CAD FILES AND THE COMPUTERIZED DESIGN OF BEAMS AND GIRDERS

by

Pierre Gasztowtt

Eleve-Ingenieur, Ecole Nationale des Ponts et Chaussess

SUBMITTED TO THE DEPARTMENT OF CIVIL ENGINEERING IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

April 1986

Copyright (c) 1986 Massachusetts Institute of Technology

Signature of Author ______________________________

Department of Civil Engineering
April 7, 1986

Certified by ______________________________

Professor John.H. Slater
Thesis Supervisor

Accepted by ______________________________

Professor Ole Madsen
Chairman, Departmental Graduate Committee

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Archives JUN 02 1986

CAD FILES AND THE COMPUTERIZED DESIGN OF BEAMS AND GIRDERS

by

Pierre Gasztowtt

Submitted to the Department of Civil Engineering on April 7, 1986 in partial fulfillment of the requirements for the degree of Master of Science.

Abstract

In this thesis, a set of analysis programs is presented, which link a computer aided drafting system (CAD) with a structural analysis and design package for the automatic design of floor systems. The work builds on FLOOR, a program for the computerized analysis of floor systems, presented by Bleakley in his 1984 MIT Civil-Engineer's Degree thesis.

Since its conception, FLOOR had been missing a user interface. In addition to a revised version of the original FLOOR program, this thesis presents new programs to provide this interface. These new programs:

- read the framing plan representing a floor layout from a CAD file,
- generate a linked data structure describing this floor layout in terms adequate to the FLOOR program,
- analyse and design the floor system and write the results to a generic CAD file, for subsequent drafting.

FLOOR and the design interface package are written in C and operate under UNIX and PC/DOS for the IBM/PC-AT computer.

Thesis Supervisor: Professor John.H. Slater
Title: Assistant Professor of Civil-Engineering

Dedication

To Al Bleakley.

Acknowledgement

I am grateful to Prof. Slater, my thesis advisor, for making this work possible; to Prof. Logcher, my faculty advisor, for letting me know about Al Bleakley's work; to Mr. Steve Lew, from Weidlinger Associates, for exposing me to professional practice; and to Mr. Matthew Alexander, from the Department of Linguistics and Philosophy, and Ms. Judith Sanders, from the Writing and Communication Center, for sharing with me their skillful knowledge of the English language.

Table of Contents

Abstract 2
Dedication 3
Acknowledgement 4
Table of Contents 5
List of Figures 7

1. Introduction 8
1.1 Computers in Structural Engineering 8
1.2 The computerized design of floor-beams and girders 9
1.3 The purpose and organization of this thesis 10

2. Program Flow 12
2.1 The recursive structure of a floor layout 12
2.2 Data flow 16
2.2.1 Recursive nature of the analysis 18
2.2.2 Identification of member tributary areas 18
2.2.3 Creation of member loads 19
2.2.4 Load take-down and member sizing 21
2.3 Memory allocation 22

3. Analysis and sizing modules 23
3.1 Analysis module 23
3.1.1 The initial analysis module 23
3.1.2 Engineering practice 24
3.1.3 Data structure representing member diagrams 24
3.1.4 Access to significant quantities 25
3.1.5 Member solving 28
3.2 Member sizing 28

4. The initial mode of data entry 31
4.1 The initial situation 31
4.2 Automatic generation of node-lists and frame-in-lists 32
4.3 Evaluation 32

5. An interactive Interface 34
5.1 Simultaneous display with on-line editing and plotting 34
5.2 Entry of data with a mouse 35
5.2.1 Creating new nodes with the mouse 35
5.2.2 Automatic node generation 36
5.3 Node look-up algorithm 37
5.3.1 Selecting a point on the display with the mouse 37
5.3.2 Selecting a segment with the mouse 38

5.3.3 Final algorithm 40
5.4 Drawing aids 41
5.5 In summary 42

6. Reading data from CAD files **44**
6.1 The programmer's profits 44
6.1.1 From CAD to FLOOR 44
6.1.2 From FLOOR to CAD 45
6.2 CAD files, vehicles of information 46

7. Communication with CAD files: the process **48**
7.1 Reading floor layouts 48
7.1.1 Lifting the requirement of a methodical order 48
7.1.2 Approximate locations of the member extremities 49
7.1.3 A new data structure: the entities 50
7.1.4 Order of growth 50
7.1.5 Treatment of cantilevers 51
7.2 Reading cantilevers, openings, deck-directions and loads 52
7.3 Actual implementation with Autocad 53
7.3.1 The reasons for the choice 53
7.3.2 Contents of a suitable file 53
7.3.3 A customized AUTOCAD 55
7.4 Writing the results of the analysis 57

8. Future developments **58**
8.1 Additional analysis capabilities 58
8.2 Interface with sophisticated CADD 59
8.3 FLOOR with FLODER 60
8.3.1 FLODER 60
8.3.2 The limitations of FLODER 61
8.3.3 Interfacing FLOOR and FLODER 61

Appendix A. Program Listings **63**

Appendix B. Design Example **183**

Appendix C. AUTOCAD Definition of the User Commands **190**
Index **194**

List of Figures

Figure 2-1: A floor layout data structure 15
Figure 2-2: The modular nature of the FLOOR program 17
Figure 2-3: The several areas adjacent to a member 18
Figure 2-4: clipping algorithm 21
Figure 3-1: Diagram data structure 27
Figure 5-1: Points projecting on a segment 39
Figure 5-2: Distance between a point and a segment 40
Figure 7-1: Trimmed segments 49
Figure 7-2: Contents of a suitable file 54
Figure 7-3: Customized Autocad commands 56

Chapter 1

Introduction

1.1 Computers in Structural Engineering

The fundamental problem of the structural engineer is to find how to most economically transfer loads distributed in space to a support or a foundation, given a number of architectural constraints. For a number of reasons, practical as well as cultural, buildings differ one from the other. Because the architectural constraints are different in each case, the design of each new building requires the full attention of the structural engineer.

Structural analysis is a thoroughly studied field and there are only a limited number of construction materials and techniques: most buildings are built of wood, brick, reinforced concrete, or steel, and these materials are assembled in certain standard ways. In any building, a beam remains a beam. As a result, in project after project, after he has envisioned a structural scheme, the structural engineer has to repeat computations that follow identical patterns.

Over the last two decades, computers have proven to be invaluable tools in automating these computations. Well suited to repetitive tasks, extremely accurate, and quick at retrieving information stored in tables, computers have improved the designs of the engineer by letting him conveniently compare multitudes of different systems.

They have, however, left the engineer with three sets of tasks that, simple as they may be, are nonetheless time-consuming:

- assessing the exact problem and figuring the exact design loads;

- entering data and commands into the machine;
- copying the conclusions of the analysis from a print-out on to a drawing.

The time consumed by these tasks can actually discourage an engineer from using a computer for solving small problems.

1.2 The computerized design of floor-beams and girders

This shortcoming of the computer is best illustrated by the design of the individual beams and girders that make up a flooring system. A girder supports adjacent loaded areas, as well as beams that frame onto it and support, in turn, more loaded areas. Several loads are carried by the girder: line loads originating from the loaded areas next to the girder, and point loads originating from the beams framing onto it. The choice of adequate steel sections for such girders is a function of the maximal moment in them. One could hardly find in the field of structural engineering more repetitive a task than the design of these beams and girders. However, because a simple multiplication gives the contribution of each of these loads to this maximal moment, the computer is actually of little help: entering for each of these beams or girders, each of these loads in the computer and reading the results takes more time than doing the computations manually.

In his 1984 thesis [Bleakley 84], Bleakley addressed the problem of the computerized design of floor beams and girders, attempting to computerize the entire process, from load assessment to member sizing. He presented the FLOOR program which, given a series of area loads distributed over a floor, and some adequate information about the layout of the floor, would find out all the loads arriving on each beam or girder. Properly combined with FLOOR, any member-sizing program can size these beams and girders --automatically.

1.3 The purpose and organization of this thesis

A number of lacunae, quite excusable in view of the novelty of the enterprise and the size of the computer code, unfortunately reduced the usefulness of the version of the FLOOR program actually developed. It required that the user input quite a large amount of data, in the rather awkward form of a series of text files. An inconsistency in the code actually made this amount of data excessive. In the case of one-way deck flooring systems, it made no provisions for local changes in the direction of the deck-span, and the algorithms used could lead to wrong results in certain cases. It placed restrictions on the shape of the area-loads that could be specified. Because it did no compaction of the allocated memory, it could not be run on a personal computer. Finally, falling short of computing the maximal moment in the members and from sizing them, it would not produce any immediately useful results.

This thesis is an attempt to remedy these shortcomings. In particular, a program is presented for the automatic capture of data for FLOOR from CAD files, including an interface between a CAD system and FLOOR, and resulting in an integrated, operative, package for the design of floor beams and girders. Because automatic data capture from CAD files could also be useful for frame-analysis, some of the techniques developed in this thesis may be useful to programs other than FLOOR.

The remainder of this thesis is organized in seven chapters.

The first two chapters present an overview of the FLOOR program. In Chapter 2, the flow of data and the data structures in the FLOOR program are described, and a strategy for the management of memory is suggested. (Because it focuses on data flow and data structures, and because several local modifications to

the FLOOR program have been introduced, the description completes more than it repeats, the original documentation by Bleakley.) In Chapter 3, the logic of new routines added to complete the FLOOR program is discussed: these are the routines that compute the maximal moment in the members and size them accordingly.

The following four chapters presents the particulars of the programs introduced to replace the user interface FLOOR had been missing. Chapter 4 describes the present input in the FLOOR program, and recapitulates the data required. Chapter 5, which discusses what an ideal interactive interface should do, has a dual objective:

- to introduce some primary motivations for the data capture from a CAD file,
- to introduce some algorithms that, minimizing the input from the user, will also provide the means of this data capture.

Chapter 6 further discusses the motivations for the exchange of data between FLOOR and a CAD system, while Chapter 7 presents the implementation of this exchange of data. (In particular, Chapter 7 describes the contents of a suitable AUTOCAD CAD file for input to the extended FLOOR program, and the alterations FLOOR writes to it.)

Finally, in Chapter 8, possible future developments of the FLOOR program are suggested, and a comparison is made with FLODER, an expert-system for the design of flooring systems, developed at Carnegie-Mellon University [Karakatsanis 85]. Appendix A lists the program documentation and code, while Appendix B presents as an example the results of an actual FLOOR analysis. Appendix C lists the definition of customized user commands for use with AUTOCAD.

Chapter 2

Program Flow

FLOOR is an original combination of graph theory and graphics processing algorithms. These will not be described in detail here: the interested reader is referred to the thesis by Bleakley [Bleakley 84]. Instead, focus will be on the flow of data through the program, and on how the data structures in FLOOR reflect this flow.

2.1 The recursive structure of a floor layout

A floor layout is a hierarchical arrangement of columns, and simply supported girders and beams, that could be extended on an infinite number of levels. At its top level are the columns, which are nodes into which main girders frame. These main girders have connections distributed on them, which are more nodes into which, in turn, secondary girders frame. These secondary girders may themselves have more nodes distributed on them, into which tertiary floor beams frame. Some of these tertiary floor beams may also have nodes distributed on them, into which more beams may frame; etc., although in practice, things will seldom go further than a tertiary system.

This nesting gives rise to the description in FLOOR of a floor layout structure in terms of two data structures which recursively define each other: nodes and members (C, the language in which FLOOR is written, allowing for the recursive definition of structures, this involves no technical difficulty). In a first description of the floor structure, a node will be defined by:

- a geometric location, in the form of a pointer to a very small data structure, called a point, which just contains the value of two coordinates;

- a list[1] of members framing into it.

In this description, a member, meanwhile, will be in turn defined by:

- two extreme geometric locations,

- a list of nodes distributed on it.

In terms of graph theory, this first description allows one to see the whole floor as a series of trees that share branches, and whose initial parents are the columns: to visit the whole floor, one has to start from the columns.

In the actual description of a floor layout, a node structure also contains the computer address of the member on which it is situated, while a member contains, instead of the locations of its extremities, the addresses of its extreme nodes: with this dual referencing it becomes possible to traverse the hierarchical structure entirely from any point.

Also, some members can be cantilevered. To distinguish these members from the others, the member data structure includes a flag, which is a variable that can be set equal to O or 1 --flag raised. In addition to being flagged (with a raised flag), the cantilevered member differs from the others in that:

- it does not figure in the frame-in-list of the node at its free end, while instead

- it counts this node in its node-list.

The node onto which a cantilever frames is also flagged. If this node is not a column, the member opposite the cantilever is to take a carry-over moment: consequently, a member will have room in its data structure for two pointers to two cantilevered members framing in it.

---

[1]The reader who would not be familiar with the concept of list is referred to [Abelson 85], [Kernighan 81] or [Bleakley 84].

Because the floor can be seen alternatively as a network of members and as a network of nodes, all the columns, the members and the nodes are organized as list data structures. A node structure thus contains the address of the next node structure, and a member structure contains the address of the next member structure.

This description of a floor layout, which corresponds to the input into the FLOOR program, is further structured in an introductory step of the FLOOR program. FLOOR, in this introductory step, determines for each node in turn its neighboring nodes[2] . In the final description of a floor layout data structure, each node has associated with it a list of neighbors, which are themselves other nodes[3] .

Figure 2-1 recapitulates the resulting organization of a floor data structure.

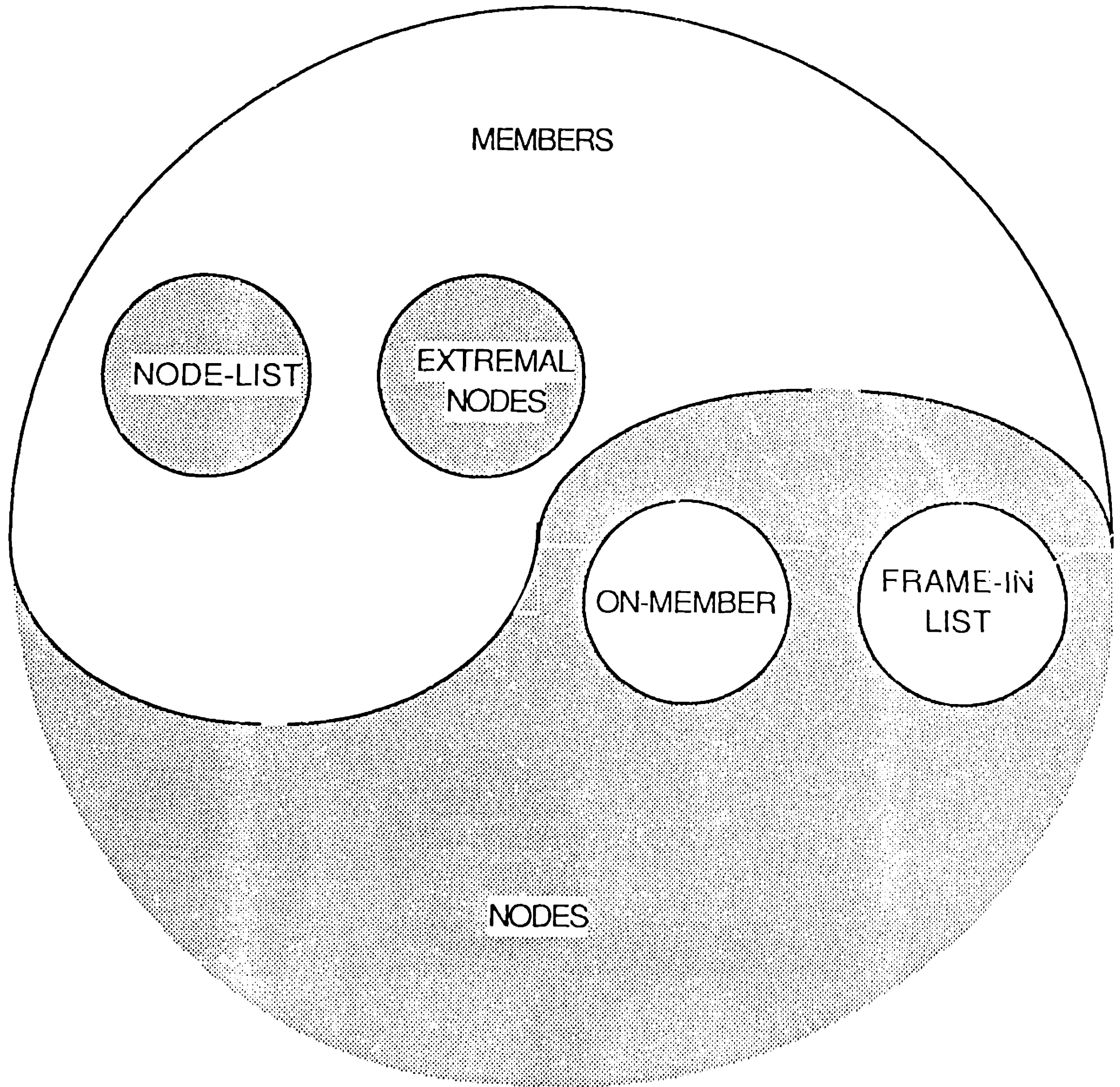
---

[2]This determination requires that:

- it be possible to visit the floor layout in both directions, and that
- the list of the nodes situated on any member be ordered.

[3]In the original version of the FLOOR program, these lists were stored in a one-dimensional array of N pointers –N being the number of nodes.

Figure 2-1: A floor layout data structure

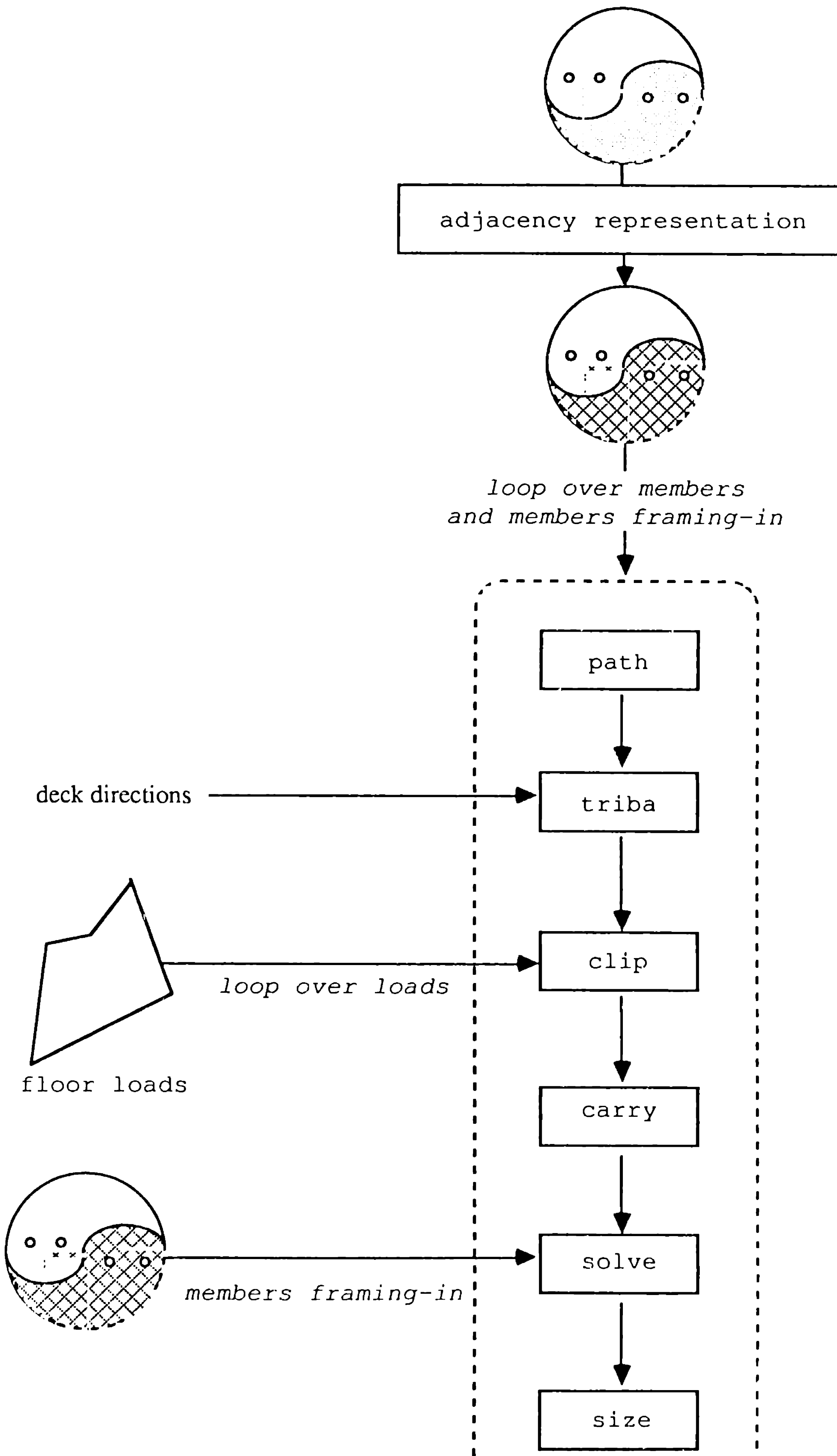


2.2 Data flow

After it has thus further structured the highly structured arrangement of columns, girders and beams, it was given as input, the FLOOR program proceeds to analyse the members. The analysis involves six steps:

- 1) the determination of the adjacent areas that on both sides of the member analyzed are enclosed by other members (or shortest paths along members from one node of this member to the next one),
- 2) the identification of the direction in which the deck spans over these areas (if the flooring system is made of a one-way deck) and the division of these areas into parts that are member tributary areas,
- 3) the intersection of these tributary areas with the floor loads,
- 4) the projection of these area-loads onto the supporting members,
- 5) the computation of the efforts in the member analyzed,
- 6) the sizing of the member analyzed.

Figure 2-2: The modular nature of the FLOOR program



2.2.1 Recursive nature of the analysis

A girder into which beams frame gets point-loads from these beams, and cannot be entirely analyzed until these beams have been analyzed themselves. A recursive function, called recurse(), regroups accordingly the steps 1 through 6) of the analysis above. Prior to analyzing any girder, recurse() goes over the members listed in the frame-in-lists of the nodes listed in the node-list of the girder, and calls itself to solve these members (if they have not been already solved).

2.2.2 Identification of member tributary areas

Because any girder is interrupted by beams, it is made of several sections delimited by nodes and it may be an edge for several areas:

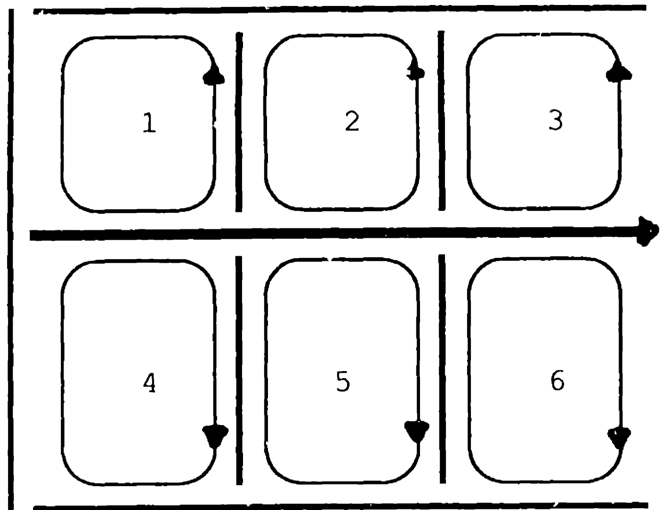


Figure 2-3: The several areas adjacent to a member

Steps 1) and 2) will therefore be repeated for each portion of a member. For any girder, several adjacent paths will be found. These paths, which are lists of nodes, will be stored in two lists: one list for the paths to the left of the oriented member, and one list for the paths to the right[4]. After Step 1), each member has thus associated with it two sets of lists of nodes, one for the left, one for the right.

---

[4]This is another difference with the original FLOOR program: its introduction will be justified below. Note that the corners of the paths to the left are listed in clockwise order, while the corners of the paths to the right are listed in counterclockwise order.

In Step 2), the direction in which the deck spans locally is identified and each of these paths is trimmed into a tributary area. A tributary area is the set of:

- a local deck direction, for use in the load carry-down routine,
- an area which is a circular list of points, identical to the points defining the location of the nodes.

C allows for storage of different kind of data in a single location, using unions. Since the paths of Step 1) are not of any further use, the tributary areas, once they have been determined, are stored in the same location. After Step 2), each member has associated with it two lists of circular lists of points instead of the list of lists of nodes[5] . The total surface of the whole tributary area directly supported by the member can be obtained by adding up the individual surfaces of each of the small tributary areas stored in the two lists.

2.2.3 Creation of member loads

Step 3) marks the entry of floor loads. At this point, it is helpful to consider the structure of a floor load.

A floor load is described by a magnitude and an area, upon which the floor load is applied. Areas have been described in Section 2.2.1: they are circular lists of points. In the design of floors, any load is traditionally described by three components: a dead part, which represents the self weight of the flooring system; a live part, which represents the weight of the future occupants of the building, and of the movable equipment they will bring; and a superimposed dead part, which represents the weight of the fixed equipment that comes with the building. Building codes typically introduce reduction factors to account for the low probability of maximal loads occuring at all points simultaneously. Because

[5] Again, the corners of the areas are listed in clockwise order if they are to the left of the member, and in counterclockwise order if they are to its right.

different reduction factors apply to the different types of loading, the distinction is a meaningful one. The floor load magnitude will therefore actually be an array of three magnitudes corresponding to these three components (this represents a change from the initial FLOOR program that allowed for, instead, an unlimited series of various load conditions).

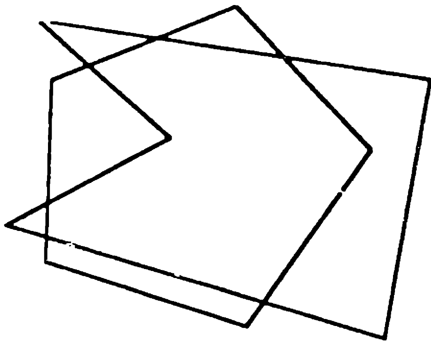
In Step 3), each of the individual tributary areas found in Step 2) is intersected with each of the areas upon which the floor loads extend themselves. In the jargon of graphics processing, one area is "clipped" against the other[6] . When the intersection of the two areas is not null, it is immediately carried down, or projected, and transformed into a line load on a member, with a beginning and an ending magnitude. (More precisely, it is projected into a series of line-loads, since FLOOR generates a line load for each of the sides of the loaded area it is carrying down [Bleakley 84][7] .)

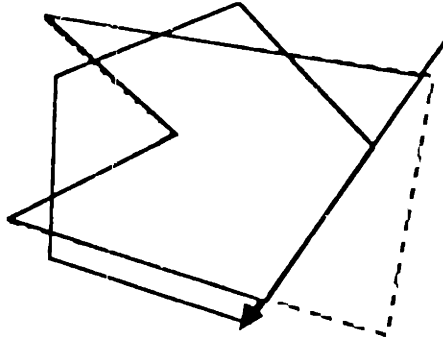
In the initial version of the FLOOR program, the line-loads generated would be stored in list form (one list of line-loads per load-case and per member). As explained in the next Chapter, (where the routines used for solving the members in the present version of FLOOR are introduced), they are now gathered in a data structure called a member diagram. Initially, the member diagrams are reduced to their simplest expression, which corresponds to a uniform load of 0: they are updated as the line loads are generated, and as the members are solved.

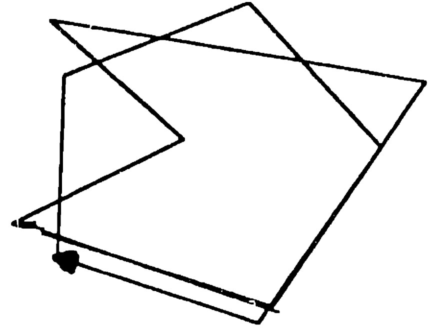
---

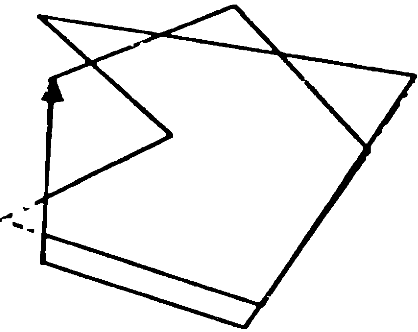
[6]As this expression suggests, the two areas do not play symetrical roles in the clipping algorithm. One area is intersected with the half plans to a given side of the oriented edges of the other area (cf. Figure 2-4). The second area must therefore have a convex shape, and its vertices must be listed in a known order. In the original version of FLOOR, this area was the floor load: this placed a limitation on the shape of the areas on which the loads could extend themselves. Tributary areas happen to always have a convex shape because of the physical layout of beams and floor carrying mechanisms. The distinction between the tributary areas to the left of the members, and those to their right, served to reverse the roles of floor loads and tributary areas in the clipping algorithm, and to lift the original limitation on the shape of the areas on which floor loads could be placed.

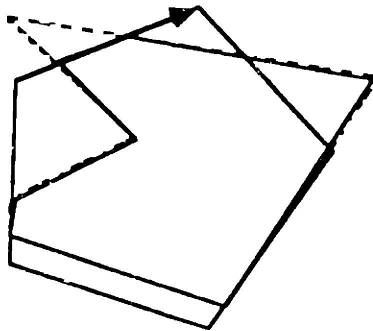
[7]The carry-down routine in the initial version of FLOOR projected the loads on the member perpendicularly to the member. This is inadequate for one-way flooring systems, and it has been supplemented by a routine that projects the loads parallel to the direction in which the deck spans locally.











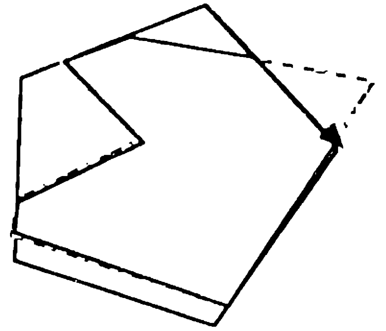


Figure 2-4: clipping algorithm

2.2.4 Load take-down and member sizing

The function recurse() calls a function called solve() to figure the efforts exerted in the members. If the member being analyzed is a girder, solve() collects the point-loads produced by the beams that frame into it. This of course supposes that recurse() has already called solve() with these beams. The current version of solve(), described in the next chapter, updates the member diagrams as new values are determined. This is substantially different from the original version.

The function size() sizes the sections for the members, as its name indicates. The member data structure includes a pointer to an entry in a table of steel sections: size() sets this pointer to a relevant value.

2.3 Memory allocation

FLOOR presently runs on the VAX II-750 operating under the UNIX operating system, as well as on the IBM PC/AT running PC/DOS 3.0. While the memory space is virtually illimited on the VAX, it is at a premium on the PC. Some routines that free memory have therefore been added to the code of the version that runs on the PC. After a member has been analyzed, the function recurse() calls the functions free-triba() and free-diagram() to reclaim the memory space taken by:

- the list of tributary areas attached to this member,
- its diagram, safe the head and the tail which, listing the reactions at the extremity of this member, may contain data of interest to the remainder of the analysis.

At any moment in the program execution, the memory of the machine contains:

- the floor layout data-structure,
- the zones prescribing local deck-directions,
- the floor-loads and openings,
- the beam-table,
- the complete representation of a dozen members at most.

Chapter 3

Analysis and sizing modules

As pointed out in the introduction, the initial version of FLOOR, stopped short of computing the maximal moments in the members and would not produce any immediately useful results for sizing of the members. Consequently, the analysis module has been re-designed, and a member sizing module has been added.

3.1 Analysis module

3.1.1 The initial analysis module

The initial version of FLOOR stored all the line-loads, generated in the Step 4) above, in a series of lists --one list of line-loads for each load case. To solve a member, it proceeded in two steps:

1. it first computed the resultant of all the loads applied on the member, and their total moment about one of the two extremities of the member,

2. these values obtained, it would compute the reactions at the two extremities of the member.

Although correct, the method presented two disadvantages:

1. it obtained the values of the reaction and moment at the two extremities, but not the value of the maximal moments in these members.

2. the best back-ups of the computations it could lead to would be a series of point-loads and line-loads. These line-loads being generated for each edge of each loaded area resting on a member, the series would be long: there would be typically eight line-loads for a simple beam, and several times as many more for a girder. Half of these line-loads would have a negative amplitude. The series would be definitely obscure.

3.1.2 Engineering practice

To overcome these two disadvantages, the present version of the FLOOR program models its computations on the actual computations taken by the engineer in sizing floor girders. Traditionally, the loading on a member, the shear, and the moment in it, are represented by three functions, q(x), V(x), M(x) defined for x ranging from zero to the length of the member. These functions are linked by the following differential equations:

$$V(x) = \int q(y)\,dy$$

$$M(x) = \int V(y)\,dy$$

To analyse a girder, the engineer begins by putting all the line-loads on a load diagram, which is the graph of q, the loading function. He then proceeds to integrate q to obtain, after allowances for the point-loads, a shear diagram, called V. Finally, integrating V he obtains a moment diagram, M.

3.1.3 Data structure representing member diagrams

q is linear by segments[8], and can be entirely described by the data of its points of discontinuity (of zero and first order), and its values to the right and to the left of these points: other values of q can be inferred from this data by linear interpolation. In all actual situations, there are only a finite number of points of discontinuity, and q can therefore be adequately represented by a list data structure, called a beam-diagram, listing the values of x and q at these points of discontinuity. (To maintain the distinction between the natures of the loads (dead, superimposed dead and live-loadings), q(x-) and q(x+) are actually arrays of three magnitudes.)

[8] In the computations of most engineers, q is taken to be constant by segments. More accurate, FLOOR generates uniformly increasing or decreasing line-loads: q is thus linear.

Given q, (which is the derivative of V), we can compute all the values of V given at least one value of V (for a boundary condition), and the values of its jumps (V(x+) - (V(x-)) at points of discontinuity. Therefore, as long as it stores these, any data structure will provide a sufficient representation of V. The knowledge of V being inseparable from the knowledge of q, the best is to store these in the beam diagram, along with the line-loads. The beam diagram will thus list:

- in addition to the values of x for which q changes, the values of x for which V jumps,
- and for all these values of x, the corresponding values of q and V to the right and to the left of x.

M is deduced from V by integration. The values of x listed in the diagram corresponding to discontinuities of V (of zero, first, or second order), we need to compute M(x) for all these values. It is therefore natural to let the diagram list these values of M, in addition to the corresponding values of L and V. Since the goal is to compute the maximal moment, an entry is added in the diagram for the value of x at which V is null and M is maximal.

3.1.4 Access to significant quantities

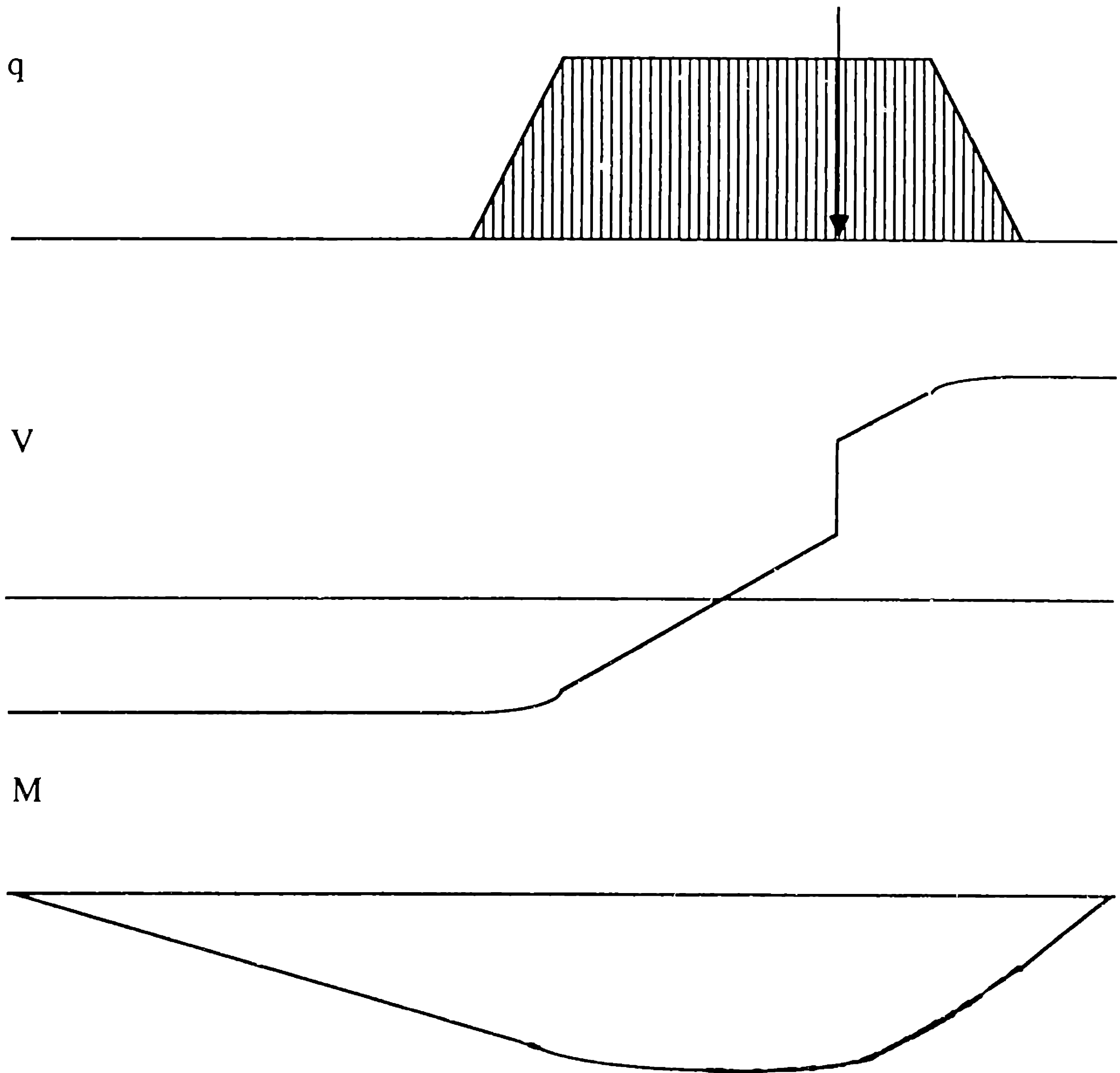
The member sizing module needs the value of three quantities: the shear at each of the two ends of the member, and the maximal moment. Since the first extremity of the member corresponds to the first entry in the diagram, the shear at this extremity can be accessed very conveniently. To provide as convenient an access to the two remaining quantities, two pointers are added to the member data structure:

- one points to the last entry in the list, providing access to the shear at the second extremity of the member (technically, the diagram is no longer a list data structure: it is a queue [Abelson 85]);
- the other points to the entry for which M is maximal, providing access

to the maximal moment (we do not think there is a technical name for
such a d[illegible]a structure as a list with three entries).

The contents of a member diagram data structure are illustrated on an example in Figure 3-1.

Figure 3-1: Diagram data structure



x = 0	x = 5	x = 6	x = 7.9	x = 9	x = 10	x = 11	x = 1[illegible]
q = 0	q = 0	q = 4	q = 4	q = 4	q = 4	q = 0	q = 0
V = - 6	V = - 6	V = - 5	V = 0	V = 4	V = 12	V = 13	V = 1[illegible]
V+ =- 6	V+=- 6	V+=- 5	V+= 0	V+= 8	V+=12	V+= 13	V+= 1
M = 0	M = 30	M = 37	M = 40	M = 38	M = 30	M = 15	M = 0

Mmax

3.1.5 Member solving

Initially, there are no loads on the members and the load diagrams are reduced to their simplest expressions: they are lists of two characteristic values of x: x equals zero, and x equals the length of the member, with correspondingly zero values to the left and the right. As line loads are found, the load diagrams are updated: several other characteristic values of x appear, along with magnitudes that are not null. To avoid repeated integrations, the shear and the moment are left equal to zero at this stage.

The function solve(), starts by collecting the point loads produced by the beams framing onto the member, creating jumps in the shear diagram.

It then integrates simultaneously the load and shear diagrams. The resulting tentative shear distribution differs from the real shear distribution by a constant, and the tentative moment distribution differs from the actual moment distribution by a linear function: solve(), therefore corrects at this point these tentative shear and moment distributions, to take into account the end conditions of the member.

Finally, solve() determines the value of x for which the shear is null. If there is not yet an entry in the diagram for this value of x, solve () creates one, computing the value of the maximal moment by means of a double integration of the shear and member diagram.

3.2 Member sizing

A member sizing module has been added to the original FLOOR program. It chooses for each member the lightest section possible, given an eventual constraint of depth. No axial force occurs in the floor beams and girders, and hence the member sizing module needs to check the various available sections for moment and

shear resistance only. Deflections may be overcome by cambering the member. Sizing a member means therefore finding a section with a sufficient web area and a sufficient section modulus[9] .

As in GROWLTIGER (an educational frame-analysis program, developed at M.I.T. [Slater]), FLOOR could sort all the available sections by weight: starting by the section with the least weight, it would keep testing sections with bigger and bigger weight until it finds an adequate one.

The simplicity of the design criteria for beams makes such a detailed method unnecessary. Indeed, a human designer proceeds more directly. He consults a table in which the sections:

- are grouped by families of sections with the same depth, and
- within these families, are sorted by decreasing weight.

In his search for an adequate steel section, he is guided by three rules:

1. The deeper the steel section, the less material will go in it to provide a given amount of moment resistance.
2. For a given depth, the more material in a section, the bigger the amount of moment resistance.
3. Generally, if, for a given depth, even the heaviest section cannot do, then, for a smaller depth, even the heaviest section will not do (this rule becoming more likely to be verified as the difference between the two depths increases).

Modeling itself on this human engineer, the FLOOR sizing module keeps the steel sections sorted in a table (in the technical sense of the term), where they are grouped in families of sections of a similar depth (depth-family), and within those families, are sorted by decreasing weight.

Given a member to size, size(), after computing a required section modulus,

[9] Allowable stresses are based on the AISC Steel Design Specification [AISC 84].

and a required web area, proceeds to go in the table, returning for the biggest depth-family allowed the acceptable section with the least weight: this section is the first tentative selection. If size() finds in the next smaller depth-family an acceptable section with a smaller weight, then, clearly the previous section was over-sized: this new section becomes the current tentative selection. After it cannot find in a depth-family an acceptable section, size() still tries in vain up to five or six depth-families before it stops its search and returns the last selection (if, in trying these five or six depth families, it finds an acceptable selection, it forgets its failure to previously find an acceptable section, and continues the search as if nothing had happened).

Most of the depth-families are such that the sections they regroup are over-sized. To speed the process, there are two entries for each depth-family in the table: one entry for the head of the list, one entry for its tail. The smallest section in the family appears there: before looking at the whole list of sections with a given depth, size() first checks if the smallest section with this depth is acceptable.

Within a family, the sections are sorted by decreasing weight. This corresponds to decreasing section modulus. Generally, this also corresponds to decreasing web areas. When this is not the case, i.e. when a section of a given depth has a larger web area thqn a section of the same depth and of a bigger weight, then the family is split (thus the W14 sections will have two table entries). There are thus 36 families. Experimentally, a program written for the purpose has counted that FLOOR tried an average of 31 sections per member when it dealt with the example shown in Appendix B.

Chapter 4

The initial mode of data entry

4.1 The initial situation

Using the original version of the FLOOR program, an engineer, interested in a FLOOR analysis, must give quite an explicit description of the floor layout to be analyzed. He has thus to enter four files to describe this floor layout. As described in Chapter 2, the floor layout data structure is made up of two lists:

- a list of nodes, with each of these nodes having attached to it a pointer to its supporting member (or a flag if it is a column) and a list of members framing in it;
- a list of members, with each of these members having attached to it pointers to two extremal nodes and a list of nodes situated on it.

The first file contains the nodes, their numbers (nodal numbers are required), their coordinates and flags to indicate if they actually stand for a column. The second file contains the members, their numbers (member numbers are as well required), the numbers of the extremal nodes, and some flags to indicate the presence of a cantilever at one extremity. The third file lists, node by node, the numbers of the members framing into it. The fourth file lists, member by member, the numbers of the nodes that sit on it.

After he has input these four files, the user must still input a last file to describe the area loads, giving for each load its magnitude and a list of the coordinates of the corners of the area the load is applied on.

Since the original version of the FLOOR program does not allow for local changes in the directions in which the one-way deck spans, there is no need for a sixth file to specify the zones and directions in which the one-way deck spans.

4.2 Automatic generation of node-lists and frame-in-lists

In fact, a simple improvement to the original FLOOR program could greatly simplify the task of the user. Appending any new member (as long as it is not a cantilever) to the frame-in-lists of its two extremal nodes, generates all the frame-in-lists automatically. While appending any new node (as long as it is not a column) to the node-list of its supporting member (which the user shall specify along with the coordinates of the nodes), generates all the node-lists.

Since a floor layout data structure is a highly recursive one, this improvement is not as straightforward to implement as might apppear. Indeed, because a node cannot appear on the node-list of a member which the system has not yet defined, it takes the system two passes over the node file to generate the node-lists:

1. in the first pass, the nodes are input,
2. in the second pass, after the members have been input, the "on-member" pointers of the node data structures are set, while the node-lists of the corresponding supporting members are updated.

4.3 Evaluation

The program places few requirements about the format in which this data is presented. The numbering of the nodes and members does not need to be sequential. It can therefore be descriptive: a member-number may thus be made from the numbers of its extremal nodes, or a node number may include the number of the member it is situated on. This descriptive numbering may ease the preparation of the data. Entering data in the initial FLOOR program does require preparation, time and attention, but the user is certainly better-of than the users of the early frame-analysis programs, who had no friendly screen-word-processors, submitted

decks of punched cards, and were required to number their nodes in a fashion that would reduce the band-width of the stiffness matrix.

Chapter 5

An interactive Interface

"If you please...draw me a ship!"
Le Petit Prince,
Antoine de Saint-Exupery.

In this Chapter, methods are introduced to further reduce the amount of data requested from the user. A conceptual interactive interface with simultaneous graphics display is presented, in which mouse functions provide a new way to:

1. create or specify nodes,
2. enter member extremities.

A number of features are provided for tasks such as on-line editing (for error elimination), plotting abilities (for output of the results) and drawing aids (making accuracy easier to attain with the mouse).

5.1 Simultaneous display with on-line editing and plotting

Drawings represent data entered and displayed in its most intuitive form (children draw before they know how to write) and a mere graphic display of the data as it is input will assist the user throughout the process in at least two ways. Since many mistakes that go undetected as text are visually obvious, it will give him an immediate check of the validity of what he has just input. It will also provide him with a global view of what he has already input.

To let the user correct the mistakes he notices, the interactive interface should provide the possibility of deleting any member or node, at any point. Ideally, all the members that frame into it, directly or indirectly, should be deleted by the same: the "delete" routine should work recursively.

The conclusions of the FLOOR analysis have to be presented in a format useful to the architect and the contractor: this means they have to be laid out on a plan. Adding a plotting ability to this graphical interface will save much work, and eliminate a few errors. Without it, the results of the analysis will be on a print-out; and an engineer will need to read these results on the print-out, consult a key and copy them on a sketch, pass this sketch to a draftsman, and finally check that the draftsman did draw what was meant.

5.2 Entry of data with a mouse

Adding a mouse to this graphics interface can transform the communication between the man and the machine.

As the reader knows, a mouse is a device that allows its user to enter very conveniently some coordinates corresponding to a point in the display (this operation is also called digitizing a point). Mice, in such interactive graphics applications as CAD systems, are not only used to let the user digitize a point, but are also used to let him refer to an entity (point or segment) which he has already input.

5.2.1 Creating new nodes with the mouse

Unless it is a column, a new node must must be appended to the node-list of a supporting member. Hence, entry of a new node (not a column) involves both the specification of a location and of a supporting member. By allowing the specification of both a location and a supporting member in one mouse-stroke, and by making member numbers unnecessary, the mouse does transform the creation of nodes.

This requires that the nodes and members be entered alternatively, in what can be called a methodical order. That is, the first nodes entered must be the columns, and the first members entered, the girders spanning between these columns. The subsequent nodes must be those situated on these main girders, and the subsequent members, the secondary girders framing into these nodes. While the last members entered must be the beams into which no other members will frame.

5.2.2 Automatic node generation

The mouse also allows the user to refer to nodes in one mouse stroke. Prior to graphic selection, a node could appear as a convenient abstraction, allowing the user to refer at once to both a supporting member and a location; however, now that digitizing a point is enough to refer to these, there is hardly any difference between creating a node on the one hand, and referring to one on the other hand. This introduces a certain redundancy, and, as far as the user is concerned, first creating nodes, and afterwards, referring to them, is no longer convenient.

Since they accept the same input from the user, the two operations, creating a new node (not a column), and referring to an existing one, can actually be merged. They will by the same token become invisible to the user who, after he has entered the locations of the columns, will enter extremal locations of new members, instead of entering extremal nodes. Such a reduction of the amount of data FLOOR requests to define a layout will be discussed further in the next Chapter, where the interface between FLOOR and a CAD system is described. The node look-up routine lies indeed at the center of everything we will do then, and it is appropriate to discuss its logic.

5.3 Node look-up algorithm

The node look-up can be implemented very straightforwardly:

1. the system attempts to return a node that would be "near"[10] the digitized point;

2. if it fails, it creates this node.

5.3.1 Selecting a point on the display with the mouse

We need to first explain the mechanism behind the graphic selection of entities with the mouse.

When he views an image, a human observer records a sorted representation in which points are grouped to form meaningful objects, which are in relationship to each another: some entities are to the foreground, and some in the background; some are above or below some others; and some to the left or to the right of still others. Coordinates are therefore a convenient way to refer to points. Although it could, the machine keeps no such sorted representation. A human observer of the New-York skyline at night told: "Look at the window with the blue light!", will have to look at each window, until he finds one with a blue light (unless he can ask "Where is it?"). Similarly, the machine referred to a point lit on the display, by its coordinates, will have, as explained in Harrington [Harrington 83], to loop over all the points it has listed in memory, until it finds one with matching coordinates (it cannot ask "When was it input?").

Because the user manipulating a mouse can only attain a limited accuracy, the machine in its search tests actually for closeness. It determines thus if the point it is testing and the digitized point lie within a same little square. Using a square

---

[10] we will return on the concept of closeness

instead of a circle considerably simplifies the computations: since the test may have to be repeated many times, this simplification is valuable.

5.3.2 Selecting a segment with the mouse

To identify a target segment next to which a point has been digitized, the machine proceeds similarly: it tests all the segments until it finds one passing close enough to the digitized point.[11] In this test, it has to check for two conditions:

- does p lie close to the line L supporting the segment ?
- but also, does the projection of p on L lie inside the segment ?

It is not necessary to compute the projection of p on L to check for the first condition. We can instead notice that it will be fulfilled if p lies in a rectangular window whose corners are close to p_1 and p_2:

[11] We mentioned above that a machine might keep, like a human, a sorted representation of the images it deals with. It would make identifying a point by its coordinates immediate. However, a difference between the human and the machine is that while the human records all the points making a drawing, and in particular, all the points making any line, the machine only keeps in memory the two extremities of any segment, and does not know of any point on it. A line can be referred to a human observer by some point on it (if the observer is shortsighted, he can always grasp the whole line letting his eye slide along the black points); but it cannot be referred in such a way to a machine.

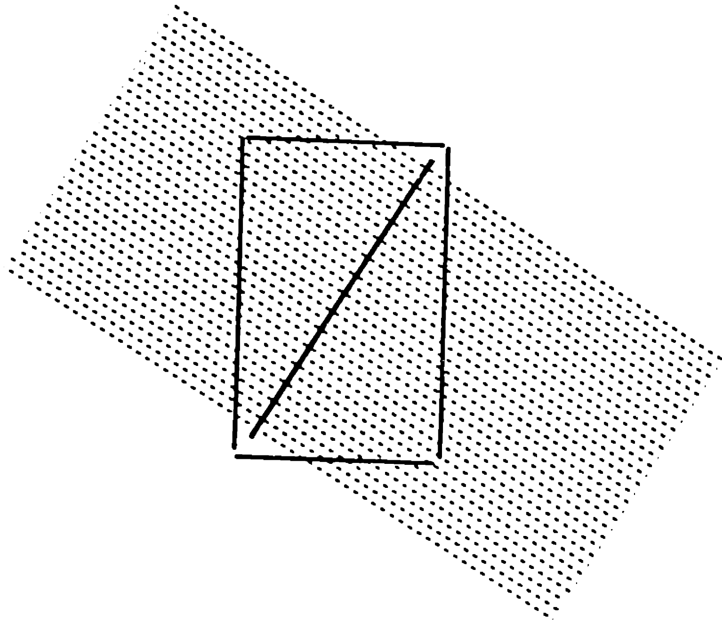


Figure 5-1: Points projecting on a segment

Testing for this in the first place allows the program to reject at once most of the segments that are too far from the digitized point.

For the remaining segments, one can test if the digitized point lies inside the area drawn on the left half of Figure 5-2 below:

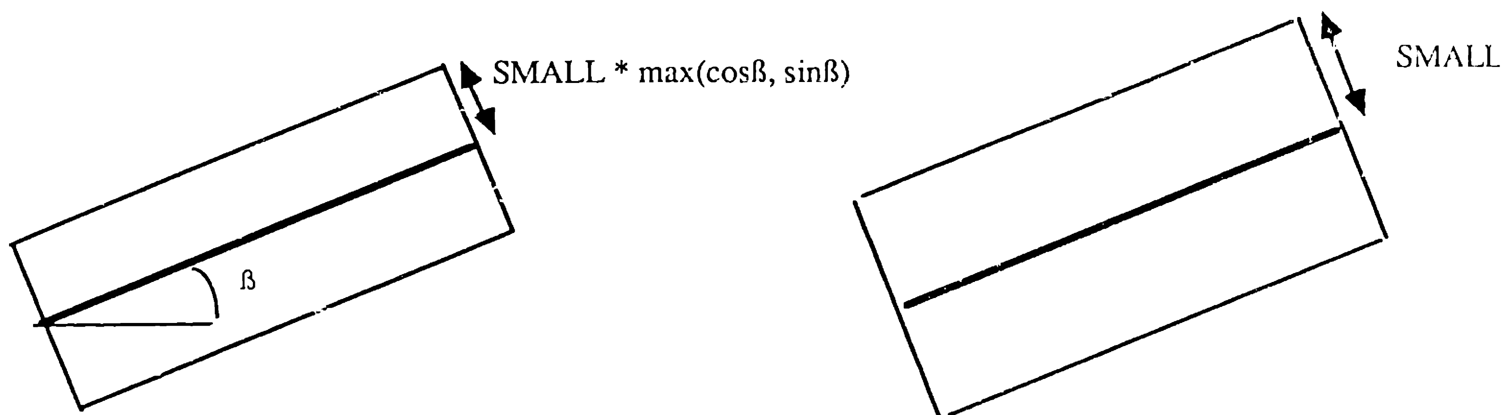


Figure 5-2: Distance between a point and a segment

Using the area drawn on the right might have seemed more natural, but it would have resulted in slightly more complicated computations. (The interested reader will find the formulas for testing against the area to the left in our listings; he can compare the formulas for testing against both areas in [Harrington 83]).

5.3.3 Final algorithm

A natural way to implement the node look-up would be to let the system go through the list of all the nodes input, testing them for closeness. With such an implementation, however, if after going through the list of all the nodes, the system fails to find a node near the digitized point, it needs to create a node, and for that, go through the list of all the members, in search of a member near the digitized point. Because a node is either a column, or it appears on the node list of a girder, the system can instead do this node look-up along the list of columns and the node-list of a proper girder, on which it will create a new node if it cannot find an adequate one.

This proper girder is of course a girder apt to counting in its node-list a node close to the digitized point. Unless it is a cantilever, a member does not count in its node-list its extremal nodes: the proper supporting girder at one extremity of a beam is thus the girder that is close enough to the extremity of this beam, and yet, unless it is a cantilever, has its extremities distant enough from it. Because two girders can meet but cannot cross each another, this set of two conditions entirely defines one unique supporting member.

We stated above that cantilevers had to be entered through a special command. Actually, it would be possible to let the system discover them as the members at one of the extremities of which no supporting entity can be found. Unfortunately, this is a feature that we will not be able to adapt in Chapter 6.

Because the user can only attain a limited accuracy with the mouse, the location of the new node created by the system, when it fails to find an existing node, is actually problematic. It has been decided to let the system create this node at the intersection of the supporting girder and the line defined by the two points digitized by the user. The accuracy of the FLOOR analysis will therefore reflect the accuracy of the user when he digitized his points. Drawing aids making accuracy easier to attain, are therefore required.

5.4 Drawing aids

The user can now input a flooring system almost as conveniently as he would draw it on paper; with the difference being that he manipulates a mouse instead of a pencil, and has no ruler. To be accurate, he can type coordinates instead of using a ruler, giving away, however, some convenience.

CAD systems also involve drawing with a mouse and without a ruler. To

make this task an easy one, CAD systems offer a choice of modes that suppress the need for accurate positioning of the mouse or pen :

- SNAP is a mode in which each of the digitized points is locked to the nearest point on an orthometric regular grid of points (the larger the spacing of the grid, the less accuracy is required). SNAP is useful for drawing very regular plans, in which all entities are separated by an entire number of the same modular unit of length.

- ORTHO is a mode in which all the lines drawn are vertical or horizontal (on the display).

- OSNAP/near (O stands for object) is a mode in which each of the digitized points is set to be the point on some entity that is the nearest from the actual location of the mouse.

- OSNAP/intersect is a mode in which each of the digitized points is set to be the point where two entities intersect that is the nearest from the actual location of the mouse.

Drawing a grid and placing the entities on this grid with the help of the OSNAP/intersect mode is a particularly efficient drawing technique that it is suitable to any drawing, no matter how irregular, and would be particularly adapted to this interactive interface.

5.5 In summary

No more frame-in-lists, no more node-lists, no more nodes: to build a floor-layout data structure this interactive interface only requests from the user a list of column locations, and a list of approximate member extremal coordinates. It then proceeds to:

- find supporting entities;

- find or create nodes on these entities;

- add, if it is relevant, these nodes to the node-lists of the supporting entities;

- create a member between these nodes;

- add it to the frame-in-lists of these nodes.

Because it creates and links the floor data-structure as its constituent entities are input, these must to be input in a methodical order.

It also displays the members as they are created, accepts input from a mouse, can lock points to a user-defined grid, and can produce plots on the terminal and a hard-copy device.

Chapter 6

Reading data from CAD files

"The sheep is in this box"
Le Petit Prince,
Antoine de Saint-Exupery.

An increasing number of CAD systems support drawing interchange files. These are ASCII files, which by describing the contents of a CAD drawing, allow other CAD systems and third-party programs, like FLOOR, to exchange data with some CAD system. In this chapter, we will discuss the profits this communication brings -reserving the discussion of its implementation to the next chapter.

6.1 The programmer's profits

6.1.1 From CAD to FLOOR

As pointed out in the previous chapter, an interactive interface should feature an immediate display of the data input, some editing capabilities, and some drawing aid functions. All these are present in any CAD system: letting FLOOR read CAD files, instead of conceiving an interactive interface, therby avoids re-doing much work.

Because it is a tool conceived specifically for drafting, the CAD system will bring many other convenient features. These include the choice of different line types, and colors for display (that can help distinguish members from loads); the possibility of zooming in and out from some part of the drawing; the possibility of creating more entities by duplicating or mirroring part of the display; the possibility of rotating some members included in part of the display; and finally the possibility

of digitizing, or taking electronic pictures of, conventional drawings, drawn with pencil and paper.

Because digitization gives the option of turning a free hand sketch, made without a ruler, into the CAD representation of a perfect drawing, with all lines straightened, and aligned to a specified grid if desired, it can bring major savings of time. It actually involves very complex hardware and software:

- hardware in the form of a device called a scanner or CAD camera, that works somehow like a photocopy machine, with this difference that instead of outputting a hardcopy, it outputs, as a description of the document it was given, a binary file telling for each point of the drawing if it is black or white.
- software in the form of pattern recognition programs that, sliding along the binary file, recognize the lines and segments that make the drawing and assemble the CAD file, which, as we already know, is a list of discrete points and bipoints[12] .

The complexity of the process made reading CAD files necessary in order to let the FLOOR program take advantage from the convenience the digitization technique can bring.

6.1.2 From FLOOR to CAD

As pointed out, a good interface to the FLOOR program should produce plots to convey the sizes it found. CAD systems have graphics routines that support several kinds of plotters: letting the FLOOR program write CAD files avoids therefore re-writing these routines.

It will also give the FLOOR user the possibility to:

- conveniently edit the resulting drawings, if, for some reason or some other, he is dissatisfied with them,

---

[12] Pattern recognition is a branch of artificial intelligence in full development. Some products are already commercialized, while more sophisticated ones, that will be more reliable, or will recognize text, are still the object of more research. The interested reader is referred to [Pavlidis 82].

- send them through electronic mail, communicating them to his client immediately.

6.2 CAD files, vehicles of information

When the engineer receives from the architect the grid on which he will place the various members in the form of a CAD file, he does not need to reenter this grid on his own CAD drawing. In turn, his sending the results of the FLOOR analysis under the form of a CAD file can be quite helpful to the architect.

As buildings have become more complex, management of data has become an increasingly important task for the architect. The architect, coordinator of the design team, has to deal with many types of data. He is expected to respect all the relevant articles of all the relevant building codes; to design for a budget; to put together specifications that can be over a thousand pages long; to know in advance the thermal efficiency of the building he designs; or check that the various subsystems that make the building do not interfere with each another (for example, that the various ducts and beams do not cross).

To assist him in these tasks, some CAD systems have become more elaborate than the others in that they have been combined with databases, and allow the association of a "number" of attributes (or characteristics) with a geometric entity. To mark the difference, they are called CADD systems: the two D's stand for drawing and design. In such a CADD system a window, for example, will be described by a unit price, a reference to some specification article, a thermal coefficient and, if the CADD system is truly sophisticated and boasts "object intelligence", a relationship to other entities (it calls for an opening in a wall). One

says that a "computerized model of the project", instead of a "drawing", is stored[13] .

Storing data makes sense only in the view of its retrieval: along with these CADD systems come programs that, extracting information from this database, automate a number of tasks such as quantity take-off and cost-estimate, energetic evaluation, specification writing and revision of drawings (if, through a change order, the shape of the window is changed at a later stage of the project, then the CADD system, if it boasts object intelligence, will warn that the concrete forming plans must be revised).

Because the steel sections computed by the structural engineer have to be processed for cost-estimating and mill-order, and because they are also required for checking that no beams and ducts cross, FLOOR generates data that it would be opportune to keep in such CADD systems. Frame-analysis has already been combined to these CADD systems to a similar end.

---

[13]Many buildings are made of literally thousands of individual items. Their computerized models are therefore large databases. Organization of large databases is quite an art, in which the goal is to avoid data redundancy and ease the updates consequent to changes in the project. Thus, if there are many similar windows in the project, one would try to store only one complete description of the window (shape and attributes), referring to "instances" of it in the project, characterized by a point of inclusion, and maybe some additional attributes, or facets. Most present personal computers are unfortunately too small machines to maintain open simultaneously both a CAD system and a database in the technical sense of the term. Yet, some CAD systems running on PC's have provisions to associate a few attributes to the entities that make the drawing. These attributes are then all listed in the drawing interchange files describing the drawing, where a real database can read them.

Chapter 7

Communication with CAD files: the process

7.1 Reading floor layouts

FLOOR, with the interface described in Chapter 4, can assemble a floor layout data structure with, for all input, the location of the columns, and the approximate locations of the extremities of the members. It is therefore not asking the user for more information than is contained in a drawing interchange file describing the framing plan of this layout.

7.1.1 Lifting the requirement of a methodical order

The reader will recall from Chapter 5 that FLOOR, with the interface described in this chapter, places a requirement on the order in which the data is input. Because drawing interchange files list the entities making the drawing in the order in which they were input, this requirement is no real hindrance. It can however become an inconvenience for three reasons:

1. whereas an interactive interface would not let its user input the entities in an inadequate order, the CAD system will do no check what-so-ever;

2. the delete routine may create perturbations;

3. the CAD file describing a digitized drawing cannot convey the order in which the lines of the original drawing were drawn.

While the interactive interface connected the members as they were input, the program reading the CAD files can instead proceed in two steps:

1. read all the entities,
    - allocating memory space for them,

- storing the points that described them,
- setting their node-lists (or frame-in-lists for columns) to be null;

and

2. then connect these entities,
   - creating or finding nodes,
   - filling the node-lists and frame-in-lists,
   - computing the actual extremal coordinates.

One can check that this dichotomy obviates the need for a methodical order.

7.1.2 Approximate locations of the member extremities

A framing plan is legible only if the segments that represent the members are trimmed at their extremities, stopping short from their "ideal" locations. The two following figures illustrate this:

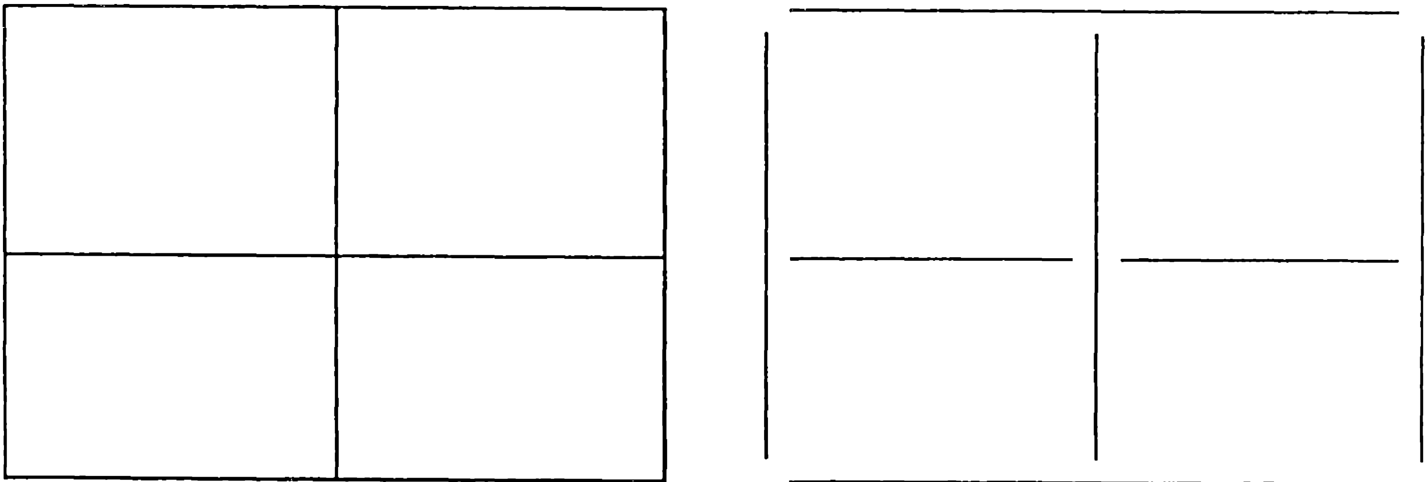


Figure 7-1: Trimmed segments

While it is impossible to tell the girder from the beams on the left side of figure 7-1, on the right side, the situation is clear. The CAD file obtained will therefore list vectors slightly shorter than the members that they actually represent. Fortunately, this will not prevent the program creating the nodes from computing the exact locations of the nodes.

7.1.3 A new data structure: the entities

Each time a member is to be connected to the other entities, the system will search for a supporting entity. This search goes through both the list of columns and the list of members. Because many users will input their entities section by section of the drawing, the search can yield faster results if the entities are tested in the order in which they were input, regardless of whether they stand for columns or members. To store the entities in the order in which they were found in the drawing file and to allow for such a search, a dummy data structure, called an entity is introduced. An entity is a simple data structure: it contains a pointer to an actual entity, a flag to tell to tell the nature of this actual entity (column or member), and a pointer to the next entity.

Incidentally, the approximate coordinates of the extremities of the members (the ones that were input) are stored in this "entity" data structure. In addition to all the above, an entity data structure includes a set of up to four coordinates.

7.1.4 Order of growth

Because the program does a double loop over the members, its time of execution can be expected to grow proportionally to the square of the number of members. The huge majority of the members are discarded after checking for one or two absolute values, so this is not excessively onerous. Also, the program models itself on graphic member selection with a mouse. If a user were doing his data entry progressively, selecting with a mouse existing nodes and members, the same number of tests would be performed. With a difference of a factor of 2 though, as inputting the members progressively reduces the number of searches at the

beginning of the process[14] .

There is no reason why an attribute could not be a pair of node numbers, or a member number, and one could consider the alternative approach of passing two lists to the FLOOR program, one listing the members, and the other the nodes. Several CAD systems can be customized. They allow their users to define user-commands, or macros, that, performing several regular commands at once, can automate attribute writing. Although defining adequate macros may involve several difficulties, such an approach is, after all, conceivable: it is indeed at the base of the combination of frame-analysis programs with CADD systems. Because this precludes digitizing conventional drawings of floor framing plans and turning them into files suitable for the FLOOR program, the current approach, however, seems preferable.

7.1.5 Treatment of cantilevers

The interactive interface could identify the cantilevered members as being those members for which no supporting member can be found. Unfortunately, with the approach taken in this Chapter, such an automatic identification is impossible. Indeed, some problems would arise with the members that frame-in at the extremity of the cantilever: if they are examined before the cantilevered member onto which they frame has been examined and flagged, they will be the ones considered to be the cantilevers. Cantilevers must therefore be entered separately. The next section describes how.

---

[14]As a rather striking similarity, the same factor of 2 is to be found in Physics when a distribution of electric loads is created: it takes twice as much energy to create the distribution at once than to create it progressively.

7.2 Reading cantilevers, openings, deck-directions and loads

As we already know, in addition to a floor layout, FLOOR also takes as data:

- zones for deck directions, which are a set of a direction and a series of points defining an area;
- loads, which are a set of a magnitude and a series of points defining an area;
- openings, which are just a series of points defining an area.

Many CAD systems treat polygons as individual entities (instead of bundles of segments), which of course will be listed as individual entities on drawing interchange files. The shapes of the openings, of the deck-directions and of the loads can be read from drawing interchange files.

Decent CAD systems allow for grouping entities by families on "layers", which can then be simultaneously displayed, or individually turned on or off. This classification is of course reflected on drawing interchange files: these files thus specify, for each entity, which layer of the drawing it belongs to. Cantilevers, placed on a separate layer can therefore be distinguished from the regular members. Openings, zones in which the deck spans in various directions, and loads of various magnitudes, as long as they are placed on different layers, may be all be grouped on the same CAD drawing.

The numerical data associated with these polygons (angle or magnitude) does not need to be passed as attributes to FLOOR since FLOOR can find in the name of the layer on which a polygon appears, a key to this numerical data.

7.3 Actual implementation with Autocad

7.3.1 The reasons for the choice

The program actually developed in this thesis reads AUTOCAD DXF (drawing interchange format) files. AUTOCAD was selected for several reasons:

- The price range of CAD systems is quite wide: it goes anywhere from 200$(MacDraw) to 20,000$(3-dimensional systems boasting object intelligence). With a complete version priced at 2,500$, AUTOCAD is a rather affordable product.

- It runs on the IBM/pc's and is widely used.

- It supports a drawing exchange format.

- It treats polygons as individual entities.

- It allows an unlimited number of layers.

- It allows its user to modify his environment and define new commands (which have to be written in "AUTOLISP").

- It can support a scanner (sold separately).

7.3.2 Contents of a suitable file

The following table summarizes these contents.

Figure 7-2: Contents of a suitable file

OBJECTS	LAYER_NAME	ENTITIES	REMARKS
MEMBERS	LAYOUT	LINE	can be represented by segments trimmed at their extremities cantilevered members do not belong to this layer
COLUMNS	LAYOUT	POINT	traditionnal I symbol should be drawn on another layer
CANTILEVERS	ENDCANT	LINE	first extremity of segment is built-in, second is free
AREA_LOADS	anyname formed as follows: Lx1/x2/x3 with: x1: dead_mag x2: live_mag x3: superimp	POLYLINE	no restrictions on shape, or order in which the vertices are listed
DECK_ZONES	anyname formed as follows: Dangle where angle is the angle between horizontal and the specified deck_direction	POLYLINE	shape must be convex, and vertices must be listed in clockwise order
	any other name		IGNORED

7.3.3 A customized AUTOCAD

Processing a conventional drawing through a scanner may be the most direct way to produce a CAD file, but, right now, scanners are expensive and not frequently used. Many users will instead go through the process of creating their CAD file step by step. To help them, the FLOOR interface takes advantage of the possibility of defining new commands, that are macros doing several standard commands. These are presented in the following table. The file listing their "AUTOLISP" definition is attached in Appendix C.[15]

[15]The user who would not have access to this customized version of AUTOCAD can always enter the appropriate commands individually.

Figure 7-3: Customized Autocad commands

NAME	ACTION	REMARKS
STARTUP	creates the various layers to be used	should be used on any new drawing
GRID	draws a line on a layer name GRID	to ensure that three grid lines that should intersect, do intersect, place AUTOCAD in OSNAPinters mode before using GRID when drawing the third line
MEMBER	places AUTOCAD in OSNAPinters mode, selects LAYOUT layer, draws a member with trimmed extremities corresponding to two points on the grid	
COLUMN	places AUTOCAD in OSNAPinters mode, prompts user for a location, locks this location to a point on the grid, draws there a point on the LAYOUT layer, and traditionnal I symbol on a layer named STUFF	
CANTILEVER	selects ENDCANT layer, asks user first for location of built-in extremity, then for location of free extremity, draws the cantilever	
LOADS menu: OFFICE,PUBLIC MECHANICAL, ROOF, etc.	select layer corresponding to adequate magnitudes and draw polyline	
DECK menu D0,D15,D30,etc.	select layer corresponding to specified deck_angle and draw polyline	

7.4 Writing the results of the analysis

As far as FLOOR is concerned, AUTOCAD has no adequate provisions for attributes, in that it can only associate attributes to "blocks", which are drawings the user inserts in his drawing. FLOOR therefore will just append strings of text to the drawing interchange file, with adequate points of insertion and orientation, depicting, on a layer named SECTIONS, the steel sections FLOOR recommends.

A text file listing the point-loads and the member diagrams provides a complete back-up of the computations. Member numbers, appended to the drawing file on a layer named KEY, and echoed in the back-up file, provide a correspondence between the drawing and the listings. Also, the shapes of the tributary areas figured by FLOOR, appended to the drawing file on a layer named TRIBAS, provide an additional, immediate check on the FLOOR analysis. An experienced designer, familiar with the member sizing process, will of course be able to tell at a glance if the results look safe to him.

A sample framing plan (with 120 members) generated by FLOOR and a few selected pages from the back-up file, attached in Appendix B, give an example of the FLOOR output. The original floor layout for this demonstration run, corresponds to a project actually designed by Weidlinger Assoc., and was selected because of its interesting complexity:

- its wedge-shape gives it a rich geometry, calling for several local changes in the direction of the deck-span,
- it is traversed in its middle by an expansion joint,
- it presents several instances of secondary girders, resting on primary girders.

It is therefore a good demonstration of the ability of the FLOOR program to deal with irregular layouts.

Chapter 8

Future developments

This chapter presents possible extensions of the FLOOR program. It distinguishes three kinds of improvements:

- improvements that by extending the reach of the FLOOR analysis, would round off the FLOOR program (these will be added to the FLOOR program in a near future);

- improvements that a tighter interface with a sophisticated CADD system could bring;

- improvements that would result from the combination of FLOOR to FLODER, a floor designer expert system developed at the Carneggie-Mellon University.

8.1 Additional analysis capabilities

Composite construction, which is more economical than non-composite steel construction, is more widely used. Composite design, leads to more complicated computations than steel design, and was not madc a part of this version of FLOOR. FLOOR can generate for any beam all the data needed for composite design (in particular, it can generate the area of the total composite cross section along all the points of the member); the economic importance of the question clearly warrants that a composite design member sizing module will be attached to FLOOR in the near future.

To allow the designer to judge at once the effect of a change in layout or in loading conditions, FLOOR should estimate the price of the designs it generates, doing a quantity take-off and counting the number of connections, distinguishing

between regular connections and difficult ones, involving for example, an oblique angle.

8.2 Interface with sophisticated CADD

Presently, the engineer using FLOOR has to define loads and openings manually, entering their locations and choosing from a table their magnitudes. For that, he needs to read and interpret the architect's drawings and the relevant building codes, and to keep himself informed of any change in the location of a piece of mechanical equipment, or in the function of a room. By introducing a risk of human error, this manual definition not only takes the engineer's time, but it also places an important responsibility on this engineer: as computer wisdom has it, "Garbage in, garbage out".

With proper input from the architect, and sufficient knowledge of applicable building codes, a sophisticated CADD system could automate the generation of loads and openings. All it needs for that is to keep a record of the function of the various rooms, and of the weight of the heaviest pieces of equipment. As rooms change functions, as mechanical equipment is moved, the sophisticated CADD system might even automatically call the FLOOR program to make sure that the affected framing plans remain valid.

By computing the actual column reactions, FLOOR generates information quite relevant to frame-analysis. It would therefore appropriately complete a CADD system already incorporating frame-analysis.

8.3 FLOOR with FLODER

8.3.1 FLODER

FLOOR leaves the user with the task of designing a floor layout, and drawing it (on the CAD system, or maybe, if he has a digitizer, manually). In the case of large projects, this can turn out to be quite a time-consuming task. An improved version of the FLOOR system would:

- add a beam when it finds an area that spans a distance greater than the decking allows;
- place beams and girders around the openings.

With this version, the user would then just need to input the girders, and a few beams: the system would then complete the floor layout for him.

FLODER, a Floor Designer Expert System, presented by Andreas Georgiou Karakatsanis in his 1985 thesis [Karakatsanis 85], goes further: given architectural plans, FLODER generates floor framing plans that will take into account the locations of the shafts and the hallways. It generates these plans for a variety of construction techniques:

- concrete flat slabs;
- concrete solid slabs;
- prestressed concrete;
- steel construction.

It makes a preliminary analysis of the various designs it generates, and evaluates them, guiding its user in his choice amongst these various designs.

8.3.2 The limitations of FLODER

FLODER has several limitations.

It can only deal with plans in which all the lines are perpendicular.

The user must break down the plan in a set of small rectangles. Each rectangle must be input separately. When two rectangles share an edge, the user has to input this edge twice.

In view of the logic of FLODER, we suspect that it will give unpractical results in a number of cases in which the designer attempts to multiply the corner offices[16] . FLODER needs probably more expertise (like the initial FLOOR program it is a pioneer work, not a commercial product).

Presently, most expert systems are excellent at inferring propositions, and good at manipulating data structures; but they are bad at carrying numerical computations. FLODER typifies this limitation of expert systems in that it only has at its command approximate methods of analysis that involve a high degree of uncertainty.

8.3.3 Interfacing FLOOR and FLODER

FLOOR and FLODER have the potential for completing each another. A mega-program, combining both programs, would:

- enable FLODER to deal with more complex shapes;
- provide for entry of geometric data more convenient than in the present FLODER program;
- unlike FLOOR, would require a minimal input from its user.

---

[16] Corner offices, providing their occupants with a wider angle of view, are more enjoyable than regular offices. In a company, they are typically sought after by the executives. As a result, developers and architects often try to multiply the corner offices in a building. Exchange Place, a recently completed high-rise in downtown Boston, has no less than 16 corner offices per floor.

Combined with accurate analysis programs, and combined with each another, such expert systems as FLODER for engineering design may bring about, some day, a transformation of the design profession. One may thus envision the architect of the future as an independent designer, seldom needing the help of an engineer.

Appendix A
Program Listings

The various routines of the FLOOR program have been regrouped in twelve files:

1. main.c lists main() and recurse();
2. r_d.c lists the function read_data() which reads the drawing interchange file;
3. connec.c lists the functions that connect the segments;
4. a_r.c lists the function adj_rep();
5. triba.c lists the functions corresponding to the Steps 1) and 2) described in Chapter 2) (paths and tributary areas);
6. cl_cl.c lists the functions corresponding to the Steps 3) and 4) described in Chapter 2) (paths and tributary areas);
7. sol.c lists solve() and set_kink();
8. siz.c lists size() and the functions it calls;
9. geom.c lists various functions, called in several other places of the program, to perform geometric manipulations;
10. li.c lists various functions, called in several other places of the program, to perform operations with the list data structures;
11. nu.c lists several functions which append the DXF file;
12. pr.c lists functions that print back-ups of the computations.

Finally, an "include-file", called strall2.h, lists the data structures used throughout the program.

```
/**/
/*********************************************************************

        Main menu driver for program 'FLOOR'

        programmed by Al Bleakley (1984) and Pierre Gasztowtt (1985)

*********************************************************************/

#include <stdio.h>
#include <ctype.h>
#include "strall2.h"

        /* external variables */

int ways = ONE;       /* one-way or two-way */


struct member *member_head = NULL;
struct segment *seg_head  = NULL;
struct f_load *opening_head  = NULL;
struct f_load *fload_head  = NULL;
struct node *node_head = NULL;
struct node *col_head  = NULL;
struct deck_zone *deck_head = NULL;
struct wf_table *wf_head = NULL;
double  maxx = 0, maxy = 0, SMALL = 0;
double d_max = 36;
FILE   *dxf_file_ptr, *back_file_ptr;
main (argc, argv)

int argc;
char *argv[];

{
extern int ways;
int  loads = 0, deck = 0;
int  tribas = 0, key = 0;
FILE *file_ptr, *fopen();
extern FILE *dxf_file_ptr, *back_file_ptr;
char filename[40];
char *s;
extern struct member *member_head;
extern double maxx, maxy, SMALL;
struct member *member_ptr;



while (-- argc > 0 && (*++argv)[0] == '-')
        for (s = argv[0]+1; *s != '\0'; s++)
```

```
                switch (*s) {
                case '2':
                    ways = TWO;
                    break;
                case 'l':
                    loads = 1;
                    break;
                case 'k':
                    key = 1;
                    break;
                case 'd':
                    deck = 1;
                    break;
                default:
                    printf("floor: illegal option %c\n", *s);
                    argc = 0;
                    break;
        }

if (argc != 1)
        /*stop here*/
        printf("Usage:.....\n");
else{
        /*proceed*/
        file_ptr = fopen(*argv, "r");
        while (file_ptr == NULL) {
                printf ( "Could not open the file %s\n", filename);
                printf (" Reenter the name, or send a ↑C: " );
                scanf("%s", filename);
                file_ptr = fopen(filename, "r");
        }
        printf("Starting to read the dxf_file\n");
        read_seg_dxf(file_ptr);
        fclose(file_ptr);

        if (deck == 1){
                list_decks();
#ifdef DEBUG
                return;
#endif
        }

        if (loads == 1)
                list_loads();

        printf("Starting to connect the members\n");
        SMALL = (maxx + maxy)/175;
        connect_segment();
```

```
        back_file_ptr = fopen("memb.res","w");
        list_member_nodes(back_file_ptr);
                                  .   /*lists  extremal nodes and nl*/
        list_nodes(back_file_ptr);



        dxf_file_ptr = fopen(*argv,"a");
        nuout(dxf_file_ptr);
#ifdef DEBUG
        if(key == 1)
                return;
#endif


printf("Building the adj_rep\n");
adj_rep();


printf("Loading the beam table\n");
beam_table();


for(member_ptr = member_head; member_ptr != NULL;
 member_ptr = member_ptr->next)
  if(member_ptr->flags.solved == NO){
        recurse(member_ptr);
  }

}

}



recurse(member_ptr)

struct member *member_ptr;

{
extern FILE *dxf_file_ptr, *back_file_ptr;
struct member  *m_ptr;
struct node_list *nl_ptr;
struct frame_in_list *fil_ptr;

for (nl_ptr = member_ptr->frame_in; nl_ptr != NULL;
  nl_ptr = nl_ptr->next)
        for (fil_ptr = nl_ptr->in_node->frame_in; fil_ptr!=NULL;
          fil_ptr=fil_ptr->next){
                m_ptr = fil_ptr->in_member;
```

```
                if(m_ptr->flags.solved == NO){ /*not solved?*/
                  printf("member %d: recurse calling itself with %d\n",
                          member_ptr->member_number,
                          m_ptr->member_number);
                  recurse(m_ptr); /* recurse calls itself*/
                }
        }

#ifdef DEBOGUE
        printf("Finding tributary areas\n");
#endif
        triba(member_ptr);
        triba_out(dxf_file_ptr, member_ptr);

#ifdef DEBOGUE
        printf("Clipping and carrying_down the loads\n");
#endif
        cliploads(member_ptr);
        free_tribas(member_ptr);

#ifdef DEBOGUE
        printf("Analysing the member\n");
#endif
        analyze(member_ptr);

#ifdef DEBOGUE
        printf("Sizing the member\n");
#endif
        size(member_ptr);
        sectout(dxf_file_ptr, member_ptr);

        alldias(back_file_ptr, member_ptr);

#ifdef DEBOGUE
        printf("next\n");
#endif

}
```

File r_d.c

```
/*
```

```
      */
/*****************************************************************

        READ_SEG_DXF()

 This is the function  read_seg_dxf(), which reads enitities from
 a dxf drawing  file and allocates memory for them.

*****************************************************************/
#include "strall2.h"
#include <stdio.h>
#include <math.h>
#define malloc getmem


read_seg_dxf(segf_ptr)

FILE                    *segf_ptr;

{
char               string[17], nature[10],layer[17];
int                deck_zone();
extern double      maxx, maxy;
int                code,   cant;
struct f_load      *fload_ptr, *add_fload();
struct deck_zone *deck_ptr, *add_deck();
double x,y, begin_x, begin_y, end_x, end_y, alpha, sin(), cos();
struct member           *member_ptr, *add_member();
struct column           *col_ptr, *enter_col();
struct segment          *seg_ptr, *add_segment();
h_area                  *read_harea();
h_line                  *direction;
char                    char_d, *malloc();
int                     strcmp();
fscanf(segf_ptr, "%s", string);
while(strcmp (string, "ENTITIES") !=0){
    /*keep searching until there is nothing left*/
    if (fscanf(segf_ptr, "%s", string) == EOF)
        printf("reached EOF without finding entities\n");
}


while( fscanf(segf_ptr, "%d %s",&code, nature) != EOF){

#ifdef WHAT
        if(code != 0)
                printf("Wrong code\n");

        else    printf("nature is %s\n", nature);
#endif
```

```
    fscanf (segf_ptr, "%*d %s", layer);
#ifdef WHAT
        printf ("layer is %s\n", layer);
#endif
    if (strcmp (layer, "LAYOUT")== 0){
        if ( strcmp(nature,"POINT")== 0)  {
                seg_ptr = add_segment();

                fscanf(segf_ptr,"%*d %lf %*d  %lf ", &x, &y);
                if (x > maxx)
                        maxx = x;
                if (y > maxy)
                        maxy = y;

                seg_ptr->begin_y = y;
                seg_ptr->type = COLUMN;
                seg_ptr->actually_c = col_ptr->node_ptr;
        }
        else if (strcmp(nature, "LINE") == 0)  {

                seg_ptr = add_segment();
                member_ptr = add_member();
                seg_ptr->type = MEMBER;
                seg_ptr->actually_m = member_ptr;
                member_ptr->flags.begin_cant = 0;
                member_ptr->flags.end_cant  = 0;
                fscanf(segf_ptr, "%*d %lf %*d %lf", &begin_x, &begin_y);
                seg_ptr->begin_x = begin_x;
                seg_ptr->begin_y = begin_y;
                fscanf(segf_ptr, "%*d %lf %*d %lf", &end_x, &end_y);
                seg_ptr->end_x = end_x;
                seg_ptr->end_y = end_y;
    }
  }

  else if (strcmp(layer, "ENDCANT")==0)  {
        if (strcmp(nature, "LINE") != 0)
                printf("error reading a cantilever\n");
        seg_ptr = add_segment();
        member_ptr = add_member();
        seg_ptr->type = MEMBER;
        seg_ptr->actually_m = member_ptr;
        member_ptr->flags.begin_cant = 0;
        member_ptr->flags.end_cant  = 1;
        fscanf(segf_ptr, "%*d %lf %*d %lf", &begin_x, &begin_y);
        seg_ptr->begin_x = begin_x;
        seg_ptr->begin_y = begin_y;
        fscanf(segf_ptr, "%*d %lf %*d %lf", &end_x, &end_y);
        seg_ptr->end_x = end_x;
        seg_ptr->end_y = end_y;
```

```
  }

  else if (strcmp(layer,"MECHANICAL") == 0 ||
          strcmp(layer,"PUBLIC")== 0 ||
          strcmp(layer,"OFFICE")== 0 ||
          strcmp(layer,"ROOF")== 0){
      if (strcmp(nature, "POLYLINE") != 0)
                     printf("error reading a load\n");
      fload_ptr = add_fload();
      if (strcmp(layer, "OFFICE") == 0){
               fload_ptr->magnitude[0] = .02;
               fload_ptr->magnitude[1] = .05;
               fload_ptr->magnitude[2] = .035;
      }
      else if (strcmp(layer,"PUBLIC") == 0 ||
                      strcmp(layer,"MECHANICAL") == 0){
               fload_ptr->magnitude[0] = .02;
               fload_ptr->magnitude[1] = .050;
               fload_ptr->magnitude[2] = .015;
      }
      if (strcmp(layer, "ROOF") == 0){
               fload_ptr->magnitude[0] = .02;
               fload_ptr->magnitude[1] = .04;
               fload_ptr->magnitude[2] = .015;
      }
#ifdef WHAT
      printf("Reading a load\n");
#endif
      fload_ptr->area = read_harea(segf_ptr);
#ifdef HAREA
      print_harea(fload_ptr->area);
#endif


   }

   else if (deck_zone(layer)){
            deck_ptr = add_deck();
            while((direction = (h_line *) malloc(sizeof(h_line)))
                    == NULL)
                    printf("memory\n");
            deck_ptr->direction = direction;
            sscanf(layer, "%c%f", &char_d,  &alpha);
            direction->x = - sin(alpha * (3.141 / 180));
            direction->y = cos(alpha * (3.141 / 180));
            direction->w =1;
            deck_ptr->harea_ptr = read_harea(segf_ptr);
   }
   else {
#ifdef WHAT
```

```
        printf("Passing stuff\n");
#endif
        if(strcmp (nature,"LINE") == 0)
                fscanf(segf_ptr, "%*d %*lf %*d %*lf %*d %*lf %*d %*lf");
        if (strcmp (nature, "TEXT") == 0){
                fscanf(segf_ptr, "%*d %*lf %*d %*lf %*d %*lf %*d %*s");
                fscanf(segf_ptr, "%*d %*d %*d %*lf %*d %*lf");
        }
    }
}
return;
}

/*
```

```
        */
/******************************************************************
        H_AREA *READ_HAREA

This function reads the coordinates of the vertices of an h_area,
allocating memory for them.

******************************************************************/

h_area *read_harea(segf_ptr)

FILE *segf_ptr;

{

h_area  *area_head, *add_point();
h_point point;
char    nature[10];
int     strcmp();

        area_head = NULL;
        fscanf(segf_ptr,"%*d %*d %*d %*d %*d %s", nature);
#ifdef WHAT
        printf("initially, nature is %s\n", nature);
#endif
        fscanf(segf_ptr,"%*d %*s"); /*ignore this layer reminder*/
        while(strcmp(nature,"SEQEND") != 0) {
#ifdef WHAT
        printf("adding a point\n");
#endif
                fscanf(segf_ptr,"%*d %lf", &(point.x));
                fscanf(segf_ptr,"%*d %lf", &(point.y));
                area_head = add_point(&point, area_head);
                fscanf(segf_ptr,"%*d %s", nature);
#ifdef  WHAT
        printf("nature is %s\n", nature);
#endif
                fscanf(segf_ptr,"%*d %*s"); /*ignore this layer reminder*/
        }
        append(area_head, area_head);

        return(area_head);
}

/*
```

```
    */
/***********************************************************************

        int deck_zone()

***********************************************************************/

int deck_zone(layer)

char layer[10];


{

return( (strcmp(layer, "D0") == 0)
      ||(strcmp(layer, "D15") == 0)
      ||(strcmp(layer, "D30") == 0)
      ||(strcmp(layer, "D45") == 0)
      ||(strcmp(layer, "D60") == 0)
      ||(strcmp(layer, "D75") == 0)
      ||(strcmp(layer, "D90") == 0)
      ||(strcmp(layer, "D105") == 0)
      ||(strcmp(layer, "D120") == 0)
      ||(strcmp(layer, "D135") == 0)
      ||(strcmp(layer, "D150") == 0)
      ||(strcmp(layer, "D165") == 0)     );

}
```

```
/**********************************************************************

                    FLOOR Program Documentation

NAME            connect_segment(), get_extr_node(), find_support()

SYNOPSIS        connect_segment()

                struct node *get_extr_node(side, seg_ptr)
                        char side;
                        struct segment *seg_ptr;

                struct segment *find_support(point)
                        h_point *point;

EXTERNAL FUNCTIONS

                fabs()
                set_point()
                line()
                intersect()
                find_point()
                add_node()
                insert_nl()
                add_fil()
                length()

STATIC VARIABLES

EXTERNAL VARIABLES

DESCRIPTION     called from  main(), connect_segment()
                transforms a list of entities into some interconnected
                list of members and list of nodes.
                For that, it loops over the members that were entered,
                setting their extremal nodes to nodes that
                get_extr_node() finds or creates
                on some supporting entities found by find_support.
                connect_segment() also
                adds the members (as long as they are not  cantilevers)
                to the frame-in-lists of their extremal nodes,
                and sets the member lengths in the
                member diagrams.

AUTHORS         Pierre Gasztowtt

SIDE EFFECTS    the list of nodes is expanded. The list of members and the
                list of nodes are connected.
```

```
BUGS            None

FILES           strall2.h, stdio.h, list_m.c, geom.c

*********************************************************************/
/*
```

```
      */

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

extern double SMALL;


/* This is  the function CONNECT_SEGMENT()                          */


connect_segment()


{

extern struct segment   *seg_head;
struct segment          *seg_ptr;
int                     m, begincant, endcant;
struct node              *begin_node_ptr, *end_node_ptr;
struct member           *member_ptr;
struct node             *get_extr_node();
double                  L, length();


for (seg_ptr =seg_head; seg_ptr !=NULL; seg_ptr = seg_ptr->next )

     if(seg_ptr->type == MEMBER) {

        member_ptr = seg_ptr->actually_m;
#ifdef DEBUG
        m = member_ptr->member_number;
#endif
        begincant = member_ptr->flags.begin_cant;
        endcant = member_ptr->flags.end_cant;

        /* take care of beginning extremity*/

        begin_node_ptr = get_extr_node('b', seg_ptr);
        if(begin_node_ptr == NULL)
            printf("%d\n", member_ptr->member_number);
        else
            member_ptr->begin_node = begin_node_ptr;

        if (begincant == 0)
                add_fil(begin_node_ptr, member_ptr);
        if (endcant == 1)
                begin_node_ptr->flags.cant = 1;
```

```
        /* take care of end extremity*/

        end_node_ptr = get_extr_node ('e', seg_ptr);
        if(end_node_ptr == NULL)
            printf("%d\n", member_ptr->member_number);
        else
            member_ptr->end_node = end_node_ptr;

        if (endcant == 0)
                add_fil(end_node_ptr, member_ptr);
        if (begincant == 1)
                end_node_ptr->flags.cant = 1;

        /*update something very obscure:*/

        L = length(member_ptr);
        member_ptr->last_kink->x = L;

        }


return;

}

/*
```

```
    */
/**/
/***************************************************************************

        STRUCT *NODE GET_EXTR_NODE()

        This function is called twice for any segment:
        once for each of its  two extremities. Depending on wether or not
        underlying member is a cantilever, it sets the member to be its
        supporting member, or calls find_support.
        It then does a node look-up: either, it finds a node on the
        node-list of the supporting member, either it adds it.

***************************************************************************/

struct node *get_extr_node(side, seg_ptr)

char side;
struct segment *seg_ptr;

{
        struct node       *extr_node_ptr, *find_point(),*add_node();
        h_line            l1, l2;
        h_point           point1, point2, point3, point4, point;
        struct segment    *support_ptr, *find_support();
        struct node_list  *insert_nl();
        int               cant;
        double            x1, y1, x2, y2, x3, y3, x4, y4;
        int               m, n;


m = seg_ptr->segment_number;

/* Preliminary: set local-variables: */

switch (side){
        case 'b':
                set_point(&point1,seg_ptr->begin_x,seg_ptr->begin_y);
                set_point(&point2,seg_ptr->end_x, seg_ptr->end_y);
                cant = seg_ptr->actually_m->flags.begin_cant;
                break;
        case 'e':
                set_point(&point1,seg_ptr->end_x, seg_ptr->end_y);
                set_point(&point2,seg_ptr->begin_x,seg_ptr->begin_y);
                cant = seg_ptr->actually_m->flags.end_cant;
                break;
        }

/* Then find support: */
```

```
if (cant == 1)
        support_ptr = seg_ptr;
else
        support_ptr= find_support(&point1);



/* Then do node look-up:*/

#ifdef DEBUG
n = support_ptr->segment_number;
#endif

switch (support_ptr->type) {
case COLUMN:
        extr_node_ptr = support_ptr->actually_c;
        break;
case MEMBER:
        /*if the extremity is cantilevered,*/
        /*its location is as shown on drawing:*/
        if (cant == 1)
                set_point(&point, point1.x, point1.y);
        /*otherwise, its location is at intersection of*/
        /*member and its support:*/
        else{
                set_point(&point3,support_ptr->begin_x,
                  support_ptr->begin_y);
                set_point(&point4,support_ptr->end_x,
                  support_ptr->end_y);
                set_line(&point1, &point2, &l1);
                set_line(&point3, &point4, &l2);
                intersect(&l1,&l2, &point);
        }
        extr_node_ptr = find_point(&point,
                              support_ptr->actually_m->frame_in);
        if (extr_node_ptr == NULL) {
                 /*create new node*/
                extr_node_ptr = add_node(&point);
                extr_node_ptr->flags.col = MEMBER;
                extr_node_ptr->on_member = support_ptr->actually_m;
                /*and insert it in node-list of support_member*/
                support_ptr->actually_m->frame_in =
                        insert_nl(extr_node_ptr, &point3,
                            support_ptr->actually_m->frame_in);
         }

         break;
default:
        printf ("illegal support_ptr returned");
        extr_node_ptr = NULL;
```

```
        break;
 }


return (extr_node_ptr);
}

/*
```

```
     */
/*index(find_support)*/
/*****************************************************************************
        STRUCT *SEGMENT FIND_SUPPORT()

        find_support() loops over all the entities, testing them for
        closeness, until it finds one which might support a node at the
        location it was passed

*****************************************************************************/

#include <math.h>


#define max(A, B) ((A) > (B) ? (A) : (B))
#define min(A, B) ((A) > (B) ? (B) : (A))

struct segment *find_support(point)

h_point *point;

{
int found, n;
extern struct segment *seg_head;
struct segment  *test_segptr;
double      x,y,x1,y1,x2,y2;
double fabs();

found = 0;
x = point->x;
y = point->y;


test_segptr = seg_head;
while (found == 0 && test_segptr != NULL){
        n = test_segptr->segment_number;
        switch   ( test_segptr->type) {
        case MEMBER:
                                /*******************************
                                underlying entity is a member
                                *******************************/
           x1= test_segptr->begin_x;
           y1= test_segptr->begin_y;
           x2= test_segptr->end_x;
           y2= test_segptr->end_y;
                                /*First test: does the point lie
                                within a rectangle enclosing the
                                segment ?                      */
           if ( x > min(x1,x2) - SMALL &&
                x < max(x1,x2) + SMALL &&
```

```
              y > min(yl,y2) - SMALL &&
              y < max(yl,y2) + SMALL) {
                              /*Second test: now that it is in
                              this rectangle, is it indeed
                              close to the segment?*/
             if (   fabs(y2-yl)<SMALL /*it will if segment is*/
                 || fabs(x2-xl)<SMALL /*horizontal or vertical*/
                 || fabs((y2-yl)*(x-xl)+ (x2-xl)*(yl-y)) <
                                    SMALL * fabs(x2-xl)
                 || fabs((x2-xl)*(y-yl)+ (y2-yl)*(xl-x)) <
                                    SMALL * fabs(y2-yl) ){
                              /*Third test: now that it is close
                              to the segment, is it however
                              far enough from the extremities?
                              Or, is there a cantilevered extremity?*/
                 if ( ( fabs(xl-x) + fabs(yl-y) > 2 * SMALL
                 ||test_segptr->actually_m->flags.begin_cant==1)
              && ( fabs(x2-x) + fabs(y2-y) >  2 * SMALL
                  || test_segptr->actually_m->flags.end_cant ==1)){
                              /*Then we have won:*/
                        found = 1;
                  }
                  else test_segptr = test_segptr->next;
              }
              else test_segptr = test_segptr->next;
          }
          else test_segptr = test_segptr->next;
          break;              /*from the switch, but stay in the loop*/

    case COLUMN:
                              /****************************
                              underlying entity is  a column
                              *****************************/
                              /* Much simpler:is the max_dist
                              from point to this column < SMALL?*/
         if ( fabs (x - test_segptr->begin_x) +
                     fabs (y - test_segptr->begin_y) < SMALL) {
                        found = 1;
         }
         else  test_segptr = test_segptr->next;
         break;
  }
}

return(test_segptr);             /*even if it is null*/
}
```

```
/*************************************************************************

                    FLOOR Program Documentation

NAME              adj_rep()

SYNOPSIS

EXTERNAL FUNCTIONS

                  add_neighbor()

STATIC VARIABLES

EXTERNAL VARIABLES

                  node_head()

DESCRIPTION        adjrep() creates an adjacency representation for the
                  floor, finding the neighbors of each node.
                  It does a double loop: on the nodes of the floor, and
                  an inner one on the members that frame into each node.


AUTHORS           Al Bleakley and Pierre Gasztowtt

SIDE EFFECTS      Each node gets a list of neighbors.

BUGS              I suspect the function does a wrong job for the
                  nodes that are on a cantilever with a long node_list.

FILES             strall2.h, stdio.h, list_m.c, geom.c

*************************************************************************/
/*
```

```
    */

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

/*********************************************************************
        ADJ_REP()

        adj_rep() collects for each node of the floor its neighbors.
        It first finds the neighbors that are on the members framing
        into the node, and then the neighbors that are on the same
        member that the node is on.

*********************************************************************/

adj_rep()


{
extern struct node        *node_head;
struct frame_in_list      *fil_ptr;
struct neighborlist       *add_neighbor();
struct node_list          *nl_ptr;
struct member             *current_member, *member_ptr;
struct node               *neighbor_ptr, *node1, *node2;
struct node               *node_ptr;
int                       m, n;

for (node_ptr=node_head; node_ptr!=NULL;
  node_ptr=node_ptr->next_node) {

#ifdef DEBUG
    n = node_ptr->node_number;
#endif

                        /****************************************
                        First, get the neighbors that are on the
                        members that frame into *node_ptr:
                        ****************************************/

    for (fil_ptr = node_ptr->frame_in; fil_ptr != NULL;
      fil_ptr = fil_ptr->next){
          current_member = fil_ptr->in_member;

                        /* if the current member has only its two
                        end nodes:      */
          if(current_member->frame_in == NULL) {
              if( current_member->begin_node == node_ptr)
                        neighbor_ptr = current_member->end_node;
              else
```

```
                    neighbor_ptr = current_member->begin_node;
                node_ptr->neighbors =
                    add_neighbor(node_ptr->neighbors,
                                            neighbor_ptr);
            }

                    /* if the current member has nodes on it, we
                    must get the first, or the last, node from
                    the node list:          */
            else {
                if(node_ptr == current_member->begin_node)
                    neighbor_ptr =
                        current_member->frame_in->in_node;
                if(node_ptr == current_member->end_node) {
                    /*search for last node*/
                    nl_ptr = current_member->frame_in;
                    while(nl_ptr->next != NULL)
                        nl_ptr = nl_ptr->next;
                    neighbor_ptr = nl_ptr->in_node;
                }
                node_ptr->neighbors =
                    add_neighbor(node_ptr->neighbors,
                                    neighbor_ptr);
            }
    }


                    /*********************************************
                    Now find the neighbors that are on the same member
                    that the node is on. If the node is on a member.
                    *********************************************/

    if (node_ptr->flags.col == MEMBER){
        member_ptr = node_ptr->on_member;
#ifdef DEBUG
        m = member_ptr->member_number;
#endif
                    /* If the member is not cantilevered at its begin-
                    ning, we search node 2, the node before node ptr*/
        if (member_ptr->flags.begin_cant == 1)
                node2 = NULL;
        else
                node2 = member_ptr->begin_node;
                for( nl_ptr=member_ptr->frame_in;
                     nl_ptr->in_node!=node_ptr;
                     nl_ptr=nl_ptr->next)
                        node2 = nl_ptr->in_node;

        if (node2 != NULL)
                node_ptr->neighbors =
```

```
                        add_neighbor(node_ptr->neighbors,node2);

                        /* Then we search the node after node_ptr */
        if( nl_ptr->next != NULL) {
                neighbor_ptr = nl_ptr->next->in_node;
                node_ptr->neighbors =
                        add_neighbor(node_ptr->neighbors,
                                neighbor_ptr);
                }
        else
                if (member_ptr->flags.end_cant == 0)
                        node_ptr->neighbors =
                              add_neighbor(node_ptr->neighbors,
                                      member_ptr->end_node);
        }
}
return;
}
```

```
/*********************************************************************

                    FLOOR Program Documentation

NAME            triba(), path(), unique(), get_dir(), one_way(), two_way()

SYNOPSIS        struct area *path(nl,n2,side)
                        struct node *nl, *n2;
                        int side;

                struct area *filter(area_ptr)
                        struct area *area_ptr;

                int unique(area_ptr, areas)
                        struct area *area_ptr;
                        struct arealist *areas;

                h_line  *get_dir(path)
                        struct area *path;

                int     interpath(path, harea_ptr)
                        struct area *path;
                        h_area *harea_ptr;

                h_area  *one_way(area_ptr, direction)
                        struct area             *area_ptr;
                        h_line  *direction;

                h_area *twoway(area_ptr)
                        struct area *area_ptr;

EXTERNAL FUNCTIONS

                add_area()
                nright_of()
                corner()
                right_of()
                add_corner()
                tail()
                area_length()
                line()
                hline()
                strict_inside()
                intersect()
                affinity()
                add_point()
                append()
```

```
EXTERNAL VARIABLES member_head

DESCRIPTION      find_triba() loops over the physical members in a
                 floor. Identifying the graphical portions that
                 make up each member, it calls for each of these
                 twice path() that returns successively the circuits
                 to the left and to the right of the portion.
                 After unique() has checked that these  do not duplicate
                 circuits previously found for another portion of the
                 member, one-way(), or two-way(), whichever is proper,
                 is called, to transform the paths in tributary areas,
                 which are added  to the member list of tributary areas.


AUTHORS          Al Bleakley and Pierre Gasztowtt (changed most of
                 the data-structures, added distinction between areas to
                 left and write in find_triba and in path(), trimmed unique()
                 changed one-way() because it gave wrong results,
                 added get_dir())

SIDE EFFECTS

BUGS             None

FILES            strall2.h, stdio.h, list_m.c, geom.c

**********************************************************************/
/*
```

```
     */

/**/

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


extern double SMALL;

/*********************************************************************

        TRIBA()

*********************************************************************/

triba(member_ptr)


struct member *member_ptr;
{

extern int ways;
struct area             *area_ptr, *path();
int                      unique();
struct arealist         *arealist_ptr, *add_area();
struct node_list        *current_nl, *next;
h_area                  *twoway(), *one_way();
h_line                  *direction, *get_dir();



                        /********************************************
                        First build list of adjacent paths:
                        ********************************************/


                        /********************************************
                        If the current member has no nodes on it,
                        it is made of one single portion:
                        ********************************************/
if(member_ptr->frame_in == NULL) {
        area_ptr = path(member_ptr->begin_node,
                        member_ptr->end_node, RIGHT);
        if (area_ptr != NULL){ /* too erly to call unique*/
                member_ptr->rightareas =
                        add_area(member_ptr->rightareas, area_ptr);
        }
```

```
        else
                member_ptr->flags.ext =1;
        area_ptr = path(member_ptr->begin_node,
                        member_ptr->end_node, LEFT);
        if (area_ptr != NULL){ /* still too erly to call unique*/
                member_ptr->leftareas =
                        add_area(member_ptr->leftareas, area_ptr);
        }
        else
                member_ptr->flags.ext = 1;
}

                        /*********************************************
                        if it has nodes on it, it  must be broken
                        into successive portions:
                        *********************************************/
else {
                        /* begin_node - 1st intermediate node*/
   if(member_ptr->flags.begin_cant == 0){
        area_ptr = path(member_ptr->begin_node,
                        member_ptr->frame_in->in_node,RIGHT);
        if (area_ptr != NULL) /* too erly to call unique*/
                member_ptr->rightareas =
                        add_area(member_ptr->rightareas, area_ptr);
        else
                member_ptr->flags.ext = 1;
        area_ptr = path(member_ptr->begin_node,
                        member_ptr->frame_in->in_node, LEFT);
        if (area_ptr != NULL) /* still too erly to call unique*/
                member_ptr->leftareas =
                        add_area(member_ptr->leftareas, area_ptr);
        else
                member_ptr->flags.ext = 1;
    }

                        /* intermediate to nth intermediate nodes*/
    next = member_ptr->frame_in;
    for (current_nl= member_ptr->frame_in;
       current_nl->next != NULL; current_nl= next) {
                next = current_nl->next;
                area_ptr = path(current_nl->in_node,
                                next->in_node, RIGHT);
                if (area_ptr != NULL &&
                   unique(area_ptr, member_ptr->rightareas) == YES)
                     member_ptr->rightareas =
                          add_area(member_ptr->rightareas, area_ptr);
                if (area_ptr == NULL)
                        member_ptr->flags.ext = 1;
                area_ptr = path(current_nl->in_node,
                                next->in_node, LEFT);
```

```
                if (area_ptr != NULL &&
                   unique(area_ptr, member_ptr->leftareas) == YES)
                     member_ptr->leftareas =
                         add_area(member_ptr->leftareas, area_ptr);
                if (area_ptr == NULL)
                         member_ptr->flags.ext = 1;
        }
                                   /* nth intermediate node - end_node*/
        if(member_ptr->flags.end_cant == 0){
                         /*if the member is cantilevered its
                         last_node figured on the member
                         node_list*/
        area_ptr = path(next->in_node,
                member_ptr->end_node, RIGHT);
        if (area_ptr != NULL &&
           unique(area_ptr, member_ptr->rightareas) == YES)
                member_ptr->rightareas =
                         add_area(member_ptr->rightareas, area_ptr);
        if (area_ptr == NULL)
                         member_ptr->flags.ext = 1;
        area_ptr = path(next->in_node,
                member_ptr->end_node, LEFT);
        if (area_ptr != NULL
        && unique(area_ptr, member_ptr->leftareas) == YES)
                member_ptr->leftareas =
                         add_area(member_ptr->leftareas, area_ptr);
        if (area_ptr == NULL)
                member_ptr->flags.ext = 1;
        }
}


                                   /***************************************
                                   then transform all these paths into
                                   tributary areas
                                   ***************************************/

#ifdef CHECK
        printf("%d ways\n", ways);
#endif

for (arealist_ptr = member_ptr->rightareas; arealist_ptr != NULL;
   arealist_ptr = arealist_ptr->next)
      switch (ways) {
        case ONE:
        direction = get_dir(arealist_ptr->uarea.area_ptr);
        if (direction == NULL)
          printf("Missing deck direction next to member %d\n",
```

```
                member_ptr->member_number);
        else{
                arealist_ptr->uarea.triba_ptr->harea_ptr =
                  one_way(arealist_ptr->uarea.area_ptr,
                                                direction);
                arealist_ptr->uarea.triba_ptr->direction =
                                                direction;
        }
        break;
/*      case TWO:
                arealist_ptr->uarea.triba_ptr->harea_ptr =
                  twoway(arealist_ptr->uarea.area_ptr);
        break;*/
      }

for (arealist_ptr = member_ptr->leftareas; arealist_ptr != NULL;
   arealist_ptr = arealist_ptr->next)
      switch (ways) {
        case ONE:
        direction = get_dir(arealist_ptr->uarea.area_ptr);
        if (direction == NULL)
          printf("Missing deck direction next to member %d\n",
                member_ptr->member_number);
        else{
           arealist_ptr->uarea.triba_ptr->harea_ptr =
             one_way(arealist_ptr->uarea.area_ptr,
                                        direction);
                arealist_ptr->uarea.triba_ptr->direction =
                                                direction;
        }
        break;
/*      case TWO:
                arealist_ptr->uarea.triba_ptr->harea_ptr =
                  twoway(arealist_ptr->uarea.area_ptr);
        break;*/
      }


return;

}

/*
```

```
    */
/**/
/****************************************************************

        STRUCT AREA *PATH()

        path finds the shortest path on one side of a portion,
        delimited  by two nodes, of a member. It does a breadth-
        first search. It starts at one node at one end  of a portion;
        it queues its neighboring nodes in an array; then it proceeds
        to visit the nodes in the array, queuing their neighbors (if
        they are not already queued). When it reaches the  node at the
        end of the portion, it reconstitutes the path it  folllowed,
        and returns it. In order to tell if a path is to the left or
        the right of a member, an  assumption is made: all circuits
        are convex; if a circuit is on  one side of a member, all its
        constitutive nodes are on this side.

****************************************************************/
#define MAXQUE 32                          /* Redefinition of MAXQUE*/


struct area *path(n1,n2,side)

struct node *n1, *n2;
int side;

{
struct area     *area_ptr, *add_corner(), *filter();
int             found, n1_n,n2_n, n_n, v_n, qsize, qposition,
                qpos, nright_of();
struct qnode    que[MAXQUE];
struct neighborlist *neighbors, *more_neighbors;
struct node     *n, *neighbor,*node_ptr;
extern struct node *node_head;
                                /**********************************
                                reinitialize queued
                                **********************************/


for (node_ptr = node_head; node_ptr !=NULL;
                node_ptr = node_ptr->next_node) {
                      node_ptr->flags.queued = 0;
}


#ifdef CHECK
n1_n =  n1->node_number;
n2_n = n2->node_number;
#endif
```

```
                                    /************************************
                                    set local variables
                                    ************************************/

que[0].node_ptr = n1;
que[0].parent = -1;
n1->flags.queued = YES;
qposition = 0;
qsize = 1;

found = 0;

                                    /************************************
                                    visit the nodes in the queue
                                    ************************************/
for(n = n1; found==0 && qposition < qsize && n != NULL;
        n = que[++qposition].node_ptr) {
#ifdef CHECK
        n_n = n->node_number;
#endif
        neighbors  = n->neighbors;
                        /*examine the neighbors of this node*/
                        /*queue them as long as qsize < MAXQUE*/
        for(more_neighbors = neighbors;
                more_neighbors != NULL && qsize<=MAXQUE;
                more_neighbors = more_neighbors->next){
                neighbor = more_neighbors->node_ptr;
#ifdef CHECK
                v_n = neighbor->node_number;
#endif
                if (neighbor == n2){
                        if (qposition == 0)
                                ; /*do nothing: in particular*/
                        /*do not queue n2, and do not recognize it*/
                        else {
                                found = 1;
                                break;  /* from this inner loop, and*/
                                /* also from the outer one,  as*/
                                /*found==1 is terminating condition*/
                        }
                }
                else if(neighbor->flags.queued == NO &&
                (nright_of(n1,n2,neighbor) * side) >=0 &&
                qsize < MAXQUE) {
                        neighbor->flags.queued = YES;
                        que[qsize].parent = qposition;
                        que[qsize].node_ptr = neighbor;
                        qsize++;
                        }
                else continue;
```

```
                }
        }

                                /******************************************
                                Breadth first search has concluded,  make
                                a path, if possible, and return it.
                                ******************************************/

if (found == 0) {
/*      printf("No path found to %d side of graphical member %d %d\n",
                side, n1->node_number, n2->node_number);*/
        area_ptr = NULL;
        }
else {
        area_ptr = NULL;
        area_ptr = add_corner(area_ptr, n2);
        for (qpos= --qposition; qpos >= 0; qpos = que[qpos].parent)
                area_ptr = add_corner(area_ptr, que[qpos].node_ptr);
        }

if (area_ptr != NULL){
#ifdef PATH
        print_path(area_ptr);
#endif
        area_ptr = filter(area_ptr);
}

return (area_ptr);       /* even if it is null*/


}

/*
```

```
        */
/***********************************************************************

        INT UNIQUE()

        unique checks that a circuit just built is not already on some
        member's list of  circuits. All the circuits it compares are
        supported by the same physical member, and were built while
        marching along the member in a constant direction: their first
        corner is somewhere up the member, their last corner,
        somewhere down. An area has no edge crossing it.
        Therefore two areas are the same if they share their first
        corner and if they lie on the same side of the member.
        Therefore, unique() only needs to check the two first corners
         of the areas it compares.

***********************************************************************/

int unique(area_ptr, areas)

struct area *area_ptr;
struct arealist *areas;

{
int unique;
struct arealist *more_areas;
struct area *this_area;


unique = YES;

for (more_areas = areas; more_areas != NULL;
  more_areas = more_areas->next) {
        this_area = more_areas->uarea.area_ptr;
        if (this_area->node_ptr == area_ptr->node_ptr &&
         this_area->next->node_ptr == area_ptr->next->node_ptr){
                        unique = NO;
                        break;
                }
}

return (unique);

}

/*
```

```
     */
/**/
/*******************************************************************

        STRUCT AREA *FILTER()

 The function filter removes  non corner
 nodes from an area found by path.  Areas found by path have indeed
 three nodes at least; if there are only three, they won't be colinear

*********************************************************************/

struct area *filter(area_ptr)

struct area *area_ptr;

{
int corner();
struct area *this_area, *imm_area, *further_area;

this_area = area_ptr;
imm_area = area_ptr->next;
further_area = area_ptr->next->next;

                        /* loop to get rid of one point at each cycle*/
while (further_area != NULL) {
         if( corner (this_area->node_ptr, imm_area->node_ptr,
              further_area->node_ptr) == 0)
                /*discard imm_area, don't move this_area*/
              this_area->next = further_area;
         else
              this_area = imm_area;   /*do move this_area*/
        imm_area = further_area;
        further_area = further_area->next;
}

                        /* at this point, we need to leave the loop
                        because further_area is null; but we still
                        have two segments to test*/
further_area = area_ptr;
if( corner(this_area->node_ptr, imm_area->node_ptr,
              further_area->node_ptr) == 0)
           this_area->next = NULL;
else
          this_area = imm_area;
imm_area = further_area;
further_area = further_area->next;
if( corner (this_area->node_ptr, imm_area->node_ptr,
              further_area->node_ptr) == 0)
    return (further_area);
```

```
else return(imm_area);
}

/*
```

```
      */
/*********************************************************************

H_LINE *GET_DIR()

This function finds the direction in which the deck spans within
a given path. It loops over the zones delimiting the directions
in which the deck spans, until it finds one that encloses the path.
*********************************************************************/

h_line *get_dir(path)

struct area *path;

{
extern struct deck_zone *deck_head;
struct deck_zone *deck_z;
h_line          *direction = NULL;

for (deck_z = deck_head;
    (deck_z != NULL && direction== NULL);
    deck_z = deck_z->next){
        if ( interpath(path, deck_z->harea_ptr) == YES)
                direction = deck_z->direction;
}

return(direction);

}

/************************************************************
        INT INTERPATH()
Interpath is a simplified version of clip. It tells if a path
lies within an h_area (which must have its points listed
in clockwise order, like in clip).
************************************************************/



int interpath(path, harea_ptr)

struct area     *path;
h_area          *harea_ptr;

{
h_area          *v1, *v2;
h_point         *p1, *p2;
int inside = YES, count = 1;
struct area     *p;
double          aright_of();
```

```
for (v1 = harea_ptr;
   (v1 != harea_ptr || count++ == 1) && inside == YES;
    v1 = v2){
        v2 = v1->next;
        p1 = v1->point;
        p2 = v2->point;

                            /* check if all the nodes making the path
                            lie inside the half_plan defined by p1,
                            p2 */

        for (p = path; p != NULL && inside == YES;
          p = p->next)
                if ( aright_of(p1, p2, p->node_ptr->point) < 0)
                            inside = NO;
                            /* arightof has a greater tolerance than right_of*/
}

return(inside);

}

/*
```

```
     */
/**/
/***********************************************************************

H_AREA *ONE_WAY()

***********************************************************************/

h_area  *one_way(area_ptr,  direction)

struct area             *area_ptr;
h_line  *direction;


{

int             strict_inside();
h_point         point;
h_line          l, ll;
struct area     *tail_ptr, *tail(), *last_left, *last_not_right,
                *first_right;
h_point         *first_point, *last_point;
h_area          *triba, *add_point();
int             n1, n2, n3, n4;
extern double   SMALL;
double          fabs();
                        /***************************************
                        preliminaries:
                        ***************************************/

triba = NULL;

tail_ptr = tail(area_ptr);
first_point = area_ptr->node_ptr->point;
last_point =  tail_ptr->node_ptr->point;
set_line(first_point, last_point, &ll);

                        /***************************************
                        if member // to deck direction, nothing
                        to do:
                        ***************************************/
if(fabs(ll.x * direction->y - ll.y * direction->x) <
                                        (3 * SMALL))
        return(NULL);

else{                   /***************************************
                        else, tributary area shares its first point
                        with the supporting segment:
                        ***************************************/
```

```
triba = add_point(first_point, triba);

                    /************************************
                    find the last node of the path
                    projecting to the left of the segment:
                    ************************************/
last_left = area_ptr;
while (strict_inside(last_left->next->node_ptr->point, first_point,
             last_point, direction) == -1)
     last_left = last_left->next;


                    /* if last_left is not the first point,
                    create a corner*/
if (last_left != area_ptr){
     set_line(last_left->node_ptr->point,
                    last_left->next->node_ptr->point, &l);
     affinity(first_point, &l, direction, .5, &point);
     triba = add_point(&point, triba);
}

                    /****************************************
                    create corners for all the points
                    projecting inside the segment:
                    ****************************************/
last_not_right = last_left;
while (strict_inside(last_not_right->next->node_ptr->point, first_point,
  last_point, direction) == 0){
     affinity(last_not_right->next->node_ptr->point,
                    &l1, direction, .5, &point);
     triba = add_point(&point, triba);
     last_not_right = last_not_right->next;
}

                    /************************************
                    find the first node of the path
                    projecting to the right of the segment:
                    ************************************/

first_right = last_not_right->next;

                    /* if first_right is not the last point,
                    create a corner*/
if(first_right != tail_ptr){
     set_line(last_not_right->node_ptr->point,
                    first_right->node_ptr->point, &l);
     affinity(last_point, &l, direction, .5, &point);
     triba = add_point(&point, triba);
}
```

```
                         /****************************************
                         finally, tributary area shares its last
                         point with the supporting segment.
                         ****************************************/
   triba = add_point(last_point, triba);

   append(triba, triba);        /*make the tributary area a
                                circular list*/
#ifdef HAREAS
        print_harea(triba);
#endif

   return(triba);
}

}


double hyp(x,y)

double x, y;

{

double sqrt();

return(sqrt(x*x + y*y));

}
```

```
/*******************************************************************

                        FLOOR Program Documentation

NAME            cliploads(), mark(), clip(), carry_down()

SYNOPSIS        cliploads()

                h_area *mark(s1, s2, area_head)
                        h_area          *area_head;
                        h_point         *s1, *s2;

                h_area *clip_half(p1, p2, ha_ptr)
                        h_point *p1, *p2;
                        h_area  *ha_ptr;

                carry_down(area_head, member_ptr, magnitude, direction)
                        h_area          *area_head;
                        struct member   *member_ptr;
                        double          magnitude[3];
                        h_line          *direction;

EXTERNAL FUNCTIONS      copy_harea(),
                        right_of(),
                        line(),
                        intersect(),
                        hp_insert(),
                        hp_delete(),
                        m_to_hlin(),
                        line_distance(),
                        distance(),
                        get_kink(),

STATIC VARIABLES

EXTERNAL VARIABLES member_head, fload_head;

DESCRIPTION       clip_loads  does  a quadruple loop:  over the members,
                their tributary areas, and over the  f_loads and their
                edges. Each edge of each f_load area is thus clipped
                against each tributary area of each member. When the
                intersection of a  tributary area and a floor load has more
                than one edge, it is carried down.

AUTHORS         Al Bleakley and Pierre Gasztowtt (for carry_down)

SIDE EFFECTS    The members now have line_loads on their load_diagrams.

BUGS            None
```

```
FILES           strall2.h, stdio.h, list_m.c, geom.c

*************************************************************************/
/*
```

```
      */

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


/**************************************************************************
 This is CLIPLOADS().
**************************************************************************/


cliploads(m_ptr)

struct member *m_ptr;

{
struct arealist          *areas_ptr;
extern struct f_load     *fload_head;
struct f_load            *fload_ptr;
h_area                   *this_harea, *copy_harea(), *clip();
int                      c_m, f_n, l_n;



#ifdef HAREAS
c_m = m_ptr->member_number;
#endif
for (areas_ptr =m_ptr->rightareas; areas_ptr != NULL;
  areas_ptr =areas_ptr->next)
    for(fload_ptr= fload_head; fload_ptr!= NULL;
        fload_ptr=fload_ptr->next) {
#ifdef HAREAS
        l_n = fload_ptr->number;
#endif

        if (areas_ptr->uarea.triba_ptr->harea_ptr != NULL){
                this_harea = copy_harea( fload_ptr->area);
                this_harea = clip(areas_ptr->uarea.triba_ptr->harea_ptr,
                                  this_harea,  RIGHT);
                if (this_harea != NULL){
                        carry_down(this_harea, m_ptr,
                                &(fload_ptr->magnitude[0]),
                                areas_ptr->uarea.triba_ptr->direction);
                        free_harea(this_harea);
                }
        }
    }
  for (areas_ptr =m_ptr->leftareas; areas_ptr !=NULL;
    areas_ptr =areas_ptr->next)
    for(fload_ptr= fload_head; fload_ptr != NULL;
```

```
        fload_ptr=fload_ptr->next) {
        if (areas_ptr->uarea.triba_ptr->harea_ptr != NULL){
                this_harea = copy_harea( fload_ptr->area);
                this_harea = clip(areas_ptr->uarea.triba_ptr->harea_ptr,
                                        this_harea, LEFT);
                if (this_harea != NULL){
                        carry_down(this_harea, m_ptr,
                                &(fload_ptr->magnitude[0]),
                                areas_ptr->uarea.triba_ptr->direction);
                        free_harea(this_harea);
                }
        }
}
return;
}

/**********************************************************************
        H_AREA CLIP()

This function intersects two h_areas. It walks along the vertices that
make the first area, intersecting  the half plans to their right or left,
depending on the third argument, and the
second h_area. The first area must  be convex.
**********************************************************************/

h_area *clip(h_area1, h_area2, side)

h_area *h_area1, *h_area2;

{
h_point                 *p1, *p2;
double                  cx1, cy1, cx2, cy2;
h_area                  *v1, *v2, *mark(), *clip_half();
int                     count = 1;



for (v1 = h_area1; v1 !=h_area1 || count++== 1; v1 = v2){
        v2 = v1->next;
        p1 = v1->point;
        p2 = v2->point;

        /*mark prepares for clip_half:*/
        h_area2 =mark(p2, p1, h_area2);
#ifdef HAREAS
        print_harea(h_area2);
#endif
                                /*clip_half intersects the area
                                with the right of (p2, p1)*/
        h_area2 = clip_half(p2, p1, h_area2, side);
```

```
#ifdef HAREAS
        print_harea(h_area2);
#endif

                                /* if clip_half returns an area with one*/
                                /* or two points, there is no need to   */
                                /* continue the clipping*/
        if(h_area2 == NULL || h_area2->next == h_area2 ||
                h_area2->next->next == h_area2)
                                break;
}


if(h_area2 != NULL && h_area2->next != h_area2 &&
   h_area2->next->next != h_area2)
        return(h_area2);

else return(NULL);
}

/*
```

```
     */
/************************************************************************
        H_AREA *MARK()

        mark insert the  points where a line meets the edge of an area
        in the list of points (or h_area sturcture) representing this
        area

*************************************************************************/

h_area *mark(s1, s2, area_head)

h_area          *area_head;
h_point         *s1, *s2;
{


int             count;
double          u1, u2, vright_of();
h_area          *prev_vert, *this_vert;
h_point         i1, *p1, *p2;
h_line          line1, line2;

                                        /* set the loop*/
        count = 0;
        prev_vert   = area_head;
        p1    = prev_vert->point;
        u1 = vright_of(s1, s2, p1);  /*since right_of is actually a
                                        determinant, the order of the points
                                            is flexible*/

                                        /* then loop over prev_vert*/
    while( prev_vert != area_head || count++ < 1) {
                this_vert = prev_vert->next;
                p2 = this_vert->point; /*p1, u1 are updated at the end*/
                u2 = vright_of(s1, s2, p2);
                if(u1 >= 0 && u2 >= 0)
                        ;               /* do nothing:
                                    both vertices are outside the line*/
                else if(u1 <= 0 && u2 <= 0)
                      ;                 /* do nothing:
                                    both vertices are inside the line*/
                else {
                                    /* clip line intersects this side        */
                        set_line(p1, p2, &line1);
                        set_line(s1, s2, &line2);
                                    /* insert the intersection in the circular*/
                                            /*list    */
                        intersect(&line1, &line2, &i1);
                        hp_insert(prev_vert, &i1);
```

```
                        }
                                 /* carry the loop forward*/
                prev_vert       = this_vert;
                                 /* in order to save some computations */
                                 /* swap p1 and p2 for next  pass: */
                u1 = u2;
                p1 = p2;
        }
return(area_head);
}


/*
```

```
    */
/*******************************************************************
        H_AREA *CLIP_HALF()

        clip filters a homogeneous area after marking to remove
        points that are outside of the loaded area.

*******************************************************************/

h_area *clip_half(p1, p2, ha_ptr, side)


h_point *p1, *p2;               /* side used for clipping       */
h_area  *ha_ptr;                /* tributary area being clipped */
int     side;
{

extern double SMALL;
int     count = 0;
double  vright_of();
h_area  *area_head;             /* current h_area               */
double  cx3, cy3;


        area_head = ha_ptr;


        while(count++ < 1 || ha_ptr != area_head) {
                while((vright_of(p1, p2, ha_ptr->next->point) * side)>0
                        && ha_ptr != ha_ptr->next) {
                        cx3 = ha_ptr->next->point->x;
                        cy3 = ha_ptr->next->point->y;
                        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);
}



/*
```

```
      */
/**********************************************************************

        CARRY_DOWN()

        carry_down() takes the clipped  intersection polygon found in
        clippoly() and projects the load onto   the member to
        update its load_diagram. It projects the load side by side:
        the polygon is een as a superposition of ficitious polygons,
        some with a positive load, others with a negative load,
        that have their base on the member (see Al''s thesis).
        This version of carry_down  differs from Al's original one in
        that:

            - it projects the loads along the deck_direction;

            - it writes a load_diagram.

**********************************************************************/



carry_down(area_head, member_ptr, magnitude, direction)

h_area          *area_head;
struct member   *member_ptr;
double          magnitude[3];
h_line          *direction;

{

int                     m, count = 0;
double                  l1, l2,  x1, x2,
                        distance();
h_point                 p1, p2;
h_line                  l;
h_area                  *prev_vert, *this_vert;


#ifdef HAREAS
m = member_ptr->member_number;
#endif

m_to_hlin(member_ptr, &l);
                                /****************************************
                                set the loop
                                ****************************************/
count = 0;
prev_vert = area_head;
```

```
project(prev_vert->point, &l, direction, &p1);
l1 = distance(&p1, prev_vert->point)/12;
x1 = distance(&p1, member_ptr->begin_node->point)/12;


                                /*************************************
                                then loop over prev_vert
                                *************************************/
while(prev_vert != area_head || ++count== 1){
        this_vert = prev_vert->next;
        project(this_vert->point, &l, direction, &p2);
        l2 = distance(&p2, this_vert->point)/12;
        x2 = distance(&p2, member_ptr->begin_node->point)/12;
        /*free(p2)*/
        if (x1 < x2) /* for improved legibility of back-ups,
                        load exerted downwards is counted as positive*/
                carry(member_ptr->dia_ptr, x1, x2, l1, l2, magnitude);
        if (x2 < x1)
                carry(member_ptr->dia_ptr, x2, x1, l2, l1, magnitude);
        prev_vert = this_vert;
        l1 = l2;
        x1 = x2;
}

return;
}

carry(dia_ptr, x1, x2, l1, l2, magnitude)

struct diagram *dia_ptr;
double x1, x2, l1, l2, magnitude[3];

{

double x, dx, dl, l, to;
struct diagram  *kink1, *kink2, *kink, *get_kink();
int i;

kink1 = get_kink(dia_ptr, x1);
kink2 = get_kink(dia_ptr, x2);
dx = x2 - x1;
dl = l2 - l1;
to = dl/dx;
if (dx < 0)
        return;

for (kink = kink1; kink != kink2; kink = kink->next){
        x = kink->x;
        l = (x - x1) * to + l1;
        for (i = 0; i < 3; i++){
```

```
                kink->right_mag[i] +=
                        1 * magnitude[i];
                kink->next->left_mag[i] +=
                        1 * magnitude[i];
        }
}

return;

}

/*********************************************************************

        CARRY_DOWN2()

        This is a previous version of carry_down, more similar to
        Al's original. It is adequate to two_way flooring systems.

#include <math.h>




carry_down(area_head, member_ptr, magnitude)

h_area          *area_head;
struct member   *member_ptr;
double          magnitude[3];

{

int                     i,m, count = 0;
double                  l1, l2, x1, x2, dx, dl, x, y,l1,
                        line_distance(),distance(),sqrt(), fabs();
h_line                  *l, *m_to_hlin();
h_point                 *p1, *p2;
h_area                  *prev_vert, *this_vert;
struct diagram          *kink1, *kink2, *kink, *get_kink();

#ifdef HAREAS
m = member_ptr->member_number;
#endif

l = m_to_hlin(member_ptr);

count = 0;
prev_vert = area_head;
x = prev_vert->point->x;
y = prev_vert->point->y;
l1 = line_distance(prev_vert->point, l)/12;
```

```
10 = distance(prev_vert->point, member_ptr->begin_node->point)/12;
x1 = sqrt(fabs(10*10 - 11*11));

kink1 = get_kink(member_ptr->dia_ptr, x1);


while(prev_vert != area_head || ++count== 1){
        this_vert = prev_vert->next;
        12 = line_distance(this_vert->point, 1)/12;
        10 = distance(this_vert->point,
                        member_ptr->begin_node->point)/12;
        x2 = sqrt(fabs(10*10 - 12*12));

        kink2 = get_kink(member_ptr->dia_ptr, x2);
        if (x1 < x2){
                dx = x2 - x1;
                dl = 12 - 11;
                for (kink = kink1; kink != kink2; kink = kink->next){
                        x = kink->x;
                        11 = (x - x1) * dl + 11;
                        for (i = 0; i < 3; i++){
                                kink->right_mag[i] +=
                                                11 * magnitude[i];
                                kink->next->left_mag[i] +=
                                                11 * magnitude[i];
                        }
                }
        }
        if (x2 < x1){
                dx = x1 - x2;
                dl = 11 - 12;
                for (kink = kink2; kink != kink1; kink = kink->next){
                        x = kink->x;
                        11 = (x - x2) * dl + 12;
                        for (i = 0; i < 3; i++){
                                kink->right_mag[i] -=
                                                11 * magnitude[i];
                                kink->next->left_mag[i] -=
                                                11 * magnitude[i];
                        }
                }
        }
        prev_vert = this_vert;
        11 = 12;
        x1 = x2;
        kink1 = kink2;

}

return;
```

```
}

************************************************************************/
```

```
/*********************************************************************

                     FLOOR Program Documentation

NAME            analyze(), set_kink()

SYNOPSIS

EXTERNAL FUNCTIONS      get_kink()
                        to_local()


STATIC VARIABLES

EXTERNAL VARIABLES member_head

DESCRIPTION

AUTHORS         Al Bleakley and Pierre Gasztowtt (put in place
                diagram data structure to compute Mmax)

SIDE EFFECTS

BUGS            None

FILES           strall2.h, stdio.h, list_m.c, geom.c

*********************************************************************/
/*
```

```
    */

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

/*
```

```
        */
/*********************************************************************

        ANALYZE()
        This function goes over a member, once all the ones
        framing in it have been solved, and solves it.

***************************************************************/



analyze(member_ptr)

struct member *member_ptr;

{

int     m_n, c;
double  L, x, dx, dv, v, v0, v1, *m, r, ml, mc;
double  to_local();
struct member  *m_ptr;
struct node_list *nl_ptr;
struct node *node_ptr;
struct diagram *kink, *prev_kink, *get_kink(), *new_kink, *make_kink();

struct frame_in_list *fil_ptr;

#ifdef DEBUG
m_n = member_ptr->member_number;
#endif

L = member_ptr->last_kink->x;

for (nl_ptr = member_ptr->frame_in; nl_ptr != NULL;
   nl_ptr = nl_ptr->next){
        node_ptr = nl_ptr->in_node;
        for (fil_ptr=node_ptr->frame_in; fil_ptr!=NULL;
          fil_ptr=fil_ptr->next){
                m_ptr = fil_ptr->in_member;
                x = to_local(member_ptr,node_ptr->point);
                kink = get_kink(member_ptr->dia_ptr, x);
                if(m_ptr->begin_node == node_ptr)
                        for (c = 0; c < 3; c++)
                                kink->right_v[c] -=
                                        m_ptr->dia_ptr->right_v[c];
                else if(m_ptr->end_node == node_ptr)
                        for (c = 0; c < 3; c++)
                                kink->right_v[c] += /* notice double
                                                    sign reversal */
                                        m_ptr->last_kink->left_v[c];
```

```
                    /* end_reaction = -1 * end_shear */
        }
}

                                /*********************************
                                We can now get  first idea of
                                shear and moment distribution:
                                *********************************/
for (prev_kink = member_ptr->dia_ptr; prev_kink->next != NULL;
  prev_kink = kink){
        kink = prev_kink->next;
        set_kink(kink, prev_kink);
}

                                /*********************************
                                Adjust to meet end conditions:
                                *********************************/

if(member_ptr->flags.begin_cant == 1)
                                /* we want shear[0] = moment[0] = 0*/
                ;               /* nothing to do: almost sad we
                                decided cantilevers should be
                                built-in at begin_extremity */

else if(member_ptr->flags.end_cant == 1){

        for(c = 0; c < 3; c++){
                v1 = member_ptr->last_kink->right_v[c];
                m1 = member_ptr->last_kink->moment[c];
                mc =  ( m1 - L * v1);
                for(kink=member_ptr->dia_ptr; kink!=NULL;
                 kink = kink->next){
                    kink->left_v[c] -= v1;
                    kink->right_v[c] -= v1;
                    kink->moment[c] -= (kink->x *v1 + mc);
                }
      }
}


else {
                                /* we want moment[0] = moment[L] = 0*/
     for(c = 0; c < 3; c++){
        v0 =  - (member_ptr->last_kink->moment[c]/ L);
        for(kink=member_ptr->dia_ptr; kink!=NULL; kink = kink->next){
                kink->left_v[c] += v0;
                kink->right_v[c] += v0;
                kink->moment[c] += kink->x  * v0;
        }
    }
```

```
}

                        /*******************************************
                        Adjust for cantilever at beginning:
                        *******************************************/
if (member_ptr->begin_camr != NULL)
     for(c = 0; c < 3; c++){
        v = member_ptr->begin_camr->dia_ptr->moment[c]/L;
                        /* cantilevers are built in at begin_extr*/
        for(kink=member_ptr->dia_ptr; kink!=NULL; kink = kink->next){
            kink->left_v[c] -= v;
            kink->right_v[c] -= v;
            kink->moment[c] += v*(L - kink->x);
        }
     }

                        /*******************************************
                        and at end:
                        *******************************************/
if (member_ptr->end_camr != NULL)
     for(c = 0; c < 3; c++){
        v = member_ptr->end_camr->dia_ptr->moment[c]/L;
        for(kink=member_ptr->dia_ptr; kink!=NULL; kink = kink->next){
            kink->left_v[c] += v;
            kink->right_v[c] += v;
            kink->moment[c] += v * kink->x;
        }
      }

                        /****************************************
                        Compute the  maximal moment:
                        ****************************************/

                        /* First, spot its location:*/
if(member_ptr->flags.end_cant == 1)
        member_ptr->kink_max = member_ptr->dia_ptr;
else{
    for (kink=member_ptr->dia_ptr; kink->next != NULL &&
      member_ptr->kink_max == NULL; kink=kink->next){

        if (kink->left_v[0]*kink->right_v[0] <= 0)
                member_ptr->kink_max = kink;

        else if (kink->next == NULL)
                        /* possible if cantilevered at begin */
```

```
                member_ptr->kink_max = kink;

        else if (kink->right_v[0]*kink->next->left_v[0] < 0){
                        /* it is between these two kinks */
                        /* let us spot exactly where:   */
                dx = kink->next->x - kink->x;
                dv = kink->next->left_v[0] - kink->right_v[0];
                x = kink->x - dx * kink->right_v[0] / dv;
                member_ptr->kink_max = new_kink =
                        make_kink(kink, kink->next,x);
                set_kink( new_kink, kink);
        }

                        /* else keep searching */
    }
}
/* to obtain begin_reac (resp end_reac), */
/* take diagram->right_v (resp - last_kink->left_v)*/
/* to obtain begin_moment (resp end_moment), */
/* take diagram_moment (resp + last_kink->moment)*/


member_ptr->flags.solved = 1;

return;
}


/*
```

```
      */
/***************************************************************

        SET_KINK()

***************************************************************/

set_kink(this_kink, prev_kink)

struct diagram *this_kink, *prev_kink;


{

double dx, v_increment[3];
int c;


dx = this_kink->x - prev_kink->x;

for(c = 0; c < 3; c++){
        v_increment[c] = prev_kink->right_v[c] -
                       .5 * dx * (prev_kink->right_mag[c] +
                                  this_kink->left_mag[c]);
                       /* We have given a positive sign to
                       loads acting downwards */
        this_kink->left_v[c] = v_increment[c];
        this_kink->right_v[c] += v_increment[c];
        this_kink->moment[c] = prev_kink->moment[c]
                              + dx * prev_kink->right_v[c]
                              - dx*dx * (prev_kink->right_mag[c]/3 +
                                  this_kink->left_mag[c]/6);
}

return;

}
```

```
/*************************************************************************

                    FLOOR Program Documentation

NAME                size(), make_adeq(), beam_table(), append_entries(),
                    append_table()

SYNOPSIS

EXTERNAL FUNCTIONS

STATIC VARIABLES

EXTERNAL VARIABLES

DESCRIPTION         Given a member, size()  goes through the table in which
                    the various WF-sections are sorted by depth.
                    For each depth_family, it calls make_adeq()
                    which returns an adequate WF-section, and specifies a
                    grade and a camber. It then compares this new proposal
                    with the previous one, and adopts the new one if it
                    is more economical.

                    beam_table() loads the beam_table.

AUTHORS              Pierre Gasztowtt

SIDE EFFECTS

BUGS                None

FILES               strall2.h, stdio.h, list_m.c, geom.c

*************************************************************************/
/*
```

```
     */
#include "strall2.h"
#include <stdio.h>
#define max(A, B) ((A) > (B) ? (A) : (B))
#define malloc getmem




size(m_ptr)

struct member *m_ptr;

{
extern double d_max;
extern struct wf_table *wf_head;
double Fy = 50;
double  Mmax, Ma, Mb, Va, Vb, L, I_defl, S_bend, Aw_shear, fabs();
struct wf_table  *sections;
int c, make_adeq();
struct section *best_sec, this_sec;
char    *malloc();
int count = 0;
                                /**************************************
                                Set design criteria:
                                **************************************/
Mmax = 0;
Ma = 0;
Mb = 0;
Va = Vb = 0;
for (c = 0; c < 3; c++){
        Mmax += m_ptr->kink_max->moment[c];
        Ma += m_ptr->dia_ptr->moment[c];
        Mb += m_ptr->last_kink->moment[c];
        Va += m_ptr->dia_ptr->right_v[c];
        Vb += m_ptr->dia_ptr->left_v[c];
}

S_bend = 12 * fabs(Mmax) / (0.66 * Fy);

Aw_shear = max(fabs(Va), fabs(Vb)) / (.4 * Fy);



L = m_ptr->last_kink->x;

I_defl = 0;
/*I_defl = 2.23 * L * fabs(Mmax - 0.1 * (Ma + Mb));*/
```

```
                                /***********************************
                                Allocate memory, set a default:
                                ************************************/
while((best_sec =  (struct section *) malloc (sizeof(struct section)))
                == NULL)
                        printf("memory\n");
best_sec->wf_ptr = wf_head->entries;
best_sec->grade = GRADE_50;


                                /***************************************
                                Now, look in the table for best section
                                possible
                                ***************************************/
for (sections = wf_head; sections != NULL && count < 4;
  sections = sections->more){
        if (sections->d <= d_max)
                if ( make_adeq(sections->entries, sections->last,
                        S_bend, I_defl, Aw_shear, &this_sec) == 0)
                  count++;
                else {
                   count == 0;
                   if (this_sec.wf_ptr->weight <=
                        best_sec->wf_ptr->weight){
                                best_sec->wf_ptr = this_sec.wf_ptr;
                                best_sec->grade = this_sec.grade;
                   }
                }
}

m_ptr->section = best_sec;

}



int make_adeq(same_d, last,  S_bend, I_defl, Aw_shear, this_sec)

struct section          *this_sec; /* a place where to write the results*/
double                  S_bend, I_defl, Aw_shear;
struct wf_entry         *same_d, *last;

{

struct wf_entry  *this_entry;
static int count = 0;
                                /***************************************
                                Find adequate wf_shape for this depth:
                                ***************************************/
```

```
if ((same_d->S ) <  S_bend){    /* if even the biggest section for
                                Depth Is Inadequate, Do Not Insist */
#ifdef COUNT
        count ++;
#endif
        return(0);
}

else {
        if (last->S >= S_bend     /* check now if smallest section */
        /*  && last->I >= I_defl            might do */
          && last->Aw >= Aw_shear){
#ifdef COUNT
                 ++count;
#endif
                this_entry = last;
        }
        else                       /* if not, search*/
            for (this_entry = same_d;
                  ( (this_entry->next_wf->compact_fact *
                        this_entry->next_wf->S  >= S_bend)
                  && this_entry->next_wf->I >= I_defl
                  && this_entry->next_wf->Aw >= Aw_shear);
                  this_entry = this_entry->next_wf){
#ifdef COUNT
                ++ count;
#endif

                                ;               /* keep updating this_entry */
        }
        this_sec->wf_ptr = this_entry;
                                /*********************************************
                                Check if A36 would be OK:
                                *********************************************/
        this_sec->grade = GRADE_50;
#ifdef COUNT
        printf("%d\n", count);
#endif
        return (1);
}
}




beam_table()

{
FILE *fp;
```

```
extern struct wf_table *wf_head;
struct wf_table *tail_table, *append_table();
struct wf_entry *entry_p, *append_entries();
char            char_w, char_x, name[10], *malloc();
int             count = 0, int_wt, int_d;
double          last_d, d, tw, last_Aw, Aw = 0;

wf_head = NULL;

fp = fopen("sections.dat", "r");

if(fp == NULL){
        printf("\nUnable to find section data file. \n");
        return;
        }




while(fscanf(fp, "%s", name) != EOF){

        sscanf(name, "%c%d%c%d",
                &char_w, &int_d, &char_x, &int_wt);
        fscanf(fp, "%*lf%lf%lf", &d, &tw);
        last_Aw = Aw;
        Aw = d * tw;
        if(++count == 1){
                last_d = int_d;
                while((wf_head = tail_table = (struct wf_table *)
                        malloc(sizeof(struct wf_table))) == NULL)
                        printf("memory\n");
                tail_table->d = ( (double) int_d);
                while((entry_p = tail_table->entries = tail_table->last =
                 (struct wf_entry *) malloc(sizeof(struct wf_entry))) == NULL
                        printf("memory\n");
        }
        else if (int_d < last_d || Aw > last_Aw){
                last_d = int_d;
                tail_table = append_table(tail_table);
                tail_table->d = ( (double) int_d);
                while((entry_p = tail_table->entries = tail_table->last =
                   (struct wf_entry *) malloc(sizeof(struct wf_entry))) == NU
                   printf("memory\n");
        }
        else {                  /* tail_table is the correct_place */
                                /* and it already has an entry*/
                entry_p = tail_table->last = append_entries(tail_table->last)
        }
                                /*********************************************
                                We can now fill in the entry
```

```
                                        *********************************/
        entry_p->weight = ((double) int_wt);
        entry_p->d = d;
        entry_p->Aw = d * tw;
        fscanf(fp,"%*lf%*lf%*lf%*lf%lf%lf%*lf%*lf%*lf%*lf%*lf%*lf%*lf",
                &entry_p->I,
                &entry_p->S);
        entry_p->compact_fact = 1; /* for the moment */

}

fclose(fp);

return;

}



struct wf_table *append_table(tail_table)

struct wf_table *tail_table;

{

struct wf_table *new_tail;
char            *malloc();

while((new_tail = (struct wf_table *) malloc(sizeof (struct wf_table)))
        == NULL)
        printf("memory\n");

new_tail->more = NULL;

tail_table->more = new_tail;

return(new_tail);
}


struct wf_entry *append_entries(tail_entries)

struct wf_entry *tail_entries;

{

struct wf_entry *new_tail;
char *malloc();

while((new_tail = (struct wf_entry *) malloc(sizeof (struct wf_entry)))
```

```
        == NULL)
        printf("memory \n");

new_tail->next_wf = NULL;

tail_entries->next_wf = new_tail;

return(new_tail);
}
```

File geom.c

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

extern double SMALL;
/*
```

```
        */
/*************************************************************************

        INT CORNER(), INT NRIGHT_OF(), DOUBLE RIGHT_OF(), INT
        ARIGHT_OF(), DOUBLE VRIGHT_OF();

  The function right_of() takes the
 homogeneous representation for a point and for the two end points of
a line segment and, computing a determinant,
  checks to see if the point is to the right of the
 segment.
 The function corner takes three node pointers and deternmines whether
they form a corner.  Used in checking paths for graph manipulation.

*************************************************************************/

int corner(n1, n2, n3)

struct node     *n1, *n2, *n3;

{
double  fabs(), flag, right_of();


        if(n1 == NULL || n2 == NULL || n3 == NULL)
                printf("Error: NULL node pointer passed to corner()");


        flag = right_of(n1->point, n2->point, n3->point);

        if(fabs(flag) < SMALL)  /* the points are colinear*/
                return(0);      /* allow tolerance for round off*/
        else                            /* the points are not colinear  */
                return(1);              /* so they must be a corner     */
}



int nright_of(n1, n2, n3)



struct node *n1, *n2, *n3;

{
double fabs(), flag, right_of();

        if(n1 == NULL || n2 == NULL || n3 == NULL)
                printf("Error: NULL node pointer passed to corner()");
```

```
        flag = right_of(n1->point, n2->point, n3->point);

        if(fabs(flag) == 0)
                return(0);      /* the points are colinear*/

        else if(flag >0)
                return(1);
        else return(-1);
}

double aright_of(p, p1, p2)

h_point *p, *p1, *p2;

{
double vright_of(), flag, fabs();
extern double SMALL;

flag = vright_of(p,p1,p2);

if(fabs(flag) < (SMALL * SMALL * 15))
        flag = 0;

return (flag);

}


double right_of(p, p1, p2)

h_point *p, *p1, *p2;

{
double vright_of(), flag, fabs();
extern double SMALL;

flag = vright_of(p,p1,p2);

if(fabs(flag) < (SMALL * SMALL))
        flag = 0;

return (flag);

}

double vright_of(p, p1, p2)

h_point *p, *p1, *p2;                   /* currently assumes 2d graphics*/

{
```

```
extern double SMALL;
double  checksum, coeffx, coeffy, coeffw, fabs();
        /* see Pavlidis p. 328                                                  */

        if(p->w != 1 && p->w != 0) {     /* ensure that all w componants */
                p->x /= p->w;            /* are positive 1               */
                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->x;
        coeffw = p1->x * p2->y - p1->y * p2->x;

        checksum = p->x * coeffx + p->y * coeffy + p->w * coeffw;

        if(fabs(checksum)< (SMALL/1000))
                return(0);
        else
                return(checksum);        /* checksum < 0 p is to right   */
                                         /* checksum = 0 p is on p1-p2   */
                                         /* checksum > 0 p is to left    */
}


/*
```

```
      */
/**************************************************************************

        H_POINT PROJECT(), AFFINITY(), INTERSECT();
        DOUBLE LINE_DISTANCE();  INT INSIDE();

 Here are a few functions that involve lines.

 The function intersect() takes the
 homogeneous representation for two lines and returns their point of
 intersection.

 line_distance() finds the perpendicular distance from a point to a
 line. project() finds its projection on this line. inside() tells if
 a point projects inside or outside of a segment.

**************************************************************************/




intersect(l1, l2, point)

h_line  *l1, *l2;
h_point *point;

{


        /* see Pavlidis p. 324                                          */

        point->x = l1->y*l2->w - l2->y*l1->w;
        point->y = l2->x*l1->w - l1->x*l2->w;
        point->w = l1->x * l2->y - l1->y * l2->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;

}

affinity(point, l, direction, r, pl)

h_point         *point, *pl;
h_line          *l, *direction;
float           r;

{

project(point, l, direction, pl);
pl->x = pl->x + (point->x - pl->x)*r;
pl->y = pl->y + (point->y - pl->y)*r;

return;

}


project(point, l, direction, pl)

h_point         *point, *pl;
h_line          *l, *direction;

{

h_line  line2;
double  fabs();
extern double SMALL;

if(fabs(point->x * l->x + point->y * l->y + point->w * l->w)
        <= (SMALL * SMALL)/2000)
        set_point(pl, point->x, point->y);

else{
        line2.x = direction->x;
        line2.y = direction->y;
        line2.w = -point->x * direction->x - point->y * direction->y;
        intersect(l, &line2, pl);
}


return;

}

int     strict_inside(point, pl, p2, direction)
```

```
h_point *point, *p1, *p2;
h_line  *direction;


{
h_line  l;
int     flag;
h_point p0;
double  d0_1, d2_1, d0_2, fabs(), right_of();
extern double SMALL;

                                /***************************
                                compute projection of p on (p1,p2)
                                **********************************/


set_line(p1, p2, &l);
project(point, &l, direction, &p0);

                                /*******************************
                                then compute distances from
                                p0 to p1 and p2, and between p1
                                and p2.
                                ********************************/
d2_1= fabs(p2->x - p1->x) + fabs(p2->y - p1->y);
d0_1= fabs(p0.x - p1->x) + fabs(p0.y - p1->y);
d0_2= fabs(p0.x - p2->x) + fabs(p0.y - p2->y);

                                /*****************************
                                draw conclusion:
                                ******************************/

if ((d0_1 < d2_1 - SMALL) && (d0_2 < d2_1 - SMALL))
                           /* p0 is half-way between p1 and p2:
                              p projects strictly inside */
        flag = 0;
else{
        if (d0_1 < d0_2)
                        /* p, closer to point 1 than point 2,
                             is to the left of segment 1_2*/
                flag = -1;
        else
                         /* p is to the right of segment 1_2*/
                flag = 1;
}


return(flag);

}
```

```
double line_distance(pl, 11)

h_point          *pl;
h_line           *11;

{
double  distance, fabs(), hyp();

        distance = fabs(11->x * pl->x + 11->y * pl->y + 11->w)
                        /hyp(11->x, 11->y);

        return(distance);       /* always return positive distance      */
}



/*
```

```
    */
/**********************************************************************
        H_LINE *LINE(), *M_TO_HLIN();

line() takes two points  and returns the homogeneous representation
for the line that they   define.

**********************************************************************/



m_to_hlin(member_ptr, line_ptr)

struct member   *member_ptr;
h_line *line_ptr;

{

h_point         *p1, *p2;


        p1 = member_ptr->begin_node->point;
        p2 = member_ptr->end_node->point;

        set_line(p1, p2, line_ptr);

        return;

}




h_line *line(p1, p2, line1)

h_point *p1, *p2;
h_line  *line1;                 /*address where to write the result*/

{


        /* 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.                     */

}


set_line(p1, p2, line1)

h_point *p1, *p2;
h_line  *line1;                     /*address where to write the result*/

{


        /* 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;                     /* return address of struct     */
                                          /* h_line.                     */

}

/*
```

```
       */
/********************************distance******************************/

double distance(p1, p2)

h_point *p1, *p2;

{
double d, dx, dy, sqrt();

dx = p1->x - p2->x;
dy = p1->y - p2->y;

d = sqrt(dx * dx + dy * dy);

return(d);
}




/*
```

```
       */
/*********************************************************************

        DOUBLE LENGTH(), TO_LOCAL()

length() finds the length of a member, to_local the local coordinate
of a point on a member.

*********************************************************************/

double length(member_ptr)                /* Find member length   */
struct member *member_ptr;

{
  double xb, xe, yb, ye, x_length, y_length, length, sqrt();
  int   m;

  m = member_ptr->member_number;


  xb=member_ptr->begin_node->point->x;
  xe = member_ptr->end_node->point->x;
  x_length = (xb - xe)/12;
  yb =member_ptr->begin_node->point->y;
  ye =member_ptr->end_node->point->y;
  y_length = (yb - ye)/12;

  length = sqrt(x_length*x_length + y_length*y_length);

  return(length);

}



double to_local(member_ptr, point)

struct member *member_ptr;
h_point  *point;


{

double  tl_sq,  tl_x, tl_y, tl_x0, tl_y0, xlength, ylength, local;

tl_x = point->x;
tl_y = point->y;
tl_x0 =  member_ptr->begin_node->point->x;
tl_y0 =  member_ptr->begin_node->point->y;
```

```
xlength = (tl_x - tl_x0)/12;
ylength = (tl_y - tl_y0)/12;


tl_sq = xlength*xlength + ylength*ylength;
local = sqrt(tl_sq);

return(local);
}

/*
```

```
        */
/**************************set_point*************************************/
set_point(point, x, y)

h_point *point;
double x, y;


{

point->x =x;
point->y =y;
point->z =0;
point->w =1;

return;
}
```

File li.c

```
#include "strall2.h"
#include <stdio.h>
#include <math.h>
#define IBM
#define malloc getmem
#define free rlsmem

extern double SMALL;
/***********************add_segment***************************/
/* First comes this function, which attaches a new segment to */
/*the floor*/

struct segment *add_segment() {

struct segment *segment1;
extern struct segment *seg_head;
char *malloc();

while((segment1 = (struct segment *)              /*    allocate memory*/
                  malloc(sizeof(struct segment))) == NULL)
        printf("memory\n");

        segment1->begin_x =segment1->end_x = segment1->begin_y =
                segment1->end_y = 0;



segment1->next = seg_head;

seg_head = segment1;


return(segment1);

}

/********************add_member********************************/
/* This function attaches a new member to the floor      */


struct member *add_member() {

struct member *member1;
extern struct member *member_head;
static int member_count = 0;
char *malloc();

while((member1 = (struct member *)              /*    allocate memory*/
                  malloc(sizeof(struct member))) == NULL)
```

```
                printf("memory\n");

member1->member_number = member_count++;
memb_init(member1);

member1->next = member_head;     /* pointers in the list */

member_head = member1;           /* beginning of list    */

return(member1);
}



memb_init(memb_ptr)

struct member *memb_ptr;

{
int c, m;
struct diagram *last;

m = memb_ptr->member_number;
memb_ptr->flags.solved = 0;
memb_ptr->flags.begin_cant = 0;
memb_ptr->flags.end_cant = 0;
memb_ptr->flags.ext = 0;
memb_ptr->begin_node = NULL;          /* invalid ptr  */
memb_ptr->end_node = NULL;            /* invalid ptr  */
memb_ptr->frame_in = NULL;            /* invalid ptr  */
memb_ptr->begin_camr = NULL;
memb_ptr->end_camr = NULL;
memb_ptr->leftareas = NULL;           /*                    */
memb_ptr->rightareas = NULL;
if( (memb_ptr->dia_ptr = (struct diagram*)
        malloc(sizeof (struct diagram))) == NULL)
                printf("memory problem\n");
memb_ptr->dia_ptr->x = 0;
for (c = 0; c < 3; c++)
                memb_ptr->dia_ptr->right_mag[c]=
                memb_ptr->dia_ptr->left_mag[c]=
                memb_ptr->dia_ptr->right_v[c]=
                memb_ptr->dia_ptr->left_v[c]=
                memb_ptr->dia_ptr->moment[c]= 0;
while((last = (struct diagram*) malloc(sizeof (struct diagram)))
                == NULL)
                printf("memory\n");
memb_ptr->dia_ptr->next = memb_ptr->last_kink = last;
/*x = length is missing here, because*/
/*length will be determined  in connect_member()*/
```

```
for (c = 0; c < 3; c++)
                memb_ptr->last_kink->right_mag[c]=
                memb_ptr->last_kink->left_mag[c]=
                memb_ptr->last_kink->right_v[c]=
                memb_ptr->last_kink->left_v[c]=
                memb_ptr->last_kink->moment[c]= 0;
}

/**********************enter_col.c******************************/

/*creates a column associated to a node in the floor node_list,*/
/*with its coordinates set to x,y*/

struct column *enter_col(x, y)

double x, y;

{
extern struct column *col_head;
struct column *new;
h_point point;
struct node *node_ptr, *add_node();
static int count = 0;
char *malloc();


while((new = (struct column*) malloc(sizeof(struct column))) == NULL)
        printf("memory problem\n");
#ifdef BOB
        printf("Returning from malloc\n");
#endif
set_point(&point, x, y);
node_ptr =add_node(&point);
node_ptr->flags.col =COLUMN;
new->col_number = count++;
new->node_ptr =node_ptr;

new->next = col_head;
col_head = new;
return (new);
}


/**********************add_node.c******************************/
/* This function attaches a new initialized node to the floor. */
/* It is called both from read_seg_dxf and connect_members. */

struct node *add_node(point)

h_point *point;
```

```
{
static int count = 0;
struct node *node1;
extern struct node *node_head;
char *malloc();
h_point *new_point;

while((node1 = (struct node *)
                malloc(sizeof(struct node))) == NULL)
                 printf("memory\n");    /*alloc memory*/
while((new_point = (h_point *)
                malloc(sizeof(h_point))) == NULL)
                 printf("memory\n");    /*alloc memory*/
set_point(new_point, point->x, point->y);
node_init(node1);
node1->node_number = count++;
node1->point = new_point;
node1->next_node = node_head;
node_head = node1;
return(node1);
}



node_init(node_ptr)
struct node *node_ptr;
{


node_ptr->flags.col = 0;        /* assumed to be on a girder */
node_ptr->flags.queued = 0;
node_ptr->flags.cant = 0;
node_ptr->frame_in = NULL;      /* no frame in list     */
node_ptr->neighbors = NULL;     /* no neighbors */
}                                       /* no nothing*/


/*****************************find_point***********************/
/*This function gets a point(er to a), and a pointer to a node_list*/
/*where a node with the coordinates of the point might lie*/
/*if it finds a node with same coordinates as the point, it returns a*/
/*pointer to the node. Otherwise, it returns the null pointer*/


struct node *find_point(point,nl_ptr)
h_point *point;
struct node_list *nl_ptr;
{
struct node *node_ptr;
```

```
double x, y, xl, yl, fabs();
extern double SMALL;

if (nl_ptr ==NULL)
   node_ptr=NULL;
else {
      x= point->x;
      y= point->y;
      xl= nl_ptr->in_node->point->x;
      yl=  nl_ptr->in_node->point->y;

      if ( fabs(x-xl)+ fabs(y-yl) < SMALL)
         node_ptr=nl_ptr->in_node;

     else
         node_ptr=find_point(point,nl_ptr->next);
}
return (node_ptr);
}

/************************insert_nl.c**************************/
/*This function takes as arguments a bunch of pointers: */
/* to a node,  to a node_list and to a segment, needed for*/
/* knowing what the local cordinates are. It returns a node_list*/
/* pointer to a node_list in which the node has been insert_nled*/
/* at the right place. It works recursively.*/



struct node_list *insert_nl (node_ptr,begin_point, nl_ptr)

struct  node *node_ptr;
h_point *begin_point;
struct  node_list *nl_ptr;
{
struct  node_list  *new;
double  x, y, x0, y0, xl, yl, fabs(), locald_node,
        locald_this_node;
int n;
char *malloc();
if (nl_ptr == NULL){
              while((new = (struct node_list *)
                  malloc(sizeof (struct node_list)))== NULL)
                  printf("memory\n");
             new->in_node = node_ptr;
             new->next = NULL;
             return (new);
 }
else {
   n = node_ptr->node_number;
```

```
   x = node_ptr->point->x;
   y = node_ptr->point->y;
   x0 =begin_point->x;
   y0 = begin_point->y;
   x1 = nl_ptr->in_node->point->x;
   y1 = nl_ptr->in_node->point->y;
   locald_node = fabs(x-x0) + fabs(y-y0);
   locald_this_node = fabs(x1-x0) + fabs(y1-y0);
   if (locald_node< locald_this_node){
        while((new = (struct node_list *)
                  malloc(sizeof (struct node_list)))
                        == NULL)
                        printf("memory\n");
             new->in_node = node_ptr;
             new->next = nl_ptr;
             return (new);
   }
   else if (locald_node ==locald_this_node){
             return (nl_ptr);
   }
   else  nl_ptr->next =
        insert_nl(node_ptr,begin_point, nl_ptr->next);
             return (nl_ptr);
}

}
/**************************add_fil*****************************/
add_fil(node_ptr, member_ptr)

struct node *node_ptr;
struct member *member_ptr;

{

struct frame_in_list *fill;
char *malloc();

while((fill = (struct frame_in_list *)
           malloc(sizeof(struct frame_in_list))) == NULL)
                printf("memory\n");

fill->in_member = member_ptr;
fill->next = node_ptr->frame_in;
node_ptr->frame_in = fill;
return;
}

/*
```

```
     */
/********************add_neighbor*********************/
/* This function is named add_neighbor. It adds another node to a*/
/* list of neighbors*/


struct neighborlist *add_neighbor(neighbors_ptr, node_ptr)

struct neighborlist *neighbors_ptr;
struct node *node_ptr;


{
struct neighborlist *new;
char *malloc();
while((new = (struct neighborlist *)
        malloc(sizeof(struct neighborlist))) == NULL)
                printf("memory\n");

new->node_ptr = node_ptr;
new->next = neighbors_ptr;

return(new);
}


/*
```

```
        */
/*********************get_kink****************************

        STRUCT DIAGRAM *GET_KINK()

        This function looks up a member diagram to find a kink
        at some given location; if it cannot find any, it creates
        it, setting its load-magnitude, its shear, and its
        moment to the right values.

*************************************************************/

struct diagram *get_kink(dia_ptr, x)

struct diagram *dia_ptr;
double x;


{
struct diagram *kink, *next, *this_kink, *make_kink();
double   x1, x2, fabs();
extern double SMALL;

kink = NULL;

for (this_kink = dia_ptr; kink == NULL; this_kink = next){
        x1 = this_kink->x;
                              /* First possibility:
                              here we are!           */
        if (fabs(x1-x)< (SMALL/12))
                kink = this_kink;

        else {
                              /* Second possibility:
                              we need to create a new kink */
        next = this_kink->next;
        x2 = next->x;
        if (x <= x2 - (SMALL/12))
                kink = make_kink(this_kink, this_kink->next, x);
                              /* Else:
                              keep searching */
        }
}

return(kink);
}


/*
```

```
        */
/*****************************************************************

        STRUCT DIAGRAM *MAKE_KINK()

*****************************************************************/

struct diagram *make_kink(prev_kink, next_kink, x)

struct diagram *prev_kink, *next_kink;
double x;

{

double dmag2, r, dx;
int c;
struct diagram *new_kink;
char *malloc();

while((new_kink = (struct diagram *)
                malloc(sizeof(struct diagram))) == NULL)
                printf("memory\n");

prev_kink->next = new_kink;
new_kink->next = next_kink;

new_kink->x = x;
dx = next_kink->x - prev_kink->x;
r = x/dx;
for (c = 0; c < 3; c++){
        dmag2 = next_kink->left_mag[c] - prev_kink->right_mag[c];
        new_kink->left_mag[c] = new_kink->right_mag[c]=
                prev_kink->right_mag[c] + r * dmag2;
        new_kink->right_v[c] = new_kink->left_v[c] = 0;
}

return(new_kink);

}

/*
```

```
      */
/***********************add_fload******************************/
/* This function is called from read_data*/
/* It adds a new uninitialized, floor_load to the floor structure*/
/* It is a duty of the calling routine to give correct values to the */
/* elements of the magnitude array and to call add_point,*/
/* build an h_area, and then set the area of thenew f_load*/

struct f_load *add_fload()


{
static int count = 0;
extern struct f_load *fload_head;
struct f_load  *new_fload;
char *malloc();

while((new_fload =  (struct f_load *)
                malloc(sizeof(struct f_load))) == NULL)
                        printf("memory\n");
new_fload->number = ++count;
new_fload->next = fload_head;
fload_head = new_fload;
return(fload_head);
}
/*
```

```
      */
/************************add_deck*******************************/
/* This function is called from read_data*/
/* It adds a new uninitialized, deck_zone to the floor structure*/


struct deck_zone *add_deck()


{
static int count = 0;
extern struct deck_zone *deck_head;
struct deck_zone  *new_deck;
char *malloc();

while((new_deck =  (struct deck_zone *)
                malloc(sizeof(struct deck_zone))) == NULL)
                printf("memory\n");
new_deck->next = deck_head;
deck_head = new_deck;
return(deck_head);
}

/*
```

```
      */
/***************************add_point**********************/
/* This functi‹ ı is called from many places: from read_data, when*/
/*it calls add_fload, from one-way and two-way, and*/
/*from copy_harea*/

h_area  *add_point(point, area)

h_point *point;
h_area   *area;

{
h_area *new;
char *malloc();
h_point *new_point;
while ((new = (h_area *) malloc(sizeof(h_area))) == NULL)
        printf("memory problem\n");
while((new_point = (h_point *) malloc(sizeof(h_point))) == NULL)
        printf("memory problem\n");
set_point(new_point, point->x, point->y);
new->point = new_point;
new->next = area;
return(new);
}

/**/
/****************************append************************/

/* This function appends a list of h_points, or h_area, to another */
/*one, thus forming a longer list of points.*/
/* append(area_head, area_head)produces a circular list*/


append(area_head, more_area)

h_area *area_head, *more_area;

{
h_area *this_one;

for(this_one = area_head; this_one->next != NULL;
        this_one = this_one->next)
                ;/*keep looping*/

this_one->next = more_area;

}


/* */
```

```
/***********************area_length*****************************/
/* This function finds the number of vertices in an area*/

int area_length(area_ptr)

struct area *area_ptr;

{
int l,n;
struct area *this_area;

l = 0;

for (this_area = area_ptr; this_area != NULL; this_area = this_area->next){
        n = this_area->node_ptr->node_number;
        l++;
}
return(l);
}

/***************************hp_insert*************************/
/* This function inserts a new homogeneous point into a*/
/* homogeneous   area */


hp_insert(ha, p2)

h_area  *ha;
h_point *p2;

{
h_area  *new_area;
h_point *new_point;
char *malloc();

while((new_area = (h_area *) malloc(sizeof(h_area)))== NULL)
        printf("memory\n");

while((new_point = (h_point *) malloc(sizeof(h_point))) == NULL)
        printf("memory\n");

set_point(new_point, p2->x, p2->y);

        new_area->point = new_point;

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

        ha->next        = new_area;

        return;
```

```
}


/************************hp_delete************************/

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


hp_delete(ha)

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

#ifdef UNIX
        free((char *) condemned_harea->point);  /* clean up*/
        free((char *) condemned_harea);         /* memory      */
#endif

#ifdef IBM
        free((char *) condemned_harea->point, sizeof(h_point));
        free((char *) condemned_harea, sizeof(h_area));

#endif
}

return;
}
/*
```

```
      */
/***************************add_area**************************/
/* This file is named add_area.c. The function add_area is called by */
/* find_triba(). It adds an area to the arealist of the member.*/


struct arealist *add_area(areas_ptr, area_ptr)

struct arealist *areas_ptr;
struct area *area_ptr;

{
struct arealist *new;
char *malloc();
while((new = (struct arealist*) malloc(sizeof(struct arealist))) == NULL)
        printf("memory problem");

new->uarea.area_ptr = area_ptr;
new->next = areas_ptr;
return (new);
}

/*
```

```
     */
/******************************************************************

        FREE_TRIBAS()

******************************************************************/

free_tribas(member_ptr)

struct member *member_ptr;

{
struct arealist *areas_ptr, *next;
h_area *triba_h;

for (areas_ptr= member_ptr->leftareas; areas_ptr != NULL;
  areas_ptr =next){
        next = areas_ptr->next;
        triba_h = areas_ptr->uarea.triba_ptr->harea_ptr;
        if (triba_h != NULL)
                free_harea(triba_h);
#ifdef UNIX
        free((char *) areas_ptr->uarea.triba_ptr);
        free((char *) areas_ptr);
#endif

#ifdef IBM
        free((char *) areas_ptr->uarea.triba_ptr, sizeof(struct triba));
        free((char *) areas_ptr, sizeof(struct arealist));
#endif
}

for (areas_ptr= member_ptr->rightareas; areas_ptr != NULL;
  areas_ptr = next){
        next = areas_ptr->next;
        triba_h = areas_ptr->uarea.triba_ptr->harea_ptr;
        if (triba_h != NULL)
                free_harea(triba_h);

#ifdef UNIX
        free((char *) areas_ptr->uarea.triba_ptr);
        free((char *) areas_ptr);
#endif

#ifdef IBM
        free((char *) areas_ptr->uarea.triba_ptr, sizeof(struct triba));
        free((char *) areas_ptr, sizeof(struct arealist));
#endif
}
```

```
return;

}

/***************************************************************

        FREE_HAREA()

***************************************************************/

free_harea(harea_ptr)

h_area *harea_ptr;

{
h_area *ha_ptr, *next, *head;
int count = 0;

head = harea_ptr;
for(ha_ptr = head; ha_ptr != head || ++count == 1; ha_ptr = next){
        next = ha_ptr->next;
#ifdef UNIX
        free((char *) ha_ptr->point);
        free((char *) ha_ptr);
#endif

#ifdef IBM
        free((char *) ha_ptr->point, sizeof(h_point));
        free((char *) ha_ptr, sizeof(h_area));
#endif
}

return;

}

/*
```

```
      */
/*************************add_corner.c***********************

 The function add_corner  adds a node to a specified path.
 It does no check as to where the node should be:
 it adds it at the front of the list. Better not call it at random.

******************************************************************/

struct area *add_corner(area_ptr, node_ptr)

struct area *area_ptr;
struct node *node_ptr;

{
struct area *new;
char *malloc();

while((new = (struct area*) malloc(sizeof(struct area))) == NULL)
        printf("memory problem\n");
new->node_ptr = node_ptr;
new->next = area_ptr;

return(new);
}

/**/
/****************************tail*******************************/
/*This function is called from one-way and two-way*/
struct area *tail(area_ptr)

struct area *area_ptr;

{
struct area *this_one;

for (this_one = area_ptr; this_one->next !=NULL;
                        this_one = this_one->next)
        ; /*keep going*/

return(this_one);
}

/*
```

```
      */
/*****************************copy_harea.c****************/
/* Function to duplicate an h_area data structure.       */



h_area *copy_harea(ha_ptr)

h_area  *ha_ptr;

{
int     i = 0;
h_area  *current, *harea_head, *add_point();

harea_head = NULL;


for(current = ha_ptr;current != ha_ptr || i++ <1;
   current = current->next)
        harea_head = add_point(current->point, harea_head);

append(harea_head,harea_head);

#ifdef BOB
print_harea(harea_head);
#endif

return(harea_head);

}
```

File nu.c

```
#include "strall2.h"
#include <stdio.h>
#include <math.h>
nuout(file_ptr)


FILE *file_ptr;

{
extern struct segment *seg_head;
extern struct node *node_head;
extern double SMALL;
double angle, atan(),ins_x, ins_y;
struct segment *seg_ptr;
struct member *m_ptr;
struct  node   *node_ptr;

for (seg_ptr = seg_head; seg_ptr != NULL; seg_ptr = seg_ptr->next)
        if(seg_ptr->type == MEMBER){
                m_ptr = seg_ptr->actually_m;
                fprintf(file_ptr,"0\nTEXT\n 8\nKEY\n");
                ins_x = .5 * (seg_ptr->end_x + seg_ptr->begin_x);
                ins_y = .5 * (seg_ptr->end_y + seg_ptr->begin_y);
                fprintf(file_ptr, "10\n %f\n 20\n %f\n", ins_x, ins_y);
                fprintf(file_ptr, "40\n %f\n", (.3 * SMALL));
                fprintf(file_ptr, "1\n %d\n", m_ptr->member_number);
                fprintf(file_ptr, "50\n 0\n");

                if(seg_ptr->end_x == seg_ptr->begin_x)
                        angle = 90;
                else
                  angle = 180 *
                       atan((seg_ptr->end_y - seg_ptr->begin_y)/
                               (seg_ptr->end_x - seg_ptr->begin_x))/
                        3.1416;

                if(m_ptr->flags.begin_cant == 1){
                        fprintf(file_ptr,"0\nTEXT\n 8\nSTUFF\n");
                        fprintf(file_ptr, "10\n %f\n 20\n %f\n",
                                seg_ptr->begin_x, seg_ptr->begin_y);
                        fprintf(file_ptr, "40\n %f\n",(.3 * SMALL));
                        fprintf(file_ptr, "1\n M\n");
                        fprintf(file_ptr, "50\n %f\n", angle);
                }

                if(m_ptr->flags.end_cant == 1){
                        fprintf(file_ptr,"0\nTEXT\n 8\nSTUFF\n");
                        fprintf(file_ptr, "10\n %f\n 20\n %f\n",
                                seg_ptr->end_x, seg_ptr->end_y);
```

```
                        fprintf(file_ptr, "40\n %f\n", (.3 * SMALL));
                        fprintf(file_ptr, "1\n M\n");
                        fprintf(file_ptr, "50\n %f\n", angle);
                }
}
for (node_ptr = node_head; node_ptr != NULL; node_ptr =node_ptr->next_node){
                        fprintf(file_ptr,"0\nTEXT\n 8\nNODES\n");
                        fprintf(file_ptr, "10\n %f\n 20\n %f\n",
                                node_ptr->point->x, node_ptr->point->y);
                        fprintf(file_ptr, "40\n %f\n", (.3 * SMALL));
                fprintf(file_ptr, "1\n %d\n", node_ptr->node_number);
                        fprintf(file_ptr, "50\n 0\n");
}

return;
}

sectout(file_ptr, m_ptr)


FILE *file_ptr;
struct member *m_ptr;
{

extern double SMALL;
double angle, atan(), cos(), sin(), ins_x, ins_y, x0, y0, x1, y1;


x0 = m_ptr->begin_node->point->x;
y0 = m_ptr->begin_node->point->y;
x1 = m_ptr->end_node->point->x;
y1 = m_ptr->end_node->point->y;
if(x1 == x0)
        angle = 3.14116/2;
else
        angle = atan((y1 - y0)/
                        (x1 - x0));
ins_x = .5 * (x1 + x0)
        - SMALL * (.7 *  cos(angle) + .25 * sin(angle));
ins_y = .5 * (y1 + y0)
        + SMALL * (.25 * cos(angle) -.7 * sin(angle));

fprintf(file_ptr,"0\nTEXT\n 8\nSECTIONS\n");
fprintf(file_ptr, "10\n %f\n 20\n %f\n", ins_x, ins_y);
fprintf(file_ptr, "40\n %f\n", (.3 * SMALL));
fprintf(file_ptr, "1\nW%.01fX%.01f\n",
        m_ptr->section->wf_ptr->d,
        m_ptr->section->wf_ptr->weight);
fprintf(file_ptr, "50\n %f\n", 180 * angle / 3.14116);
```

```
return;

}
```

File pr.c

```
#include "strall2.h"
#include <stdio.h>

/*
```

```
      */
/**********************list_members****************************/

/* This function prints out a list of the members in a structure*/



list_member_nodes(file_ptr)

FILE    *file_ptr;
{
    struct member *present_member;
    extern struct member *member_head;



        fprintf(file_ptr,"member  begin   end     node_list\n");
        fprintf(file_ptr,"  #     node #  node #  \n");


    present_member = member_head;
    while(present_member != NULL) {
        fprintf(file_ptr,"%4d", present_member->member_number);
        fprintf(file_ptr,"  %6d",
                present_member->begin_node->node_number);
        fprintf(file_ptr,"  %6d",
                present_member->end_node->node_number);
        list_nl(file_ptr, present_member);
        present_member = present_member->next;
    }
    fprintf(file_ptr,"\f");
    return;
}


list_nl(file_ptr, member_ptr)
FILE    *file_ptr;
struct member *member_ptr;

{


struct node_list        *nl;

        nl = member_ptr->frame_in;
        if(nl == NULL) {
                fprintf(file_ptr,"  NULL");
        }
        while(nl != NULL) {
                fprintf(file_ptr,"  %4d ",nl->in_node->node_number);
```

```
                nl = nl->next;
        }
        fprintf(file_ptr,"\n");
}

/*newpage*/
/****************************list_nodes***********************/
/* This function prints out a list of the nodes in a structure*/


list_nodes(file_ptr)

    FILE        *file_ptr;

{

    struct node *present_node;
    extern struct node *node_head;


    fprintf(file_ptr," List of nodes in structure\n");
    fprintf(file_ptr,"                         on       \n");
    fprintf(file_ptr,"    x      y  cant member  fil\n");

    present_node = node_head;
    while(present_node != NULL) {
        fprintf(file_ptr,"node %d\t", present_node->node_number);
        fprintf(file_ptr,"%6.2f %6.2f ", present_node->point->x,
                present_node->point->y);
        fprintf(file_ptr," %2u", present_node->flags.cant);
        if (present_node->flags.col == COLUMN)
             fprintf(file_ptr,"   xx");
        else
             fprintf(file_ptr,"%5d",present_node->on_member->member_number);
        list_fil(file_ptr, present_node);
        present_node = present_node->next_node;
    }

    fprintf(file_ptr,"\f");
    return;
}



list_fil(file_ptr, node_ptr)

    FILE        *file_ptr;
struct node *node_ptr;

{
```

```
struct frame_in_list    *fil;
int                     m;


        fil = node_ptr->frame_in;
        while(fil != NULL) {
                m = fil->in_member->member_number;
                fprintf(file_ptr," %4d ",m);
                fil = fil->next;
        }
        fprintf(file_ptr,"\n");

}

/*
```

```
    */
/*********************list_loads********************************/
/* This function lists the floor loads*/




list_loads()    {


   extern struct f_load *fload_head;
   struct f_load  *this_load;
   int c;

printf("List of the loads input:\n\n");

for (this_load=fload_head; this_load!=NULL;
  this_load=this_load->next) {
     printf("load %d magnitudes", this_load->number);
     for(c = 0; c < 3; c++)
        printf(" %4.2f", this_load->magnitude[c]);
     printf("\n");
     print_harea(this_load->area);
     }

printf("\f");

 return;
 }


/*********************list_decks********************************/
/* This function lists the floor decks*/




list_decks()    {


   extern struct deck_zone *deck_head;
   struct deck_zone  *this_deck;
   int c = 0;

printf("List of the decks input:\n\n");

for (this_deck=deck_head; this_deck!=NULL;
  this_deck=this_deck->next) {
     printf("deck %d direction", ++c);
     printf(" %4.2f", this_deck->direction->x);
```

```
     printf(" %4.2f", this_deck->direction->y);
     printf("\n");
     print_harea(this_deck->harea_ptr);
     }

printf("\f");

 return;
 }


/************************print_ha.c******************************/
/* This function prints an h_area representation.              */



print_harea(a_ptr)

h_area  *a_ptr;

{
int     count = 0;
h_area  *head;

        head = a_ptr;
        printf("a_ptr\tnext\tpoint\tx\ty\tw\n");
        while((a_ptr != head || count++ <1) && a_ptr != NULL) {
                printf("%6.2f %6.2f\n",
                  a_ptr->point->x,
                  a_ptr->point->y);

                a_ptr = a_ptr->next;
        }
        return;
}


/*
```

```
      */
/*******************************************************************

        TRIBA_OUT()

*******************************************************************/
triba_out(file_ptr, member_ptr)

FILE *file_ptr;
struct member *member_ptr;

{

struct arealist *areas;


for (areas=member_ptr->rightareas; areas!=NULL; areas=areas->next)
        if( areas->uarea.triba_ptr->harea_ptr != NULL)
                ha_out(file_ptr, areas->uarea.triba_ptr->harea_ptr);
for (areas=member_ptr->leftareas; areas!=NULL; areas=areas->next)
        if( areas->uarea.triba_ptr->harea_ptr != NULL)
                ha_out(file_ptr, areas->uarea.triba_ptr->harea_ptr);

return;
}


/*******************************************************************/
/* This function appends, to a DXF file, the entity
 representation of an h_area.*/


ha_out(file_ptr, a_ptr)

FILE *file_ptr;
h_area  *a_ptr;

{
int     count = 0;
h_area  *head;

        head = a_ptr;
        fprintf(file_ptr," 0\nPOLYLINE\n 8\nTRIBA\n 66\n   1\n 70\n 1\n");
        while((a_ptr != head || count++ <1) && a_ptr != NULL) {
                fprintf(file_ptr,"0\nVERTEX\n 8\nTRIBA\n");
                fprintf(file_ptr," 10\n%6.3f\n20\n%6.3f\n",
                  a_ptr->point->x,a_ptr->point->y);

                a_ptr = a_ptr->next;
        }
```

```
        fprintf(file_ptr," 0\nSEQEND\n 8\nTRIBA\n");
        return;
}


/*
```

```
        */
/*****************************************************************
        ALLLDIAS()
*******************************************************************/

alldias(file_ptr, m_ptr)

FILE *file_ptr;
struct member *m_ptr;

{

struct member *member_ptr;
struct diagram *kink;
struct node_list *nl_ptr;
struct frame_in_list *f_l;
double left_m, right_m, left_v, right_v, moment, point1;
int c;


        fprintf(file_ptr,"member %d:", m_ptr->member_number);

        if (m_ptr->flags.ext == 1)
                fprintf(file_ptr, "EXTERIOR");
        fprintf(file_ptr,"\nmembers framing in and associated point loads\n")
        for( nl_ptr= m_ptr->frame_in; nl_ptr != NULL;
          nl_ptr= nl_ptr->next){
                for(f_l = nl_ptr->in_node->frame_in; f_l != NULL;
                 f_l = f_l->next){
                   member_ptr = f_l->in_member;
                   point1 = 0;
                   if (member_ptr->begin_node == nl_ptr->in_node)
                     for (c = 0; c < 3; c++)
                        point1 += member_ptr->dia_ptr->right_v[c];
                   else
                     for (c = 0; c < 3; c++)
                        point1 -= member_ptr->dia_ptr->right_v[c];
                   fprintf(file_ptr, " %6d %6.21f\n",
                        member_ptr->member_number, point1);
                }
        }
        fprintf(file_ptr,"\n");

        fprintf(file_ptr,"\n\n\n x  load_l load_r shear_l shear_r  moment \n"
        for(kink=m_ptr->dia_ptr; kink!=NULL; kink=kink->next){
                left_m = right_m = left_v = right_v = moment = 0;
                for ( c= 0; c < 3; c++){
                        left_m += kink->left_mag[c];
                        right_m += kink->right_mag[c];
                        left_v += kink->left_v[c];
```

```
                                right_v += kink->right_v[c];
                                moment  += kink->moment[c];
                        }
                        fprintf(file_ptr,"%4.2f %6.2f %6.2f %6.2f %6.2f %6.2f\n",
                                kink->x,
                                left_m,
                                right_m,
                                left_v,
                                right_v,
                                moment);
                }

                fprintf(file_ptr,"\n\n\n Selected section: w%.01fx%.01f",
                                m_ptr->section->wf_ptr->d,
                                m_ptr->section->wf_ptr->weight);
                fprintf(file_ptr,"\n\n\n\n\n");


        return;

        }
```

File strall2.h

```
/* 7 April 86        FLOOR DATA STRUCTURES        Gasztowtt &   Bleakley*/

/********** #defines for graph construction and manipulation. ***********/

#define NO      0
#define YES     1
#define COLUMN  0
#define MEMBER  1
#define RIGHT 1
#define LEFT -1
#define MAXPATH 32
#define GRADE_50 2
#define A_36  1
#define ONE  1
#define TWO  2
/************************ Type Definitions *****************************/

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

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


/****************** Data Structure Definitions ***********************/

        struct segment {

                int             seg_number;
                double          begin_x, begin_y, end_x, end_y;
                int             type;
                struct member   *actually_m;
                struct node     *actually_c;
                int             segment_number;
                struct segment  *next;
        };

        struct member {
                int              member_number;
                struct {
                    unsigned solved     :1;     /* solved flag          */
                    unsigned end_cant   :1;     /* cantilever flag      */
```

```
            unsigned begin_cant :1;     /* unused right now */
            unsigned ext        :1;
        } flags;
        struct node     *begin_node;    /* pointers to begin &   */
        struct node     *end_node;      /* end nodes             */
        struct node_list *frame_in;     /* list of nodes on memb*/
        struct arealist *leftareas;   /*list of tributary areas resp*
        struct arealist *rightareas;   /*list of tributary areas resp
        struct diagram  *dia_ptr;
        struct diagram  *last_kink;
        struct diagram  *kink_max;
        struct section  *section;       /*selected section     */
        struct member   *next;
        struct member   *begin_camr;    /* cantilever framing in */
        struct member   *end_camr;      /* at one extremity and the
                                        other */
};


struct node {
        int             node_number;
        h_point          *point;
        struct {                        /* K&R p.137 bit fields */
            unsigned col : 1;     /*      1 - column      */
            unsigned cant :1;           /* a cantilever frames in*/
            unsigned queued : 1; /*used in the adjacency search*/
        } flags;                        /* 1 is cantilever      */
        struct frame_in_list *frame_in; /*members framing into node:
                                   /* this is the essence of a node *
        struct neighborlist *neighbors;  /*which nodes are adjacent*/
        struct member *on_member;
        struct node *next_node;        ./*next node in structure*/
};


struct neighborlist {
        struct node *node_ptr;
        struct neighborlist *next;
};


struct frame_in_list {
        struct member *in_member;
        struct frame_in_list *next;
};

struct node_list {
        struct node *in_node;           /* pointers to nodes on */
        struct node_list *next;         /* the member           */
};
```

```
struct triba {
                h_area *harea_ptr;
                h_line *direction;
};

struct arealist {
        union {
                struct area *area_ptr;
                struct triba *triba_ptr;
        } uarea;
        struct arealist *next;
};

struct area {           /* would more correctly be called path*/
        struct node *node_ptr;
        struct area     *next;
};

struct diagram {
        double  x;                      /*local coordinate*/
        double  left_mag[3], right_mag[3];  /*distinguish load cases*
        double  left_v[3], right_v[3];
        double  moment[3];
        struct  diagram *next;
};

struct wf_table {
        double  d;
        struct wf_entry *entries;
        struct wf_entry *last;
        struct wf_table *more;
};

struct wf_entry {
        double d;
        double weight;
        double S;
        double tw;
        double Aw;
        double I;
        double compact_fact;
        struct wf_entry *next_wf;
};


struct section{
        struct wf_entry *wf_ptr;
        int grade;
        double camber;
};
```

```
/****************************************************************************/

        /*  This header defines the structure for columns                   */

        struct column {
                int     col_number;             /* change to alpha-num? */
                struct node *node_ptr;
                double load[3];
                double orientation;             /* allow arbitrary ???  */
                struct column *next;    /* pointer to next col  */
        };


/***********************floor loads  and deck_zones**********************/
        /*   This header defines the structure for floor loads              */
        struct f_load {
                int     number;
                double  magnitude[3];           /* distinguish load cases*/
                h_area  *area;
                struct f_load   *next;
        };

        struct deck_zone{
                h_line  *direction;
                h_area  *harea_ptr;
                struct deck_zone *next;
        };
```

Appendix B
Design Example

The following layout corresponds to an actual project, designed by Weidlinger Assoc..

The grid was entered very conveniently in the CAD system with a large scale tablet, on which was attached a preliminary plan, drafted at Weidlinger: the lines already drawn served as marks for the positionning of the mouse. The members were more delicate to enter: the gap initially left between the segments representing the members and the segments representing their supports (cf. Chapter 6) was too large in view of the width of the joint that traverses the layout. Indeed, the smaller the amount by which the segments representing the members are trimmed, the more reliably the program connecting the members works: the members were input a second time, with a slimmer trimming.

Composite construction was used throughout the project, except on the roof-level. FLOOR does not deal yet with composite construction: as a result, the framing plan attached next page corresponds to this roof level. It was designed for a uniform total load of 75psf, broken down into:

- a dead-load of 20psf,
- a live-load (snow) of 40psf, and
- a superimposed dead-load of 15psf.

When the layout was communicated to us, the members had not yet been sized. In the default of a comparison with an actual design, one can a few members. The following members, located in the upper-left corner of the drawing can be checked easily:

- member 12 is a beam 31.14ft long and loaded with a uniform load of 1.03K/ft. The maximal moment in it is

  $1.03 * (31.14)^2 / 8 = 124$ ft.K

  A W16x31 section was selected for member 12: according to the AISC

charts for allowable moment in beams, a W16x31 can resist a moment of 130ft.K.

- member 16 is a girder 27.35ft long, which supports members 12 and 4. The only load on member 16 is a concentrated load of 31K in its middle (member 4 is slightly shorter than member 12). The maximal moment in member 16 is:

  ```
  31 * (27.35)/4 = 213 ft.K
  ```

  A W21x44 section, which can resist a moment of 225ft.K, was selected for member 16.

- The maximal moment in member 9, which is an exterior beam, is exactly the half of the maximal moment in member 10, which is the interior beam next to it: the moments are 62ft.K. and 124ft.K.

- member 14 is a 52 ft. long secondary girder, which receives from beams 46, 57,123, 3,42, 4 a load of 44K. It also supports an average line-load of .35K/ft. The reactions applied at the extremities of member 14 are 27.3K and 35.3K. Their sum is indeed equal to the sum of the loads applied. In turn, member 39, the primary girder supporting member 14, does receive a point load of 27.3K from member 14.

A comparison between the other members and these validates by extension the design of the other members.

member 12:
members framing in and associated point loads

x	load_l	load_r	shear_l	shear_r	moment
0.00	0.00	1.03	15.97	15.97	0.00
15.57	1.03	1.03	0.00	0.00	124.32
31.14	1.03	0.00	-15.97	-15.97	0.00

Selected section: w16x31

member 19:
members framing in and associated point loads
12 -15.97

x	load_l	load_r	shear_l	shear_r	moment
0.00	0.00	0.49	14.76	14.76	0.00
13.73	0.49	0.49	8.02	-7.95	156.34
27.52	0.49	0.00	-14.72	-14.72	0.00

Selected section: w18x35

member 10:
members framing in and associated point loads

x	load_l	load_r	shear_l	shear_r	moment
0.00	0.00	1.02	15.95	15.95	0.00
15.57	1.02	1.02	-0.00	-0.00	124.20
31.14	1.02	0.00	-15.95	-15.95	0.00

Selected section: w16x31

Selected section: w16x31

member 11:
members framing in and associated point loads

x	load_l	load_r	shear_l	shear_r	moment
0.00	0.00	1.02	15.89	15.89	0.00
15.57	1.02	1.02	-0.00	-0.00	123.72
31.14	1.02	0.00	-15.89	-15.89	0.00

Selected section: w16x31

member 9:EXTERIOR
members framing in and associated point loads

x	load_l	load_r	shear_l	shear_r	moment
0.00	0.00	0.51	7.98	7.98	0.00
15.57	0.51	0.51	0.00	0.00	62.10
31.14	0.51	0.00	-7.98	-7.98	0.00

Selected section: w10x22

member 8:EXTERIOR
members framing in and associated point loads

x	load_l	load_r	shear_l	shear_r	moment
0.00	0.00	0.37	4.80	4.80	0.00
13.08	0.37	0.37	0.00	0.00	31.40
26.16	0.37	0.00	-4.80	-4.80	0.00

Selected section: w18x35

member 17:
members framing in and associated point loads

```
       5 -13.82
      27  15.96
```

x	load_l	load_r	shear_l	shear_r	moment
0.00	0.00	0.00	15.05	15.05	0.00
13.52	0.00	0.00	15.05	-14.73	203.45
27.33	0.00	0.00	-14.73	-14.73	0.00

Selected section: w21x44

member 16:
members framing in and associated point loads

```
       4 -14.99
      12  15.97
```

x	load_l	load_r	shear_l	shear_r	moment
0.00	0.00	0.00	15.74	15.74	0.00
13.56	0.00	0.00	15.74	-15.46	213.30
27.35	0.00	0.00	-15.46	-15.46	-0.00

Selected section: w21x44

member 2:
members framing in and associated point loads

x	load_l	load_r	shear_l	shear_r	moment
0.00	0.00	1.03	9.19	9.19	0.00
8.96	1.03	1.03	-0.00	-0.00	41.19
17.93	1.03	0.00	-9.19	-9.19	0.00

member 14:
members framing in and associated point loads

46	0.00
57	-6.05
123	8.41
3	12.85
42	2.14
4	14.99

x	load_l	load_r	shear_l	shear_r	moment
0.00	0.00	0.41	27.30	27.30	0.00
13.86	0.41	0.41	21.63	21.63	339.10
21.84	0.41	0.00	18.37	2.94	498.62
28.55	0.00	0.26	2.94	-9.92	518.34
36.42	0.26	0.41	-12.00	-14.14	432.17
40.73	0.41	0.41	-15.89	-30.87	367.39
51.83	0.41	0.00	-35.37	-35.37	0.00

Selected section: w24x84

member 39:
members framing in and associated point loads

14	-27.30
117	-12.63

x	load_l	load_r	shear_l	shear_r	moment
0.00	0.00	0.00	37.73	37.73	0.00
3.79	0.00	0.00	37.73	2.36	142.98
11.99	0.00	0.00	2.36	-10.26	162.36
14.69	0.00	0.41	-10.26	-10.26	134.69
25.50	0.41	0.00	-14.65	-14.65	0.00

Selected section: w16x40

member 38:
members framing in and associated point loads

65	-16.66

Appendix C

AUTOCAD Definition of the User Commands

```
[STARTUP]snap off;+
units 3;;;;+
(setq l (getpoint "Enter lower corner coordinates:"));\+
(setq u (getpoint "Enter upper corner coordinates:"));\+
limits !l !u;+
zoom a;+
(setq small (/ (min (- (car u) (car l))  (- (cadr u) (cadr l))) 250));+
layer new grid colour blue grid new layout colour yellow layout +
new endcant new stuff colour yellow stuff +
new office,public,mechanical,roof,d0,d135,d75,d60,d45,d30,d15,d90 +
color red office,public,mechanical,roof;;
[GRID]layer set grid;;line
[COLUMN](setq a (osnap (getpoint "Enter location:") "inters"));\+
layer set layout;;+
point !a;+
layer set stuff;;+
text c (list (car a) (- (cadr a) (/ small 2))) !small 0 I;
[MEMBER]layer set layout;;+
(setq a (osnap (getpoint "Enter first extremity:") "inters"));\+
(setq b (osnap (getpoint "Enter second extremity:") "inters"));\+
(setq smallx (* small (/ (- (car b) (car a)) (distance a b))));+
(setq smally (* small (/ (- (cadr b) (cadr a)) (distance a b))));+
(setq c (list (+ (car a) smallx) (+ (cadr a) smally)));+
(setq d (list (- (car b) smallx) (- (cadr b) smally)));+
line !c !d;;
[CANT]layer set endcant;;+
(setq a (osnap (getpoint "Enter built-in extremity:") "inters"));\+
(setq b (osnap (getpoint "Enter cantilevered extremity:") "inters"));\+
(setq smallx (* small (/ (- (car b) (car a)) (distance a b))));+
(setq smally (* small (/ (- (cadr b) (cadr a)) (distance a b))));+
(setq c (list (+ (car a) smallx) (+ (cadr a) smally)));+
line !c !b;;
[LOADS]$S=Loads
[DECK]$S=Deck
**Loads
[OFFICE]layer set office;;pline
[PUBLIC]layer set public;;pline
[MECHANICAL]layer set mechanical;;pline
[ROOF]layer set roof;;pline
[LAYOUT]$S=
**Deck
[VERTICAL]osnap inters;layer set d90;;pline
[D75]osnap inters;layer set d75;;pline
[D60]osnap inters;layer set d60;;pline
[D45]osnap inters;layer set d45;;pline
[D30]osnap inters;layer set d30;;pline
[D15]osnap inters;layer set d15;;pline
[HORIZONT]osnap inters;layer set d0;;pline
[LAYOUT]$S=
```

This page intentionally left blank.

References

[Abelson 85] Harold Abelson and Gerald Jay Sussman.
The Structure and Interpretation of Computer Programs.
MIT Press, Cambridge, 1985.

[AISC 84] American Institute for Steel Construction.
Manual of Steel Construction, Eight Edition.
A.I.S.C. Inc., 1984.

[Bleakley 84] Bleakley, Albert.
Computer Aided Design of Structural Flooring Systems.
Master's thesis, Massachusetts Institute of Technology, December, 1984.

[Harrington 83] Harrington, Steve.
Computer Graphics, a Programming Approach.
MacGrawHill, New-York, 1983.

[Karakatsanis 85] Karakatsanis, Andreas.
FLODER, a Floor Designer Expert System.
Master's thesis, Carneggie-Mellon University, December, 1985.

[Kernighan 81] Brian W.Kernighan and Dennis M.Ritchie.
The C programming Language.
Prentice-Hall, 1981.

[Pavlidis 82] Pavlidis, Theo.
Algorithms for Graphics and Image Processing.
Computer Science Press, 1982.

Index

Add_area 161
Add_corner 164
Add_deck 157
Add_fload 156
Add_neighbor 153
Add_point 158
Adj_rep 85
Append 158
Area_length 158

Carry_down 114
Clip 112
Cliploads 107
Copy_harea 165

Deck_zone 73
Distance 143

Filter 98
Free_tribas 162

Get_dir 100
Get_extr_node 79
Get_kink 154

Hline 141

Length 144
List_members 170

Main 64
Make_kink 155
Mark 110

One_way 102

Path 94
Project 137

Read_seg-dxf 69
Right_of 134

Set_kink 125
Set_point 146

Solve 121

Tail 164
Triba 90
Triba_out 175

Unique 97