d", &member_number);
    member_ptr = to_nptr(member_number);
    member_ptr->member_number = member_number;
fscanf(memf_ptr,"%*id %*id %*id" );
fscanf(memf_ptr,"%d %*d" );
end = fscanf(memf_ptr,"%d", &member_number);  /* EOF? */
}

/* list_members(); */
rewind(nodf_ptr);  /* reset to beginning */
rewind(memf_ptr);  /* of files */

/* Second pass -- read data into list */

last_node = next_node = NULL;
end = count = 0;
while (end != EOF) {
  if (count++ == 0)  /* first time through? */
    fscanf(nodf_ptr,"%d", &node_number);
  if (last_node == NULL) {
    node_ptr = to_nptr(node_number);
  } else {
    last_node->next_node = node_ptr = to_nptr(node_number);
  }
  node_ptr->node_number = node_number;
  fscanf(nodf_ptr,"%If %If %If", &node_ptr->x, &node_ptr->y, &node_ptr->z);
  fscanf(nodf_ptr,"%d %d %d %d %d %d %d %d %d",
         &type, &x, &y, &z, &rx, &ry, &rx, &cantilever);
  node_ptr->flags.node_type = type;  /* This approach is */
  node_ptr->flags.fix_x = x;  /* necessary because */
  node_ptr->flags.fix_y = y;  /* the flags are bit */
  node_ptr->flags.fix_z = z;  /* fields, and therefore*/
  node_ptr->flags.fix_rx=rx;  /* do not have addresses*/
  node_ptr->flags.fix_ry=ry;  /* e.g. fscanf(...,",%u",*/
  node_ptr->flags.fix_rz=rz;  /* &node_ptr->flags.fix */
  node_ptr->flags.cantilever = cantilever;  /* *

  fscanf(nodf_ptr,"%d %d %d %d", &on_member, &on_column);  */
  fscanf(nodf_ptr,"%d %d", &on_member, &on_column);

  /*
  node_ptr->next_node = to_nptr(next_node);
  node_ptr->last_node = last_node;
  last_node = node_ptr;
  node_ptr->on_member = to_mptra(on_member);
  node_ptr->on_column = node_ptr;  /* not implemented */
  end = fscanf(nodf_ptr,"%d", &node_number);  /* more nodes? */
} */

end = count = 0;
while (end != EOF) {  /* read member data */
  if (count++ == 0)
    fscanf(memf_ptr,"%d", &member_number);
  member_ptr = to_mptra(member_number);
  fscanf(memf_ptr,"%id %id %id", &solved, &fiction, &cantilever);
  member_ptr->flags.solved = solved;
  member_ptr->flags.fiction = fiction;
  member_ptr->flags.cantilever = cantilever;
```c
/*
 * fscanf(memf_ptr,"%s d", &member_ptr->section->table,
 *        &member_ptr->section->index);
 *
 * fscanf(memf_ptr,"%d %d ", &begin_node,&end_node);
 * member_ptr->begin_node = to_nptr(begin_node);
 * member_ptr->end_node   = to_nptr(end_node);
 * member_ptr->end_node   = to_nptr(end_node);
 * end = fscanf(memf_ptr,"%d", &member_number);    /* EOF? */
 */

}

read_memberload(lodf_ptr);

floadr = read_floorload(flfr_ptr);

opening_head = read_floorload(opf_ptr);

end = count = 0;
member_ptr = member_head;
while(end != EOF) {
    /* read node_list file*/
    if(count++ == 0)
        fscanf(nlf_ptr,"%d", &member_number);
    fscanf(nlf_ptr,"%d", &node_number);
    member_ptr = to_mptr(member_number);
    n1_ptr = add_n1(member_ptr);
    n1_ptr->in_node = to_nptr(node_number);

    end = fscanf(nlf_ptr,"%d", &member_number);    /* EOF? */
}

end = count = 0;
nodestr = node_head;
while(end != EOF) {
    /* read fil_list file*/
    if(count++ == 0)
        fscanf(filf_ptr,"%d", &node_number);
    fscanf(filf_ptr,"%d", &member_number);

    node_ptr = to_nptr(node_number);
    fil_ptr = add_fil(node_ptr);
    fil_ptr->in_member = to_mptr(member_number);

    end = fscanf(filf_ptr,"%d", &node_number);    /* EOF? */
}

/* Close everything and return */
```
fclose(nodf_ptr);
fclose(memf_ptr);
fclose(lodf_ptr);
fclose(nif_ptr);

/*
 * fclose(lcf_ptr);
 */
fclose(lcdf_ptr);
fclose(flf_ptr);
if(flf_ptr != NULL)
    fclose(flf_ptr);
if(opf_ptr != NULL)
    fclose(opf_ptr);

return;
}
/* This function finds the next load on a member */

#include "strail2.h" /* standard header for functions*/

struct load *get_load(lc_ptr)
struct load_case *lc_ptr;

/* find all loads on the member being analyzed */
{
    int i, j;
    static int count;
    static struct load *current_load;

    if(lc_ptr == NULL || lc_ptr->loads == NULL)
        return(NULL);

    if (count++ == 0) { /* first time function called */
        current_load = lc_ptr->loads;
    }

    else if (current_load->next_load != NULL) { /* there are more loads */
        current_load = current_load->next_load;
        count++;
    }

    else { /* there are no more loads */
        current_load = current_load->next_load;
        count = 0;
    }

    return(current_load); /* pointer to next load. Null if* /
    /* there isn't one */
}
/* This routine finds the reactions on a member with all */
/* loads on the member known */

#include "stra12.h"
#include <stdio.h>
#include <math.h>

#define total_load lc_ptr->begin_reac.shear
#define sum_moment lc_ptr->begin_reac.moment

get_reactions(lc_ptr, p1, p2, x1, x2) /* multiple load cases */

struct load_case *lc_ptr;
double p1;    /* begin magnitude */
double p2;    /* end   magnitude */
double x1;    /* begin position */
double x2;    /* end   position */

{
    double fabs();  /* floating point absolute value*/

    /* define local x coordinate along long axis of beam */
    /* clockwise moment is positive */
    if((p1 == p2) && (x1 == x2)) /* point load? */
        total_load += p1;
        sum_moment -= p1 * x1;

    }
    else /* distributed load */
        total_load += (p1 + p2)/2 * (x2 - x1);
        sum_moment -= (p1 * (x2 - x1) * (x1 + (x2 - x1)/2)
            + .5 * (p2-p1) * (x2-x1) * (x1 + (x2-x1) * 2/3));

    return;
}

/* This function deletes a homogeneous point from a homogeneous */
/* area (linked list). */

#include "stral12.h"

hp_delete(ha)

h_area *ha;
    /* deletes the h_area *after* */
    /* ha */

{ // condemn_harea;

    if(ha->next != ha) ( // make sure there is more than */
        /* one element in the list */
        condemned_harea = ha->next;
        ha->next = condemned_harea->next;

        /* free(condemned_harea->point); */ // clean up
        /* free(condemned_harea); */ // memory
    }

    return;
}
/** This file is named hp_equate.c. This function sets one h_point equal to a second. In some versions of C, structures cannot be directly assigned, e.g. structure1 = structure2 is not legal. */

#include "strall2.h"

hp_equate(p1, p2)

h_point *p1, *p2;
{
    p1->x = p2->x;
    p1->y = p2->y;
    p1->z = p2->z;
    p1->w = p2->w;
    return;
}
/* This function inserts a new homogeneous point into a homogeneous */
/* area (linked list). */

#include "strall2.h"

hp_insert(ha, p2)

h_area *ha;
h_point *p2;
{
char *calloc();

h_area *new_area;

new_area = (h_area *) calloc(1, sizeof(h_area));

new_area->point = (h_point *) calloc(1, sizeof(h_point));

hp_equate(new_area->point, p2); /* set new point's coordinates */

new_area->next = ha->next; /* adjust pointers in list */

ha->next = new_area;

return;
}
This function takes input for nodes and creates a linked list.

```c
#include "strall2.h"

input_node() {
    char coordinates[51];
    char answer[51];
    int stop = 0;
    int count = 0;
    int member_number, how_many;
    int to_nnum(), to_mnum();
    struct node *node_ptr;
    struct node *add_node(), *to_nptr();
    struct member *memb_ptr;
    struct member *add_memb(), *to_mptr();
    struct frame_in_list *add_fil(), *fil_ptr;

    fgets("Rotation and translation end conditions are initialized\n",);
    fgets("as follows: no trans x,y,z; rotation allowed about x,y,z\n",);
    fgets("There is currently no way to change these defaults.\n",);
    while(stop != 1) {
        node_ptr = add_node();
        node_init(node_ptr);
        printf("Enter the member that this node is on (0 or 0 if a column)\n");
        scanf("%d", &member_number);
        if (member_number != 0)
            node_ptr->on_member = to_mptr(member_number);
        else
            node_ptr->on_member = NULL;

        printf("Enter 'g' for global or 'l' for local coordinates\n",);
        scanf("%s", coordinates); /* does nothing */

        printf("Enter coordinates x,y,z\n");
        scanf("%f %f %f", &node_ptr->x, &node_ptr->y, &node_ptr->z);
        printf("How many members frame into this node?\n");
        scanf("%d", &how_many);
        while(how_many-- > 0) {
            fil_ptr = add_fil(node_ptr);
            if(count++ == 0)
                node_ptr->frame_in = fil_ptr;
            printf("Enter the member number that frames in\n");
            scanf("%d", &member_number);
            fil_ptr->in_member = to_mptr(member_number);
        }
        count = 0;
        printf("Enter 1 to stop, any other number to continue\n");
        scanf("%d", &stop);
        printf("scanf read response as \%d\", stop);
    }
    return;
}
```

/* This file is called intersec.c. The function intersect() takes the */
/* homogeneous representation for two lines and returns their point of */
/* intersection. */

#include "strail2.h"

h_point *intersect(11, 12, point)

h_line *11, *12;               /* currently assumes 2d graphics*/
volatile h_point *point;
[
    /
    * see Pavlidis p. 324  *
    /
    point->x = 11->y*12->w - 12->y*11->w;
    point->y = 12->x*11->w - 11->x*12->w;
    point->z = 0;
    point->w = 11->x * 12->y - 11->y * 12->x;
    if(point->w != 0) { /* normalize so that point->w =1*/
        point->x /= point->w;
        point->y /= point->w;
        point->w /= point->w;
    }
    else if(point->x * point->y == 0) { /* Special case for */
        point->x = 0; /* coordinate (0,0) */
        point->y = 0;
        point->w = 0;
    }
]

return(point);                /* return address of (ptr to) */
/* h_point */
/* This function is named lc_init. It initializes the lc */
/* structure (space for lc must be allocated by add_lc()) */

#include "strall2.h"     /* standard header for functions */

lc_init(lc_ptr)
{
    struct load_case *lc_ptr; /* pointer to lc to */
    /* be initialized */

    lc_ptr->begin_reac.shear = NULL; /* invalid reaction ptr */
    lc_ptr->end_reac.shear = NULL;    /* invalid reaction ptr */
    lc_ptr->begin_reac момent = NULL; /* invalid reaction ptr */
    lc_ptr->end_reac.moment = NULL;   /* invalid reaction ptr */
    lc_ptr->begin_reac.axial = NULL;  /* invalid reaction ptr */
    lc_ptr->end_reac.axial = NULL;    /* invalid reaction ptr */
    lc_ptr->loads = NULL;             /* invalid ptr */
}

/* This function initializes the lc. To set the lcer parameters to */
/* the correct values, use function input_lc */
/ * This routine finds the length of a member * /

#include "strall2.h"
#include <math.h>

double length(member_ptr) {
    struct member *member_ptr;
    double x_length, y_length, z_length, length;
    x_length = member_ptr->begin_node->x - member_ptr->end_node->x;
    y_length = member_ptr->begin_node->y - member_ptr->end_node->y;
    z_length = member_ptr->begin_node->z - member_ptr->end_node->z;
    length = sqrt(x_length*x_length + y_length*y_length + z_length*z_length);
    return(length);
}

/* Note: node coordinates are in global coordinates */
/* This file is called line.c. The function line() takes two points and returns the homogeneous representation for the line that they define. */

#include "stral12.h"

h_line *line(p1, p2, line1)

h_point *p1, *p2;

h_line *line1; /* currently assumes 2d graphics */

/* line is defined in local.h */

{

/* see Pavlidis p. 324 */

line1->x = p1->y - p2->y;
line1->y = p2->x - p1->x;

line1->z = 0;
line1->w = p1->x * p2->y - p1->y * p2->x;

return(line1);

/* return address of struct */
/* h_line. */

}
/* This function finds the perpendicular distance from a point to a line. */

#include "stralign.h"
#include <math.h>

double line_distance(p1, l)
{
    h_point *p1;
    h_line *l;

    double distance;

    distance = (l->x * p1->x + l->y * p1->y + l->w) / hypot(l->x, l->y);
    if(distance < 0)
        distance *= -1;

    return(distance); /* always return positive distance */
}
/* This function prints out a list of members' tributary area */

#include "strall2.h"
#include <stdio.h>

list_areas()
{
    int to_mnum();
    struct corner *corner, *corner_head;
    struct area *area;
    struct member *present_member;
    extern struct member *member_head;

    printf("List of areas in structure\n");
    printf("member\n");
    printf("area @ x y x y x y \n");

    present_member = member_head;
    while(present_member != NULL) {
        printf("member %d\n", to_mnum(present_member));
        area = present_member->area;

        while(area != NULL) {
            printf("%d", area);
            corner = corner_head = area->corner;
            while(corner != NULL) {
                printf("%.2f %.2f", corner->x, corner->y);
                corner = corner->next;
                if(corner == corner_head) break;
            }
            printf("\n");
            area = area->next;
        }
        present_member = present_member->next;
    }
    return;
}
/** This function is called list_fil(). It prints out a list */
/* of all node fil by load_case. */

#include "strall2.h"       /* standard header for functions */

list_fil() {

    extern struct node        *node_head;
    struct frame_in_list      *fil;
    struct node               *node_ptr;

    printf("Node \e members that frame into node\n");

    node_ptr = node_head;
    while(node_ptr != NULL) {
        fil = node_ptr->frame_in;
        printf("%d\t", node_ptr->node_number);
        while(fil != NULL) {
            printf("%7u %4d\t", node_ptr->frame_in,
                    node_ptr->frame_in->in_member->member_number);
            fil = fil->next;
        }
        printf("\n");
        node_ptr = node_ptr->next_node;
    }
}
/* This function clears the screen and lists the main menu of commands */

#include "strail2.h"

main_menu() {
    int i;
    for(i=0; i<25; i++) {
        printf("\n");
    }
    printf("\n\n\n\n");
    printf("Edit - edit or create a floor 1\n");
    printf("File - save or retrieve a floor 2\n");
    printf("Loads - edit or assign floor loads 3\n");
    printf("Analyze - perform analysis 4\n");
    printf("Results - view or file results 5\n");
    return;
}
/* This function is called list_n1(). It prints out a list */
/* of all member n1 by load_case. */

#include "strall2.h"  /* standard header for functions*/

list_n1() {

extern struct member *member_head;
struct node_list *n1;
struct member *member_ptr;

printf("Member & nodes on member\n");

member_ptr = member_head;
while (member_ptr != NULL) {
    n1 = member_ptr->frame_in;
    printf("%4d, %d\n", member_ptr->member_number);
    if (n1 == NULL) {
        printf("NULL");
    }
    while (n1 != NULL) {
        printf("%4d, %4d, %4d\n", member_ptr->frame_in, member_ptr->frame_in->in_node->node_number);
        n1 = n1->next;
    }
    printf("\n");
    member_ptr = member_ptr->next;
}
}
This function prints out a list of the members in a structure

```c
#include "str112.h"
#include <stdio.h>

list_members() {
    struct member *present_member;
    extern struct member *member_head;

    printf("List of members in structure\n");
    printf("member frame begin node end node\n");
    printf("\n # sol sect loads _in ptr #\n");
    printf("ptr next last\n");

    present_member = member_head;
    while (present_member != NULL) {
        printf("%4d", present_member->member_number);
        printf("%4u", present_member->flags.solved);
        printf("%4u", present_member->section);
        printf("%6u %6u", present_member->loads,
                present_member->frame_in);
        printf("%6u %3d", present_member->begin_node,
                present_member->begin_node->node_number);
        printf("%6u %3d", present_member->end_node,
                present_member->end_node->node_number);
        printf("%6u %6u\n", present_member->next,
                present_member->last);
        printf("%6.2lf %6.2lf\n", present_member->begin_reac.shear,
                present_member->end_reac.shear);
        present_member = present_member->next;
    }
    return;
}
```
/* This function is called list_reactions(). It prints out a list */
/* of all member reactions by load_case. */

#include "strai2.h" /* standard header for functions*/

list_reactions() {

extern struct member *member_head;
struct member *member_ptr;
struct load_case *lc_ptr;

printf("Member # load case 1 load case 2\n");
printf("Vbeg Mbeg Vend Mend Vbeg Mbeg Vend Mend\n");

member_ptr = member_head;
while(member_ptr != NULL) {
    lc_ptr = member_ptr->loads;
    printf("%4d",member_ptr->member_number);
    while(lc_ptr != NULL) {
        printf("%8.2lf %8.2lf %8.2lf %8.2lf ",
               lc_ptr->begin_reac.shear, lc_ptr->begin_reac.moment,
               lc_ptr->end_reac.shear, lc_ptr->end_reac.moment);
        lc_ptr = lc_ptr->next_lc;
    }
    printf("\n");
    member_ptr = member_ptr->next;
}
/* This function initializes the values in a struct load */

#include "strall2.h" /* standard header for functions*/

load_init(date, who, cf, bx, ex, bm, em, load_ptr)

int date;
int who;
int cf;
double bx, ex;
double bm, em;
struct load *load_ptr;

{
    load_ptr->date_assigned = NULL;
    load_ptr->who_assigned = who;
    load_ptr->coord_flag = cf;
    load_ptr->begin_x = bx;
    load_ptr->end_x = ex;
    load_ptr->begin_mag = bm;
    load_ptr->end_mag = em;

    return;
}

/* This is the entry point main program */

#include "strall2.h"
#include <stdio.h>
#include <ctype.h>
#define INMAIN 1
#define MAXLINE 80
#define tolower(c) ((isupper(c)?((c)-('a'-'A')):(c))

int is_floor_solved = NO;
struct member *member_head, *member_tail;
struct node *node_head, *node_tail;
struct f_load *fload, *opening_head;

main()
{

extern struct member *member_head, *member_tail;
extern struct node *node_head, *node_tail;
struct member *member_ptr;
char c, s[MAXLINE];

while(tolower(*s) != 'q') {
    main_menu();    /* clear screen and list menu */
    gets(s);       /* read an input string */
    c = tolower(*s);
    switch(c) {
    case 'e':
        /* edit_menu */
        printf("Out of order\n");
        break;

    case 'f':
        /* file_menu */
        printf("Calling file_menu\n");
        file_menu();
        break;

    case 'l':
        /* load_menu */
        load_menu();
        break;

    case 'a':
        /* do_analysis */
        printf("doing analysis\n");
        do_analysis();
        break;

    case 'r':
        /* results() */
        results_menu();
        break;
    }
}
}
case 'q':
  /* edit_menu */
  printf("Quitting\n");
  break;

default:
  printf("%c is not a valid choice\n", *s);

})

nodelist();
list_members();
list_reactions();
/*fprint_loads();
/*fprint_reactions();
/* save_floor(1);
*/

/* This function clears the screen and lists the main menu of commands */
main_menu() {

int i;

for(i=0; i<20; i++) {
  printf("\n");
}

printf("\t\t\t\tMAIN MENU\n\n");
printf("\t\tEdit   edit or create a floor\n\n");
printf("\t\tFile    save or retrieve a floor\n\n");
printf("\t\tLoads   edit or assign floor loads\n\n");
printf("\t\tAnalyse perform analysis\n\n");
printf("\t\tResults view or file results\n\n");
printf("\t\tQuit    end program\n");

return;
}

/* This function clears the screen and lists the edit menu of commands */
edit_menu() {

int i;
char c, s[MAXLINE];

for(i=0; i<25; i++) {
  printf("\n");
}
printf("\ttStartNew
printf("\ttNode
printf("\ttMember
printf("\ttColumn
printf("\ttQuit
gets(s);    /* read an input string */
c = tolower(*s);
switch(c) {
    case 's':
        /* Start new floor */
        printf("start new floor not available\n");
        break;
    case 'n':
        /* edit_node */
        printf("edit nodes not available\n");
        /* edit_node() */
        break;
    case 'm':
        /* edit_members */
        printf("edit members not available\n");
        /* edit_member() */
        break;
    case 'c':
        /* edit_column */
        printf("edit columns not available\n");
        /* edit_column() */
        break;
    case 'q':
        /* return to main menu */
        printf("Quitting\n");
        break;
    default:
        printf("'\%c' is not a valid choice\n", *s);
}
return;

/* This function clears the screen and lists the file menu of commands */
file_menu() {
    int i;
    char c, s[MAXLINE];
    extern struct f_load *fload;
    struct f_load *get_fload();
    for(i=0; i<25; i++) {
        printf("\n");
printf("\t\tSave
- save a floor \n\n");
printf("\t\tRetrieve
- retrieve a floor \n\n");
printf("\t\tDirectory
- view file directory \n\n");
printf("\t\tQuit
- return to main menu\n\n");
gets(s);
// read an input string
"/
c = tolower(*s);
switch(c) {
   case 's':
      /* Save floor */
      printf("save floor not available\n");
      break;
   case 'r':
      /* Retrieve floor */
      printf("retrieving floor\n");
      get_floor();
      break;
   case 'd':
      /* directory of saved floors */
      printf("directory not available\n");
      break;
   case 'q':
      /* return to main menu */
      printf("Quitting\n");
      break;
   default:
      printf("'%c' is not a valid choice\n", *s);
}
return;

/* This function clears the screen and lists the file menu of commands */
load_menu() {

int i;
char c, s[MAXLINE];

for(i=0; i<25; i++) {
   printf("\n");
}

printf("\t\tMember - assign member loads 1\n\n");
printf("\t\tFloor - assign floor loads 2\n\n");
printf("\t\tQuit
- return to main menu\n\n");

/*
printf("\t\tCalling get_load()\n");
float load = get_load();
printf("\t\tReturn from get_load()\n");
*/
gets(s);
// read an input string
"/
c = tolower(*s);
switch(c) {
    case 'm':
        /* Member loads */
        printf("Assign member loads not available\n");
        break;
    case 'f':
        /* Floor loads */
        printf("\t\tCalling get_fload()\n");
        fload = get_fload();
        printf("\t\tReturn from get_fload()\n");
        break;
    case 'q':
        /* return to main menu */
        printf("Quitting\n");
        break;
    default:
        printf("\%c is not a valid choice\n", *s);
}
return;

/* This function clears the screen and lists the file menu of commands */
results_menu() {
in
    i;
    char c, s[MAXLINE];
    struct member *to_mptr();

    for(i=0; i<25; i++) {
        printf("\n");
    }

    if(is_floor_solved == NO) {
        printf("The floor has not yet be analyzed\n");
        printf("Strike any key to continue\n");
        gets(s);        /* pause for input */

        printf("\t\tShear\n");
        printf("\t\tMoment\n");
        printf("\t\tFile\n");
        printf("\t\tView\n");
        printf("\t\tQuit\n");

        printf("\n\t\tview shear diagrams \n\n");
        printf("\n\t\tview moment diagrams \n\n");
        printf("\n\t\tsave results in a file\n\n");
        printf("\n\t\tView results \n\n");
        printf("\n\t\treturn to main menu \n\n");

        gets(s);
        /* read an input string */
        c = tolower(*s);
        switch(c) {
            case 's':

/ * Shear diagram */
printf("Shear diagram not available\n");
break;

case 'm':
    /* Retrieve floor */
printf("Moment diagram not available\n");
break;

case 'f':
    /* Save results */
list_reactions();
fsprint_loads(to_mptr(5));
fsprint_loads(to_mptr(5));
break;

case 'v':
    /* View results */
list_reactions();
break;

case 'q':
    /* return to main menu */
printf("Quitting\n");
break;

default:
    printf("\%c is not a valid choice\n", *s);
}

return;

/* This function clears the screen and lists the file menu of commands */
do_analysis() {

    int
    struct member *member_ptr;
    struct node *node_ptr;
    extern int
    is_floor_solved;
    extern struct member
    *member_head;
    extern struct f_load
    *fload;

    for(i=0; i<20; i++) {
        printf("\n");
    }

    reset();    /* set member->flags.solved = NO for all members*/
    if(fload == NULL) {
        printf("Floor loads must be assigned before analysis\n");
        printf("Strike any key to continue\n");
        getchar();
    }
check_cantilevers(); /* Check to see if cantilevers have */ /* opposing members, columns. */

/* See if there are any openings in the floor. If so, make */ /* adjustments in the floor loads as appropriate. */

clip_openings();

printf("\t\tCalling testadj1()\n");
testadj1();

printf("\t\tCalling clip_polygon()\n");
member_ptr = member_head;
while (member_ptr != NULL) {
    clip_polygon(fload, member_ptr);
    /* printf("\t\tCalling carrydown()\n"); */
    /* carrydown(); */
    member_ptr = member_ptr->next;
}

printf("\t\tCalling analyze()\n");
analyze();
printf("\t\treturned from analysis\n");

is_floor_solved = YES;

return;
This function is called memberlp(). It loops over the physical members in a floor, identifies graphical members, and puts paths in the form of a single array of unique paths. The protocol for passing paths back from path() is given a to what it thinks is an array of union pathnode paths[2][MAXPATH] the paths found are placed in the paths[0][] and paths[1][] rows for use by other functions. Unique checks to see if 1)path() actually found two paths, and 2) whether either of the paths duplicate an existing path. Unique returns the array location in allpaths[][] that the next path() call will use.

#include "strall2.h"

memberlp(adjlist, allpaths)

adjnode *adjlist[1];
union pathnode allpaths[1][][MAXPATH];

{
extern int one way_frame, two way_frame;
extern struct member *member_head;

int i, j, one way_frame, two way_frame,
path(), to,mnum(), unique();
direction;
struct area
struct member
*oneway(), *two way();
*member_ptr;

struct node_list
struct node
*currentnl;
*last;

i = j = 0;
one way_frame = 0;
two way_frame = 1;
direction.x = 1;
direction.y = 0;
direction.w = 50;

member_ptr = member_head;
while(member_ptr != NULL) {
   /* The following routine treats the case where the physical member has no nodes on it. In this case, the physical and graphical members coincide. */
   for(i=0; i<MAXPATH; i++) {
      /* initialize to 0 */
      allpathst[1][0].length = 0;
   }

   i = 0; /* initial position in allpaths[][] */
   if(member_ptr->frame_in == NULL) {
      i += path(member_ptr->begin_node,member_ptr->end_node, adjlist, &allpaths[1][0]);
   }
}
/* path() will find the shortest legal */
/* path on each side of the member. inserts */
/* the 2 paths in the allpaths array staring at */
/* position i. */
/* No call to unique() is needed here because */
/* there are only 2 possible paths. If path() */
/* did not find a legal path, allpaths[i][0] */
/* and/or allpaths[i][1][0] will be NULL */

} /* The following treats the case where a physical */
/* member contains nodes. In this case, the physical */
/* member must be broken into its constituent graphical */
/* members for analysis. */

/* begin_node - 1st intermediate node */
else {
  current_nl = member_ptr->frame_in;
  path(member_ptr->begin_node, current_nl->in_node,
       adjlist, &allpaths[i][0]);
  i += path(member_ptr->begin_node, current_nl->in_node,
             adjlist, &allpaths[i][0]);
  
  i = unique(allpaths, i);

  /* intermediate to nth intermediate nodes */

  last = current_nl->in_node;
  current_nl = current_nl->next;
  while (current_nl != NULL) {
    path(last, current_nl->in_node,
         adjlist, &allpaths[i][0]);
    i = unique(allpaths, i);
    last = current_nl->in_node;
    current_nl = current_nl->next;
  }

  /* nth intermediate node - end_node */

  path(last, member_ptr->end_node, adjlist, allpaths[i]);
  i = unique(allpaths, i);
}

#undef DEBUG

/* Transform circuit representation into area */
/* representation, and link to physical member. */

/* Must allocate space for area ??? */
if(twoway_frame) {
  member_ptr->area = twoway(member_ptr, allpaths);
}
else if(oneway_frame) {
member_ptr->area =
  oneway(member_ptr, allpaths, &direction);
}

member_ptr = member_ptr->next;

/* At this point, I have a complete set of */
/* circuit lists for all of the graphical members*/
/* that make up a physical member *member_ptr. */
/* Now I need a function that will perform a */
/* union of these circuits to get the minimum */
/* unique set. This set of circuit paths */
/* can then be used to find the member's tribu- */
/* tary area polygon. */

return;
/* This function is named memb_init. It initializes the memb */
/* structure (space for memb must be allocated by add_memb()) */

#include "strall2.h" /* standard header for functions*/

memb_init(memb_ptr)
{
    struct member *memb_ptr;
    /* pointer to memb to */
    /* be initialized */
    memb_ptr->section = NULL; /* */
    memb_ptr->loads = NULL; /* invalid ptr */
    memb_ptr->frame_in = NULL; /* invalid ptr */
    memb_ptr->begin_node = NULL; /* invalid ptr */
    memb_ptr->end_node = NULL; /* invalid ptr */
}

/* This function initializes the memb. To set the member parameters to */
/* the correct values, use function input_memb() */
/* This function is named node_init. It initializes the node */
/* structure (space for node must be allocated by add_node()) */

#include "strall2.h" /* standard header for functions*/

node_init(node_ptr)
struct node *node_ptr;
/* pointer to node to */
/* be initialized */
{
    node_ptr->x = -1.0; /* invalid coordinate */
    node_ptr->y = -1.0; /* invalid coordinate */
    node_ptr->z = -1.0; /* invalid coordinate */
    node_ptr->flags.node_type = 1; /* assumed to be girder */
    node_ptr->flags.fix_x = 1; /* fixed for translation*/
    node_ptr->flags.fix_y = 1; /* fixed for translation*/
    node_ptr->flags.fix_z = 1; /* fixed for translation*/
    node_ptr->flags.fix_rx = 0; /* not fixed for rot */
    node_ptr->flags.fix_ry = 0; /* not fixed for rot */
    node_ptr->flags.fix_rz = 0; /* not fixed for rot */
    node_ptr->frame_in = NULL; /* no frame in list */
}
/* This function initializes the node, the function input_node() can set*/
/* node parameters to the correct values. Pointers to the next and last*/
/* nodes are set in function add_node(). */
/* This function prints out a list of the nodes in a structure */

nodelist()
{
    struct node *present_node;
    extern struct node *node_head;

    printf("List of nodes in structure\n");
    printf("\t fixed 0 free last next fill on\n");
    printf("\t x y z x y z node node\n");
    printf(" fill next member\n");

    present_node = node_head;
    while(present_node != NULL) {
        printf("node %d\t", present_node->node_number);
        printf("%6.2f %6.2f ", present_node->x,
            present_node->y);
        printf(" %lu %lu %lu", present_node->flags.fix_x,
            present_node->flags.fix_y, present_node->flags.fix_z);
        printf(" %lu %lu %lu", present_node->flags.fix_rx,
            present_node->flags.fix_ry, present_node->flags.fix_rz);
        printf(" %lu", present_node->flags.cantilever);
        printf(" %5u", present_node->laut_node);
        printf("%5u ",present_node->next_node);
        printf("%5u ",present_node->frame_in);
        printf("%5u ",present_node->frame_in->next);
        printf("%5u\n",present_node->on_member);
        present_node = present_node->next_node;
    }
    return;
}
/* This file is called oneway.c. This function takes a member and */
/* its list of circuits, and creates a tributary area polygon assuming */
/* a one-way flooring system. */

#include "strall2.h" /* standard header for functions*/
#include <math.h>

#define DEBUG 1

h_point
  corner1, corner2, corner3, cornern,
  point1, point2, point3;

h_line
  line1, line2, line3;

typedef struct hareas {
  h_area *area;
  struct hareas *next;
}h_areas;

/* This function works in h_areas representation, but for compatibility */
/* with the existing member data structure, converts the h_areas into */
/* a area/corner representation. */

struct area *oneway(member_ptr, all_paths, direction)
  /* floor direction must be known*/
struct member
  *member_ptr;
union pathnode
  all_paths[1][MAXPATH]; /* must straighten this out */
  *direction; /* direction = h_line */
{
  int i, j, length, n, path_length;
  char *alloc();
  h_areas *all_areas, *areas1, *last_area, *areas_head;
  h_area *area1, *area_head;
  h_line bisect[MAXPATH], *i1, *i2, *i3, *line();
  h_point *c1, *c2, *c3, *cn, *p1, *p2, *p3,
    *intersect(), *node_to_hpoint(), *set_corner();
  struct area *ca_ptr, *harea_to_area();

  i = j = length = n = 0;

  /* direction and *i1 must be normalized for valid comparison */
  /* This step is done to get the equation, and hence the slope of*/
  /* the line between ends of the member being analyzed. */
  length = all_paths[0][0].length;
  cl = node_to_hpoint(all_paths[0][11], &corner1);
  cn = node_to_hpoint(all_paths[0][length], &cornern);
  ll = line(cl, cn, &line1);

  if((ll->x * direction->y == ll->y * direction->x)) { /* Member must have a load_case to sum reactions */
member_ptr->loads = add_lc(member_ptr);
return(NULL); /* should there be a status returned? */

all_areas = (struct hareas *) calloc(1, sizeof(struct hareas));
areas_head = all_areas;
area1 = (h_area *) calloc(1, sizeof(h_area));
all_areas->area = area1;
area_head = area1;

while(all_paths[j][0].length != NULL) {
    path_length = all_paths[j][0].length;

    if(path_length != 0) {
        c1 = (h_point *) calloc(1, sizeof(h_point));
        area1->point = c1;

        /* first point is the end of the graphical member */
        node_to_hpoint(all_paths[j][1].node_ptr, c1);

        c1 = node_to_hpoint(all_paths[j][1].node_ptr, &corner1);
        c2 = node_to_hpoint(all_paths[j][2].node_ptr, &corner2);
        c3 = node_to_hpoint(all_paths[j][3].node_ptr, &corner3);
        cn = node_to_hpoint(all_paths[j][length].node_ptr, &cornern);

        /*
        * area1->next = (struct harea *) calloc(1, sizeof(struct harea));
        * area1 = area1->next;
        *
        * area1->point = set_corner(cn, c1, c2, c3, direction);
        */

        c1 = node_to_hpoint(all_paths[j][1].node_ptr, &corner1);
        c2 = node_to_hpoint(all_paths[j][length-2].node_ptr, &corner2);
        c3 = node_to_hpoint(all_paths[j][length-1].node_ptr, &corner3);
        cn = node_to_hpoint(all_paths[j][length].node_ptr, &cornern);

        p1 = (h_point *) calloc(1, sizeof(h_point));

        area1->next = (struct harea *)
                    calloc(1, sizeof(struct harea));

        area1 = area1->next;

        area1->point = set_corner(c1, cn, c3, c2, direction);

        /* last corner is end node of the path */
        printf("last corner: \n");

        area1->next = (struct harea *)
                     calloc(1, sizeof(struct harea));
areal = areal->next;

c1 = (h_point *) calloc(1, sizeof(h_point));
areal->point = c1;

node_to_hpoint(all_paths[j][length].node_ptr, c1);
areal->next = area_head; /* list is circular */
}

j++; /* continue while to do next path */
all_areas->next = (struct hareas *)
    calloc(1, sizeof(struct hareas));
last_area = all_areas;
all_areas = all_areas->next; /* linking adds 1 too many areas */
areal = (struct harea *)
    calloc(1, sizeof(struct harea));
all_areas->area = area_head = areal;
}
last_area->next = NULL; /* area's memory should be freed */
ca_ptr = harea_to_carea(areas_head);
return(ca_ptr);
}

/* This routine takes a node and returns the h_point representation */
/* for its coordinates. */

h_point *node_to_hpoint(n1, h1)

struct node   *n1;
struct harea  *h1;
{
    h1->x = n1->x;
    h1->y = n1->y;
    h1->z = n1->z;
    h1->w = 1;

    return(h1);
}

/* This function takes list of nodes and finds the corner of the */
/* tributary area for a one way flooring system. */

h_point *set_corner(c1, c2, c3, c4, direction)

h_point  *c1, *c2, *c3, *c4; /* floor direction must be known*/
...
int i, j, length, n;
char * calloc();
h_line bisect(MAXPATH), * i11, * i12, * i13;
h_point * p1, * p2, * p3, * intersect();

i = j = length = n = 0;

11 = line(c1, c2, & line1);

if(11->x * direction->y != 11->y * direction->x) {
    return; /* should there be a status returned? */
}

/* Construct line in strong direction at 1st point in path */
line2.x = direction->x;
line2.y = direction->y;
line2.w = -c2->x * direction->x - c2->y * direction->y;

12 = & line2;

13 = line(c3, c4, & line3);
p1 = (h_point *) calloc(1, sizeof(h_point));
p1 = intersect(12, 13, p1);

/* if c3, and p1 are on the same side of c1 - c2, the */
/* product of the two right_of() tests is positive */
if(right_of(c2, c3, c4) * right_of(c2, c3, p1) >= 0) {
    p1->x = (p1->x + c2->x)/2;
    p1->y = (p1->y + c2->y)/2;
    return(p1);
}
else {
    /* Construct perpendicular at 2nd point in path */
    line2.x = direction->x;
    line2.y = direction->y;
    line2.w = -c3->x * direction->x - c3->y * direction->y;

    /*11 = line(c1, c2, line3);*/
}
pi = intersect(11, 12, pi);
ipi->x = (pi->x + c3->x)/2;
ipi->y = (pi->y + c3->y)/2;
return(pi);
}

/* This function converts an h_area/h_point area representation into a */
/* corner based representation. */

struct area *harea_to_area(a_ptr)
    h_areas       *a_ptr;
{
    int            count = 0;
    char           *calloc();
    h_area         *harea_ptr, *harea_head, *last_harea;
    h_point        *hp, *point;
    struct area    *area_head, *ca_ptr, *ca_head, *last_ca_ptr;
    struct corner  *corner_ptr, *c_head, *last_corner, *hpoint_to_corner();

    ca_ptr = ca_head = (struct area *) calloc(1, sizeof(struct area));

    while(a_ptr != NULL) {

        harea_ptr = harea_head = a_ptr->area;
        point = harea_ptr->point;

        corner_ptr = c_head = (struct corner *)
            calloc(1, sizeof(struct corner));

        ca_ptr->corner = corner_ptr;

        while(harea_ptr != harea_head) { count++ < 1 ) {

            corner_ptr = hpoint_to_corner(point, corner_ptr);
            corner_ptr->next = (struct corner *)
                calloc(1, sizeof(struct corner));

            last_corner = corner_ptr;
            corner_ptr = corner_ptr->next;
            harea_ptr = harea_ptr->next;
            point = harea_ptr->point;

        }

        count = 0;
        last_corner->next = c_head;
        ca_ptr->next = (struct area *) calloc(1, sizeof(struct area));
        last_ca_ptr = ca_ptr;
        ca_ptr = ca_ptr->next;
        a_ptr = a_ptr->next;
last_ca_ptr->next = NULL;
    
    return(ca_head);
);

/* THIS function converts an h_point representation (homogeneous point) */
/* into a corner representation of a point */

struct corner *hpoint_to_corner(h1, c1)

h_point    *h1;
struct corner    *c1;

{
    c1->x = h1->x;
    c1->y = h1->y;
    c1->z = h1->z;

    return(c1);
}
/* This function is named path.c. The function finds the shortest */
/* (based on number of edges traversed) path between two vertices an */
/* adjacency list representation of the floor graph. */

#include "strall2.h" /* standard header for functions*/

#define MAXQUE 32 /* Redefinition of MAXQUE */

path(n1,n2,adj1,paths)
struct node *n1, *n2;
adjnode *adj1[];
union pathnode paths[][MAXPATH];
{
    int i, legal, path_length, qsize, qpos, qposition;
    int while1count, while2count, while3count, while4count, ifcount;
    int visited[MAXV];
    struct qnode que[MAXQUE];
    adjnode *a_node;
    struct node *n, *neighbor;

    for(i=0; i<MAXV; i++)
        visited[i] = 0; /* Initialize to 0 */
    for(i=0; i<MAXQUE; i++)
        que[i].parent = 0;
        que[i].node_ptr = NULL; /* */

    if(n1 == n2) {
        return;
    } /* initialize beginning of search que */
    que[0].node_ptr = n1;
    que[0].parent = NULL;
    visited[to_nnum(n1)] = YES;
    path_length = 0;
    qposition = 0;
    qsize = 1;
    n = n1;

    while1count = while2count = ifcount = 0;

    while((n != n2 && qsize <= MAXQUE && n != NULL) { /* */
        while1count = while2count = ifcount = 0;
        while1count++;
        a_node = adj1[n->node_number];
        while2count = 0;
        while(a_node != NULL) { /* */
            while2count++;
            /* */
            printf("\"initial\"while2count = %dn", while2count);
            printf("\n136 a_node %s\ta_node->next%sn", a_node,
a_node->next);

/*
 neighbor = a_node->node_ptr;
 if(count == 0);
 /* These steps effectively delete the n1-n2 edge*/
 /* because the destination, n2 in only recog-
 /* ized as reached if we're past the first */
 /* level of the search (n1's neighbors). */
 if(neighbor == n2) {
   if(qposition > 0) {
     n = n2;
     break; /* stop adding to que */
   }

 } else if(visited[neighbor->node_number] == NO) {
   ifcount++;
   visited[neighbor->node_number] = YES;
   que[qsizel].parent = qposition;
   que[qsizel].node_ptr = neighbor;
   qsize++;
 }

 a_node = a_node->next;

/*
 if(a_node != NULL)
   printf("\t148 a_node->next = %lu\n", a_node->next);
 else
   printf("while2 next a_node undefined\n");
 */

if(qposition > qsize) {
   printf("qposition > qsize - aborting search\n");
   break; /* run into dead end */
} else if(n == n2) {
   n = n;
   /* do nothing */
} else {
   qposition++;
   n = que[qposition].node_ptr;
}

/* If n1 had no neighbors, the program will hang here */
/* because n below will get garbage or cause error. */
/* Breadth first search has concluded, see if there's a path */
i = 0;

if(qsize > MAXQUE || qposition > qsize) {
  return(NULL);
  /* no path found from n1 to n2 */
}
else if(n == n2) { /* There is a path */
paths[0][++i].node_ptr = n2; /* add n2 to make circuit */
path_length++;
qpos = qposition;
while(qpos != NULL) {
    paths[0][++i].node_ptr = que[qpos].node_ptr;
    qpos = que[qpos].parent;
    path_length++;
}
paths[0][++i].node_ptr = que[qpos].node_ptr;
paths[0][0].length = ++path_length;

filter(paths[0]);                 /* ptr to 1st array */
}
else {
    return(NULL);
}

/* first path found and examined above, start search for 2nd */
legal = NO;

while(legal == NO) {
    visited[n2->node_number] = NO;      /* make n2 unvisited */
    qposition++;                        /* continue the search */
    n = que[qposition].node_ptr;       /*visit next node in que*/
    path_length = 0;
    i = 0;

    while3count = while4count = ifcount = 0;

    while(n != n2 && qsize <= MAXQUE && n != NULL) {
        a_node = adj1[n->node_number];
        while3count++;
        while2count = 0;
        while(a_node != NULL) {
            while2count++;
            neighbor = a_node->node_ptr;
            if(neighbor == n2) {
                if(qposition > 0) {
                    n = n2;
                    break; /* stop adding to queue */
                }
            }
            else if(visited[neighbor->node_number] == NO) {
                ifcount++;
                visited[neighbor->node_number] = YES;
                que[qsize].parent = qposition;
                que[qsize].node_ptr = neighbor;
                qsize++;
            }
            a_node = a_node->next; /* next in adj1 */
        }
    }
}
if(qposition > qsize) {
    printf("qposition > qsize - aborting search\n");
    break;                   /* run into dead end        */
}
else if(n == n2) {
    n = n2;                   /* do nothing      */
}
else {
    qposition++;
    n = que[qposition].node_ptr;
}

/* Breadth first search has concluded, see if path's legal */

if(qsize > MAXQUE :: qposition > qsize) {
    paths[1][0].length = 0; /* no 2nd path found */
    return(1);              /* return integer 1 */
} /* break out of loop, return */

/* Test to see if the path found is a legal one */

else if(n == n2) {
    paths[1][++i].node_ptr = n2;
    path_length++;
    qpos = qposition;
    while(qpos != NULL) {
        paths[1][++i].node_ptr = que[qpos].node_ptr;
        qpos = que[qpos].parent;
        path_length++;
    }
    paths[1][++i].node_ptr = que[qpos].node_ptr;
    paths[1][0].length = ++path_length;
    filter(paths[1]);          /* ptr to list array */
    legal = test(paths);        /* If 2nd path is found */
}

if(legal == YES) {
    return(2);                 /* 2 paths found */
}
else {
    return(1);
}
/** This function prints an adjacency list data structure */
#include "strall2.h"

print_adjlist(jimbob)

adnode *jimbob[];
{

int i, j, neighbor, to_nnum();
adnode *a_node;

/*
   for(i=1; i<=10; i++) {
       printf("adjlist[%d]\tadjlist[%u]\tadjlist[%u]\n", i, &jimbob[i], jimbob[i]);
   }
*/
i = 1;

/*
while(jimbob[i] != NULL && i < 55) {
    a_node = jimbob[i];
    printf("adjlist%d", i);
    while(a_node != NULL) {
        printf("\t\t%u", a_node, a_node->next);
        a_node = a_node->next;
    }
    i++;
    printf("\n");
}
*/
printf("node# a_node neighbor neighbor\n");

for(i=1; i<=55; i++) {
    a_node = jimbob[i];
    printf("node %d", i);
    while(a_node != NULL) {
        //printf("%u", a_node);
        neighbor = to_nnum(a_node->node_ptr);
        printf(" %d %u", neighbor, a_node->node_ptr);
        a_node = a_node->next;
    }
    printf("\n");
}
return;
/* This function prints a floor load (f_load) structure. */

#include "strall2.h"

print_fload(fl_ptr)

struct f_load *fl_ptr;
{
    int count = 0;
    struct f_load *head;

    head = fl_ptr;
    printf("%s\n", "fl_ptr\nnext\ntarea\ntname\ntdate\ntwho\ntcoord\ntmagnitude\n");
    while((fl_ptr != head || count++ (i)) && fl_ptr != NULL) {
        printf("%d\t%6d\t%6d\t%6d\t%4d\t%.2f\n", fl_ptr,
            fl_ptr->next, fl_ptr->area, fl_ptr->name, fl_ptr->date_assigned,
            fl_ptr->who_assigned, fl_ptr->coord_flag, fl_ptr->magnitude);

        fl_ptr = fl_ptr->next;
    }

    return;
}
/* This function prints an h_area representation. 

#include "strail2.h"

print_harea(a_ptr)

h_area *a_ptr;
{
    int count = 0;
    h_area *head;
    head = a_ptr;
    printf("a_ptr\nnext\npoint\tx\nty\tw\n");
    while((a_ptr != head || count++) && a_ptr != NULL) {
        printf("%d\%d\%d\%d\n", a_ptr, a_ptr->next, a_ptr->point, x, a_ptr->point->y, a_ptr->point->w);
        a_ptr = a_ptr->next;
    }
    return;
}
/* This function prints the loads on a member. */

#include "strall2.h"

print_loads(member_ptr)

struct member *member_ptr;
(
    int count = 0;
    struct load_case *loadcase;
    struct load *load;

    printf("loads for member %d\n", member_ptr->member_number);
    loadcase = member_ptr->loads;
    while(loadcase != NULL) {
        printf("\tloadcase %d %n", ++count, loadcase);
        printf("\tt\t\begin_x\t\end_x\tt\begin_mag\t\end_mag\n");
        load = loadcase->loads;
        while(load != NULL) {
            if(load->begin_mag != 0 || load->end_mag != 0) {
                printf("\tt\%6.2f\t%6.2f\t", load->begin_x, load->end_x);
                printf("\t%6.2f\t%6.2f\n", load->begin_mag, load->end_mag);
            }
            load = load->next_load;
        }
        loadcase = loadcase->next_lc;
    }

    return;
}
/* This function prints paths found by the path() function */

#include "strali2.h"

print_path(paths)

union PathNode paths[][MAXPATH];

int i, path_length, to_nnum();

i = path_length = 0;

printf("path: ");

printf("length %d ", paths[0][0].length);
for(i=1; i<= paths[0][0].length; i++) {
    printf("\t%d %u", to_nnum(paths[0][i].node_ptr),
        paths[0][i].node_ptr);
    if(i>5) {
        printf("i>5 aborting\n");
        break;
    }
}

printf("\n");
return;
/** This function is used to retrieve either the floor loads or the
 * openings for a floor. It returns a pointer to the recreated list. */

#include "strall2.h"
#include <stdio.h>

struct f_load *read_floordload(flf_ptr)

FILE *flf_ptr; /* pointer to loads or openings */

{
    int area_tag, count, name, end;
    char *malloc(),
    h_area *area_head, *current_area, *last_area;
    struct f_load *fload_ptr, *fload_head;

    end = count = 0;
    fload_ptr = fload_head = (struct f_load *)
        malloc(1, sizeof(struct f_load)); /* allocate space */

    end = fscanf(flf_ptr,"%d", &name);

    while(end != EOF) ( /* read floorload file */
        fload_ptr->name = name;

        fscanf(flf_ptr,"%d", &fload_ptr->date_assigned);

        fscanf(flf_ptr,"%d %d", &fload_ptr->who_assigned,
            &fload_ptr->coord_flag);

        fscanf(flf_ptr,"%lf", &fload_ptr->magnitude);

        /* The next step is to read in the coordinates of the vertices
        * of the h_area for this load. These coordinates are in the
        * floorload.dat file interspersed with the other floor load
        * data. Each coordinate begins with an integer tag. When the
        * end of the linked list is reached, the tag is 0.
        * floor loads are linked together when read from the file so
        * that they remain in the same order as they were written in. */

        current_area = area_head = (h_area *)
            malloc(1, sizeof(h_area));

        current_area->point = (h_point *)
            malloc(1, sizeof(h_point));

        fscanf(flf_ptr,"%d", &area_tag);

        while(area_tag != 0) ( /* read floorload file */
            fscanf(flf_ptr,"%lf", &current_area->point->x));

            fscanf(flf_ptr,"%lf", &current_area->point->y));

            fscanf(flf_ptr,"%lf", &current_area->point->z));

            fscanf(flf_ptr,"%lf", &current_area->point->w));

            fscanf(flf_ptr,"%d", &area_tag);

            current_area->next = (h_area *)
malloc(1, sizeof(h_area));
last_area = current_area;
current_area = current_area->next;
current_area->point = (h_point *)
malloc(1, sizeof(h_point));
}
last_area->next = area_head; /* make list circular */
floor_ptr->area = area_head; /* add area to floorload*/

end = fscanf(flf_ptr, "%d", &name);
if(end != EOF) {
    floor_ptr->next = (struct f_load *)
    malloc(1, sizeof(struct f_load));
    floor_ptr = floor_ptr->next;
}

return(floor_head);
#include "strall2.h"
#include <stdio.h>

struct f_load *read_memberload(mlf_ptr)
{
    FILE *mlf_ptr;
    /* pointer to loads or openings */

    int name, end;
    struct load_case *lc_ptr, *to_lcptr();
    struct member *member_ptr, *to_mptr();
    struct load *load_ptr, *add_load();

    end = name = 0;

    end = fscanf(mlf_ptr,"%d", &name);

    while (end != EOF) /* read memberload file */
    {
        member_ptr = to_mptr(name/10); /* integer division */
        /* name modulo 10 = loadcase # must be (<= 9 */
        lc_ptr = to_lcptr(name%10, member_ptr);
        load_ptr = add_load(lc_ptr);

        fscanf(mlf_ptr,"%d", &load_ptr->date_assigned);

        fscanf(mlf_ptr,"%d %d", &load_ptr->who_assigned,
               &load_ptr->coord_flag);
        fscanf(mlf_ptr,"%lf %lf", &load_ptr->begin_mag,
               &load_ptr->end_mag);
        /* the order of end_mag, begin_mag is intentionally */
        /* reversed because the program consider a positive */
        /* load to have a begin_x > its end_x */

        fscanf(mlf_ptr,"%lf %lf", &load_ptr->end_x, 
               &load_ptr->begin_x);

        end = fscanf(mlf_ptr,"%d", &name);
    }

    return;
}
/ * This function loops over the members in the floor and solves */
/* for member end reactions */

#include "strall2.h"
#include (stdio.h)

#define total_load ic_ptr->begin_reac.shear
#define sum_moment ic_ptr->begin_reac.moment
#define KLUDGE
#define KLUDGE2

recurse(member_ptr)

struct member *member_ptr;
{
    int i, j;
    double coordinates[3], L, x1, x2, p1, p2;
    double length(), *to_global, *to_local();
    extern struct member *member_head;
    struct member *find_opposite(), *memb_ptr, *opposite;
    struct node *node_ptr;
    struct node_list *ni_ptr;
    struct load *load_ptr, *get_load();
    struct load_case *lc_ptr, *takedownlc_ptr;
    struct frame_in_list *fil_ptr;
    struct reaction *get_reac();

    if(member_ptr != NULL) {
        /* more members? */
        ni_ptr = member_ptr->frame_in;
        /* Loop over nodes */
        while(ni_ptr != NULL) {
            node_ptr = ni_ptr->in_node;
            /* more nodes on memb? */
            if(node_ptr != NULL) {
                fil_ptr = node_ptr->frame_in;
                while(fil_ptr != NULL) {
                    memb_ptr = fil_ptr->in_member;
                    if(memb_ptr->flags.solved == NO) {
                        /* solved? */
                        recurse(memb_ptr);
                    }
                }
            }
            /* What should recurse return, if anything? */
            /* must set p1,p2,x1,x2 prior to next call */
            /* may be the begin or end reaction how to tell? */
        }
    }
    lc_ptr = member_ptr->loads;
    takedownlc_ptr = memb_ptr->loads;
    while(lc_ptr != NULL && takedownlc_ptr != NULL) {
        if(member_ptr->begin_node == node_ptr) {
            p1 = p2 = takedownlc_ptr->begin_reac.shear;
            /* must convert to local coordinates */
            coordinates[0] = memb_ptr->begin_node->x;
            coordinates[1] = memb_ptr->begin_node->y;
            coordinates[2] = memb_ptr->begin_node->z;
            x1 = x2 = *(to_local(member_ptr, coordinates));
            x1 = x2 = x1 * length(member_ptr);
        }
    }
}
else if(memb_ptr->end_node == node_ptr) {
    p1 = p2 = takedown(lc_ptr->end_reac_shear;
    /* must convert to local coordinates */
    coordinates[0] = memb_ptr->end_node->x;
    coordinates[1] = memb_ptr->end_node->y;
    coordinates[2] = memb_ptr->end_node->z;
    x1 = x2 = *(to_local(member_ptr, coordinates));
    x1 = x2 = xi*length(member_ptr);
}
else {
    printf("Error in load takedown in recurse\n");
    get_reactions(lc_ptr, p1, p2, x1, x2);
    lc_ptr = lc_ptr->next_lc; /* next load case */
    takedown(lc_ptr = takedown(lc_ptr->next_lc);
    fil_ptr = fil_ptr->next;
}
}
nl_ptr = nl_ptr->next;
/*
printf("in recurse nl_ptr->next =\n", nl_ptr);
*/
/* Will this loop work correctly? Check get_load() */

L = length(member_ptr);
lc_ptr = member_ptr->loads;
while(lc_ptr != NULL) {
    while((load_ptr = get_load(lc_ptr)) != NULL) {
        /* while there are more loads*/
        p1 = -(load_ptr->begin_mag);  /* positive load acts */
        p2 = -(load_ptr->end_mag);    /* downward. This */
        x1 = load_ptr->begin_x * L;   /* should be changed */
        x2 = load_ptr->end_x * L;     /* load positions are in*/
        get_reactions(lc_ptr, p1, p2, x1, x2);   /* local coord */
        /* increment end moment and shear*/
    }
    lc_ptr = lc_ptr->next_lc; /* ***** new ***** */
}
}
lc_ptr = member_ptr->loads;
while(lc_ptr != NULL) {  /* sum_moment & total_load are macros */
    if(member_ptr->end_node->flags.cantilever
        && member_ptr->flags.cantilever) {
    }
    else if(member_ptr->begin_node->flags.cantilever
        && member_ptr->flags.cantilever) {
reverse(member_ptr, lc_ptr); /* make end fixed, not begin */
}

else { /* simply supported member */
  lc_ptr->end_reac.shear = -sum_moment/L;
  lc_ptr->begin_reac.shear = total_load + sum_moment/L;
  lc_ptr->end_reac.moment = 0.00;
  lc_ptr->begin_reac.moment = lc_ptr->end_reac.shear * L + sum_moment;

  /* member is simply supported so moments balance to 0 */
}

lc_ptr = lc_ptr->next_lc;

/* Check here to see if there are any cantilevers opposite the ends */
/* of this member. */

adjust_for_cantilever(member_ptr);

member_ptr->flags.solved = 1;
return;

/* There's a problem with the summation since this is a */
/* recursive function. get_reactions() will get called in the */
/* recursion of the upper loop before getting to the lower loop */
/* This problem is solved by summing in the loadcase of the */
/* member being analyzed. */
/* This function is named reset(). It performs analysis of */
/* a floor using the recurse() function */

#define floor_ptr

reset(/*floor_ptr*/)

/* struct floor *floor_ptr */
{
    extern struct member *member_head;
    struct member *member_ptr;

    member_ptr = member_head;
    while(member_ptr != NULL) {
        member_ptr->flags.solved = NO;
        member_ptr = member_ptr->next;
    }
    return;
}
/* Change a cantilever fixed at the beginning into an equivalent */
/* cantilever fixed at the end. */

#include "strall2.h"

reverse(m_ptr, lc_ptr)

struct member *m_ptr;
struct load_case *lc_ptr;

{
    double length();

    if(lc_ptr == NULL || m_ptr == NULL) {
        return(0);
    }

    lc_ptr->end_reac.shear = lc_ptr->begin_reac.shear;
    lc_ptr->end_reac.moment = lc_ptr->begin_reac.moment +
    lc_ptr->begin_reac.shear * length(m_ptr);
    lc_ptr->begin_reac.shear = 0;
    lc_ptr->begin_reac.moment = 0;

    return;
}
/* This file is called right_of.c. The function right_of() takes the */
/* homogeneous representation for a point and for the two end points of */
/* a line segment and checks to see if the point is to the right of the */
/* segment. */

#include "straight.h"

double right_of(p, p1, p2)

h_point *p, *p1, *p2;               /* currently assumes 2d graphics*/
{
    extern int exkludge;
    double checksum, coeffx, coeffy, coeffz, coeffw;
    /* see Pavlidis p. 328 */

    if(p->w != 1 && p->w != 0) {
        /* ensure that all w components */
        /* are positive */
        p->x /= p->w;
        p->y /= p->w;
        p->z /= p->w;
        p->w /= p->w;
    }
    if(p1->w != 1 && p->w != 0) {
        p1->x /= p1->w;
        p1->y /= p1->w;
        p1->z /= p1->w;
        p1->w /= p1->w;
    }
    if(p2->w != 1 && p->w != 0) {
        p2->x /= p2->w;
        p2->y /= p2->w;
        p2->z /= p2->w;
        p2->w /= p2->w;
    }

    coeffx = p1->y - p2->y;
    coeffy = p2->x - p1->z;
    coeffz = 0;
    coeffw = p1->z * p2->y - p1->y * p2->x;

    checksum = p->x * coeffx + p->y * coeffy + p->z * coeffz + p->w * coeffw;
    return(checksum);               /* checksum < 0 p is to right */
    /* checksum = 0 p is on p1-p2 */
    /* checksum > 0 p is to left */
/* This file is named (SAVEFLOOR.C) The save_floor() function */
/* saves a floor structure to a disk FILE */

#include "strall2.h" /* standard header for functions*/

save_floor(f_number)

int f_number;           /* not used currently */

{
    extern struct node *node_head;
    extern struct member *member_head;
    struct node *node_ptr;
    struct member *member_ptr;
    struct frame_in_list *filf_ptr;
    struct node_list *nlf_ptr;
    struct load *load_ptr;
    FILE *fopen();
    FILE *nodf_ptr,*memf_ptr,*filf_ptr,
    *nlf_ptr,*loadf_ptr;          /* func returns file ptr */

    nodf_ptr = fopen("nodelist.out","w");
    memf_ptr = fopen("memblist.out","w");
    filf_ptr = fopen("fil_list.out","w");
    nlf_ptr = fopen("nl_list.out","w");
    loadf_ptr = fopen("loadlist.out","w");

    node_ptr = node_head;              /* write node file */
    while(node_ptr != NULL) {
        fprintf(nodf_ptr,"%ld %ld %ld %ld %ld %ld
", node_ptr->node_number,
                node_ptr->x, node_ptr->y, node_ptr->z);
        fprintf(nodf_ptr,"%lu %lu %lu %lu ", node_ptr->flags.node_type,
                node_ptr->flags.fix_x, node_ptr->flags.fix_y,
                node_ptr->flags.fix_z);
        fprintf(nodf_ptr,"%lu %lu %lu ", node_ptr->flags.fix_rx,
                node_ptr->flags.fix ry, node_ptr->flags.fix rz);
        fprintf(nodf_ptr,"%lu %lu %lu ", node_ptr->flags.cantilever);
        fprintf(nodf_ptr,"%d %d %d
", node_ptr->next_node->node_number,
                node_ptr->last_node->node_number,
                node_ptr->on_member->member_number,
                node_ptr->on_column);
        node_ptr = node_ptr->next_node;
    }

    member_ptr = member_head;          /* write member file */
    while(member_ptr != NULL) {
        printf("Saving member %d\n", member_ptr->member_number);
        fprintf(memf_ptr,"%d %ld %ld %ld %ld %ld %ld
", member_ptr->member_number,
                member_ptr->begin_reac(moment, member_ptr->begin_reac.shear,
                member_ptr->begin_reac.axial);
        fprintf(memf_ptr,"%ld %ld %ld ", member_ptr->end_reac.moment,
                member_ptr->end_reac.shear, member_ptr->end_reac.axial);
        fprintf(memf_ptr,"%ld %ld %ld ", member_ptr->flags.solved);
        fprintf(memf_ptr,"%s %s
", member_ptr->section-table,
                member_ptr->section_index);
    }
*/
load_ptr = member_ptr->loads;
while(load_ptr != NULL) {
    /* write load list file */
    printf(lodf_ptr,"%d ", member_ptr->member_number);
    printf(lodf_ptr,"%d %d %d ", load_ptr->date_assigned,
           load_ptr->who_assigned, load_ptr->coord_flag);
    printf(lodf_ptr,"%lf %lf %lf %lf\n", load_ptr->begin_x,
           load_ptr->end_x, load_ptr->begin_mag,
           load_ptr->end_mag);
    load_ptr = load_ptr->next_load;
}

nl_ptr = member_ptr->frame_in;
while(nl_ptr != NULL) {
    /* write node list file */
    printf(nlf_ptr,"%d %d\n", member_ptr->member_number,
           nl_ptr->in_node->node_number);
    nl_ptr = nl_ptr->next;
}
member_ptr = member_ptr->next;
}

fclose(nodf_ptr);
fclose(memf_ptr);
fclose(lodf_ptr);
fclose(nlf_ptr);
fclose(lodf_ptr);
fclose(fill_ptr);

return;
/* Controller to try adjacency list functions */

#include "str112.h"

#define MAKEADJL
#undef ARRAY
#undef READADJL

struct member *member_head;
struct node *node_head, *node_tail;

testadjl() {
    char c, answer[10], *calloc();
    int keep_going, i, j, node1, node2;

    extern struct member *member_head;
    extern struct node *node_head;

    struct node *n1, *n2, *to_nptr();
    union pathnode *allpaths;
    adjnode *a_node;
    adjnode *adjlist;

    /* I'm still confused about proper declaration of a ptr to array*/
    /* See K&R p.199. */

    adjlist = (adjnode *) calloc(MAXV, sizeof(adjnode));
    allpaths = (union pathnode *)
        calloc(MAXPATH*MAXPATH, sizeof(union pathnode));

    printf("Calling member1p() adjlist = %u\t allpaths = %u\n",
        adjlist, allpaths);

    member1p(adjlist, allpaths);
}

/* This file is named testpaths.c. The function test() tests a pair of */
/* paths found in path() to see if they are legal. */

#include "strall12.h"

test(paths)
union pathnode  paths[2][MAXPATH];
{
    int i, j;
    h_point p1, p2, p3;
    double flag[2], right_of();

    /* process 1st path */
    for(j=0; j<2; j++) {
        i = 0;
        flag[j] = 0;
        while(flag[j] == 0) {
/*
    printf("(paths[j][i].node_ptr)->x %f\t\tpaths[j][i+1].node_ptr->x %f\n",
        (paths[j][i].node_ptr)->x, paths[j][i+1].node_ptr->x);
*/
            p1.x = (paths[j][i].node_ptr)->x;
            p1.y = (paths[j][i].node_ptr)->y;
            p1.w = 1;
            p2.x = (paths[j][2+i].node_ptr)->x;
            p2.y = (paths[j][2+i].node_ptr)->y;
            p2.w = 1;
            p3.x = (paths[j][3+i].node_ptr)->x;
            p3.y = (paths[j][3+i].node_ptr)->y;
            p3.w = 1;
            flag[j] = right_of(&p1, &p2, &p3);
            if(flag[j] == 0) { /* points are colinear */
                i++;
                /* keep going until a corner is */
                /* reached. */
            } else if(paths[j][2+i].node_ptr == 0) {
                break;
            }
        }
    }

    if(flag[0] * flag[1] < 0) { /* one path is to right, the */
        return(1); /* other to the left */
    } else if(flag[0] * flag[1] > 0) { /* the points are not colinear */
        return(0); /* so they must be a corner */
    }
/* More complicated error checking could be used here to indi- */
/* cate whether the path crosses from left to right of the */
/* member. This could be done by using path(j)[1] & path(j)[n-1] to test all of the points on a path to see if right_of() returns the same sign (or zero) for all of the path nodes. */
/* This function is called to_global.c. It converts global coordinates */
/* to global coordinates for a specified member. */

#include "strall2.h"  /* standard header for functions*/

double *to_global(member_ptr, xl, yl, zl)
struct member *member_ptr;
double xl, yl, zl;
{
  double global[3], length();

  global[0] = (xl/length(member_ptr))*
     (member_ptr->end_node->x - member_ptr->begin_node->x);
  global[1] = (xl/length(member_ptr))*
     (member_ptr->end_node->x - member_ptr->begin_node->y);
  global[2] = (xl/length(member_ptr))*
     (member_ptr->end_node->x - member_ptr->begin_node->z);

  return(global);
}
/* This function converts an lc number to the pointer to the lc */
/* If the lc is not found, a new one is created by calling add_lc() */

#include (stdio.h)
#include "strai12.h"

struct load_case *to_lcptr(lc_number, member_ptr)

int lc_number;
struct member *member_ptr;            /* ptr to memb */

{
    struct load_case *last_lc, *this_lc; /* ptr to current memb */
    struct load_case *add_lc();
    struct member *to_mptr();

    if(lc_number == 0 || member_ptr == NULL) /* if passed null number */
        this_lc = NULL;
    else {
        this_lc = member_ptr->loads;

        while(this_lc != NULL && this_lc->lc_number != lc_number) {
            last_lc = this_lc;
            this_lc = this_lc->next_ic;
        }

        if(this_lc == NULL) {                  /* lc ptr not found */
            this_lc = add_lc(member_ptr);    /* create a new lc */
            this_lc->lc_number = lc_number;
            this_lc->next_ic = NULL;
        }
    }
    return(this_lc);
}
/* This function is called to_local.c. It converts global coordinates */
/* to local coordinates for a specified member. */

#include "strall2.h"
#include "math.h"

double *to_local(member_ptr, global)
{
    struct member *member_ptr;
    double global[];

    double length(), xlength, ylength, zlength;

    xlength = global[0] - member_ptr->begin_node->x;
    ylength = global[1] - member_ptr->begin_node->y;
    zlength = global[2] - member_ptr->begin_node->z;

    global[0] = sqrt(xlength*xlength + ylength*ylength + zlength*zlength);
    // length(member_ptr);
    global[1] = 0.0;
    global[2] = 0.0;

    return(global);
}

/* This routine assumes that the coordinate desired is the distance */
/* of (xg, yg, zg) along the member (*member_ptr). */
/* This function converts a member number to the pointer to the member */
/* Returns NULL pointer if member is not found */

#include "strail2.h"   /* standard header for functions*/

to_mnum(member_ptr)

struct member *member_ptr;
{
    extern struct member *member_head;
    struct member *this_member;
    this_member = member_head;
    while(this_member != member_ptr && this_member != NULL)
        this_member = this_member->next;

    if(this_member == NULL)    /* member not found */
        return(0);
    else
        return(this_member->member_number);
}
/* This function converts a member number to the pointer to the member */
/* Returns NULL pointer if the member is not found */

#include "strail2.h" /* standard header for functions*/

struct member *to_mptr(member_number)

int member_number;
{
    extern struct member *member_head; /* ptr to head of memb */
    struct member *this_member; /* ptr to current memb */
    struct member *add_member();

    if(member_number == 0)
        this_member = NULL;
    else
    {
        this_member = member_head;
        while((this_member->member_number != member_number
              && this_member != NULL) (this_member = this_member->next;
    }
    if(this_member == NULL) /* memb ptr not found*/
        this_member = add_member(); /* create a new member */
        memb_init(this_member);
        this_member->member_number = member_number;
    
    return(this_member);
}
/* This function converts a node pointer to the number to the node */
#include "strall2.h"
to_nnum(node_ptr) /* returns an integer */
struct node *node_ptr;
{
    extern struct node *node_head;
    struct node *this_node; 

    this_node = node_head;
    while(this_node != node_ptr && this_node != NULL)
        this_node = this_node->next_node;
    if(this_node == NULL) /* node not found */
        return(0);
    else
        return(this_node->node_number); /* node found */
}
/* This function converts a node number to the pointer to the node */
#include "strall2.h"

#undef DEBUG

struct node *to_nptr(node_number) /* returns ptr to node */
int node_number;
{
    extern struct node *node_head;
    struct node *this_node;
    struct node *add_node();

    if(node_number == 0)
        this_node = NULL;
    else {
        this_node = node_head;
        while((this_node->node_number != node_number)
             && this_node != NULL)
            this_node = this_node->next_node;

        if(this_node == NULL) { /* if node ptr not found */
            this_node = add_node(); /* create a new node */
            node_init(this_node);
            this_node->node_number = node_number;
        }
    }

    return(this_node); /* node found */
/* This file is called Zwayarea.c. This function takes a member and its list of circuits, and creates a tributary area polygon assuming a two way flooring system. */

#include "strarr2.h" /* standard header for functions*/
#include <math.h>

struct area *twoway(member_ptr, all_paths)

struct member *member_ptr;
union pathnode all_paths[][MAXPATH]; /* must straighten this out */

{
    int i, j, length, n;
    char *malloc();
    h_line bisect[MAXPATH];
    h_point point1, point2, point3, *intersect(), *p1, *p2, *p3;
    double lx, ly, l1, l2, l3;
    struct area *area1, *area_head, *last_area;
    struct corner *corner1, *corner_head;
    struct vector {
        double dx, dy;
    } v[MAXPATH];

    i = j = length = n = 0;

    area1 = (struct area *) malloc(sizeof(struct area));

    while(all_paths[j][0].length != NULL) {

        if(j == 0) {
            area_head = area1;
        }

        /* find slopes for sides of the all_paths[j] */

        length = all_paths[j][0].length;
        for(i=1; i <= length; i++) {
            n = i%length; /* modulo division */
            v[i].dy = all_paths[j][n+1].node_ptr->y
                - all_paths[j][i].node_ptr->y;
            v[i].dx = all_paths[j][n+1].node_ptr->x
                - all_paths[j][i].node_ptr->x;
            v[i].dy /= hypot(v[i].dx, v[i].dy);
            v[i].dx /= hypot(v[i].dx, v[i].dy);

        }

        /* find the bisectors at each corner */
/* see interesting notes below about how many inter- */
/* sections are needed.

length = all_paths[j][0].length;
for(i=1; i <= length; i++) {
    n = i % length;
    bisect[n+1].x = v[n+1].dy - v[i].dy;
    bisect[n+1].y = v[i].dx - v[n+1].dx;
    bisect[n+1].w = -all_paths[j][n+1].node_ptr->x * bisect[n+1].x
                    - all_paths[j][n+1].node_ptr->y * bisect[n+1].y;
}

/* First corner is the begin node of the path */
corner_head = (struct corner *) malloc(sizeof(struct corner));
area->corner = corner_head;
corner_head->x = all_paths[j][1].node_ptr->x;
corner_head->y = all_paths[j][1].node_ptr->y;
corner1 = (struct corner *) malloc(sizeof(struct corner));
corner_head->next = corner1;

/* Find the next corner(s) */

i = 1;
p1 = intersect(&bisect[i], &bisect[length], &point1);
/* K&R p. 129*** */
lx = (all_paths[j][i].node_ptr->x - p1->x);
ly = (all_paths[j][i].node_ptr->y - p1->y);
l1 = hypot(lx, ly);

p2 = intersect(&bisect[i], &bisect[i+1], &point2);
lx = (all_paths[j][i].node_ptr->x - p2->x);
ly = (all_paths[j][i].node_ptr->y - p2->y);
l2 = hypot(lx, ly);

if(l1 <= l2) {
    /* Area is a triangle */
    corner1->x = p1->x;
    corner1->y = p1->y;
}
else {
    corner1->x = p2->x;
    corner1->y = p2->y;

    i = length;
    lx = (all_paths[j][i].node_ptr->x - p2->x);
    ly = (all_paths[j][i].node_ptr->y - p2->y);
l2 = hypot(lx, ly);

    p3 = intersect(&bisect[i], &bisect[i-1], &point3);
lx = (all_paths[j][i].node_ptr->x - p3->x);
ly = (all_paths[j][i].node_ptr->y - p3->y);
l3 = hypot(lx, ly);

corner1->next = (struct corner *)
    malloc(sizeof(struct corner));
corner1 = corner1->next;
if(l3 <= l2) {
    corner1->x = p3->x;
    corner1->y = p3->y;
}
else {
    corner1->x = p2->x;
    corner1->y = p2->y;
}

/* Last corner is end node of the path */

corner1->next = (struct corner *)
    malloc(sizeof(struct corner));
corner1 = corner1->next;
corner1->x = all_paths[j][length].node_ptr->x;
corner1->y = all_paths[j][length].node_ptr->y;

/* make list circular */

j++;
/* continue while to do next path */
area1->next = (struct area *) malloc(sizeof(struct area));
last_area = area1;
area1 = area1->next; /* list linking adds one too many areas */

last_area->next = NULL; /* area1's memory should be freed */
return(area_head);

/* Interesting notes to myself: */
/* I only need to make two checks: is the distance to the */
/* intersection of the bisectors at node 1 and node 2 less than */
/* the distance to the intersection of the bisectors at node 1 */
/* and node n; and the same check for 4-1 and 3-4. */
/* If the path has four members, then only one check is */
/* necessary to decide whether the area contribution is a */
/* triangle or a polygon. Life gets sticky if there are more */
/* than four sides. However, I cannot tell from the path */
/* how many sides there are. I could sum the interior angles */
/* as I go. Does this cause problems on members with multiple */
/* nodes? Circuit/path definition requires more thought in this*/
/* case. */
/* This file is called unique.c. This function accepts an array of */
/* pointers to the circuit lists for a physical member, and compares */
/* the circuits to ensure that only unique ones are kept. */

/* standard header for functions */

#include "strall2.h"

unique(allpaths, i)
    /* first item in a path */
    /* gives its length (# edges) */

union pathnode allpaths[i][MAXPATH];
int i;

{
    int j, k, l, m, n, length, linear, match,
        nomatch, same, duplicate_flag[2];
    struct node *n1, *n2, *n3, *nomatches[MAXPATH];

    j = k = l = m = n = length = match = nomatch = same = 0;
    duplicate_flag[0] = duplicate_flag[1] = NO;

    while(allpaths[i][0].length != NULL) {
        /* if two lists are the same length, check to see if */
        /* all of their nodes are the same. */
        for(j=0; j<i; j++) {
            /* check against all existing */
            /* paths */
            if(duplicate_flag[n] == YES) {
                break;
            }
        }

        length = allpaths[i][0].length;
        if(allpaths[i][0].length == allpaths[j][0].length) {

            /* Compare every node in the ith list */
            /* to every node in the jth list */
            /* until all have been compared */
            nomatch = match = 0;
            for(l=1; l<length; l++) {
                same = NO;
                k = 0;
                while(same == NO & & k<length) {
                    k++;
                    same = (allpaths[i][l].node_ptr ==
                        allpaths[j][k].node_ptr);
                }

                /* all nodes must match to be same */
                if(same == YES) {
                    match++;
                } else {
                    /* no match was found */
                    nomatch++;
                    nomatches[nomatch] =
                        allpaths[i][l].node_ptr;
                }
            }
        }
    }
}
/* if the paths are the same, then any unmatched */
/* nodes must be colinear with the begin and */
/* end nodes of the graph member */

if(nomatch == 0) {
    duplicate_flag[n] = YES;
}
else if(match >= 3) {
    linear = 1;
    n1 = allpaths[i][1].node_ptr;
    n2 = allpaths[i][length].node_ptr;
    for(m=1; m<=nomatch; m++) {
        n3 = nomatches[m];
        linear = corner(n1, n2, n3);
        if(linear != 0) {
            printf("different paths\n");
            break;
        }
    }
}
else
    duplicate_flag[n] = NO;

}
i++;
n++;
APPENDIX D

Data Structure Documentation
AREA DATA STRUCTURE

double area

pointer to corner

pointer to next area
COLUMN DATA STRUCTURE

column data structure

double x, y, z

double orientation

int column number

pointer to next column
CORNER DATA STRUCTURE

corner data structure

double \( x, y, z \)

pointer to next corner
DESIGN_VALUES DATA STRUCTURE

- design_values
  - data
  - structure

- double
  - moment

- double
  - shear

- double
  - axial
FLOOR DATA STRUCTURE

floor data structure

pointer to member

pointer to node

pointer to column

pointer to f_load (floor loads)

pointer to f_load (openings)
FRAME_IN_LIST DATA STRUCTURE

: frame_in_list :
  data
  structure

: pointer to member :

: pointer to next :
  frame_in_list
LISTNODE DATA STRUCTURE

listnode
  data
  structure

pointer to node

pointer to next listnode
NODE DATA STRUCTURE

- node data structure
  - double coordinates x, y, z
    - pointer to on member
      - fix x, y, z
        - cantilever flag
  - integer node number
    - pointer to frame in list
      - fix rx, ry, rz
        - node_type flag
  - pointer to loads
    - pointer to next node
    - pointer to previous node
    - bit flags
NODE_LIST DATA STRUCTURE

- node_list
  - data
  - structure

- pointer to node

- pointer to next
  - node_list
Note: This data structure is a union - it may contain either an integer or a pointer.
GNODE DATA STRUCTURE

node data structure

int parent

pointer to next qnode
REACTION DATA STRUCTURE

- reaction data structure
  - moment
  - shear
  - axial
SECTION DATA STRUCTURE

- section
  - data
    - structure
      - char[10] name
      - char[10] table
      - int index in table
APPENDIX E

Data Structure Listings
.8 December 1984 THESIS DATA STRUCTURES Bleakley

********* defines for graph construction and manipulation. ***********/

fine NO 0
fine YES 1
fine NULL 0
fine MAXV 1000
fine MAXQUE 256
fine MAXPATH 10

******************** Type Definitions ********************

edef struct h_coord {
    double x;
    double y;
    double z;
    double w;
    _point, h_line;
}

edef struct harea {
    h_point *point;
    struct harea *next;
    _area;
}

***************** Data Structure Definitions *******************/

struct node {
    int node_number;
    double x,y,z;
    struct {
        unsigned node_type : 1;
        unsigned fix_x : 1;
        unsigned fix_y : 1;
        unsigned fix_z : 1;
        unsigned fix_rx : 1;
        unsigned fix_ry : 1;
        unsigned fix_rz : 1;
        unsigned cantilever : 1;
    } flags;
    struct load *loads;
    struct frame_in_list *frame_in;
    struct node *next_node;
    struct node *last_node;
    struct member *on_member;
    struct column *on_column;
};

/* This header defines the structure for members framing into */
/* a joint(node) or into a member */
struct frame_in_list {
    struct member *in_member; /* member that frames in*/
struct frame_in_list *next; /* next member that */ /* frames to joint */
};

/* This header file declares a data structure for member */ /* reactions. Use with <member.h> */

struct reaction {
    double moment; /* end moment on mem */
    double shear; /* end shear on mem */
    double axial; /* -1 global coord */
};

struct load {
    int date_assigned; /* date load assigned */
    int who_assigned; /* user who assigned */
    int coord_flag; /* -1 global coord */
    double begin_x, end_x; /* +1 local coord */
    double begin_mag, end_mag; /* */
    struct load *next_load; /* pointer to next load */
};

/* It might be a good idea to include a pointer that could */ /* indicate the source of the load e.g. a member, a floor */ /* live load, or superimposed dead load. */ /* This structure only allows distributed loads along */ /* local x-axis (the long axis of the member). The load must */ /* be in the local z direction. For floor framing, the local */ /* z direction should always coincide with the global z axis. */ /* All loads are represented using the same data structure */ /* point loads are just special cases of distributed loads. */

struct node_list {
    struct node *in_node; /* pointers to nodes on */
    struct node_list *next; /* the member */
};

/*************************************************************/

struct load_case {
    int lc_number; /* # # # # # : # # # - member # */
    /* # - load case */
    struct reaction begin_reac; /* begin reactions */
    struct reaction end_reac; /* end reactions */
    struct load *loads; /* pointer to load list */
    struct load_case *next_lc; /* pointer to next lc */
};

/*************************************************************/

/* This header file contains the declaration of a structure */ /* member representation */

struct corner {
    double x, y, z;
    struct corner *next;
};

struct area {
    double area;

struct corner
  *corner;
struct area
  *next; /* * /
};

struct member {
  int member_number; /* */
  struct {
    unsigned solved : 1; /* solved flag */
    unsigned fiction : 1; /* fictitious edge beam */
    unsigned cantilever : 1; /* cantilever flag */
  } flags;
  struct section *section; /* selected section */
  struct load_case *loads; /* load_case list */
  struct node_list *frame_in; /* list of nodes on memb*/
  struct node *begin_node; /* pointer to begin & */
  struct node *end_node; /* end nodes */
  struct area *area;
  struct member *next;
  struct member *last;
};

struct section {
  char name[101]; /* name of section */
  char table[10]; /* name of table to look*/
  int index; /* number of entry in */
  /* table */
};

/* This header file declares a data structure for design */
/* moments and shears. Use with <member.h> */

struct design_values {
  double moment; /* largest moment on mem*/
  double shear; /* largest shear on mem */
  double axial; /* largest axial force */
};

/* It should be possible to calculate these quantities from */
/* the known end reactions and the loads. */

/* This header defines the structure for columns */

struct column {
  int column_number; /* change to alpha-num? */
  double x,y,z; /* global coordinates */
  double orientation; /* allow arbitrary ??? */
  struct column *next_column; /* pointer to next col */
};

/* Column is a special case of a node. */

/* This header defines the structure for floor openings */
struct openings {
    double x[4], y[4]; /* specify 4 corners */
};

/* Limited to polygonal openings - should be adequate */

/* This header defines the structure for floor loads */
struct f_load {
    int date_assigned;
    int who_assigned;
    int coord_flag; /* -1 global +1 local */
    double magnitude; /* constant over area */
    h_area *area;
    struct f_load *next;
} *fload_head, *opening_head;

/* This header defines the framing plan for a complete floor */

struct deck_structure {
    char frame_type[15]; /* type of framing */
    struct member *member_head; /* pointer to memb list */
    struct node *node_head; /* pointer to node list */
    struct f_load *fload_head; /* pointer to loads */
};

/* This header defines the structure for a complete floor */

struct floor {
    struct member *member_head; /* pointer to memb list */
    struct node *node_head; /* pointer to node list */
    struct column *column_head; /* pointer to column list */
    struct f_load *opening_head; /* pointer to opening list */
} *floordata; /* pointer to floor */

/***************************** Typedef and data structures for graph manipulation *****************************/

typedef struct listnode {
    struct node *node_ptr;
    struct listnode *next;
} adjnode, circuit_node, node_list, qnode;

union pathnode {
    int length; /* used to store paths */
    struct node *node_ptr; /* between nodes */
};

struct qnode {
    int parent; /* used to make priority */
    struct node *node_ptr; /* que in breadth list */
};