Durham E-Theses

The distinct element analysis of soil masses

Watson, Colin Richard

How to cite:

Watson, Colin Richard (1990) *The distinct element analysis of soil masses*, Durham theses, Durham University. Available at Durham E-Theses Online: http://etheses.dur.ac.uk/6467/

Use policy

The full-text may be used and/or reproduced, and given to third parties in any format or medium, without prior permission or charge, for personal research or study, educational, or not-for-profit purposes provided that:

- a full bibliographic reference is made to the original source
- a link is made to the metadata record in Durham E-Theses
- the full-text is not changed in any way

The full-text must not be sold in any format or medium without the formal permission of the copyright holders.

Please consult the full Durham E-Theses policy for further details.

Academic Support Office, Durham University, University Office, Old Elvet, Durham DH1 3HP
e-mail: e-theses.admin@dur.ac.uk Tel: +44 0191 334 6107
http://etheses.dur.ac.uk

THE DISTINCT ELEMENT ANALYSIS OF SOIL MASSES

by

Colin Richard Watson

B.Sc., M.Sc. *(Dunelm)*

ABSTRACT

The conventional Distinct Element Analysis of Cundall and Belytschko and their respective co–workers are prone to vibrations which must be damped out artificially if numerical problems are to be avoided. An alternative approach to this method is developed which eliminates such problems by allowing the elements to consolidate without gain in velocity. In the method employed here the contact forces, together with body forces due to gravity give rise to accelerations of the elements which in turn cause them to change position. Normally this change in position will produce an increase in the contact forces. Once these new contact forces have been calculated the elements are then returned to their original positions prior to the next iteration. The contact forces, therefore, increase during the analysis to counter the effects of gravity. Two methods using this new approach are described, for which computer programs have been written.

The first program, SLICES, is designed to analyse slopes divided in to slices with a predetermined failure arc. During the analysis the program generates the stress profile acting on the failure arc and predicts the stability or otherwise of the slope. Program SLICES is compared with a traditional slice method under conditions of total and effective stress with cohesive and frictional soils. An analysis using a non–linear failure criterion is also carried out with program SLICES. The second program, CIRCLES, uses circles as the distinct element type and does not require a predetermined failure arc. It is shown that edge effects cause an incorrect stress regime to be set up that masks the failure process. However a sliding type failure is demonstrated where the edge effects do not mask the analysis.

Submitted in accordance with the regulations for the degree of Ph. D. of the University of Durham. October 1989.

THE

DISTINCT ELEMENT ANALYSIS

OF SOIL MASSES

by

Colin Richard Watson

B.Sc., M.Sc. *(Dunelm)*

The copyright of this thesis rests with the author. No quotation from it should be published without his prior written consent and information derived from it should be acknowledged.

12 AUG 1990

List of Contents

		page
	List of Figures	i
	List of Tables	iv
	Acknowledgements	vi
Chapter 1	**Chapter 1 Introduction**	1
1.1	**Distinct element analysis**	1
1.1.1	*The need for Distinct Element Analysis*	1
1.1.2	*Distinct Element Analysis and discontinuous rock masses*	2
1.1.3	*Relation of Distinct Element Analysis to other analysis*	2
1.2	**A Rigid Block Model for rock masses**	3
1.2.1	*Corner formulations*	3
1.2.2	*Edge formulations*	4
1.2.3	*Hybrid formulations*	4
1.2.3.1	The Distinct Element – Boundary Element hybrid	5
1.2.3.2	The Distinct Element – Finite Element hybrid	6
1.2.3.3	Explicit – Implicit time integration	6
1.3	**A Rigid Ball Model for soil**	7
1.3.1	*Soil particle modelling*	7
1.4	**Initial aims of this work**	8
1.4.1	*Development of the Rigid Block Model at Durham*	8
1.5	**A final approach**	12
1.5.1	*The relevance of Distinct Element Analysis to soil masses*	12
1.5.2	*Soil slices as rigid blocks*	13
1.5.3	*Soil masses as circles of influence*	13
1.5.4	*Organisation of this work*	13

Chapter 2	**The implementation of the Distinct Element Analysis**	15
2.1	**Cundall's Cyclic Process**	15
2.1.1	*Cundall's Concept*	15
2.1.1.1	The Cyclic Process	15
2.1.1.2	A Simple Implementation	19
2.1.2	*The Behaviour of a Single Contact*	20
2.1.3	*Controlling Numerical Instability*	23
2.1.4	*Towers of Contacts*	24
2.1.5	*Some Recommendations*	25
2.2	**An Alternative Approach**	25
2.2.1	*Consolidation*	25
2.2.2	*A Statement of the New Approach*	28
2.2.3	*Machine Accuracy*	30
2.2.3.1	The Relevance to Discretization	30
2.2.3.2	Bringing Consolidation to a Close	31
2.2.4	*Propagating Effects Through the Matrix*	32
2.2.4.1	A Simple Tower Problem	32
2.2.4.2	Some Recommendations	35
2.2.5	*The Role of Damping*	35
2.2.6	*Concluding Remarks*	38
Chapter 3	**Chapter 3 Distinct element method of slices**	39
3.1	**Introduction**	39
3.2	**Theory extensions for SLICES**	42
3.2.1	*The edge formulation employed*	42
3.2.2	*The Force Displacement and Motion Laws*	47
3.3	**Using Program SLICES**	50
3.3.1	*Introduction*	50
3.3.1.1	Overview	50

3.3.1.2	Input and output unit summary	51
3.3.1.3	Outline of facilities	53
3.3.2	*Input Command Language*	55
3.3.2.1	Introduction	55
3.3.2.2	Control commands	56
3.3.2.3	The debug command set	58
3.3.2.4	The set command set	59
3.3.2.5	The calculator command set	61
3.3.2.6	The plot command set	62
3.3.2.7	The map command set	63
3.3.2.8	The mesh command set	64
3.3.2.9	Syntax table	65
3.3.3	*Input command file*	66
3.3.3.1	File format	66
3.3.3.2	Defining tasks	68
3.3.3.3	Input error handling	71
3.3.4	*Utility files*	74
3.3.4.1	Repeat file	74
3.3.4.2	Command list file	74
3.3.4.3	Restart file	74
3.3.4.4	Trace output file	76
3.3.4.5	Debug output	76
3.3.4.6	Oscillation output	79
3.3.4.7	The running commentary	80
3.4	**Structure of Program SLICES**	83
3.4.1	*Memory structure*	83
3.4.2	*Program structure*	86
3.4.2.1	Procedural elements	86
3.4.2.2	Main relationships	86
3.4.2.3	Recursion structures	87

3.4.2.4	Structure that maps structured variables	91
3.5	**Validation**	93
3.5.1	*Introduction*	93
3.5.2	*Validation Methods*	93
3.5.3	*Discussion of results*	95
3.5.3.1	Results involving total stress conditions	96
3.5.3.2	Results involving effective stress conditions	97
3.5.3.3	Conclusions	98
3.5.4	*Interpretation of SLICE output*	99
Chapter 4	**Distinct element method of circles**	101
4.1	**The Concept**	101
4.1.1	*Circles as Areas of Influence*	101
4.1.2	*Contacts in detail*	102
4.1.3	*The Distinct Element Analysis formulation for CIRCLES*	104
4.1.3.1	Consolidation formulation	106
4.1.3.2	The traditional Distinct Element Analysis formulation	108
4.2	**Implementation**	110
4.2.1	*The Program Memory Structure*	110
4.2.2	*Program structure*	114
4.2.2.1	Program structure that maps the memory	114
4.2.2.2	The updating of contacts	116
4.2.3	*Input command language*	117
4.2.3.1	Introduction	117
4.2.3.2	Control commands	118
4.2.3.3	The debug command set	121
4.2.3.4	The set command set	122
4.2.3.5	The calculator command set	124
4.2.3.6	The plot command set	125
4.2.3.7	The map command set	126

4.2.3.8 The mesh Command Shell 127
4.2.4 *The utility files* 128
4.3 **Validation** 132
4.3.1 *Introduction* 132
4.3.2 *The contact behaviour* 133
4.3.3 *The Mesh Edge effects* 134

Chapter 5 **Conclusions** 141

References 145

Appendices

Appendix A Mathematical Notation A.1
Appendix B Structure charts for SLICES B.1
Appendix C Results for program SLICES C.1
Appendix D Program SLICES D.1
Appendix E Program CIRCLES E.1

List of Figures

		Page
Figure 1.1	Analysis of a tower. After Rouse (1982)	10
Figure 2.1	The Distinct Element Analysis Calculation Cycle	16
Figure 2.2	The forces associated with a contact	17
Figure 2.3	The behaviour of a single contact	21
Figure 2.4	The use of fixed blocks to promote consolidation	27
Figure 2.5	The new calculation order	29
Figure 2.6	The tower of circles analysed	33
Figure 3.1	A typical slope for analysis by SLICES	40
Figure 3.2	Stress profile produced by SLICES	43
Figure 3.3	The SLICES calculation cycle	44
Figure 3.4	A Comparision of the data required to define a contact	46
Figure 3.5	Bachmann diagram of SLICES memory items	85
Figure 3.6	Stress profiles for result set 1	B.2
Figure 3.7	Stress profiles for result set 2	B.9
Figure 3.8	Stress profiles for result set 3	B.11
Figure 3.9	Stress profiles for result set 4	B.13
Figure 3.10	Stress profiles for result set 5	B.20
Figure 3.11	Stress profiles for result set 6	B.22
Figure 3.12	Stress profiles for result set 7	B.24
Figure 3.13	Stress profiles for result set 8	B.30
Figure 3.14	Stress profiles for result set 9	B.32
Figure 3.15	Stress profiles for result set 10	B.34
Figure 3.16	Stress profiles for result set 11	B.40
Figure 3.17	Stress profiles for result set 12	B.42
Figure 3.18	Stress profiles for result set 13	B.43

Figure 3.19 Stress profiles for result set 14 B.48
Figure 3.20 Stress profiles for result set 15 B.49

Figure 4.1 Contact definition in program CIRCLES 103
Figure 4.2 The Mohr construction 105
Figure 4.3 An high level view of the memory structure 111
Figure 4.4 A Bachmann diagram of the program memory elements 113
Figure 4.5 Analysis of embankment without contact correction 135
Figure 4.6 Analysis of embankment using a contact correction 138
Figure 4.7 Analysis showing partial wedge failure 139

List of Structure charts in Appendix C

Figure C.1 Chart for procedure *error_simple* C.3
Figure C.2 Chart for procedure *word_scan* C.4
Figure C.3 Chart for procedure *skipblks* C.5
Figure C.4 Chart for procedure *skipcomment* C.6
Figure C.5 Chart for procedure *trapper* C.7
Figure C.6 Chart for procedure *get_command* C.8
Figure C.7 Chart for function *onoff* C.9
Figure C.8 Chart for procedure *headers* C.10
Figure C.9 Chart for procedure *factors_of_safety* C.11
Figure C.10 Chart for function *sign* C.12
Figure C.11 Chart for procedure *initialise* C.12
Figure C.12 Chart for procedure *plots* C.13
Figure C.13 Chart for procedure *map_space* C.15
Figure C.14 Chart for procedure *setup_plot* C.14
Figure C.15 Chart for procedure *disp_plot* C.16
Figure C.16 Chart for procedure *fram_plot* C.16
Figure C.17 Chart for procedure *slice_plot* C.17
Figure C.18 Chart for function *utohead* C.17
Figure C.19 Chart for procedure *force_profile* C.18

Figure C.20	Chart for procedure *init_fm*	C.19
Figure C.21	Chart for function *ptrd_fm*	C.19
Figure C.22	Chart for procedure *lims_fm*	C.20
Figure C.23	Chart for procedure *cycle*	C.21
Figure C.24	Chart for procedure *fordsl*	C.23
Figure C.25	Chart for procedure *fconsolsl*	C.22
Figure C.26	Chart for procedure *start_shut*	C.24
Figure C.27	Chart for procedure *update_area*	C.26
Figure C.28	Chart for procedure *update_message*	C.25
Figure C.29	Chart for procedure *cold_contact*	C.27
Figure C.30	Chart for procedure *get_apex*	C.27
Figure C.31	Chart for procedure *mesh*	C.28
Figure C.32	Chart for procedure *cre_platen*	C.29
Figure C.33	Chart for procedure *cre_slices*	C.30
Figure C.34	Chart for procedure *read_restart_file*	C.31
Figure C.35	Chart for procedure *write_restart_file*	C.32
Figure C.36	Chart for procedure *write_r_el*	C.33
Figure C.37	Chart for procedure *complete*	C.34
Figure C.38	Chart for procedure *debug_slice*	C.35
Figure C.39	Chart for procedure *write_con*	C.36
Figure C.40	Chart for procedure *wr_con*	C.36
Figure C.41	Chart for procedure *write_sli*	C.34
Figure C.42	Chart for procedure *parameters*	C.37
Figure C.43	Chart for procedure *calculator*	C.38
Figure C.44	Chart for function *intcalc*	C.39
Figure C.45	Chart for procedure *repeater*	C.40
Figure C.46	Chart for procedure *control*	C.41
Figure C.47	Chart for program *SLICES*	C.42

List of Tables

		Page
Table 2.1	Contact forces for a tower	32
Table 2.2	The expansions for 3 cicles	34
Table 3.1	A Typical Command File	41
Table 3.2	Input Command Language Parsing Symbols	66
Table 3.3	Input Command Language Parsing Definition	67
Table 3.4	An Example of Error Correction	72
Table 3.5	An example of interactive input	73
Table 3.6	The restart file line tags	75
Table 3.7	The debug format table	78
Table 3.8	The running commentary screen lines	81
Table 3.9	Running commentary messages	82
Table 3.10	Trace of Program Behaviour During Simple Use	88
Table 3.11	Program behaviour during Repeat processing	89
Table 3.12	Program behaviour during error processing	91
Table 3.13	Table of Results for Program SLICES	95
Table 3.14	Input commands for result set 1	B.1
Table 3.15	Input commands for result set 2	B.8
Table 3.16	Input commands for result set 3	B.10
Table 3.17	Input commands for result set 4	B.12
Table 3.18	Input commands for result set 5	B.19
Table 3.19	Input commands for result set 6	B.21
Table 3.20	Input commands for result set 7	B.23
Table 3.21	Input commands for result set 8	B.29
Table 3.22	Input commands for result set 9	B.31
Table 3.23	Input commands for result set 10 and 13	B.33
Table 3.24	Input commands for result set 11 and 14	B.39

Table 3.25	Input commands for result set 12 and 15	B.41
Table 4.1	Input Command Language Parsing Symbols	118
Table 4.2	Input Command Language Parsing Definition	119
Table 4.3	The restart file line tags	129
Table 4.4	Debug output formats	130
Table 4.5	The running commentary screen lines	131
Table 4.6	The running commentary messages	132
Table 4.7	Output from CIRCLES after 2000 cycles	133

ACKNOWLEDGEMENTS

I have cause to thank several people for their help during the course of this work. In particular I am greatly indebted to the Late Dr. R. K. Taylor and to Dr. J. M. Wilson for their supervision and encouragement. Dr. J. M. Wilson has had the difficult task of taking over the supervision of my thesis at a late stage and this I greatly appreciate. I also owe a great deal to my wife, family and friends for their invaluable support throughout.

THE

DISTINCT ELEMENT ANALYSIS

OF SOIL MASSES

by

Colin Richard Watson

B.Sc., M.Sc. *(Dunelm)*

CHAPTER 1
INTRODUCTION

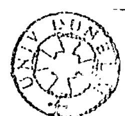
1.1 Distinct element analysis

1.1.1 The need for Distinct Element Analysis

It may appear strange that a thesis principally concerned with the analysis of the stability of soil masses should begin by discussing rock masses. Indeed discontinuous rock masses shall frequently be referred to throughout the theoretical sections of this discussion. The reason for this hybridisation is simply that the analysis techniques developed here are abstract formulations of those used for some years for the analysis of discontinuous rock masses, namely the Distinct (or Discrete) Element Analysis (DEA).

Distinct Element Analysis is a numerical model which utilises the time explicit integration of the second order difference equations for reduced degrees of freedom of distinct geometric elements, for example rectangular blocks, within the problem. Normally the reduction in the degrees of freedom is due to ignoring the internal deformation of the elements, the elements being connected by their boundaries across which deformation of the mass is considered to take place.

The major advantages of Distinct Element Analysis over finite element analysis are speed of execution, ease of incorporation of non–linear material properties and its explicit relation to time allowing the progressive failure of the system to be studied. These three properties make Distinct Element Analysis a tool worth developing for the analysis of soil masses.



1.1.2 Distinct Element Analysis and discontinuous rock masses

Distinct Element Analysis was originally developed for the analysis of discontinuous rock masses by Cundall (1971), this model was known as the Rigid Block Model (RBM) due to the reduction of the degrees of freedom by elimination of the internal deformation of the elements. In terms of rock masses, the problem elements are bounded by the joints and bedding planes to form blocks. Each block is allowed rotational and translational displacements and move under the influence of gravity and the forces between neighbouring blocks at contacts.

The contact is fundamental to the understanding of the Distinct Element Analysis as it is these which govern the behaviour of the mass as a whole. The Distinct Element Analysis or Rigid Block Model is a dynamic relaxation method for it is at the contacts that inter–element forces are produced by multiplying the small overlaps of the elements (due to previous movements) by the relaxation constant (or stiffness). These new inter–element forces are summed for each element to give rise to new accelerations, velocities and displacements, and hence to new inter–element forces. As it is the contacts which govern the overall behaviour of the model, and the contact conditions are recalculated at the end of each time step, it can readily be seen that this technique lends itself to the modelling of large scale movements.

1.1.3 Relation of Distinct Element Analysis to other analysis methods

There are several possible classes of techniques facing the Engineer, deciding which analysis to choose can often be difficult. There are finite elements, boundary elements, distinct elements, displacement discontinuity methods and various hybrid versions. As Meek and Beer (1984) point out, all these methods should provide reasonable approximations to the elastic stress around an excavation.

For excavations in blocky rock systems the problem of choice is further compounded by the fact that the near field and far field behave very differently. The near field is non–linear, non–elastic in its behaviour while the far field is linear elastic. No individual technique can accurately model both behaviour types at once. The Distinct Element Analysis requires all geometric data to be known throughout the model domain which is a major problem for large excavations. It is ideally suited to model the near field where the data is normally the most readily available. To incorporate both behaviour types Distinct Element Analysis may be coupled to a far field modelled by either Finite Element or Boundary Element methods.

1.2 A Rigid Block Model for rock masses

The Rigid Block Model has undergone much work since its inception in 1971, most published work concentrates upon modelling the behaviour and estimating the support requirements of underground openings in jointed rock masses, almost all the literature concerns itself with promoting the technique in a theoretical fashion and rarely presents the analysis of real cases. There are three types of Rigid Block Model, which shall be referred to here as Corner, Edge and Hybrid formulations.

1.2.1 Corner formulations

The corner formulation is the original two dimensional formulation as developed by Cundall. It is so named because the contacts are defined when a corner of one block touches the edge of another. The contact of two blocks along an edge was defined simply as two contacts, unfortunately this gave rise to multiple contact problems, and hence multiple force problems, Rouse (1982). This formulation led to straightforward housekeeping algorithms for contacts allowing them to be made, broken, and remade as necessary. Corner formulations were used in all

cases where large displacements required to be modelled, Cundall (1976), Voegele (1978).

1.2.2 Edge formulations

In response to work with a Cundall corner formulation by Dowding *et al.* (1983), to model the transient behaviour of rock caverns, the edge formulation was developed, Belytschko *et al.* (1983). Here the definition of a contact was always as two edges, one from each block. The physical length of contact allows the calculations to be in terms of stress, a major advantage over corner formulations. The rationale behind this work argued that the initial failure of the mass was the most important feature and that the contacts could not be modelled accurately over large displacements due to the simple failure criteria in use. It was also pointed out that failure may take place at two to three percent strain rendering large strain modelling inappropriate. Due to this the housekeeping of the corner formulation could be dropped to give a much more compact code.

1.2.3 Hybrid formulations

In the same way as Finite Element analysis has difficulty analysing the far field around an excavation due to the number of elements required to model it, so too does Distinct Element Analysis, a further complication for Distinct Element Analysis was how to model the behaviour of excavation support satisfactorily. This led to hybrid formulations of Distinct Element Analysis with Boundary Elements and Finite Elements.

1.2.3.1 The Distinct Element – Boundary Element hybrid

Lorig and Brady published work in 1982, 1983 and 1984 coupling Boundary Elements with Distinct Elements. Lorig, Brady and Cundall (1986) discuss this method utilising a sophisticated form of Rigid Block Model.

Much of the work enhancing the Distinct Element Analysis was concerned with increasing efficiency and combating problems discovered during use. Corner formulations are subject to interlocking at corners as corner to corner contacts may often lead to abnormally high forces. These forces can then be propagated throughout the Distinct Element mesh. To overcome this problem an edge formulation was adopted in addition to the corner formulation allowing edge to edge and corner to edge contacts. It is assumed here that corner to corner contacts were not entertained but it is not explicitly stated. A new damping regime was also introduced, that of adaptive density scaling, whereby the element densities are modified to allow the application of the mass proportional part of the damping system across a greater spread of element masses. The contact housekeeping routines have been modified to transfer the contact between a sliding block and each successive neighbour across which it slides, thus preventing the sudden collapse of the forces on a block as previously described by Watson (1983).

The boundary element method determines the behaviour of the mass from the boundary conditions imposed on surfaces within it. This allows for the modelling of semi–infinite regimes as the far field boundaries need not be known. The Boundary Element Analysis is an elastic analysis and it has been found from field measurements that the far field domain (two to three excavation radii from the excavation) does indeed act elastically with the discontinuities playing little or no part. Boundary Element Analysis is therefore used to model the far field and the Distinct Element Analysis to model the near field. To couple the two methods care

has been taken by Lorig *et al.* to preserve kinematic continuity at the interface, by equating block corner and nodal displacements in the Boundary Element domain.

1.2.3.2 The Distinct Element – Finite Element hybrid

The work of Dowding *et al.* (1983a,b) and Belytschko *et al.* (1983) coupled Distinct Elements with Finite elements. The aim of their work was to model the transient behaviour of caverns under the influence of seismic activity. The propagation of waveforms through large stacks of Distinct Element was problematic as the mechanism was not understood and was also expensively time consuming. The excitation therefore, was propagated from the far field Finite Elements to the near field Distinct Elements. The Finite Element and Distinct Element domains were coupled by silent boundaries, (that is producing no reflection), after the Lysmer and Kuhlemeyer method (1969), while the cavern linings were modelled by beam elements.

1.2.3.3 Explicit – Implicit time integration

Explicit time integration schemes as utilised in the Distinct Element Analysis as described so far have a low over–head per time step compared with Finite Element or Boundary Element analyses. However for long duration analysis requiring small time steps, the cost of simulation may still be prohibitive. As will be seen later the time step size is critical for numerical stability and the Distinct Element Analysis has been found to be non convergent for many parameter combinations, Lorig *et al.* (1986).

Plesha (1986) has proposed that simulations utilise a constitutive implicit — explicit time splitting operator, whereby the linear portion of the analysis is modelled using an implicit time integrator and the non–linear portions (for example post contact failure) use the standard explicit, time marching integration of the

normal Distinct Element Analysis. He found that under certain long duration simulation conditions, considerable computer cost savings could be made with little difference in the general behaviour of the rock mass. He also suggested that transient behaviour could be modelled by changing to and from the usual methods when necessary throughout the simulation.

1.3 A Rigid Ball Model for soil

The Distinct Element Analysis is not restricted to the Rigid Block Model or derivatives and as a general concept has applications elsewhere. From the beginning Cundall developed a program where the calculation elements were simple discs or circles. The same degrees of freedom were allowed to the discs as to the blocks enabling them to be used to model the collapse of a set of cylinders for instance, as in Cundall (1971). It appears that this BALL program was the development route to the Rigid Block Model as the elements are significantly less complex due to them having no corners.

1.3.1 Soil particle modelling

By equating disk elements with soil particles Cundall used Program BALL as an easily controlled test apparatus to investigate the properties of soil particles under various loading conditions. From the data collected from this computer model he hoped to develop continuum constitutive laws governing soil particle behaviour, Cundall and Strack (1979). Although using a model for the basis of this research he argued that the superior control of loading conditions in the program over that of an experimental situation gave the program a valid role.

Apart from investigating the general non–linear properties of assemblies of soil particles, Cundall and Strack turned their attention to modelling the process of soil consolidation. Here a circular assembly of discs was subjected to two orthogonal

forces, firstly in an isotropic state and then under a deviatoric load. The velocities and displacements of the particles were plotted so that changes in the fabric could easily be seen.

All Distinct Element Analysis work as far as can be found dealing with soils concerns the modelling of assemblies of individual particles and not with soil masses. This thesis aims to do so.

1.4 Initial aims of this work

As pointed out in section 1.2 most published work uses idealised theoretical problems to promote the latest development in the analysis technique. Very little uses the Distinct Element Analysis / Rigid Block Model to solve a real design problem. This is perhaps, as Meek and Beer (1984) point out, because the technique has been extensively used in the commercial environment.

The initial aims of this work were to continue the development of Distinct Element Analysis theory and an edge formulation derived from Dames and Moore (1978) with a view to validation against simple physical models and then real situations. To understand the starting point of this work it is necessary to appreciate the development of the Rigid Block Model at Durham.

1.4.1 Development of the Rigid Block Model at Durham

The beginning of the work at Durham was with the Rigid Block Model implementation of Dames and Moore report (1978). This work, by Rouse (1982), utilised a corner formulation as described above which was initially unusable due to it being in single precision. The program was therefore modified to double precision, whereupon several unexpected effects were found.

The first of these effects was noted when a simple tower of blocks was modelled. Figure 1.1 shows the tower before analysis and after 7000 calculation cycles. A tower is expected to topple rigidly to begin with and then break one third up the height with the lower third rotating more rapidly than the upper part. In this simulation Rouse found that the tower broke in more than one place and therefore fell incorrectly.

It was proposed that the original damping regime employed was at fault. This regime consists of two separately controllable viscous damping factors. Firstly a stiffness–proportional damping to control contact vibration and analogous to dashpots at the contacts between blocks both in the shear and normal directions. Then secondly, a mass–proportional damping representing dashpots from the block centroids to the coordinate origin. The damping regimes are provided to remove the kinectic energy of the system generated during collapse, if this did not take place the elements would continue moving perpetually. The cause of the incorrect tower collapse was attributed to the stiffness–proportional damping giving rise to a standing wave lying the length of the tower, in turn causing localised high stress where the tower broke.

Rouse also reported the 'locking up' of certain configurations of blocks before equilibrium could be reached. This was caused by corner to corner contacts as well as corner to edge contacts being used in the formulation.

At this stage it was realised that this version of the Rigid Block Model, at least contained some very serious fundamental inaccuracies and could not be used to reliably model real situations.

An investigation by Watson (1983) showed that the corner to corner contact problem led to some pairs of blocks having up to eight contacts between them and so was remedied first. This was achieved by allowing a pair of blocks to

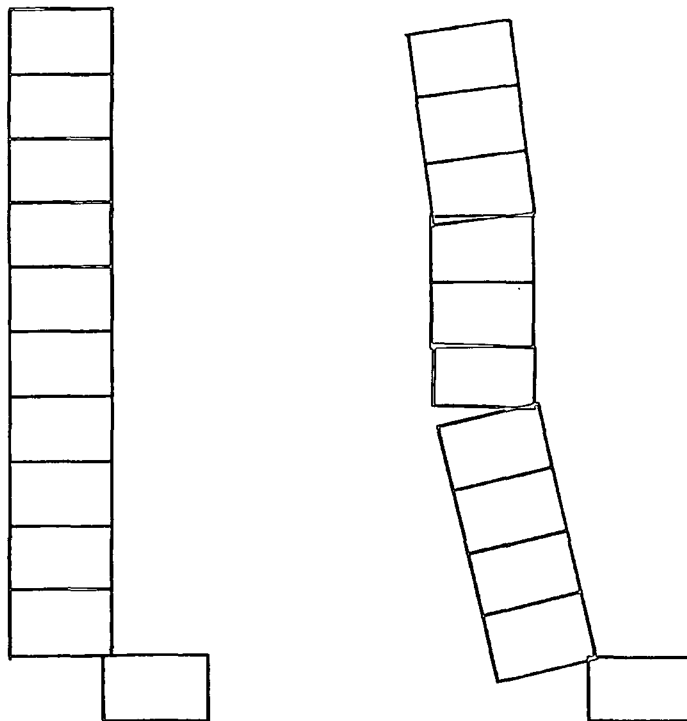


Figure 1.1 Analysis of a tower. After Rouse (1982)

have either, a single corner to edge contact or, two such contacts forming an edge contact or, a corner to corner contact. In the case of the latter contact type, the corner combination promoting the greatest ease of movement was chosen to reduce the risk of 'locking up'. Finally a comparison of Distinct Element Analysis with simple sliding physical models was carried out.

For a complete discussion of the problems encountered, together with the Rigid Block Model program versions used, reference is made to Rouse (1982) and Watson (1983). The next problems on the agenda and hence the beginning position for this work were those caused by the original Distinct Element Analysis damping regime. It was reported by Rouse (1982) that blocks that differed in mass by a factor of two from the mean were effectively undamped, even if the mean masses were heavily damped.

To begin this investigation of the effects of damping on elements, and towers of elements, a Pascal Distinct Element Analysis program was written. This implementation was extremely simple in nature allowing each element one degree of freedom. These point mass elements were all positioned at a common origin, each having one contact with the next created element. Obviously, the last formed had no contacts. This regime represents a set of elements which form a tower of contacts. The program shall be referred to as Program CVS, a mnemonic representing Contact Vibration Simulation.

It was during this stage of the damping investigation that a final approach was conceived.

1.5 A final approach

1.5.1 The relevance of Distinct Element Analysis to soil masses

For some time before Distinct Element Analysis, boundary elements and finite elements were used to analyse blocky rock systems, despite these being discontinuous systems and quite unlike the continuum systems more suited to these methods. Some attempts were made to include the discontinuities by slide lines in finite difference, Wilkins (1969) and by joint finite elements as described by Goodman (1976). To use a method designed for discontinuous materials and include increasing degrees of continuum to solve a continuum problem is simply the reverse of this.

The advantages of using a Distinct Element Analysis based solution for soil mass stability analysis are similar to the main advantages of Distinct Element Analysis for blocky rock systems, namely a low overhead per iteration, a time explicit integrator leading to easy analysis of progressive failure and easy inclusion of non–linear material properties. It is for these three reasons that an attempt has been made to develop Distinct Element Analysis programs suitable for the analysis of soil slopes and their progressive failure.

There are two such Distinct Element Analysis programs designed to model the behaviour of soil masses, developed during this work, namely SLICES and CIRCLES. The names referring to the fundamental calculation element. Program SLICES is most akin to a traditional Rigid Block Model and to a traditional limit equilibrium analysis such as Bishop (1955), Fellenius (1936) or Janbu (1973), while Program CIRCLES, the more general of the two, is quite unlike either.

1.5.2 Soil slices as rigid blocks

Like limit equilibrium analyses, the user is expected to provide a surface slope topography and a proposed failure surface. The failure sector is divided into vertical slices which are interpreted as rigid blocks. The blocks have reduced degrees of freedom, those of body displacements only. The removal of rotation is desirable as the slices often have an high aspect ratio, which would lead to problems tracing the positions of the corners. Furthermore, toppling of the slices high on the failure arc would tend to occur, which is problematic in a analysis designed to model sliding only.

Unlike limit equilibrium analysis inter–slice forces are fully incorporated. SLICES provides graphical and written output allowing the build up of stresses on the failure arc to be monitored. From this it can readily be seen which portions have reached their limit and so the progress of the failure can be traced, and the mechanisms inferred.

1.5.3 Soil masses as circles of influence

Program Circles is not a limit equilibrium analysis and employs a Distinct Element Analysis where the elements are circles of influence. The circles are not particles and have reduced degrees of freedom, rotation being ignored. As areas of influence the circles may overlap to a large extent. CIRCLES is far more sophisticated than SLICES as it is not limited to a predetermined failure arc. If failure occurs then the failure zones are displayed as they form.

1.5.4 Organisation of this work

There is much in common between CIRCLES and SLICES, both in the theory and the implementation of the Distinct Element Analysis techniques employed.

Both have similar degrees of user friendliness, input and output requirements and other features. The areas of common theory can be found in the next chapter.

Information for programs SLICES and CIRCLES is contained in Chapters three and four respectively. These chapters contain information on extensions to the theory specific to the program, its use, structure, memory requirements and validation. Finally Chapter five draws the discussion to a close containing a summary of conclusions.

CHAPTER 2
THE IMPLEMENTATION OF THE DISTINCT ELEMENT ANALYSIS

2.1 Cundall's Cyclic Process

2.1.1 Cundall's Concept

2.1.1.1 The Cyclic Process

The underlying aim of Distinct Element Analysis is to model the displacement of individual elements with time. This is cyclic or iterative in nature and allows the use of simple force displacement laws using an explicit integration scheme. As reported by Cundall (1971) several procedures are followed during each cycle. In the broadest sense these procedures are a force displacement relation to give the forces in the system, followed by a motion law to give the displacements. Furthermore the forces may be modified by force boundary conditions and the displacements by displacement boundary conditions. Figure 2.1 illustrates the process. Each complete cycle around these procedures takes one time step. So, in theory, as the values for all degrees of freedom are known at each time step the displacement state at any time can be found by cycling round an appropriate number of times.

To illustrate this cyclic process the laws used by Cundall (1971) are followed. The force displacement law is determined for each contact for each block for each cycle and the motion law for each block in each cycle. Figure 2.2 shows the forces associated with a contact. The force displacement relator is the contact stiffness,

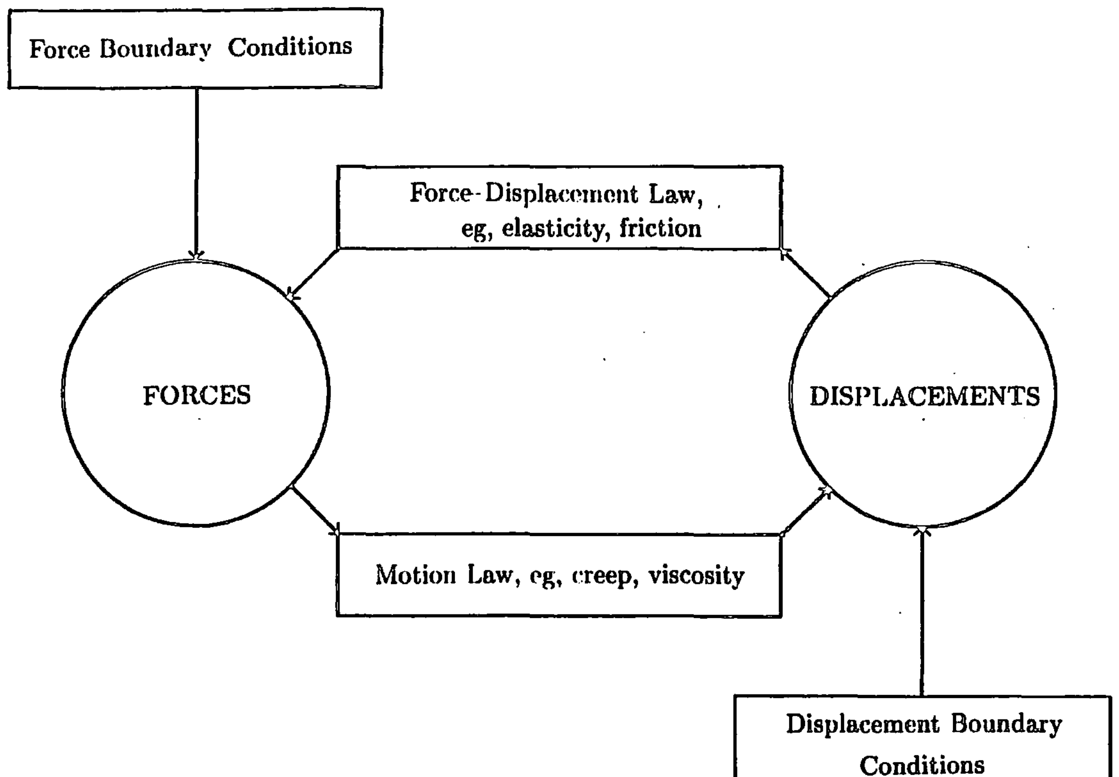


Figure 2.1 The Distinct Element Analysis Calculation Cycle
After Cundall (1971)

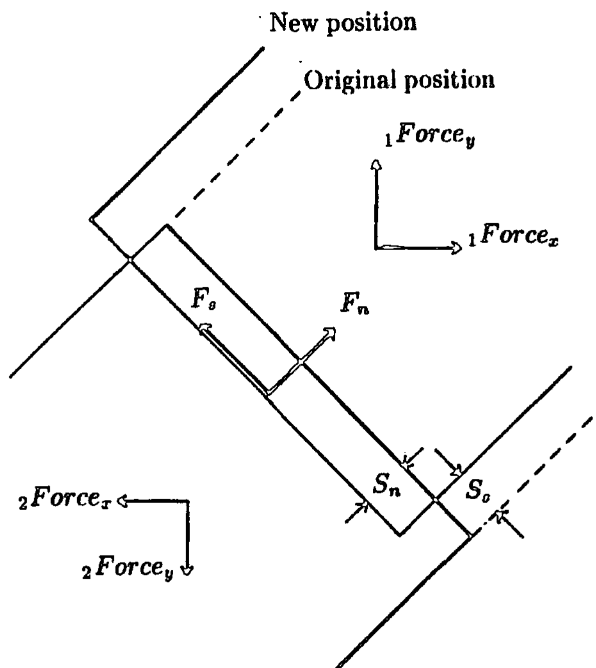


Figure 2.2 The forces associated with a contact

so that the normal force on the contact is given by the normal penetration (for example the movement of one block edge into another) multiplied by the contact stiffness, as in equation (1). A list of the mathematical notation used throughout this discussion is contained in Appendix A.

$$F_n = -S_n \times k_n \tag{1}$$

Likewise the shear force is given by the product of the shear movement and shear stiffness, equation (2).

$$F_s = S_s \times k_s \tag{2}$$

If the dashpot contact damping is in force the normal and shear dashpot forces are calculated in like manner, equations (3) and (4).

$$H_n = -S_n \times K_n \tag{3}$$

$$H_s = S_s \times K_s \tag{4}$$

These normal and shear forces are constrained by the following failure criteria. Firstly if the contact is in tension, that is $F_n < 0$ then F_n, H_n, F_s, and H_s are set to zero. Secondly the shear forces are restrained by a friction law so that if $|Fs| > \mu \times F_n$ then $F_s = \mu \times F_n \, |F_s| \, / F_s$ where μ is the coefficient of friction of the contact.

Having obtained the contact forces, they are resolved to give forces in the x and y directions which are then summed onto the blocks involved. The moment about the block centroid is also calculated and summed.

Once the force displacement law has been executed for all of the contacts on an element, the forces on the element are known. The motion law can relate these forces to element movements.

The facility for imposing force boundary conditions on the problem allows for modelling of systems including rock bolts, these being simulated as constant body forces on certain blocks. Displacement boundary conditions permit some blocks to be immovable for either the whole or part of the simulation, enabling the system to consolidate and preventing it from acting as a rigid body under gravity.

The time step, the unit of time that each cycle is deemed to have modelled, cannot be made arbitrarily large in the hope of reducing the number of iterations required for the simulation time, for, in doing so numerical instability will be encountered. This instability manifests itself as small contact oscillations. To control these oscillations numerical damping has been used, although this removed energy from the system in an apparently arbitrary fashion, Rouse (1982).

In dealing with these problems an observational investigation was carried out, the findings of which are presented in the following sections. This investigation was carried out using the Program CVS and a graphical module SOP (simulated output program) written especially for plotting the output from CVS. Program CVS uses the same undamped motion and force displacement laws as Cundall. Basic input is the number of elements, the particle mass, the stiffness and the time step size. Options include Simple Harmonic Motion simulation, Contact simulation, shear force inclusion or exclusion, contact slope angle, and the number of iterations.

2.1.1.2 A Simple Implementation

The essential feature of Distinct Element Analysis is the force, acceleration, velocity, displacement cycle. In this case the force – displacement law is executed as the starting point in the cycle. Part of the elegance of a Distinct Element Analysis solution is the simplicity with which these quantities can be calculated at each cycle in rotation. It is not easy to calculate them independently of previous

cycles, but a general formula may be derived for very simple cases in terms of solutions to sums of series and difference equations.

There also exist mathematical solutions for systems using non discrete integrating operators, but again they are extremely complex for anything but the simplest cases. The principal difference in these two approaches is the period of time step, in the Distinct Element Analysis this is the time step size whilst in the Calculus it can be considered as zero. These two types of solution therefore deal with quantised and continuous time respectively.

It follows that any system with constant acceleration shall have equal solutions in quantised and continuous time and that the quantised solution to a system with variable acceleration shall be an approximation to the true solution found in continuous time.

2.1.2 The Behaviour of a Single Contact

Figure 2.3 shows the behaviour of a single contact between two point masses. The 'lower' mass is fixed and has a zero initial overlap with the 'upper' which is allowed to move under the influence of both gravity and the contact force. The contact force is given by the product of the overlap and the stiffness and acts to separate the masses. In this case it directly opposes gravity. The waveform can be split into two portions, firstly a simple harmonic motion and secondly a freefall condition.

For such a simple case it can be seen that a contact force equal to the weight would counter gravity and thus represents an equilibrium condition (acceleration = 0). This contact force corresponds to an overlap of mg/k. In an undamped system this is never attainable because to gain an overlap of mg/k the particle

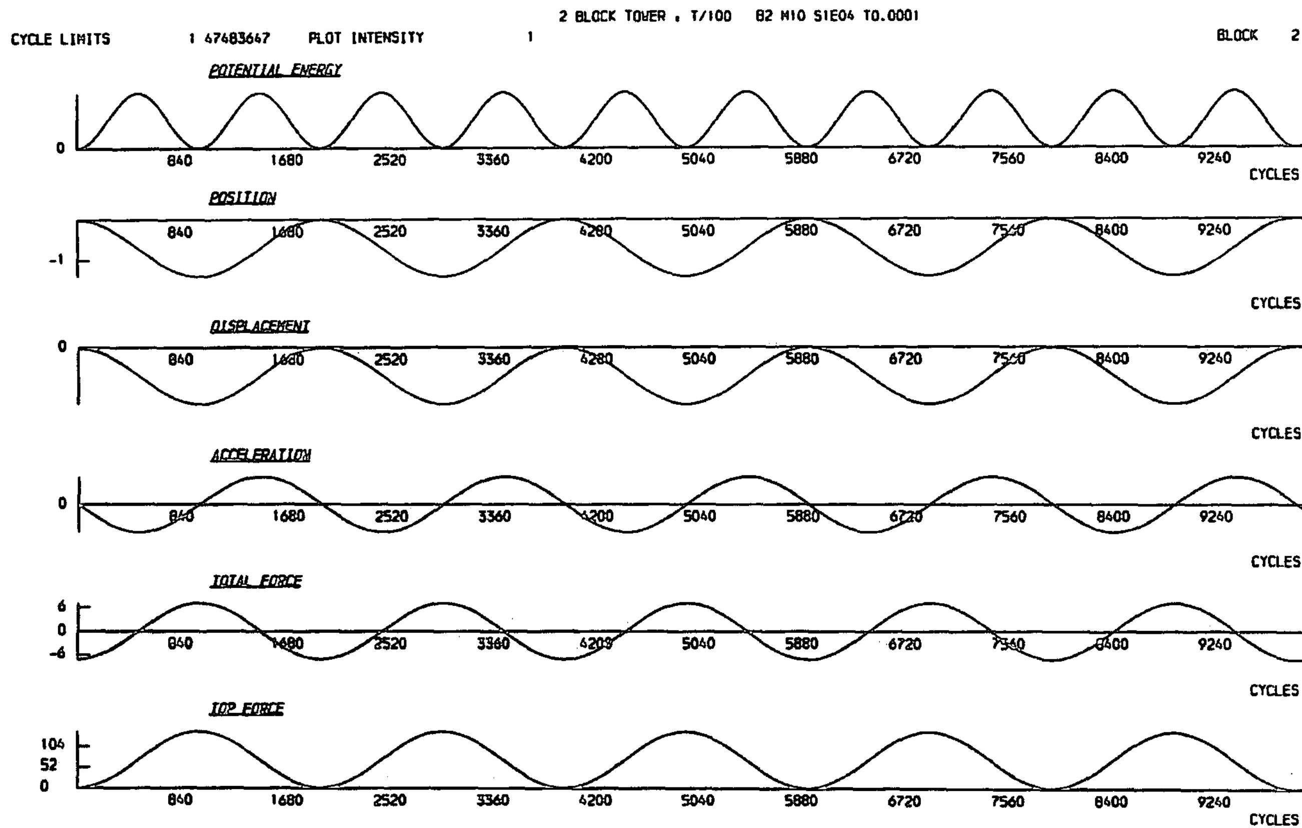


Figure 2.3 The behaviour of a single contact

must have a downwards velocity, which will carry it beyond mg/k during the following cycles.

The non–zero velocity at zero acceleration positions leads the particle always to overshoot this crucial position. The contact force gives rise to an acceleration larger than gravity, causing the mass to decelerate, the velocity eventually becomes zero and then changes direction. The mass moves upwards towards the mg/k position, and for the same reason the mass overshoots on its return and again begins to decelerate. The velocity changes direction again when there is no overlap and the mass moves down to begin a new period of oscillation.

This motion would continue indefinitely for continuous time with constant period and amplitude. However the time is quantised and it is unlikely that the incremental displacements would sum exactly to mg/k, $2mg/k$, mg/k and 0 during the first period. So then, the change of direction of the acceleration and velocity do not occur precisely at these overlaps but rather at those corresponding to the end of the time step which includes these overlap values. The overlaps are therefore a little greater than mg/k, $2mg/k$, and a little less than mg/k and 0. In the case of the last value the masses have separated and freefall ensues until contact is regained. This inaccuracy applies during freefall as well, so that the upper mass regains contact with the lower at a slightly higher velocity than expected.

The second period of oscillation is slightly different from the first, in that the particle begins this period with a downwards velocity. It therefore travels further in the first time step of the new period than in the old one. The particle decelerates more rapidly due to the increased overlap and as it approaches mg/k it does so with a lower velocity and does not overshoot as far. Consequently at the end of second period the separation or 'jump' is less.

Successive periods of oscillation alternate those similar to the first, that is beginning with a small or zero jump, and those like the second with a larger jump.

Assuming suitable parameters for mass, gravity, time step and stiffness will lead to the jump being restricted to a small value, the separation occurring for a single calculation cycle. In this case the motion is essentially that of Simple Harmonic Motion. However each of the parameters will affect the motion principally by increasing the size of the jump.

The finite difference method gives rise to a movement of $g\delta t^2$ in the first cycle if the initial velocity is zero. This is true for all cases here. Consider the motion after the first cycle if $\delta t^2 > m/k$. The initial contact force will be greater than mg. In this case the particles will separate, free fall and regain contact. The high regain velocity will cause a larger overlap than that of the first period. It is clear that both the amplitude and wavelength of this asymmetric oscillation will increase with successive periods.

This instability may be caused by a time step and / or stiffness that is too big, or alternatively by a too small a mass. The gravity determines the time that the particles are separate. The two extremes of motion described, the quasi–stable oscillations and these large jumps are end members of a series of oscillation types.

2.1.3 Controlling Numerical Instability

In an undamped system the large oscillations may be avoided by shrewd use of the problem parameters. The quasi stable oscillations, however, cannot. Damping is required to control the quasi stable oscillations for the following reason. As collapse of a real system occurs, the energy release due to collapse is absorbed through noise, heating, grinding, breakage of material and loss by vibration to the far field. Unless a damping regime is imposed on the numerical analysis the system

energy will never decrease, and not reflect reality. In particular the analysis will be too liberal in passing effects from one area to another.

Although we have only considered oscillations in the contact normal direction, for an inclined contact they exist in the contact shear direction as well. These oscillations are in phase, the stresses varying from zero to the maximum values at the same time. When a failure law is imposed on the shear stress the oscillations distort. What is of greatest importance is that the stress path followed by the contact will be cyclic and unlike the correct, or even a sensible one.

Cundall's implementations of Distinct Element Analysis involve the use of damping factors to control these oscillations, the processes of which have been adequately described by Cundall (1976), Rouse (1982) and referred to in section 1.4.1. Rouse reports that these damping regimes are unsatisfactory on two counts. Firstly that the mass proportional damping factors for the whole mesh do not damp masses differing by a factor of two from the mean. Standing waves were also encountered in towers of contacts, and finally self exciting oscillations were easily produced by the viscous damping, the use of which she strongly discouraged.

2.1.4 Towers of Contacts

A tower of contacts is a set of elements with each resting on one immediately below, the bottom one being immovable. In the first time step, all the movable elements fall by $g\delta t^2$. In the second, the lowest contact is in compression, so the first movable particle moves down less than $g\delta t^2$. All of the other elements move down by the same amount as in the first cycle. In the third, the bottom two contacts are in compression. It can be seen readily that the onset of contact compression travels up the tower at the rate of one contact per calculation cycle. It is for this reason that all effects propagate through the mesh at the same rate as this.

In this system an element's oscillations are coupled to its neighbours only while the contacts are compressive. If jumping should occur at any stage, the jump would cause a compression wave to travel up the contacts above the element. The top element would separate from the column, free fall and begin a compression wave travelling down the column. It is this effect which gives rise to the possibility of standing waves.

2.1.5 Some Recommendations

In deciding the values of the parameters, care should be exercised to ensure the following conditions:

(i) that $g\delta t^2$ is modest.

(ii) that the time step is sufficiently small for the oscillations to be traced with a reasonable degree of accuracy.

(iii) that the contact overlaps are small relative to the size of the elements at all times.

(iv) that the time step is sufficiently large for the computing cost to be acceptable.

(v) that the damping quenches all quasi stable oscillations in a reasonable fashion.

(vi) that energy is dissipated from the system during collapse.

A new approach was researched because current Distinct Element Analysis implementations seem to be oscillation prone. These oscillations lead to incorrect stress paths being followed and the oscillations are difficult to adequately damp.

2.2 An Alternative Approach

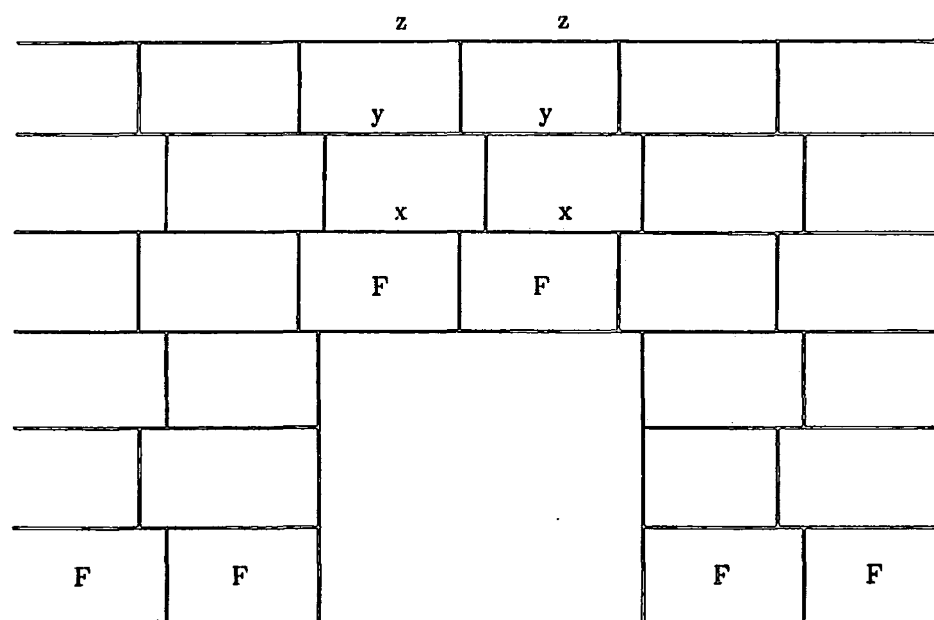
2.2.1 Consolidation

Consolidation, in this context, is the relaxation of a stable configuration of elements until equilibrium is attained. Cundall (1971) recommends that a problem

be analyzed in two parts, firstly by allowing consolidation to take place and then by collapse. The process of consolidation must take place under different conditions to that of collapse, otherwise both processes would prove to be identical. The aim of consolidation is to stabilise the contact forces to values which counter the self weight of the mesh so that during collapse the initial contact normal forces give rise to the correct limiting shear forces and that shearing may take place under the correct conditions.

Cundall outlined two methods of promoting consolidation, under conditions of artificially high friction and by the use of fixed elements to prevent collapse. The high friction method may be utilised for problems where the failure mechanism is principally sliding. Here the elements do not collapse because the contact shear forces are allowed to be large to prevent contact failure. By using this a philosophical problem is encountered, for when the friction is lowered to normal for collapse to ensue, the contacts fail immediately with no chance for the normal forces to compensate smoothly or, more importantly to follow the appropriate stress path. When the consolidation forces are much higher than the contact failure limits it may be argued that the system is as removed from the correct failure force system as it was before consolidation took place.

Where the failure mechanism is that of a toppling and sliding mixture then additional fixed elements are used to prevent movement. These are removed after consolidation has taken place. In Figure 2.4 fixed blocks have been used to allow consolidation throughout the whole of the problem. On unfixing the central supporting blocks the first row will drop, reducing the normal forces on the edges marked x. It can be seen that a wavefront of reduced normal forces will propagate through the mesh at a rate of one element per cycle. Although the side forces may restabilise the system, this wavefront seems to be an added complication to an already complex damping system. Furthermore, it must be questioned whether consolidation by fixed blocks is appropriate as in the real case the lowest blocks



x – joints relaxed in first collapse cycle

y – joints relaxed in second collapse cycle

z – joints relaxed in third collapse cycle

Figure 2.4 The use of fixed blocks to promote consolidation

would be held by side friction, interlocking joints and cohesion. Again it may be argued that this method leads to an incorrect force system.

2.2.2 A Statement of the New Approach

Any new method should allow for the correct gradual build up of forces both before and during collapse. The method should be inherently numerically stable unlike the usual Distinct Element Analysis implementations and it should correctly model the progressive nature of the collapse.

The new approach controls the consolidation of the elements in a more conservative fashion than the usual Distinct Element Analysis in that neither element velocities nor displacements are allowed to build up across time step boundaries. In essence, the incremental displacements of the previous cycle are used in a force – displacement law to give an increment of contact stress, which is then added to the contact stress. A very simple motion law is executed to give rise to new incremental displacements. The contact stresses gradually increase until the incremental displacements fall to very low levels. A contact failure law may be included in the process but for simple consolidation it need not. If no contact failure law is used and a collapse algorithm, such as a standard Distinct Element Analysis method, is processed afterwards, the artificially high shear forces for the contacts that will fail, must be reduced carefully to prevent shocking the system. Just as in normal Distinct Element Analysis this new method relies upon fixed elements to form an immovable platform.

This force orientated system requires a slightly new calculation order which is shown in Figure 2.5. To interface between the consolidation and collapse phases consolidation forces are used in the collapse motion law.

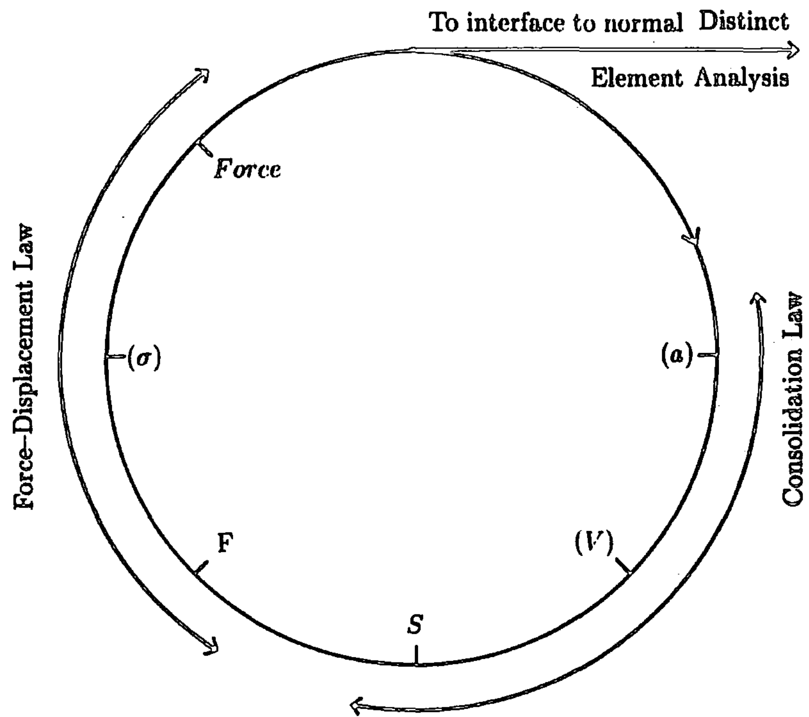


Figure 2.5 The new calculation order

The consolidation process provides as an end result contact forces, whereas normal Distinct Element Analysis provides element overlaps and hence element body forces. As the incremental forces are added to the consolidation forces in each cycle there is a more complex zero tension condition because a separation between elements implies a tensile increment rather than necessarily a tensile contact. A tensile condition occurs when a tensile increment is added to the consolidation forces which causes the result to be less than zero.

It is essential, of course that the correct forces are produced by the analysis. To check this Program CIRCLES was used to consolidate a single contact, a tower and a triangle involving two contacts. In the case of the single contact, the force summed to mg, for the tower jmg (where j is the number of circles above the contact) and for the sixty degree triangle of elements the combined contact forces resolved to give mg.

2.2.3 Machine Accuracy

2.2.3.1 The Relevance to Discretization

Machine accuracy may be expressed in absolute terms as the number of bits used for a real number, or more usefully as the number of significant figures held, or the smallest number added to 1.0 which gives a result greater than 1.0.

The computer used throughout this study was a System 370, Amdahl 470 V/8 Serial Number 70435 at Durham using the Michigan Terminal System (MTS) of the University of Michigan Computer Center, Ann Arbor, Michigan. The Pascal compiler used was PASCALJB of Plug Compatible Software, Inc. The machine accuracy in this case is such that if $2.220446049 \times 10^{-16}$ is added to 1, the result is

just discernible. Therefore, to maintain the accuracy of the algorithmn it is essential to keep calculations involving small quantities separate from those involving large ones for as long as possible.

For example, the position of the elements varies during the analysis by adding incremental displacements to the centre of gravity. Small increments are easily lost, so these are stored separately to ensure that they maintain their integrity. To illustrate,

$$(1.0 + 10^{-17}) - (1.0 + 10^{-18}) = 0$$

whereas

$$1.0 - 1.0 + 10^{-17} - 10^{-18} = 9.0 \times 10^{-18}$$

2.2.3.2 Bringing Consolidation to a Close

For a convergent, stable system, there comes a point during consolidation when the incremental forces become very small and it is necessary to terminate the process while all the quantities are above the machine accuracy. To do this the maximum increment displacement is determined in each cycle and when this has fallen to the limiting arbitrary value of 10^{-14} consolidation is considered complete. At this point the force matrix gives rise to extremely small displacements and the system may be thought of as at equilibrium.

For a convergent, unstable system the maximum incremental displacement with time becomes asymptotic to a constant value. Consolidation is brought to a close under conditions of constant displacement. That is, it is not possible for the consolidation forces to counter the effect of gravity in at least part of the mesh.

For a divergent system the process is halted if the maximum incremental displacement reaches an arbitrary high value of 10^{6}. Such a system is considered

to be numerically unstable, halting the process prevents the program crashing in an uncontrolled fashion.

2.2.4 Propagating Effects Through the Matrix

2.2.4.1 A Simple Tower Problem

What occurs in one part of the mesh is very likely to affect another part. It is essential that these effects are correctly propagated through the mesh. Plesha *et al.* (1986) report that the propagation mechanism of waves through a distinct element mesh is not understood. In this study it was found that propagation may be very limited and is controlled by machine accuracy, stiffness, time step and unit length.

Figure 2.6 shows a tower of CIRCLE elements. Table 2.1 shows the consolidation forces at completion, it can be seen that no contact forces exist above the seventh contact. As explained in section 2.1.4 propagation of the onset of consolidation travels at one element per cycle up the tower, those elements above falling under the influence of gravity only.

0.0000000000000E+00
0.0000000000000E+00
0.0000000000000E+00
7.1642691779061E-14
4.7029380390028E-12
1.0319038623585E-09
1.8433017642260E-07
2.4740616541751E-05
2.2250740175403E-03

Table 2.1 Contact forces for a tower

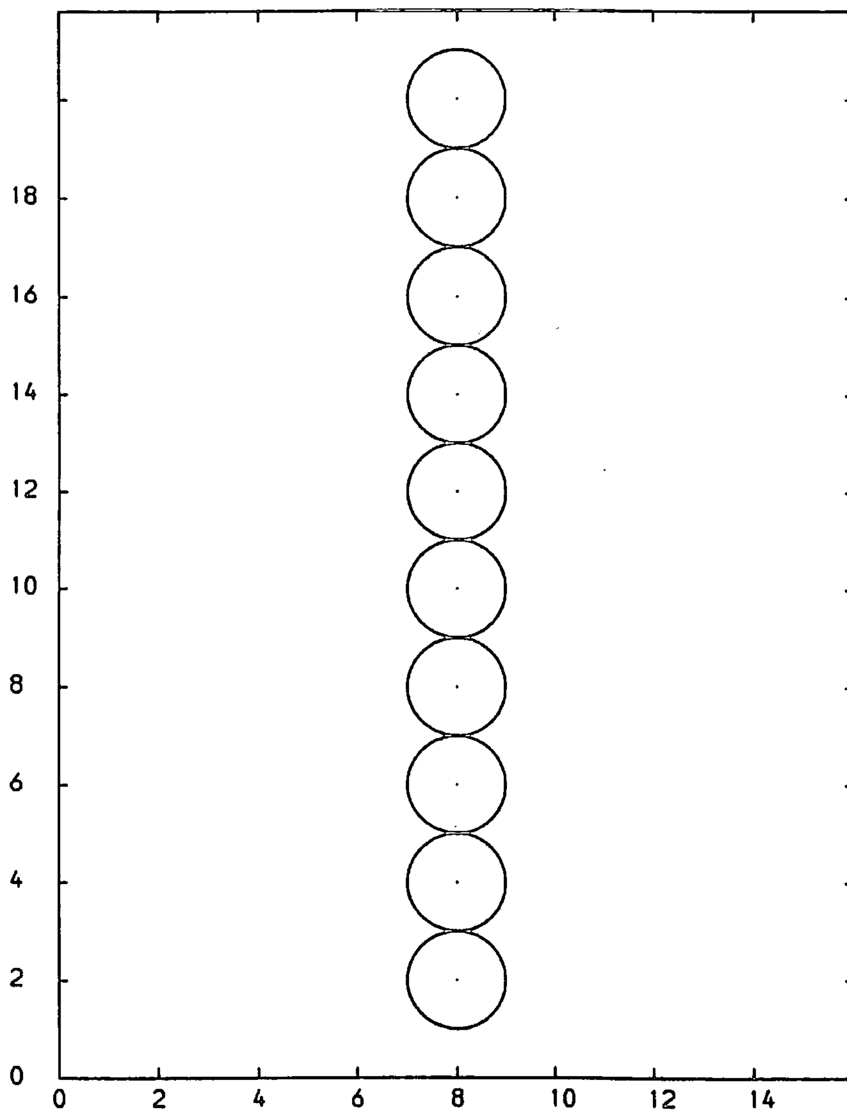


Figure 2.6 The tower of circles analysed

The relative displacement between two elements is given by

$$_1P_y - {}_2P_y + {}_1S_y - {}_2S_y$$

where $_1P_y$ and $_2P_y$ are the y positions and $_1S_y$ and $_2S_y$ are the incremental displacements of the two elements. At the limit of propagation the value $_1S_y - {}_2S_y$ is 'lost' when added to $_1P_y - {}_2P_y$ resulting in a zero relative displacement. From this point onwards propagation ceases as the contact force is also zero.

Cycle	Circle 1	Circle 2	Circle 3
0	$a = -g$ $S = -g.\delta t^2$	$a = -g$ $S = -g.\delta t^2$	$a = -g$ $S = -g.\delta t^2$
1	$F = k.d.g.\delta t^2$ $a = \frac{k.d.g.\delta t^2}{m} - g$ $S = \frac{k.d.g.\delta t^4}{m} - g.\delta t^2$	$F = 0$ $a = -g$ $S = -g.\delta t^2$	$F = 0$ $a = -g$ $S = -g.\delta t^2$
2	$F = k.d.g.\delta t^2 - \frac{2.k^2.d^2.g.\delta t^4}{m}$ $a = \frac{k.d.g.\delta t^2}{m} - \frac{2.k^2.d^2.g.\delta t^4}{m^2} - g$ $S = \frac{k.d.g.\delta t^4}{m} - \frac{2.k^2.d^2.g.\delta t^6}{m^2} - g.\delta t^2$	$F = \frac{k^2.d^2.g.\delta t^4}{m}$ $a = \frac{k^2.d^2.g.\delta t^4}{m^2} - g$ $S = \frac{k^2.d^2.g.\delta t^6}{m^2} - g.\delta t^2$	$F = 0$ $a = -g$ $S = -g.\delta t^2$

Table 2.2 The expansions for 3 circles

Table 2.2 shows the expansions for the acceleration, displacement and forces on three circles in a tower for three cycles. Of the controlling parameters used, time step affects the size of the incremental displacement most. As expected, as the time step is decreased the displacement decreases also, unfortunately it can quickly disappear. Likewise as the stiffness is increased smaller displacements are required to represent the same forces. Gravity affects the displacements proportionally whereas the other parameters have a greater effect.

2.2.4.2 Some Recommendations

To promote propagation through a large number of elements it is suggested that stiffness and time step size be equal to unity and that the numerical stability of contacts be controlled by a single damping factor. The size of the problem should be limited so that small quantities are added to modest element positions. It is suggested that the elements are of the order of one unit in radius for CIRCLES and one unit in width for SLICES.

2.2.5 The Role of Damping

The difference equation solution for a single contact under the influence of gravity is shown below. The governing equations for the displacement, acceleration and force are shown in equations (5) to (7). The sign convention of gravity as positive is used.

(5) $$ {}^{n}S_y = {}^{n}a_y \times \delta t^2 $$

(6) $$ {}^{n}a_y = \frac{{}^{n}F_y}{m} + g_y $$

(7) $$ {}^{n}F_y = -\sum_{i=0}^{n-1} k \times d \times {}^{i}S_y $$

On substituting equations (7) and (6) in (5) it is found that

(8) $$ {}^{n}S_y = -\frac{k \times d \times \delta t^2}{m} \times \sum_{j=0}^{n-1} {}^{j}S_y + g_y \times \delta t^2 $$

Let $A = \frac{k \times d \times \delta t^2}{m}$ and $B = g_y \times \delta t^2$ then the difference between two consecutive incremental displacements becomes

(9) $$ {}^{n+1}S_y - {}^{n}S_y = -A \times \left(\sum_{j=1}^{n} {}^{j}S_y - \sum_{i=1}^{n-1} {}^{i}S_y \right) $$

Which simplifies to

$$^{n+1}S_y - {}^{n}S_y = -A \times {}^{n}S_y \tag{10}$$

Let $^{n}S_y = p_y^n$ so that

$$p_y^n \times [p_y - (1 - A)] = 0 \quad \Rightarrow \quad p_y = (1 - A) \tag{11}$$

Now $^{0}S_y = g_y \times \delta t^2 = C(1 - A)^0$ which implies that $C = B$ So that

$$^{n}S_y = B \times (1 - A)^n \tag{12}$$

The number of calculation cycles for a system to converge to a limiting difference may be derived. The limiting factor may be force, acceleration or displacement. A different equation is required for each, they are given below.

The derivation of N_f, the number of cycles needed to converge to a limiting positive difference in force of Lim_f is shown. Care needs to be exercised regarding the gradients of the acceleration, displacement and force time graphs as gravity is taken as positive downwards. The limit is defined as $-Lim_f = {}^{N_f+1}F_y - {}^{N_f}F_y$

$$Lim_f = k \times d \times \left(\sum_{j=0}^{N_f} {}^{j}S_y - \sum_{l=0}^{N_f-1} {}^{l}S_y \right) \tag{13}$$

which gives $Lim_f = k \times d \times {}^{N_f}S_y$ and $Lim_f = k \times d \times B(1 - A)^{N_f}$ and hence by logarithmns

$$N_f = \frac{\lg(Lim_f/(k \times d \times B))}{\lg(1 - A)} \tag{14}$$

The equivalent equations (15) and (16) show the number of cycles required when the limiting factor is acceleration and displacement respectively.

$$N_a = \frac{\lg(Lim_a \times m/(k \times d \times B))}{\lg(1 - A)} \tag{15}$$

(16) $$N_s = \frac{\lg(Lim_s/(A \times B))}{\lg(1 - A)}$$

Furthermore for $N_s > 0$ implies $0 < kd\delta t^2/m < 1$ and assuming $k = 1$ and $\delta t = 1$ then it follows that $0 < d/m < 1$ and $N_s = \frac{\lg(Lim_a \times m/(d \times g))}{\lg(1-d/m)}$ which implies that N_s varies with mass. So that all elements take equal time under the same conditions to consolidate the damping factor actually employed is element mass multiplied by the global damping factor: $D_f = d/m$. This makes the consolidation process time independent of mass, if this were not so the time would 'warp' over the mesh with lighter elements more advanced than others. A further consequence would be that heavy elements would be lightly damped compared with light ones. Therefore $N_s = \frac{\lg(Lim_s/(D_f \times g))}{\lg(1-D_f)}$

To show the effect of the number of contacts upon these equations, the difference equation solution for an isosceles triangle where the lower two circles are fixed, now follows. The internal angle between the horizontal and the contact lines is θ. The derivation is carried out for the y direction only as the x can be shown to cancel out and have no effect. ΔG is used to represent the movement along the contact lines that join the circles. The subscripts 1 and 2 are used to differentiate between the two active contacts. The governing equations for the displacement and acceleration are as shown previously in equations (5) and (6). The equation for the force is given in (17).

(17) $${}^{n}F_y = -\sum_{i=0}^{n-1} k \times d \times ({}^{i}_{1}\Delta G + {}^{i}_{2}\Delta G) \times \sin\theta$$

However the radial displacements are given by

(18) $${}^{n}_{1}\Delta G = {}^{n}S_y / \sin\theta$$

Equation (17) simplifies to give

(19) $${}^{n}F_y = -\sum_{i=0}^{n-1} 2 \times k \times d \times {}^{i}S_y$$

By following the previous method it may be shown that

$$^{n}S_y = B \times (1 - 2 \times A)^n \tag{20}$$

and that $N_s = \frac{\lg(Lim_s/(D_f \times g))}{\lg(1-2D_f)}$. Hence generally

$$N_s = \frac{\lg(Lim_s/(D_f \times g))}{\lg(1 - I \times D_f)} \tag{21}$$

where I is the number of contacts. It should be noted that this applies to symmetrical contacts sharing the weight of elements above. It can be concluded that the damping factor, D_f, must vary according to the mesh geometry used in Program CIRCLES.

As damping increases so does the required number of cycles to reach equilibrium, and hence the computer time increases also. An overall damping factor of 0.5 to 0.9 has proved to be satisfactory.

2.2.6 Concluding Remarks

In this chapter some general theory has been investigated for Distinct Element Methods as used previously. A new method has been outlined and aspects of the theory have been detailed, in the following two chapters the theory is extended for SLICES and CIRCLES and their implementation described.

CHAPTER 3

DISTINCT ELEMENT METHOD OF SLICES

3.1 Introduction

Program SLICES is designed to analyse the behaviour of a soil slope and a failure arc in much the same way as more traditional methods, such as Janbu. The program requires geometric data to define the slices. The tops of these form the soil slope, and the bottoms form the failure arc. The slices may vary in width and should normally have vertical sides.

In addition cohesion, friction and pore water pressure values are needed for each slice. As the slope cross section may be considered as one unit length thick, and if the unit length is the metre, densities in tonnes $/m^3$, gravity as $-10m/s^2$, then the cohesions and porewater data should be entered as kN/m^2 and m. However any self consistent units may be used.

Finally control parameters and control commands govern the conditions of the solution and the production of the various outputs. Table 3.1 shows a typical command file while Figure 3.1 shows the slope and arc to be analysed.

It must be emphasised that Program SLICES is a development program, and, although hopefully, reasonably user friendly this is for ease of use rather than as an indication of a packaged production program. Program SLICES is therefore limited in its applications. It is only able to analyse situations involving one soil type and has a simple contact failure law. It is envisaged that non–linear soil properties could be incorporated quite simply and that layered situations be

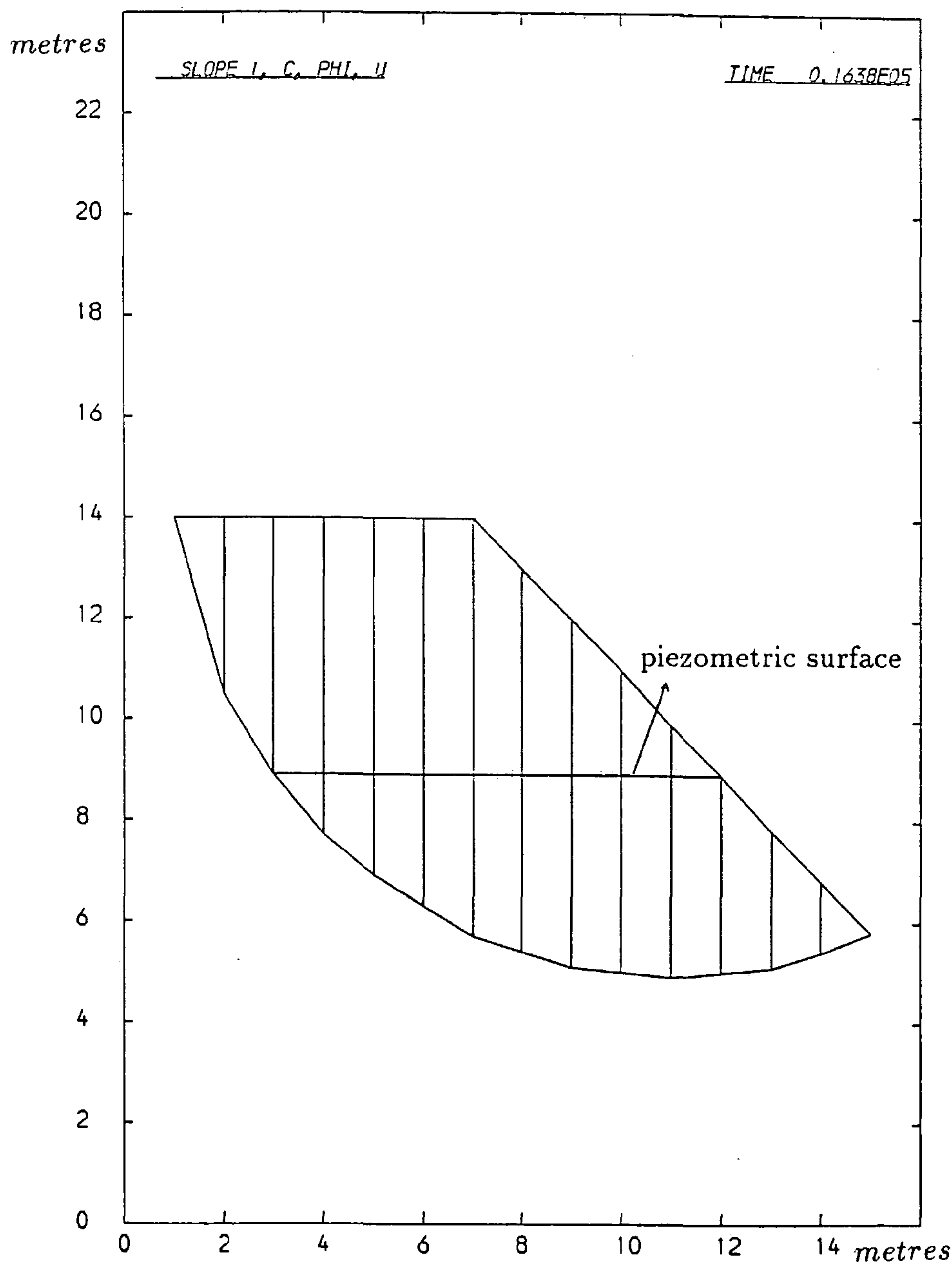


Figure 3.1 A typical slope for analysis by SLICES

```
start  SLOPE 1, C, PHI, U
 0 16 0
 create free  20 5.0 2.0 1.0 20 5.0 0.0  0.0  0.23  1  14    1  14
                                                   2  10.5  2  14
 create free  20 5.0 2.0 1.0 20 5.0 0.0  0.0  0.23  3  8.9   3  14
 create free  20 5.0 2.0 1.0 20 5.0 0.6  0.6  0.23  4  7.7   4  14
 create free  20 5.0 2.0 1.0 20 5.0 1.6  1.0  0.23  5  6.9   5  14
 create free  20 5.0 2.0 1.0 20 5.0 2.3  1.3  0.23  6  6.3   6  14
 create free  20 5.0 2.0 1.0 20 5.0 2.9  1.6  0.23  7  5.7   7  14
 create free  20 5.0 2.0 1.0 20 5.0 3.35 1.75 0.23  8  5.4   8  13
 create free  20 5.0 2.0 1.0 20 5.0 3.65 1.9  0.23  9  5.1   9  12
 create free  20 5.0 2.0 1.0 20 5.0 3.85 1.95 0.23 10  5.0  10  11
 create free  20 5.0 2.0 1.0 20 5.0 3.95 2.0  0.23 11  4.9  11  9.9
 create free  20 5.0 2.0 1.0 20 5.0 3.95 1.95 0.23 12  5.0  12  8.9
 create free  20 5.0 2.0 1.0 20 5.0 3.3  1.35 0.23 13  5.1  13  7.8
 create free  20 5.0 2.0 1.0 20 5.0 2.05 0.7  0.23 14  5.4  14  6.8
 create free  20 5.0 2.0 1.0 20 5.0 0.7  0.0  0.23 15  5.8  15  5.8
meshend
set time 1 gravity -10 cmdproc on framelimit 100
  writegap 128  interval 128
  cmdlist plot standard set calc writegap * 2 interval * 2 cend
  echo on
go 32383
stop
```

Table 3.1 A Typical Command File

accommodated by subcontacts along the interslice edges. In addition Program SLICES is intended for slopes failing by sliding only, as opposed to toppling. There is no check made by the program for toppling. Program SLICES is particularly suited for situations where the failure surface is already known, as in back analysis and post-mortem analysis of failed slopes.

The program does not display the problem solution in a fixed format, but, rather a series of possible outputs may be requested by the user. The calculation marches through time producing data and to a very great extent which data are examined, and how, is left to the user. Of the several possibilities, perhaps the

most useful is shown in Figure 3.2. Here the stresses are plotted along the length of the arc. Normal, shear and limiting shear stresses for both the base and interslice contacts are shown. The symbols $+$, $\diamondsuit$ and $*$ are used to represent the pore water pressure, mobilised stress and limiting stress respectively. Plot possibilities include incremental displacements, slice geometry and the stress profiles as illustrated.

Much written output can be produced for debugging and general information. To complement the stress profile plots, written output can be produced independently. This output consists of the Factor of Safety (Limiting stress over mobilised stress), for each of the base contacts and shows which slices are stabilising the slope. The limiting state is represented by a factor of safety of unity.

The analysis may be considered static in nature in so far as the slices are not allowed to collapse, only to consolidate. The term consolidation is being used here to describe the process whereby the contact forces increase over successive cycles to counter the self weight of the slope. The cyclic process employed includes a failure law for the contacts which is executed during the force displacement law. The complete cycle is shown in Figure 3.3. It may be seen that in addition to the calculation sections there is a controlling section which is capable of terminating a run and to administer the production of output.

3.2 Theory extensions for SLICES

3.2.1 The edge formulation employed

The Distinct Element Analysis formulation used is an Edge formulation. As stated in the Introduction, program SLICES is a Distinct Element Analysis implementation of fewer degrees of freedom than usual as rotation is ignored. As the problem types are of sliding only, rotation or toppling of slices can be safely discarded. Both edges involved in a contact, therefore, shall be parallel at all

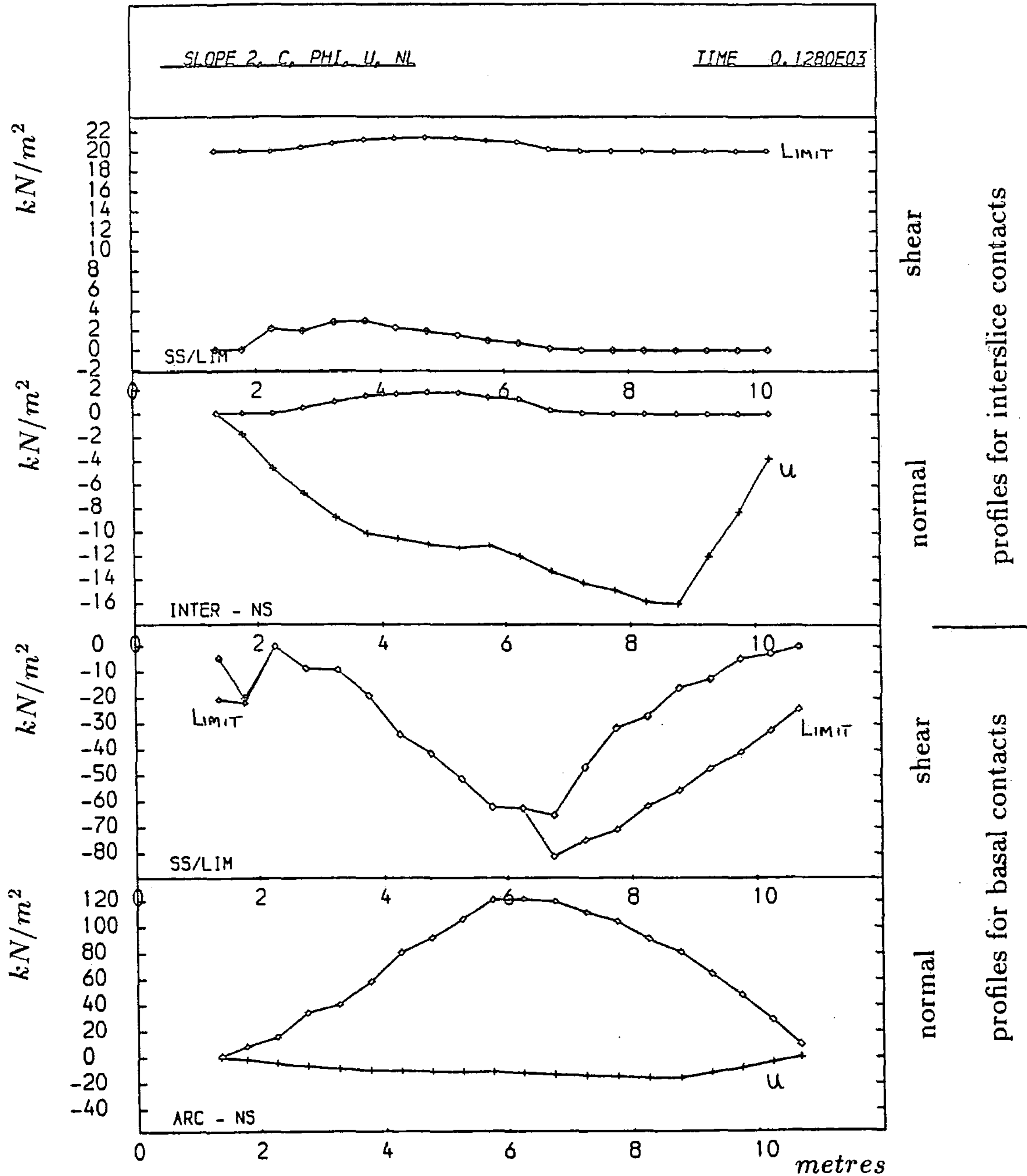


Figure 3.2 Stress profile produced by SLICES

A negative normal stress is a tensile stress.

A negative shear stress is dextral shear for basal contacts and is sinistral shear for interslice contacts.

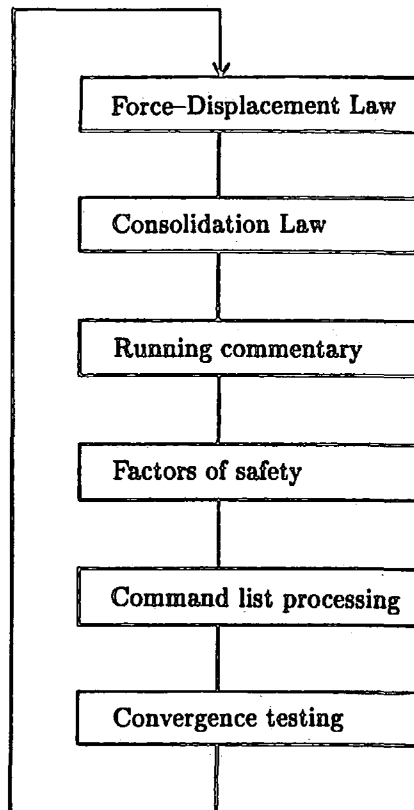


Figure 3.3 The SLICES calculation cycle

times and this simplifies the edge formulation allowing a contact to be defined by a single subcontact. A further simplification to usual edge formulations is possible, as initially both edges involved are closed along their whole lengths. This renders the storage of a contact origin unnecessary. Figure 3.4 compares a general Distinct Element Analysis edge contact with that employed in SLICES. The lefthand diagram shows the information required for a contact in the traditional Distinct Element Analysis implementation of Watson (1983). The righthand diagram shows the contact definition used here. It may also be noted that the contact length is deemed to be a constant throughout the simulation and is used to convert forces to stresses by division.

During the definition of the problem the program creates the slices from left to right and allocates a contact to the base of each slice and to the righthand edge of all except the last slice. The base contacts are made with a fixed hypothetical element, the platen, which consists of a series of edges identical to the base edges of the slices. As the leftmost slice has no lefthand contact and the rightmost slice has no righthand contact, slopes facing either left or right may be modelled.

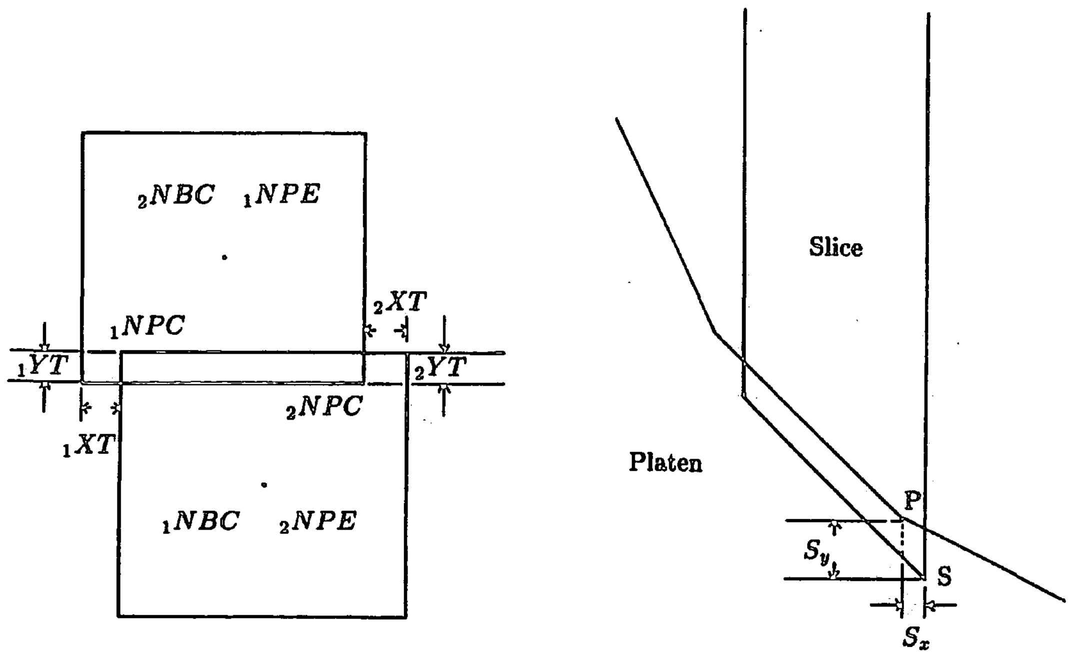
The main extension of the theory for slices is the incorporation of the edge length into the difference equation solution. The inclusion of the contact length L to convert from stress to force is shown in equations (1) and (2).

$$ {}^{n}F_y = - {}^{n}\sigma_y \times L \tag{1} $$

$$ {}^{n}\sigma_y = \sum_{i=0}^{n-1} \frac{k \times d \times {}^{i}S_y}{L} \tag{2} $$

By following the method used in chapter 2, it is found that

$$ {}^{n}S_y = B \times (1 - A)^n \tag{3} $$



NBC – Block number that owns corner NPC

NPE – Block number that owns edge involved

NPC – Corner number involved in contact

P – Platen corner involved

S – Slice corner involved

Figure 3.4 A Comparision of the data required to define a contact

and N_σ, the number of cycles needed to converge to a limiting difference in stress of Lim_σ is given by

$$N_\sigma = \frac{\lg(Lim_\sigma \times L/(k \times d \times B))}{\lg(1 - A)} \tag{4}$$

This number is an useful indicator of the rate of damping on the system. It has already been shown that the rate of damping is dependent on the element masses. For a typical slice problem this would mean that all the slices would be at different stages of the contact stress history. This problem is overcome by applying an individual damping factor to each block, such that $D_f = d/m$, where D_f is the overall damping factor.

A second problem is encountered with damping in that the rate of damping is also dependent upon the contact length. Ideally two contacts of the same slice with different lengths should have different damping factors. In practice, for SLICES the contacts may be considered as two sets of contacts, basal which converge quickest, and interslice contacts which are slower to converge and gradually influence the basal contacts. Due to the geometrical consistency of SLICE problems this side effect is an advantage, however is does complicate the choice of damping factor as it must satisfy the convergence criteria for all contacts.

3.2.2 The Force Displacement and Motion Laws

Procedure *fordsl* defines the force displacement law and is executed once per slice cycle. *fordsl* executes a force displacement law once for the base contact and once for the side contact. The relative movement in the normal and shear directions is calculated from the incremental displacements of the slices involved (the platen may be considered as a slice and has zero displacements at all times). These relative movements are converted to contact forces by the relaxation constant, k.

$$\begin{pmatrix} F_n \\ F_s \end{pmatrix} = \begin{pmatrix} \sin\theta & \cos\theta \end{pmatrix} \times \begin{pmatrix} {}_eS_x - {}_cS_x \\ {}_cS_y - {}_eS_y \end{pmatrix} \times k \tag{5}$$

The contact is judged to be active (that is in contact) if ${}^{n-1}\sigma_n > -D_f \times F_n/L$ (compression is taken as positive), and the following executed to give contact stresses.

$$(6) \qquad {}^{n}\begin{pmatrix} \sigma_n \\ \tau \end{pmatrix} = {}^{n-1}\begin{pmatrix} \sigma_n \\ \tau \end{pmatrix} + \frac{D_f}{L} \times \begin{pmatrix} F_n \\ F_s \end{pmatrix}$$

Slices that are fully submerged in the traditional method of slices, are bouyant which normally causes the toe of the slope to under contribute to the mobilised stress of the failure arc. This leads to the factor of safety to be under valued. If the pore pressure due to the water table is applied to the slices in the distinct element analysis method a similar effect is experienced. Consider equation 7.

$$(7) \qquad \tau = c + (\sigma_n - u)\tan\phi$$

At the beginning of the analysis $\sigma_n = 0$ so that for non–zero pore water pressures $(\sigma_n - u) < 0$ and hence the slices would float upwards. To ensure that $(\sigma_n - u) \geq 0$, the water is applied gradually by increasing its value at an arbitrary rate of 0.1% of the required pressure u per calculation cycle. This rate cannot be currently altered by the user.

$$(8) \qquad {}^{n}u = {}^{n+1}u + 0.001u \quad \text{where} \quad {}^{n+1}u \leq u$$

If the Critical State option is in use then failure of the contact is assumed once ${}^{n}u = u$ and $|\,{}^{n}\tau| > \hat{\tau}$. Once the contact reaches the failure condition the contact remains in the failed state and the cohesion for each successive cycle is given by ${}^{n+1}c = {}^{n}c \times 0.85$, that is $c \to 0$. The failure logic may be represented by the boolean logic of $failure = failure \vee (({}^{n}u = u) \wedge |\,{}^{n}\tau| > \hat{\tau})$.

The shear stress is limited by a Coulomb friction law such that

$$(9) \qquad 0 \leq ({}^{n}\sigma_n - {}^{n}u)$$

and

(10) $$\hat{\tau} = {}^{n}c + ({}^{n}\sigma_n - {}^{n}u)\tan\phi$$

and if $|{}^{n}\tau| > \hat{\tau}$ then ${}^{n}\tau = \hat{\tau} \times \frac{\tau}{|{}^{n}\tau|}$ ensuring sign continuity.

The stresses are converted to contact forces by

(11) $$\begin{pmatrix} C_n \\ C_s \end{pmatrix} = L \times {}^{n}\begin{pmatrix} \sigma_n \\ \tau \end{pmatrix}$$

and finally these forces are resolved in x and y and added to the forces acting on the slice, to be known here as the body forces. Additionally, in the case of a side contact, the resolved contact forces are subtracted from the body forces of the other slice involved.

(12) $$\begin{pmatrix} Force_x \\ Force_y \end{pmatrix} = (\sin\theta \quad \cos\theta) \times \begin{pmatrix} C_n \\ C_s \end{pmatrix}$$

As the incremental displacements are not used to update the slice positions the contact stress increments may be compressive or, by the influence of other slices, tensile. In the tensile case a contact is still active if the summed normal stress state is compressive. A small tensile stress is permissible so that transient numerical jumping may be more effectively damped.

The consolidation motion law is very simple compared to the motion law of normal Distinct Element Analysis. The Procedure *consolsl* defines this law and is executed once per slice per cycle. *consolsl* is called after *fordsl* in the calculation cycle.

The incremental displacements are calculated from the body forces, mass, gravity and time step (recommended as unity).

(13) $$\begin{pmatrix} S_x \\ S_y \end{pmatrix} = \frac{\delta t^2}{m} \times \begin{pmatrix} Force_x \\ Force_y \end{pmatrix} + \delta t^2 \times \begin{pmatrix} g_x \\ g_y \end{pmatrix}$$

The body forces are then set to zero ready for the next cycle

$$\begin{pmatrix} Force_x \\ Force_y \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix} \tag{14}$$

3.3 Using Program SLICES

3.3.1 Introduction

3.3.1.1 Overview

Program SLICES is written in the PASCAL programming language. For some time previously implementations of Distinct Element Analysis have been written in FORTRAN, Dames and Moore (1978), Rouse (1982) and Watson (1983). Whichever language is used it should provide efficient object code and readable, easily maintained, modular source code. Some features of PASCAL enable these objectives to be more easily attainable than many other languages. Such attributes as structured variables and records make PASCAL particularly useful in this respect.

For example *el@.force.x* is equivalent to *A(IA(J)+11)* of Watson (1983). The PASCAL version more clearly indicates that an element force in the x vector is meant than in the FORTRAN version. Furthermore, if 'force' is mistyped then the Pascal Compiler will indicate an error, whereas if 'J' was accidentally replaced with 'K' in FORTRAN a run time error might eventually occur. For a development program where the source is continually being modified these advantages are very great. For a reader unfamiliar with PASCAL, reference is made to Grogono (1980).

Meek and Beer (1986) report that a large percentage of the programming effort in a Distinct Element Analysis implementation development may concentrate upon user orientated features. Much of the structure of programs SLICES and

CIRCLES is designed for the ease of input of data, output of results, debugging features, error handling and restart facilities.

The plotting routines employed are library routines called from the GHOST library. Almost all references to these library routines are contained in Procedure *plot* so that they may be changed or replaced easily if the need should arise. To call FORTRAN library routines such as these from a PASCAL program procedure head definitions are included in the PASCAL code.

Finally, the source code for Program SLICES is in est8:p.slice.s and the object code may be found in est8:p.slice.

3.3.1.2 Input and output unit summary

The program requires various input and output files with which to communicate with the user. There are nine such channels each of which should be assigned on the Run command, for example:

Run est8:p.slice 1=resti 2=resto scards=commands sprint=*msink*
7=-debug 8=-trace 9=-plot 10=-osc 11=*msink*

In MTS the run command should occupy a single line, in addition *msink* is the pseudodevice name for the terminal screen (if run from a terminal) and '-' infront of a filename indicates that it is temporary in nature. Any channel required, but not assigned is prompted for by MTS except for scards and sprint, here the MTS default values are *source* (normally the terminal keyboard), and *sink* (normally the terminal screen). In use it may be noted that channels 8, 10 and 11 are often not needed and so need not be assigned, channel sprint should usually be the default, and finally channels 1 and 2 may be assigned to the same file. Each of the nine input / output channels are described below.

Channel 1 is the restart file input channel. The file attached to this should contain all of the information required to restart a previous problem run which terminated under normal conditions. If a new problem is to be started then a file need not be assigned.

Channel 2 is the restart file output channel. The file attached to this will receive the data required to resume the problem at a later date. This must be assigned as restart information is output at the end of a normal termination as well as when requested.

Channel scards is the command file input channel. The file or device attached to this unit contains the input command language commands that define the task to be done.

Channel sprint is the running commentary output channel. The terminal screen (*msink*) is the default value for this. As control commands are written to the device, to position the output on to various parts of the screen, the network (NUNET at Durham) should be set to allow these control commands to be passed to, and executed by, the terminal. To do this at Durham the NUNET commands Ctrl–p passall=on and Ctrl–p chc=off should be issued prior to the run and Ctrl–p passall=off and Ctrl–p chc=` afterwards.

It should be noted that the control commands used are suitable for TeleVideo terminals and that no other terminal types have been tested, as the commands are contained in two constant strings at the beginning of the program they are easy to modify. For more details see section 3.3.4.7.

Channel 7 is the debug output channel. The file attached will contain all of the debugging output requested. In some cases the amount of information may be

very large and it is best if the file is of a temporary nature so that personal disk space is not exceeded.

Channel 8 is the trace output channel. This file contains the trace of the program, if requested. It contains a message on entry to and on exit from each procedure or function as they are used. This is particularly useful on debugging recursive structures as each level of recursion used is recorded.

Channel 9 is the plot output channel. The file attached receives the plot output stream from the GHOST plotting routines. It contains control codes and unformatted values and is a plot description file and is device independent. It must be reinterpreted for the plotting device to be used. At Durham this is accomplished by the public programs *PLOTSEE and *MTSPLOT. (See MTS Volume 2, Public File Descriptions.)

Channel 10 is the oscillation output channel. The file used here will contain the oscillation output requested. The information may be reinterpreted by the program SOP, simulated output plots, originally used in the study of oscillations of traditional Distinct Element Analysis contacts. (See program comments in file est8:p.sop.s for use, and sections 3.3.2.3 and 3.3.4.6.)

Finally, Channel 11 is the error communication channel. This channel is used during error handling, for input of corrected commands or for a termination message if a non–recoverable error occurs, such as an unexpected end–of–file condition.

3.3.1.3 Outline of facilities

The program is designed to be flexible in the tasks it performs and, on the whole, is not preprogrammed to solve the problem in a set fashion. The user is

in control of what is to be done, and when. Research use of the program further emphasises the need for such flexibility and so several user orientated features have been incorporated.

A principal feature is the comprehensive input command language to define problems and the manner of solution. In addition a running commentary is produced providing information on the current program status, (for example the number of pages of plot produced so far, iterations completed and requested), current problem stability information, and the current command in progress. This allows a user to abort the run if it is not satisfactory.

Certain parameters may require changing during the problem lifetime and this may be achieved by the using Procedure *calculator* allowing, for example, intervals between plots to be multiplied by a value. This same facility allows a restart file to be examined and the parameters inspected or changed.

Experience of some previous Distinct Element Analysis implementations led to the realisation that input error checking and handling is very important. A crash caused by a mistyped command part way through an expensive run is particularly annoying. Error checking is included in SLICES and on encountering an invalid command the user is prompted for a replacement. It is not always possible to retrieve the situation, or it may be laborious to do so, but the opportunity is there.

Often in investigating the progressive nature of the solution it is necessary to do the same things repeatedly. There are two repetition structures. One is a simple repeat loop, which repeats all of the commands enclosed, as many times as instructed. The other causes a command list to be repeated after every interval of so many cycles.

Another user orientated feature, which is included as standard in development programs is that of a debug facility. With this the user may request a whole variety of information to locate bugs or to supplement the normal solution.

Finally a process orientated feature has been incorporated. The maximum individual displacement of all the slices in each cycle is monitored. The behaviour of the maximum displacement with time shows characteristic patterns under certain conditions. This value is written to the running commentary and by internally monitoring its change a final verdict on the stability may be made by the program. Under conditions of constant sliding and stable equilibrium this displacement becomes constant and the program terminates. An upper limit on the number of cycles to be executed can be issued with the realisation that termination should occur early without any waste of resources. This feature also checks for numerical instability and will automatically terminate the program before it crashes.

3.3.2 Input Command Language

3.3.2.1 Introduction

All program tasks are controlled or defined by the Input Command Language As shall be explained later the program requires some commands in a particular order, but on the whole the majority of commands may be used at any time. Although the program is not designed to run interactively, it is possible with care. Normally, however, the commands should be contained in a file prior to use.

The commands may be categorised into broad sections, dealing with program control, plotting, meshing, debugging, and the setting of options and parameters, these corresponding to the major procedures of the program. The commands are hierarchical, forming a tree system. The highest level is the control level which allows access to the lower levels of commands, such as the plotting and debugging

command sets, which are at the second level. Furthermore, the plot command set, for example, contains a level three command set, the mapping commands.

Once in a low level set the permissable commands are those of the present set and of any higher level set which contains the current set. The highest level or outermost command set, the control commands, are available immediately on entry to the program. Access to lower sets must always be made though this control level. Once in a lower command set as many commands of that set may be issued as required. To exit from a lower level, a command of a higher level set containing the present set should be issued, often this will be a control command.

To complete the picture it should be noted that on correction of an input error, it is as if the program is being re–entered, so that the only applicable commands are control commands, that is those of the outermost command set.

The following sections, 3.3.2.2 to 3.3.2.8 describe the functions of the commands of each command set. Section 3.3.2.9 describes a syntax table for the Input Command Language. How best to use combinations of the commands is not discussed here, but rather in Section 3.3.3, under the heading 'Input Command File'.

3.3.2.2 Control commands

The control commands are situated in the outermost command set, all other commands are accessed through this set. The commands are **set**, **restart**, **save**, **start**, **stop**, **debug**, **plot**, **go**, **repeat**, **rend**, **cend** and **return**. Each of these is now described in detail.

The **set** command enters the parameter procedure to allow parameters to be set up, altered or inspected.

The command **restart** causes a restart of a previous problem run. A file containing the restart information must be attached to unit 1. Within the command file the mapping information must follow.

save causes a restart file to be written. It may either overwrite or append the file attached to channel 2 according to the setting of the **overwrite** command (a set command). This is used to save the solution to the task so far found for a large job, thus avoiding loss in the case of a system crash.

The **start** command starts a new problem. A title up to 80 characters long may follow, but the next line must contain the mapping information and then mesh information is required. Section 3.3.2.8 describes the meshing commands.

stop should be the final command in the input command stream as it causes the geometry to be plotted, a restart file to be written and the program run terminated.

The command **debug** causes the debug procedure to be entered, so that debug options can be set or general information generated.

Command **plot** causes the plot procedure to be entered, which allows requests for the manipulation of the plot format, size, and the production of the different plot types available.

go causes the calculation cycle to be entered and it must be followed by an integer, the number of cycles to be executed.

The command **repeat** is the opening statement of the **repeat n** commands **rend** loop structure. It must be followed by an integer, which is the number of

times the loop is to be executed. There are certain commands for which inclusion in this structure would be pointless. These are explained in section 3.3.3.

To balance the repeat loop structure the command **rend** is used in two ways. As regards to input, it terminates input to the repeat controlling procedure and is the last statement in the repeat loop, in this case it is not a control level command. The second way in which it is used is internally, during execution of the loop, here it signifies the end of the loop so that the commands may be repeated again.

cend, like **rend**, is used in two ways. Firstly, it terminates input to the command list structure of the set command set, and secondly it terminates execution of the command list during use. Section 3.3.2.4 describes the **set cmdlist** commands **cend** facility in detail.

Finally, the **return** command terminates interactive input during input error handling, and is described together with this facility in section 3.3.3.3.

3.3.2.3 The debug command set

To gain access to these second level commands the **debug** command must be entered at the control set level. This facility falls into two parts, one outputs information at the point of issue of the command, while the other assigns options which provide data during the subsequent execution of the program. If used carelessly, this latter part may produce a very large amount of information, so it is intended that these options be switched on and off as required. All output from this routine is written to the file attached to unit 7 unless otherwise stated. There are eleven debug commands, each of which are now described.

The command **contacts** writes out the contact information. **general** produces some general problem and program information. The **flagson** command

sets all the debug options on, and should be used with care. Command **flagsoff** turns all of the debug options off.

All of the following commands must be followed by the third level commands of either **on** or **off**, which clearly sets the option on or off.

The command **update** produces contact information as the contacts are created. As creation of the contacts occurs during meshing, which is after the issue of the **start** command, but before the issue of the next control level command, this must be issued before **start**, otherwise it will do nothing.

Command **motion** controls the production of debug output from the procedure *consolsl* (the motion law) during execution of the calculation cycle. The **ford** command controls the production of the debug output from the procedure *ford* (the force displacement law) during execution of the calculation cycle. The command **consolidate** produces limited information from both *consolsl* and *ford*, again during execution of the calculation cycle. **cycle** produces information from all procedures within the calculation cycle and procedure *cycle* itself. The command **trace** causes a message to be written on entering and exiting all procedures and functions. Output is written on the file attached to the unit 8. Lastly the command **oscillate** causes information from *ford* and *consolsl*, formatted for input to the Program SOP, to be written onto the file attached to channel 10.

3.3.2.4 The set command set

To gain access to these second level commands the command **set** must be issued at the control level. This set of commands falls into two groups, problem parameters such as **gravity** and options such as **framelimit**. The set commands are as follows.

The command **echo**, if set to on this parameter enables all input commands to be echoed on the running commentary. The command must be followed by the third level commands of either **on** or **off**. The default is on.

The **overwrite** command controls the restart file output. If set, the file attached to unit 2 is emptied prior to use, otherwise the file is appended by the restart information. The default is off.

cmdlist sets up a subsidiary file and copies all command input to it until the command **cend** is entered. The execution of this secondary command file is controlled by two further set commands, **cmdproc** and **interval**. Transfer of control is passed from the file attached to the unit scards to the secondary file (always named internally as the temporary file -sass.cmd), during the execution of procedure *cycle*. The default value is null.

The command **interval** must be followed by an integer, the number of cycles to be executed between successive executions of the command list secondary command file. The default value is 100.

The **cmdproc** command must be followed by either of the commands **on** or **off**. If it is set to on, the command list secondary file is executed whenever the total cycles executed so far divided by the interval, (as set by the command **interval**), is an integer value. If set to off this facility is not used. The default value is off.

framelimit should be followed by an integer. The GHOST library limits the number of frames of plot output to twenty. If this is exceeded the program will terminate. This command allows this limit to be reset. The default is 20.

The command **writegap** sets the interval of cycles between display of some of the running commentary information. The default is 100.

The **gravity** command is followed by a real number, which represents the value of gravity in the positive y direction. The default value is 0. **damp** is followed by two real values, this sets the global damping values for base and side contacts respectively. Ideally they should be between 0.001 and 0.2 per unit mass. There is no default value other than the initialisation of zero. **time** is followed by a real value this sets the time step size, it is recommended that a value of unity is used. The default value is 1.

The **calculate** command allows the values of some parameters and options to be modified or inspected rather than simply reset. Calculator commands are described in the following section, 3.3.2.5, and are level three commands.

3.3.2.5 The calculator command set

This set is at the third level and is accessed by the command string set **calculate**. Almost all the calculator commands have the same format, that of ≺parameter≻ ≺operator≻ ≺real≻ with the exception of when the enquiry **?** is used, when a value is not required. Permissible parameters, which are in fact third level commands, are **interval**, **writegap**, **gravity**, **time** and **damp**. The operators, fourth level commands, to be precise, are = replace, * multiply, / divide, + add, - subtract, ^ exponentiation. The **?** enquiry although not an operator is used here.

The values are read in assuming a real number format. For parameters which are integer in nature, conversion takes place to give an integer result. The final value of a calculation command is written to the running commentary output stream.

3.3.2.6 The plot command set

To gain access to this second level set the command **plot** must be issued. As all the GHOST library routines are contained in the procedure *plot* to ease maintenance, and many plotting functions are automatically carried out by the program, it has been necessary for some of these and map commands to be issued internally. Although these internal commands are described, it may be that they will never need to be issued externally. They are **initialise**, **endplot**, and most map commands with the exception of **zoom**.

initialise sets the initial plotting parameters and turns the plot output stream on. This command is issued automatically on receipt of the **start** or **restart** control commands and should not need to be used normally. The **slices** command draws the slices in the current plot space. The **displacement** command draws the current slice incremental displacement vectors. **forces** draws the normal and shear stress, limiting shear stress profiles for the base contacts. The **standard** command produces a standard plot of a border and profiles for the base and side contacts. **page** calls for a new frame, or in physical terms a new sheet of paper. **border** produces a border with the problem title and current problem time. The **map** command enters the tertiary level map set and enables the modification of plot formats, it is used internally for the most part.

The unusual command **zoom** must be followed by three real numbers, *xmin*, *xmax*, and *ymin* which form the mapping limits. *xmin* is the minimum value of x, *xmax* is the maximum value of x, and *ymin* is the minimum value of y of the problem geometry to be plotted. As the plots are in fixed proportions in both landscape and portrait mode it is not necessary for the maximum y value to be provided. This command enables portions of the problem to be examined in more detail. Mapping limits are expected as part of the input after both the **start** and **restart** control commands. **zoom** is in fact a tertiary level command valid

at the secondary level as it is passed straight to the mapping procedure without processing.

The **endplot** command is issued internally during closedown of the program under normal termination, it produces a frame with a slice plot and turns the plot output stream off.

3.3.2.7 The map command set

These tertiary level commands are accessed by first issuing the command string **plot map**. They are 10 commands in this set and are described as follows.

The command **bottom** sets the plotting space to the lowest quarter of the physical page. It is used internally for the production of the normal stress profiles. A border is drawn together with axes scaled to the mapping limits (x) and normal stress limits (y).

To set the plotting space to the second lowest quarter of the physical page the command **lowermiddle** is used. It is used internally for the production of the shear stress, limiting shear stress profiles. A border is drawn together with axes scaled to the mapping limits (x) and limiting shear stress limits (y).

uppermiddle is used to set the plotting space to the second highest quarter of the physical page. It is used internally for the production of the normal stress profiles. A border is drawn together with axes scaled to the mapping limits (x) and normal stress limits (y).

To complete this suite, the command **top** is used to set the plotting space to the topmost quarter of the physical page. Again it is used internally for the production of the shear stress, limiting shear stress profiles. A border is drawn

together with axes scaled to the mapping limits (x) and limiting shear stress limits (y).

picture sets the plotting space to the upper half of the physical page. It is used internally for the production of a slice plot, normally above stress profiles. A border around the space is drawn together with axes scaled to the current mapping limits as set by the **zoom**, **start** or **restart** commands.

The command **horizontal** sets the page format to lie along the A4 sheet of paper as in a landscape picture. The default size is (0.06,0.96,0.05,0.65) expressed in a *(xmin,xmax,ymin,ymax)* format.

The **vertical** command sets the page format to lie down the A4 sheet as in conventional portrait picture. This is the default format, the default size is (0.15,0.75,0.06,0.96).

fill sets the plotting space to the maximum permitted page size suitable for A4 paper. A border is drawn around this area together with axes scaled to the current mapping limits. A variation to **fill**, **fullnoscales** does the same as the **full** command but does not draw scaled axes.

The last mapping command **zoom** has been described in the previous section.

3.3.2.8 The mesh command set

This is the only set of commands that cannot be accessed at random by a user. It is automatically entered after the issue of the level one command **start**, a further oddity is that this set can only be exited by issuing the **meshend** command.

There are two mesh commands which lie at level two, **create** and **meshend**, they are described below.

The command **meshend** causes the meshing routines to terminate. The contacts are found and initial plots are produced before the next command in the input stream is executed.

The **create** command triggers the creation of a new slice, the information for which must follow, 14 pieces are required, the first of which is strictly a tertiary level command describing the type. There are two tertiary commands.

These two tertiary level commands associated with **create** are **free** and **track**. The **free** command is the normal slice type and is used almost exclusively, **track**, on the otherhand, in conjunction with the debug oscillation option, permits dumps of the slice information to be made during processing.

The remaining information required by **create** is both geometric and geotechnical. Nine pieces of geotechnical information are required to describe the geotechnical state of the slices. These are as follows, base cohesion, base ϕ, dry density, numerical stiffness, side cohesion, side ϕ, pore water pressures at the middle of the base and side contacts and the void ratio. The geometric information needed is the x and y coordinates of the points defining the top and then the bottom of the righthand edge of the slice. In the case of the first created slice the coordinates of the lefthand edge are given first, followed by the data for the righthand edge. It must be remembered that slices are created from left to right.

3.3.2.9 Syntax table

The following description of the Input Command Language is based upon the symbols as defined in Table 3.2 with the syntax in Table 3.3.

Symbol	Definition
...	indicates possible repetition of the clause
[ ]	indicates an optional clause
( )	indicates a group of clauses
$\prec$ $\succ$	indicates substitution by a value, which may be either a clause or literal
‘ ’	indicates a literal value
$\vert$	indicates an alternative
IS	is the definition operator

Table 3.2 Input Command Language Parsing Symbols

3.3.3 Input command file

3.3.3.1 File format

The input command file contains the task to be performed by the program, defined by the input command language and syntax described in section 3.3.2. There are very few format conditions, and some of them are imposed by PASCAL.

All commands must be separated by at least one space. The maximum word length is 12, so no string of non–blank characters should exceed this. Real numbers may be as 1 1.0 -1.0 -1 1E10 -1E-10 and must be separated by a blank or the negation.

End of line conditions are automatically skipped by the input routines and so there is only one time when a new line must be started. This occurs after the **start** command when the remainder of the line is read as a title. Further information must begin on a new line. Word length may be exceeded in the title.

Text, including numbers may be commented out by the { and }, a blank must precede the open brace. An unbalanced open brace will cause an end of file error

```
task IS [≺com≻ ...] 'stop'
correction IS [≺com≻ ...]('return' |'stop')
limits IS ≺real≻ ≺real≻ ≺real≻
reply IS 'on' | 'off'
com IS ('set' [≺set command≻...])
    | ('restart' ≺limits≻)
    | ('start' ≺start block≻)
    | ('plot' [≺plot command≻...])
    | ('debug' [≺debug command≻...])
    | ('repeat' ≺integer≻ [≺com≻...] 'rend')
    | ('go' ≺integer≻) | 'save', 'cend' | 'rend'
parameter IS 'framelimit' | 'writegap' | 'interval' | 'gravity'
    | 'damp' | 'time'
oper IS '*' | '+' | '−' | '/' | '∧' | '='
set command IS (('echo' | 'cmdproc' | 'overwrite') ≺reply≻)
    | (('framelimit' | 'writegap' | 'interval') ≺integer≻ )
    | (('gravity' | 'damp' | 'time') ≺real≻ )
    | ('calculate' [≺parameter≻ ((≺oper≻ ≺real≻) | '?')])
    | ('cmdlist' [≺com≻ ...] 'cend')
plot command IS ('initialise' ≺limits≻)
    | 'slices' | 'displacement' | 'forces' | 'standard' | 'page'
    | 'border' | ('map' ≺map command≻) | 'endplot' | ('zoom' ≺limits≻)
map command IS 'picture' | 'horizontal' | 'vertical'
    | 'full' | 'fullnoscales' | ('zoom' ≺limits≻)
debug command IS 'contacts' | 'energy' | 'general' | 'flagson'| 'flagsoff'
    |(('update' | 'motion' | 'consolidate' | 'ford' | 'cycle'
    | 'trace' | 'oscillate') ≺reply≻)
type IS  'free' | 'track'
geom IS ≺real≻ ≺real≻ ≺real≻ ≺real≻
geotechnical IS ≺real≻ ≺real≻ ≺real≻
     ≺real≻ ≺real≻ ≺real≻
     ≺real≻ ≺real≻ ≺real≻
meshinfo IS ('create' ≺type≻ ≺geotechnical≻ ≺geom≻ ≺geom≻ )
    | ['create' ≺type≻ ≺geotechnical≻ ≺geom≻ ] ...
start block IS ≺heading≻ ≺limits≻ [≺meshinfo≻] 'meshend'
```

Table 3.3 Input Command Language Parsing Definition

termination. It should be noted that a comment does not act as a word delimiter, only a blank or an end of line fulfils this function.

During error handling the same format rules apply, comments may be entered but there is little point. Apart from these points the input format is left to the user, but it is recommended that the file can be read and understood by the user. To illustrate the commands some examples are given.

plot zoom 0 14 0 slices displacement border page

This causes the plotting space to map to new limits, produces a slice plot with incremental displacements and border and finally requests a new page.

set calculate writegap * 2 calculate writegap ? go 3000

This example shows how to multiply the present value of **writegap** by two, display the new value and then request 3000 calculation cycles. The second **calculate** is not strictly necessary, but may be used for clarity.

3.3.3.2 Defining tasks

Tasks fall into two categories, starting a new problem and restarting an old one. Both types of task may be divided into three, initialisation, solution and closedown. Initialisation for the two categories is different.

Starting a new problem calls for input of a title, plot limits, meshing information and problem parameters. In restarting, plot limits only need be supplied, as all the other initialisation took place in the first run, the command **restart** followed by the limits should be adequate.

After creating the slices and finding the contacts the start up procedure sets the plot format to the default of vertical and then produces a slice plot. If a horizontal format is required for this first page then **plot format horizontal** should be issued prior to the **start**. Another command to be issued at this point is **debug update on**, otherwise it will do nothing in the current run.

Meshing information for a new problem has been discussed in section 3.3.2.8. The optimum number of slices is between 10 and 25. Too many, and the overhead per cycle increases as does the number of cycles required for a steady state to be attained. Too few and the resolution is poor.

Solution types for start and restart task categories are similar, in the restart case the solution type may already be mostly set up, but can be altered. To monitor the progressive nature of the solution, plots, factors of safety or debug information may be required at various times.

The interval at which factors of safety are produced is controlled by the **writegap** parameter. This also controls when the total cycles and maximum displacement values are updated on the running commentary. This information is generated whenever the total cycles executed is an integer multiple of the **writegap** parameter. **writegap** has a default value of 100 cycles, so factors of safety are produced every 100 cycles.

By using **set cmdlist plot standard cend interval 100 cmdproc on** a standard plot of stress profiles is produced every 100 cycles. The interval parameter operates in the same way as **writegap**.

Having decided upon this solution type all that is necessary to consider is the upper limit to the number of cycles to be executed. This should be between 2000 and 5000 for typical problems. On issuing **go 1000**, up to 1000 cycles will be executed, 10 standard plots and 10 sets of factors of safety produced. An equivalent to this command list structure would be to use this repeat structure, **repeat 10 go 100 plot standard rend**.

The consolidation process converges to constant displacements for all of the slices. In the case of constant movement, that is when stability of the slope is

not attained, experience has shown the displacements to be 10^{-1} to 10^{-4} times $g\delta t^2$. For stable systems the values are about ten orders of magnitude smaller. In both cases the early cycles, give the largest contributions, while the later cycles make small differences. In the light of this it would be better to generate more information in the early stages and less later on. A series of **plot**, **set**, and **go** commands could program this but it is more elegant to use the calculator to change the values of the intervals. For example to produce plots and factors of safety at the powers of 2 cycles this could be used.

set interval 1 writegap 1 cmdproc on

cmdlist plot standard set calculate interval * 2 writegap * 2 cend

As a final note to the command list structure, it is possible to include a **go** command. This is particularly useful for the production of debug information during cycling. Much information can be produced, but normally it is only needed for a few cycles. A command list string of **debug ford on go 1 debug ford off** with an interval of 100 would produce force displacement information for one cycle in every hundred. If instead of 1, 100 was used, then the command list would not execute beyond the **go** before executing again. As this facility is programmed recursively, such a combination could eventually lead to a program crash and should not be used. The program structure of this facility is explained in section 3.4.2.3 under Recursion Structures.

Finally, to complete the command file, program termination must be considered. The program monitors the maximum displacements and terminates under constant conditions. If these conditions do no prevail, then termination is accomplished by the **stop** command which should always close the task definition. If for any reason it is required to halt the program prematurely, the use of the 'break' key causes an attention interrupt. This is trapped by the program and the user is then asked to confirm his wish to stop. To confirm, enter 'y'. Attention trapping is checked at the end of each calculation cycle and also during the input of a new

command. Termination involves the automatic production of stress profile plots, factors of safety, a restart file and job statistics on the running commentary.

3.3.3.3 Input error handling

Inevitably, occasional mistakes are made during production of a command file. If these are due to commands being mis-spelled, or even missing, then an error handling facility provides an opportunity for correction.

On encountering a command error, the user is informed via the running commentary and is prompted for new commands. Regardless of the level of the command in error, the replacement must be a control command. Once the replacement has been executed, the user is again prompted, and the next replacement read. When no further commands need to be entered, the user should reply to the prompt with the **return** command. This returns control to the command file at the point immediately after the original error. An immediate reply of **return** to an error causes the command to be ignored.

If a further error occurs during the input of replacement commands, correction of it is possible in the same manner as if it had occurred from within the command file. If the correction process becomes laborious or impossible the **stop** command will cause program termination immediately.

Any numerical input required is prompted for by individual messages to the user but has no correction facility. Any numbers following a command in error are treated as commands on return to the command file. They should be ignored by using **return**.

Not all mistakes in the command file need be accidental, a deliberate wildcard may be included at any stage to give control to the user. This may range from

complete interactive use of the program to interaction occurring at the end of a repeat structure. Tables 3.4 and 3.5 shows some examples of error correction and interactive use.

```
→        PROGRAM SLICES RUNNING COMMENTARY ON :
→
→        Command : plot
→        Command : sliceplot
→
→        Error 'sliceplot' found in routine get_command
→        Input corrected commands ... ≺RETURN≻ ...
→             Input a command please ............
←        plot
→        Command : plot
→             Input a command please ............
←        slices
→        Command : slices
→             Input a command please ............
←        return
→        Command : return
---------------------------------------------------
Lines marked → are output from the program
Lines marked ← input from the keyboard
```

Table 3.4 An Example of Error Correction

If an error occurs during the processing of either of the loop structures there are two possible options. Either to correct the error each time it occurs or to replace the whole structure. During complete replacement it should be borne in mind that the **rend** and **cend** commands have two functions. To replace a command list the following should be issued, **set cmdlist** commands **cend**. The **cend** terminates the input to the structure. If replacement is taking place during the execution of the previous command list, it is now necessary to terminate this invocation by issuing a second **cend**.

```
→      Command : ????
→      Error '????' found in routine get_command
→      Input corrected commands ... <RETURN> ...
→            Input a command please ............
←      go
→      Command : go
→      Enter no of cycles required...
←      1
→            Input a command please ............
→      plot
→      Command : plot
→            Input a command please ............
←      forces
→      Command : forces
→            Input a command please ............
←      go
→      Command : go
→      Enter no of cycles required...
←      1
→            Input a command please ............
←      plot
→      Command : plot
→            Input a command please ............
←      forces
→      Command : forces
→            Input a command please ............
←      slices
→      Command : slices
→            Input a command please ............
←      stop
→      Command : stop
→            total slices     10 contacts       20
→            total cycles      2 restarts        0
→            total frames      5 plots           7
→      Number slices at limit     0 not at limit   10
→      A restart file has been written
-----------------------------------------------------
Lines marked → are output from the program
Lines marked ← input from the keyboard
```

Table 3.5 An example of interactive input

Likewise during the execution of a repeat loop two **rend** commands are needed. Only one **rend** or **cend** is needed if these structures are being replaced

when they are not being executed.

3.3.4 Utility files

3.3.4.1 Repeat file

The repeat file is a secondary command file. It it emptied on issue of the repeat command, and all subsequent input is copied from the primary command source to this file up to and including the command **rend**.

During execution of the repeat loop control is passed to this file which is reset to the beginning at the start of each pass through the loop. The repeat file has an exceedingly simple structure, containing only one word or number on each line. This file is temporary in nature and is set internally always to be called '-sass.rep'.

3.3.4.2 Command list file

The command list file works on the same basis as the repeat file, it has the same structure and is named internally as '-sass.cmd'. It is a secondary command file containing the command list commands and control is passed to it on execution of the command list facility. It receives all commands from the primary command source on issue of **set cmdlist** up to and including **cend**.

3.3.4.3 Restart file

Unlike all other input and output files the restart file facility uses non–text files. As the file must contain all the numbers required for the program to restart, the numbers must be stored in a way that exactly represents the full accuracy of the computer. The numbers are, therefore, written in a binary format.

Furthermore, PASCAL restricts file definitions to be of a single type, that is they can only contain one type of record. This complicates the issue, as the program variables are of many types — combinations of reals, integers, pointers and strings. As described in section 3.4.1, many of the variables are records of various types. There is a problem then in writing many record types to a file containing only one. To overcome this, the restart file is of type buffer, where buffer is defined as an union of all the other record types defined. A single character, known as the tag field, and part of the buffer, denotes which sort of record is being handled. This enables the buffer record, read in from a restart file, to be interpreted to the correct program record type. As the tag is an ASCII character within the restart file it is easy to see which lines refer to which variables. The various tags are listed in Table 3.6.

Tag	Restart record type
G	the general information
c	a command list word
r	a repeat list word
F	the slice body data
R	the right hand contact data
B	the base contact data
P	the platen data
a	the apex coordinates
*	the end of the restart data

Table 3.6 The restart file line tags

One side effect of the buffer type is that all the records in the file are the same length, so that the smallest variables take just as much room as the longest, which defines the record size.

A further complication in implementing a restart facility in PASCAL is that pointers, which are memory addresses, are no longer valid once read back in. As Program CIRCLES and SLICES use pointers extensively, this is a significant

complication. On writing a restart file the memory structure may be thought of as being dismembered, and on being read in, the severed portions must be linked together with pointers in the same order as before.

3.3.4.4 Trace output file

This file only need be attached to unit 8 if the debug tracing option is to be used. It is emptied prior to use and receives a message on entry and exit to each procedure and function. It can, therefore, become very large if used extensively. The input and output of data is a slow operation and so the use of this facility will slow the rate of problem solution considerably. The primary purpose of this is as a debugging tool, particularly of the recursive structures, as it reveals which levels of recursion have been attained.

The file contains one message per line which take the formats of

```
'Entered procedure xxxxx'
' Exited procedure xxxxx'
```

In the case of procedure *word_scan*, the word read from the command source is appended to the exit message. The facility is accessed by **debug trace on** and is turned off by **debug trace off**.

3.3.4.5 Debug output

The debug utility file is largely unformatted as it contains information produced mostly in response to instructions from the user. As a set of safety factors is automatically generated on shut down, this file will contain the title, the closing factor of safety values and the current maximum individual slice displacement, even if no output is requested. The information may be divided into three types,

that produced during cycling by setting the debug flags, that produced immediately on demand, and that produced periodically and controlled by the **writegap** parameter.

Output generated periodically is restricted to the factors of safety. The cycle number starts a banner showing that the factor of safety, shear, normal, limiting and pore–water stress values are produced. However, these values are for the base contacts only. If the value of shear stress is zero then the factor of safety is also. There follows one line for each slice, containing the values as shown in the first entry in the format Table 3.7. The current maximum displacement is produced afterwards.

Output generated during cycling is produced in the iterative solution of the motion and force displacement laws, as well as in the controlling procedure cycles.

Refer to entry 2 in the format table, this produced when the ford flag is set and is generated by the procedure *fordsl.* It provides values for the incremental forces (Fn, Fs), geometry of the contact edge (sin, cos, l), current stresses (ss, ns, lims) and current body forces (nf, sf) during the processing of each contact.

The motion flag produces the information as shown in the third entry which is generated from procedure *motionsl.* The slice body forces in x and y as well as the displacements are produced.

The cycle flag produces the information shown in entries 2, 3 and 4. Entry 4 is generated from the procedure *cycle.*

The following output is generated on demand by the debug commands. The **update** command causes the mass and surface area values to be generated for each slice. The format for this is shown in entry 5 of the format table.

Entry	Format
1	999999 Slice no FOS shear normal limit pwp 9 9.999ES99 9.999ES99 (occurs 3 more times)
2	Fn,Fs,sin,cos,l 9.99ES99 9.99ES99 (occurs 7 more times) ss,ns,lims,nf,sf 9.9ES99 9.9ES99 9.9ES99 9.9ES99 9.9ES99
3	bforces 9.99ES99 9.99ES99 disp 9.99ES99 9.99ES99
4	max individual disp 9.9999999999999ES99
5	mass,surf 9.9999999999999ES99 9.9999999999999ES99
6	BASE Contact created edge, corn 999999 999999 edge x,y 9.9ES99 9.9ES99 corn x,y 9.9ES99 9.9ES99 sin, cos 9.9ES99 9.9ES99 len, dam 9.9ES99 9.9ES99 pwp, wt 9.9ES99 9.999ES99
7	total number of contacts 9999999999
8	Element data : mass force x y disp x y n 9.9ES99 9.9ES99 9.9ES99 9.9ES99 9.9ES99 9 9.9ES99 9.9ES99 9.9ES99 9.9ES99 9.9ES99 9
9	Contact information : slice home, other, damp 999999 999999 9.9ES99 corner coordinates x, y 9.9ES99 9.9ES99 edge coordinates x, y 9.9ES99 9.9ES99 stresses - n, s, l, u 9.9ES99 9.9ES99 9.9ES99 9.9ES99
10	SLOPE 9, C mapping xmin 9.9ES99 xmax 9.9ES99 mapping ymin 9.9ES99 ymax 9.9ES99 plot interval 999999 gravity x 9.9ES99 y 9.9ES99 damping base 9.9ES99 side 9.9ES99 totals slices 999999 contact 999999 cycles 999999 restarts 999999 frames 999999 plots 999999

Table 3.7 The debug format table

Entry 6 shows the contact information which is generated as each contact is

made. At the end of the meshing process the total number of contacts is written out as shown in the seventh entry.

Slice information is formatted as shown in the eighth entry, it includes the slice mass, body forces and incremental displacements as well as the slice number. There is one line of information for each slice.

The format of the contact informatiom is shown in entry 9 of the format table. There are four lines of data for each contact. The first contains the slice numbers for the two slices involved, the home slice contains the base of the contact linked list in which the contact is to be found, and also the damping factor used in the calculation sequence. The second and third lines contain the coordinates for the corner and edge involved. The last line contains the stress data, normal, shear, limiting and pore water stresses.

The general information is generated in accordance with the format shown in the tenth entry of the table. The general information shows the mapping limits, current plot interval set by the **interval** parameter, the values of the gravity and damping, and the numbers of slices, contacts, cycles completed, restarts of the task, plot frames and plot types generated.

3.3.4.6 Oscillation output

This file need only be attached to the unit 10 if the oscillation facility is to be used. This allows for information of track type blocks to be investigated. The file is a text file and may be visually inspected, in addition the format is compatible with program SOP which can produce graphs of the values.

The file is emptied prior to use and contains the problem heading on the first line followed by one line of data for each cycle during which the facility was in use.

This facility is useful for monitoring the progress of critical slices, damping effects and contact behaviour. The data items produced are shown below.

1 slice number
2 total cycles
3 body displacement x
4 body displacement y
5 base shear stress
6 base normal stress
7 base limiting stress
8 side shear stress
9 side normal stress
10 side limiting stress

3.3.4.7 The running commentary

The running commentary writes various information concerning the current task status to the device attached to unit sprint. As control codes are written to this device, it should be a Televideo 910 series terminal. On entering program SLICES the terminal screen is cleared. Status information is then written to appropriate lines, and in this fashion the screen is continually updated. The screen line positions are reserved for the data as shown in Table 3.8. The specific messages that can occur on lines 12 to 20 are shown in Table 3.9.

To clear the screen and turn the cursor on or off requires three separate control codes. Screen positioning is achieved by moving the cursor to the home position, at the top left hand corner, and then down the appropriate number of lines. The total number of codes used is five. They are shown below where the first two

Line	Content
1	Blank
2	The program running commentary heading
3	Blank
4	The task title
5	Blank
6	The number of cycles requested
7	The number of plot frames completed
8	The number of plot types completed
9	The number of cycles completed
10	Blank
11	The command under execution
12	General messages
13	Slices at limit messages
14	Error messages
15	Message requesting replacement commands
16	Prompts to user for command or parameter data
17	The totals for cycles, frames and so on
18	As line 17
19	As line 17
20	Messages dealing with the restart files

Table 3.8 The running commentary screen lines

characters are shown in hexadecimal format.

control code 1A	clears screen
control code 1B .0	cursor off
control code 1E	cursor home
control code 0A	cursor down
control code 1B .1	cursor on

The program string constants close to the beginning of the source contain these codes. *clearoff* is 1A1B.0, *pos_str* is 1E with twenty 0A and *curson* is 1B.1. The cursor is moved several lines at once by using a substring of *pos_str*.

On MTS the network, NUNET traps all control codes and echoes them on to a terminal with a check character. To enable the codes to be executed the network

Line	Message
4	PROGRAM SLICES RUNNING COMMENTARY ON :
12	Decreasing stability 9.99999999999E99
12	Increasing stability 9.99999999999E99
12	Stability has been gained 9.99999999999E99
12	Constant sliding now occurring 9.9999999999999E99
12	This is numerically unstable 9.9999999999999E99
12	A restart file has been read
12	The value is : 99999.9999999
12	Frame limit is now : 99999999
12	Cycle gap is now : 99999999
12	Gravity is now : 999999
12	Time increment is : 9.9999999999999E99
12	Damping factor is : 9.9999999999999E99
12	Process interval is: 99999999
13	Number slices at limit 9999 not at limit 9999
14	Attn! : Do you want to stop ?
14	
14	Error XXXXXXXX found in routine get_command
15	Input corrected commands ... <RETURN> ...
16	Input a command please ....
16	Enter xmin, xmax, and ymin ...
16	Enter no of cycles required...
16	Enter heading ....
16	Enter value ....
16	Enter frame limit ....
16	Enter gap between writing.....
16	Enter gravity values x, y ....
16	Enter time step increment ....
16	Enter value for damping ......
16	Enter cmd process interval ...
17	total slices 999999 contacts 999999
18	total cycles 999999 restarts 999999
19	total frames 999999 plots 999999
20	A restart file has been written

Table 3.9 Running commentary messages

must be configured to pass them to the terminal. To do this the network commands chc=off and passall=on must be issued. After use the network commands chc=` and passall=off should be used to reset this.

3.4 Structure of Program SLICES

3.4.1 Memory structure

The memory requirement of the program varies according to the number of slices used in the problem. As memory is dynamically allocated it is possible to use only as much memory as necessary. The number of global variables is quite few, but some of them, such as *slice_list* and *platen* are pointers leading into potentially large data structures.

On entry to the program all variables are initialised to zero, default, and for pointers, nil values. The data structure is built in the procedures *mesh*, *cre_platen* and *update_area*. Procedure *mesh* creates the slices, *cre_platen* creates the platen and *update_area* creates the contacts.

A slice is defined as a record of type *element*. This type is a combination of smaller records and pointers. *force* and *s* (displacement) are records of type *vector*, containing values for the x and y directions. Another record is *data* which contains cohesion, friction, mass and pore water information. The remaining memory of type *element* is made up of three types, a record of two pointers for the contacts, a pointer, *apexes*, to the corner coordinates of the slice, and finally *next*, a pointer to the slice to the right of the current one. The value for the rightmost slice is 'nil'.

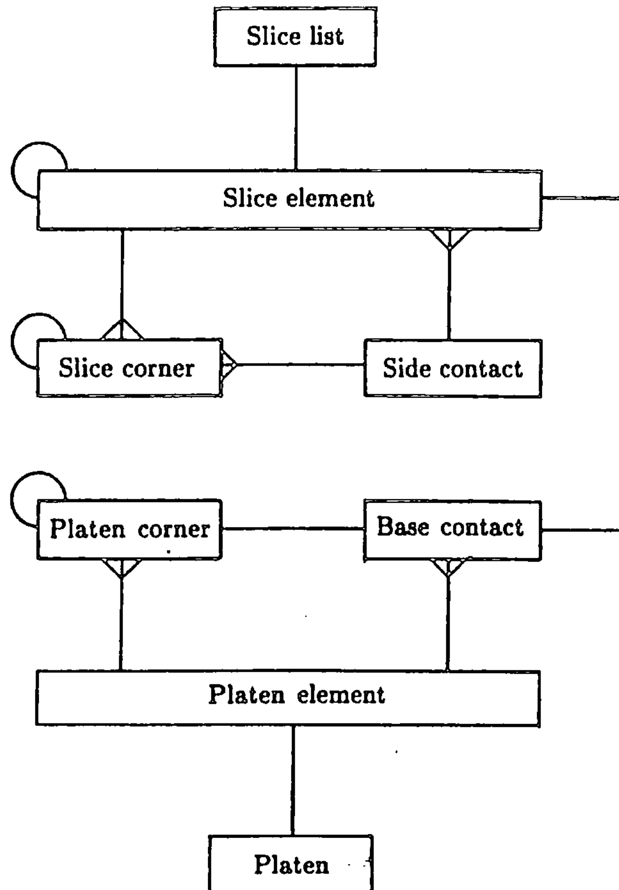
Variable *slice_list* points to the first (or lefthand) slice. The variable *next* of this slice points to the following one, and so on. The *element* type records are linked by the *next* pointers to form a list of elements, with the base pointed to by *slice_list*. This list is a FIFO (first in first out) list as the first created element is closest to this base pointer.

The corner coordinates are contained in a doubly linked ring of type *corner* records. Each of these records includes a coordinate type record of two reals, for the x and y values, and two corner pointers. These point to the adjacent corners in the clockwise and anticlockwise directions, thus it is possible to traverse the ring in either direction continuously. The *element* record field *apexes* points to the bottom left corner of the slice.

Once the slice list has been formed, complete with corner rings, the platen is created. To simplify contact processing, *platen* is an *element* pointer type and points to an *element* which has values set to nil or zero, except for *apexes*. *apexes* points to a doubly linked ring of *corner* records where the corners are copies of the base corners of the slices.

Each slice has contact pointers for the right and base contact information. The contact information is contained in a record type and consists of six real numbers and three further data pointers. Of the six numbers, three are grouped to form the consolidation force information and are segregated into a record type. Of the pointers, one, *other*, points to the other slice involved, while the others point to the corner, and to the first corner of the edge, which form the contact.

Figure 3.5 shows a Bachmann diagram of the complete data structure. It should be noted that the method of storing corners and contacts is extremely flexible and is suitable for a general Rigid Block Model style implementation. The Bachmann method of representing data relationships here is more commonly used in data–base design. The links between the elements indicate that the elements are related, in this case linked by pointers. Crows feet on the end of the links indicate that many elements are related to the element at the other end of the link. It should be noted that these diagrams are logical representations of the data. The side contact to slice relationship is a one to many relationship. This is



The crows feet indicate the relationship is many to one

Figure 3.5 Bachmann diagram of SLICES memory items

because such a contact is related to two slices. The recursive link which joins an element to itself represents a linked list.

There are many other variables used in the program but they have restricted scope. Often these are loop counters, temporary storage and pointers. Some of the more common are *el*, *ele*, *elem* as pointers to slices, *apex* a corner pointer and *condir* a contact pointer. The program structure is often imprinted by the data structures. This influence is discussed later in section 3.4.2.4.

3.4.2 Program structure

3.4.2.1 Procedural elements

The structure of the program is necessarily large but may be broken down into smaller, similar units. The program itself defines the details of the structure, so rather than merely represent this by complex diagrams, aspects of the structure and some common structures used shall be discussed.

Many different sorts of tasks can be performed by the program, and to an extent the input command language may be thought of as a language to program the tasks. As SLICES is very flexible, much of the large scale structure is concerned with parsing the input command language. Brief descriptions of the main procedures of the program together with a structural diagram may be found in Appendix C. The source code of Program SLICES may be found in Appendix D.

3.4.2.2 Main relationships

The main body of SLICES is short and as described contains a repeat *control* forever construct. This is the highest level of control in the program. Procedure *control* executes primary level commands and in doing so may call *plots*,

debug_slice, and *parameters*. These three execute secondary level commands and may call procedures *map_space* and *calculator* to get and execute tertiary level commands. At all levels this is achieved by using the procedure *get_command* followed by a case statement. In a similar fashion *start_shut* uses a case construct to execute the four primary level commands that may be passed to it. The execution of these involves branching further into the 'tree' structure, to *mesh*, *write_restart_file* and so on.

By the use of commands, the user causes the tree to be traversed, always by moving from one level, down to the next and then, eventually by retreating back to the starting point, to choose another branch. However, internally, the program may occasionally flit from one branch to another. This occurs particularly from *start_shut* when frequent calls to *plots* are made.

Essentially the structure is that of a tree with multiple branching at nodes, and where higher nodes can only be attained by visiting the node one level lower. Entry to this tree is always made at the primary node, the control level. At most nodes *get_command* is visited to ascertain which branch to traverse next. Table 3.10 shows typical simple behaviour of this structure while the commands **plot map zoom 99 99 99 stop** are executed.

3.4.2.3 Recursion structures

Under normal conditions this tree is traversed such that the primary node is regained by falling back along the traversed branches. Under three conditions this does not happen simply. The conditions are during repeat, command list and error processing.

When repeat loop processing is encountered, procedure *repeater* acts like the main body of SLICES, repeatedly jumping to the primary node until the repeat

Trace of procedures	Comments
Entered procedure CONTROL	The primary node is entered.
Entered procedure GET_COMMAND	plot is retreived from storage.
EXIT procedure GET_COMMAND	
Entered procedure PLOTS	A secondary node is entered.
Entered procedure GET_COMMAND	There is no stored word so
EXIT procedure GET_COMMAND map	map is read.
Entered procedure MAP_SPACE	A tertiary node is entered.
Entered procedure GET_COMMAND	There is no stored word so a
EXIT procedure GET_COMMAND zoom	fourth level command is read.
Entered procedure GET_COMMAND	After zoom processing there is
EXIT procedure GET_COMMAND stop	no stored word. The next command
EXIT procedure MAP_SPACE	is read and stored.
Entered procedure GET_COMMAND	stop is retrieved but is not used
EXIT procedure GET_COMMAND	as it belongs to a lower level.
EXIT procedure PLOTS	As a consequence the primary
EXIT procedure CONTROL	node is returned to, exited and
Entered procedure CONTROL	entered from the main program.
Entered procedure GET_COMMAND	stop is retrieved and is then
EXIT procedure GET_COMMAND	executed.
Entered procedure START_SHUT	The run is brought to a close

Table 3.10 Trace of Program Behaviour During Simple Use

file has been executed. Strictly the jump is made to the primary node of a second identical tree. This process is indirect recursion, as an invocation of procedure *repeater* lies between the two invocations of the procedure *control*. That is, *main* has called *control* has called *repeater* has called *control*. The second call to *control* is made with a different file device unit buffer to that used originally. The file device unit buffer used belongs to the secondary command file -sass.rep so that when *word_scan* (or anywhere else) reads input, it now reads from here. Once the repeat facility ends, *repeater* exits back to the primary node of the 'first' tree. The commands of **repeat 1 debug general go 1 rend** illustrates the program behaviour. An edited trace taken during the execution of these commands is given in Table 3.11

Trace of procedures	Comments
Entered procedure CONTROL	The primary node is entered.
Entered procedure REPEATER	After reading the repeat string
Entered procedure CONTROL	this calls control recursively.
Entered procedure GET_COMMAND	This now reads from the repeat
EXIT procedure GET_COMMAND debug	file the command debug.
Entered procedure DEBUG_SLICE	
Entered procedure GET_COMMAND	
EXIT procedure GET_COMMAND general	The general command is processed.
Entered procedure GET_COMMAND	
EXIT procedure GET_COMMAND go	As go is not a debug command
EXIT procedure DEBUG_SLICE	this routine is left.
EXIT procedure CONTROL	
Entered procedure CONTROL	Repeater calls control again.
Entered procedure GET_COMMAND	go is retrieved from storage.
EXIT procedure GET_COMMAND	
Entered procedure CYCLES	One cycle is executed.
EXIT procedure CYCLES	
EXIT procedure CONTROL	
Entered procedure CONTROL	Repeater calls control again.
Entered procedure GET_COMMAND	The command rend is read which
EXIT procedure GET_COMMAND rend	terminates the repeat loop after
EXIT procedure CONTROL	exit from control.
EXIT procedure REPEATER	Repeat also exits and normal
EXIT procedure CONTROL	processing contiues with
Entered procedure CONTROL	a normal invocation of control.
Entered procedure GET_COMMAND	
EXIT procedure GET_COMMAND stop	The stop command causes the
Entered procedure START_SHUT	execution to complete.

Table 3.11 Program behaviour during Repeat processing

During command list processing exactly the same thing occurs, this time *control* is repeatedly called from *cycle* with the file device unit buffer belonging to the file -sass.cmd.

Error correction is more complex. Consider the following. A node has been reached at any level within the tree. Procedure *get_command* is called and an error is encountered. So, *get_command* is called again (direct recursion), *word_scan* is

executed to obtain a command from the user and, if the replacement is in error then this sequence is repeated until a valid replacement is found. At present, program control is in the third invocation of *get_command*, that is the third level of recursion, a stack of invocations is produced until a valid command is entered. If the command entered is **return** then *get_command* is exited three times and the calling node reached and exited with program control being passed back to the primary node. If the command is not **return** then *get_command* exits once, procedure *control* is called repeatedly and the trees traversed until **return** is input. Procedure *control*, in this case, is called with the screen file device unit buffer pointer. This recursive invocation of *control* calls *get_command* (the fourth entry), a command is gained, *get_command* exited and the tree traversed. If an error were to be encountered from the user at this point exactly the same thing would happen as before, *get_command* would call itself directly until a valid command was gained then *control* would be called again (the third level of recursion for *control*) and the tree traversed normally. In this example an error does occur but **return** is entered immediately. In this case program control falls back to the primary node of the present recursion level, *control* exits to the previous recursion level, *get_command* exits three times in this case, (input now reverts to the primary command file), the tree is traversed back to *control*, *control* exits back to the main body of SLICES which then calls *control* as normal. Table 3.12 shows a trace of this scenario as produced by the commands **plot ????** within a file and **ploterr plot ploterr return** input interactively.

Combinations of the three recursion possibilities may occur. For example a **repeat go 1000 rend** structure causes the execution of *cycles*, which, in turn executes a command list. This command list is found to be in error and the user inputs **plot sliceplot** so that processing may continue. In this combination the repeat invokes *control*, *cycle* invokes *control*, *get_command* invokes *control*, then **plot** and **sliceplot** are executed at this third level of recursion. **return** is then

Trace of procedures	Comments
Entered procedure CONTROL	The primary node is entered.
Entered procedure PLOTS	An error occurs in plotting
Entered procedure GET_COMMAND	when ???? is encountered.
Entered procedure GET_COMMAND	A further error ploterr causes
Entered procedure GET_COMMAND	a second level recursive call.
EXIT procedure GET_COMMAND plot	A correct command of plot gives
Entered procedure CONTROL	a recursive call to control.
Entered procedure GET_COMMAND	
EXIT procedure GET_COMMAND	The command plot is retrieved
Entered procedure PLOTS	and plots entered recursively.
Entered procedure GET_COMMAND	A further error occurs and
Entered procedure GET_COMMAND	return is read.
EXIT procedure GET_COMMAND return	This causes all a return to
EXIT procedure GET_COMMAND	to the primary node.
EXIT procedure PLOTS	The recursive call to control
EXIT procedure CONTROL	was from get_command which is
Entered procedure GET_COMMAND return	returned to. The return command
EXIT procedure GET_COMMAND	is retreived and the three
EXIT procedure GET_COMMAND	invocations of get_command are
EXIT procedure GET_COMMAND	exited.
EXIT procedure PLOTS	The initial call to plot and
EXIT procedure CONTROL	then to control are exited.
Entered procedure CONTROL	Normal execution continues.

Table 3.12 Program behaviour during error processing;

input and program control falls back to *cycle*, ready to carry on processing the command list.

3.4.2.4 Structure that maps structured variables

There are three structures that map the structure of the memory. One, which is used extensively causes the slice list to be traversed. Another, often used in conjunction with the first enables both the base and side contacts to be reached and the third allows the corner rings to be traversed.

To traverse the slice list a separate procedure is written containing a while loop controlled by a pointer such as *el*. This pointer, a parameter to the procedure is seeded by the base or anchor of the list by the calling procedure. The while loop is constructed as follows. *while el ,= NIL do begin ... el := el@.next; end;*. The loop will continue until *el* becomes NIL, which will occur at the end of the list. Most procedures with headers of the form *procedure procname(el : ptr_type);* use this construct to traverse the slice list. These procedures are *factors_of_safety*, *disp_plot*, *slice_plot*, *force_profile*, *fordsl*, *fconsolsl*, *update_area*, *cre_platen*, *write_r_el*, *write_con*, and *write_sli*.

To look at the two contacts of each slice a for loop is used as follows

```
for contdir := righthand to based do begin
  case contdir of
    righthand : condir := el@.contacts.right;
    based : condir := el@.contacts.base;
  end;
  writeln(condir@.consol.ns);
end;
```

The case statement causes the right or base contact pointer to be placed in the variable *condir* (contact direction), a pointer. This may then be used to access the contact information. The righthand contact is processed first and then on the second pass of the for loop the base contact is used. This construct is used in the procedure *fordsl.* In *force_profile* a local function *ptrd_fm* uses the case construct to return the contact pointer.

Corner rings are traversed in the procedures *cre_slices*, *slice_plot*, *write_r_el* and *update_area*. In the first three the corners of a slice are traversed once, by using a repeat until loop as follows

```
apex := el@.apexes;
repeat
  ...
  apex := apex@.cw;
until apex = apexes;
```

In *update_area* the corners of the platen are traversed in forming the slice base contacts. The termination of the slice list traverse is used to terminate the traverse of the platen corners. The corners are inspected once only with each shift caused by *platapex := platapex@.cw;*.

3.5 Validation

3.5.1 Introduction

The aim of the following discussion is to show that Program SLICES is capable of predicting the factor of safety and the mechanism of failure of soil slopes. The validation has not been exhaustive nor is it intended that program SLICES is used as if the results are guaranteed correct. Furthermore this discussion does not include all the program testing carried out to prevent program failure during normal operation. Rather, these discussions are meant to show that this technique is viable when applied to problems in soil mechanics and that the results are comparable with traditional methods.

3.5.2 Validation Methods

To gauge the viability of this method three soil slope geometries were used. Each was tested under total and effective stress conditions. These problems were analysed by program FOS of Garrard (1984) and by SLICES.

Program FOS, (Factor of Safety), provides a slope stability analysis by a traditional method of slices. Factors of safety according to the Janbu, Fellenius and Bishop formulae are produced as well as an average. Slice geometry and soil parameters of cohesion, friction and density are required. In this respect program FOS was modified slightly from the source so that the slice density was input directly for each slice. This alteration was necessary to ensure that exactly the same situations were analysed by both programs. As with all traditional limiting equilibrium methods this program will under estimate the factor of safety for those slopes where some or all of the slices are submerged by the water table.

For comparison purposes the average factor of safety has been taken as the best guide to the stability of the slopes. Program SLICES does not produce an overall factor of safety, so the FOS results are quoted in the unusual manner of the values for cohesion and friction which gave a factor of 1.

In determining the stability with program SLICES an iterative method was adopted. Estimates for cohesion and friction, normally taken from the FOS analyses where used as initial values and SLICES used to determine if the configuration was stable. The parameters were then adjusted to bring the slope configuration closer to limiting equilibrium and SLICES used once more. This was repeated until the configuration was just stable. In practice a binary split method was used to reduce the number of runs, which was normally in the region of eight. It was therefore possible to standardise the results on the runs that indicated that the configuration was just unstable. For SLICE results friction is quoted to the nearest half degree and cohesion to the nearest kN/m^2.

Tests were carried out on the following combinations of parameters for each of the test slopes.

Total stress conditions with variable ϕ.

Total stress conditions with variable cohesion.

Effective stress conditions with variable ϕ.

Effective stress conditions with constant cohesion variable ϕ.

Effective stress conditions non–linear critical cohesion and variable ϕ.

Here the variable parameter is the one operated on by the binary split method. The failure circles correspond to the three main types of arc failures, steep ($\alpha > 0$), horizontal ($\alpha = 0$) and deep ($\alpha < 0$).

3.5.3 Discussion of results

Results for Method of slices											
		Total Stress				Effective stress				Non-linear	
Slope	Type	ϕ	Cr	C	Cr	ϕ	Cr	C=20,ϕ	Cr	C=20,ϕ	Cr
Slope 1 (deep)	SLICES	20.5	0	24	1	25/25	0	4/4	1	20	1
	FOS	25	0	23	0	27	0	10	0		
	FOS Cr			31	1			10	1		
Slope 2 (horiz.)	SLICES	31	0	36	3	46/46	0	21/22	2|1*	37	2|1*
	FOS	32	0	29	0	48	0	28	0		
	FOS Cr			41	3			28	1		
Slope 3 (steep)	SLICES	48	0	39	3	65.5/69	0	41.5/44	1	20/56	1
	FOS	48	0	24	0	69	0	49j	0		
	FOS Cr			40	3			44j	1		

Notes :
* most convincing alternate
/ alternates produced by increasing the damping
| alternative tension cracks to the right of the slices quoted
j indicates result for Janbu input factor of safety 1.13 is quoted
ϕ is in degrees C is in kN/m^2

Table 3.13 Table of Results for Program SLICES

The program results for the test conditions are summarised in Table 3.13. For each slope there are three lines of data. The SLICE result, the FOS result and a FOS result for a modified slope taking in to account any predicted tension crack

from the SLICES result. The tension cracks are applicable to cohesive conditions only. The stress profiles and geometries generated by SLICES are given in Figures 3.6 through 3.20, and the program input may also be found in Tables 3.14 through 3.25. Due to the quantity of output, only the geometry and final profile plots are provided for slopes 2 and 3, as the principal features are illustrated adequately by the slope 1 results which are given complete. The figures and tables are contained in Appendix B. The tables are placed before the corresponding SLICE output. The non–linear command files are not shown as they are the same as the c–ϕ effective stress examples.

3.5.3.1 Results involving total stress conditions

Column one of Table 3.13 shows the results obtained for total stress conditions with zero cohesion. The ϕ values required to stabilise the slopes are quoted. They are as expected, increasing with α. The only significant discrepancy is for the deep slope where FOS predicts an higher ϕ for safety than SLICES. It should be noted that these total stress conditions are not realistic, but were included in the validation for comparison and to see how SLICES behaved throughout the parameter spectrum. The second column headed 'Cr' shows that no tension cracks are predicted.

In column three the behaviour under purely cohesive conditions was investigated. A single result for SLICES was obtained in each slope case together with a prediction of a vertical tension crack forming near to the top of the slopes. Two results for FOS are quoted, the upper for the whole slope and the lower for the slope below the tension crack as predicted by SLICES. The assumptions in the second case being that the soil above the crack plays no part in the behaviour of the main body and hence that the crack penetrates to the failure arc. In all cases FOS predicts a much higher cohesion for slopes with a tension crack. Generally SLICE results do not correspond well with FOS analyses of the intact slopes. An

exception to this is the deep slope where the crack occurs high up on the slope. Conversely on comparison with the FOS results for the cracked slopes the best correlation is at the other end of the slope spectrum, that is the steep slope.

It is worthwhile considering why discrepancies occur. The tension crack is determined by the resolution of the slices, this sometimes leads to two adjacent vertical slice contacts being in tension, indicating a crack in between them, or perhaps a tension zone. Both cases are difficult to convert to FOS problems with certainty. This problem is particularly relevant in considering the effective c–ϕ results where difficulty was encountered in deciding which contact to choose as the tension crack.

The definition of tension cracks is clearest under conditions of high cohesion and steep slopes, which appeals to the rationale. For a well defined tension crack the FOS result for the cracked slope corresponds well to the SLICE result, but for a badly defined crack the integral FOS analysis is close to SLICES. As may be expected these circumstances are found for the steep slope and the deep slope respectively.

3.5.3.2 Results involving effective stress conditions

The first column of effective stress results of Table 3.13 contains the results for the effective frictional conditions. Unlike the total stress equivalent this is a real possibility in the field.

The results correspond well between the two methods, broadly FOS indicates that a higher ϕ is required for stability, but this may correspond to the under estimation of the factor of safety under effective conditions by traditional methods. The two values of ϕ quoted for SLICES are for two damping values. The similarity

of these indicates that as long as numerical stability is maintained the results do not differ largely with the damping factor.

The effective c–ϕ results are in the third column for this set. In all cases the cohesion was fixed at $20kN/m^2$ and again two damping values were used and hence two similar results are given for SLICES. The FOS values are considerably higher than for SLICES, although it should be noted that tension crack definition was uncertain in the horizontal slope case and that the FOS result for the Janbu 1.13 case is broadly in agreement with SLICES. It should also be remembered that effective conditions with submerged slices causes problems for the FOS methods.

Finally, the last two columns of results refer to the 'non–linear' analysis of SLICES. Critical cohesion is applied in this case, that is on contact failure the cohesion is set to zero. The initial value for cohesion is again $20kN/m^2$. There can be no comparison with FOS as non–linear parameters are not permitted. However, this set of results should be between the effective $c = 0$ and c–ϕ SLICE results. In each case this is true.

3.5.3.3 Conclusions

Overall the results from SLICES compare favourably with traditional methods. Initially the results from cohesive conditions caused concern until it was realised that SLICES can predict a tension crack. However, it is not always possible to precisely define the position and some discrepancies are inevitable. For cohesive conditions, high α slopes provide the most similar results. Friction is underestimated by SLICES relative to FOS. It has not been the intention of this discussion to show that these methods produce identical results, it would benefit no one if they did, furthermore the non–linear analysis indicates that SLICES can provide a facility not available in traditional methods.

3.5.4 Interpretation of SLICE output

The interpretations of the results discussed in the previous section were based upon two principal features. Firstly, the factor of safety lists for each slice base contact printed at the close of the run. Then, secondly, the stress profiles drawn at intervals throughout the run. The end of the analysis was determined by convergence of the maximum cycle displacement displayed as part of the running commentary, and by the number of slices at limiting friction. It is possible to terminate the run early by observing this latter number as, when all slices have reached this limit the slope has failed. Under conditions where a tension crack is formed, the uppermost slices will not reach the limit so notice is taken of the maximum displacement. When this value is almost constant the run may be halted.

A safe slope will have at least the toe slice with a factor of safety greater than unity. The safety of the slope increases with this value and with the number of slices which are safe.

Tension cracks are observed by large 'V's' in the side contact stress profiles where the stress is negative for a single contact and by safe factors for slices at the top of the slope.

It may be noted from the results that the stress profiles gradually build up during the analysis, converging on final values. Typically a steep slope shows the following behaviour. The top slices are glued by cohesion to platen and never fail, below this the large slices, due to their weight build the largest normal and shear stresses, which are passed down the slope by a 'knock on' effect. This is seen by factors of safety for the slices decreasing with time for slices near to the toe, until the toe slice factor reaches one. This implies a stress distribution largely, although not wholely, related to the weight distribution of slices. This also implies deformations occurring first in the bulk of the slope, rather than at the toe.

This is not what would be expected in reality, however as discussed previously, the results provided by SLICES are too close to the traditional methods without dismissing both techniques.

It would seem that SLICES essentially models a set of blocks on a curved surface. To overcome this the contact laws need further modification to more nearly model soil failure rather than contact failure. The constraints of time upon this project coupled with the SLICES development time has prevented further analysis of this slice technique.

CHAPTER 4

DISTINCT ELEMENT METHOD OF CIRCLES

4.1 The Concept

4.1.1 Circles as Areas of Influence

Program SLICES attempts to model soil slopes by dividing it into slices and using these as discrete elements. As has been seen this analysis still essentially deals with physical elements, the soil slices. Program CIRCLES differs from SLICES fundamentally, not solely in a different geometrical element, but by considering the circles, not as physical soil elements but as areas of influence of calculation points. This abstraction is a long way from Cundall's BALL program which has largely been used to model the behaviour of sand particles, as discussed previously in the first Chapter.

A further fundamental difference is that CIRCLES does not require a predetermined failure arc. The principle is that if the slope is unsafe, the failure arc or slip zone will be generated during the analysis. The program input is similar to SLICES excepting the meshing of the circles where several additional commands are needed.

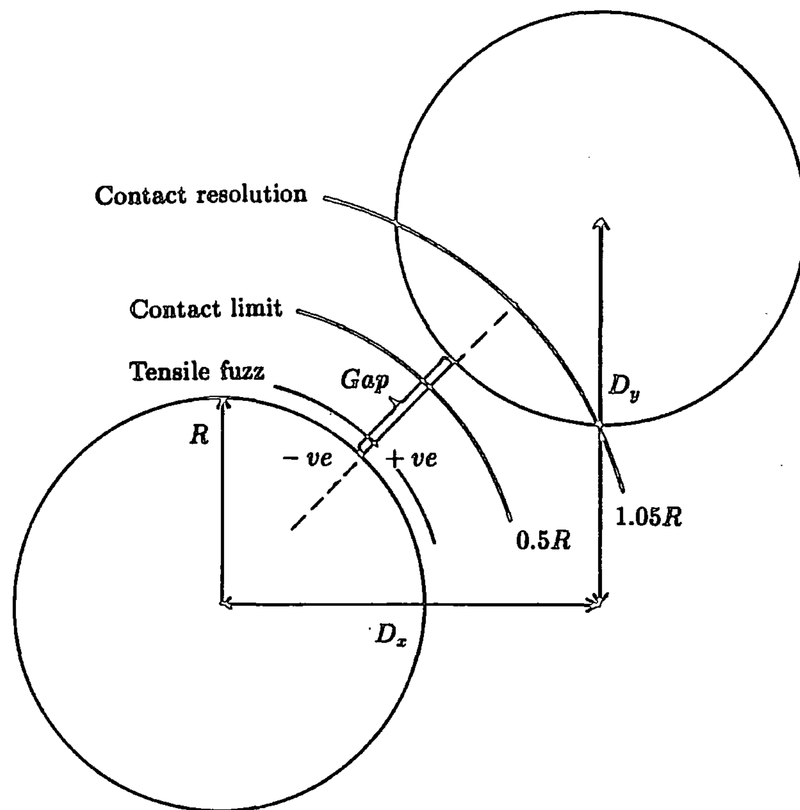
An area of influence may be defined as a circular area around a calculation point with the physical attributes of radius, mass, friction and cohesion. A contact exists with another circle if the circumferences interfere, that is touch or overlap. As the circles are areas of influence and not physical representations a large overlap

at the start of modelling does not give an initial separating force. The centre of the circular area of influence is taken as the centre of gravity.

4.1.2 Contacts in detail

All potential contacts are considered for storage. A potential contact occurs when the centres of two circles are less than the sum of the radii plus the contact resolution apart. The contact resolution is an arbitrary tolerance of 1.05 times the maximum circle radius. The maximum circle radius is the radius of the largest circle in the problem mesh. Another arbitrary tolerance, the contact limit is half the maximum circle radius and is used in a similar fashion to the contact resolution, but this time to distinguish between contacts to be stored and those to be deleted. These tolerances allow potential contacts to be stored in case movement causes a real contact to be formed later. This is further explained in the section on updating of contacts, section 4.2.2.2.

The contact point need not be defined as no rotational forces are considered and all x and y quantities are resolved from the line between the centres. As CIRCLES has been written to incorporate traditional Distinct Element Analysis and consolidation methods, full housekeeping, force displacement law and motion law routines have been included. The housekeeping routines require that a stored contact be found and deleted if the contact gap is greater than the contact limit but less that contact resolution. A further restriction upon a contact is that a small separation of the contact is allowed in the form of a tensile 'fuzz' to help damp transient jumps in the traditional Distinct Element Analysis formulation. Beyond this limit the contact is deemed to have failed in tension. The detail of a contact is shown in Figure 4.1.



Positive gap – Tensile contact

Negative gap – Compressive contact

Figure 4.1 Contact definition in program CIRCLES

4.1.3 The Distinct Element Analysis formulation for CIRCLES

The Distinct Element Analysis formulation employed may be conveniently considered in two parts, the consolidation formulation and the traditional Distinct Element Analysis formulation found in sections 4.1.3.1 and 4.1.3.2 respectively.

Currently forces are converted to stresses very crudely by dividing by the circle radius. As the initial overlap may vary largely, the contact chord formed by the intersection of the circles was deemed unsuitable as a contact surface. There being no other readily available method the current method was employed. Here large circles will have smaller stresses than small ones for the same overlap.

Presently pore water pressure is not accommodated due to the constraints of time. Further work should include this enhancement. The failure criterion for the contact is based upon the Mohr construction shown in Figure 4.2. Here the lesser stress of σ_x and σ_y is taken as σ_3 and the greater as σ_1. The failure σ_1 is calculated from σ_3, c and $\tan\phi$ as shown in equations (1) through (4). If this value is greater than σ_1 then the contact has not failed. If a failure has occurred the appropriate contact force is limited to the equivalent force of the failure σ_1.

(1)
$$q = \frac{1 + \tan\phi - \sqrt{1 + \tan^2\phi}}{\tan\phi - 1 + \sqrt{1 + \tan^2\phi}}$$

(2)
$$\sigma_n = \frac{\sigma_3 + q \times c}{1 - q \times \tan\phi}$$

(3)
$$\tau = \tan\phi \times \sigma_n + c$$

(4)
$$\sigma_1 = 2 \times (\sigma_n + \tau \times \tan\phi) - \sigma_3$$

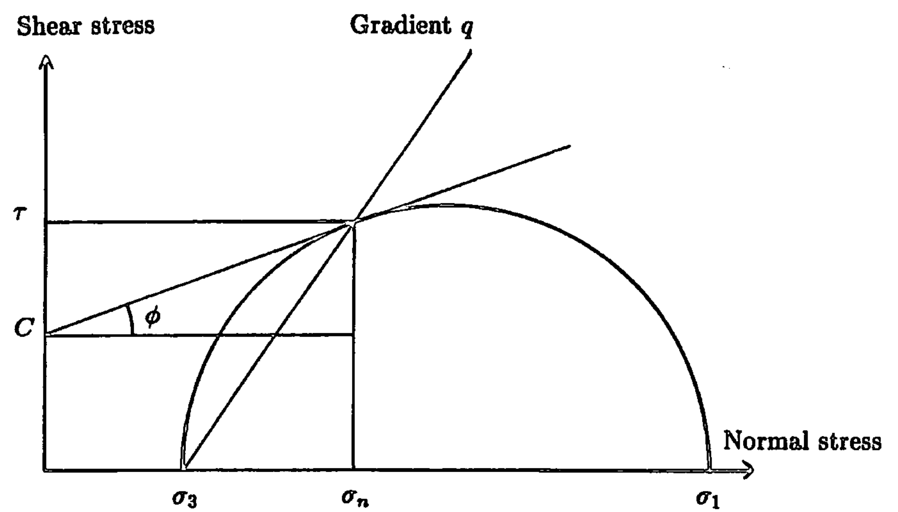


Figure 4.2 The Mohr construction

4.1.3.1 Consolidation formulation

The force displacement law for the consolidation formulation is executed for each contact every calculation cycle. It has four parts to determine the contact movement, the contact force, the limiting stress at failure, and the body force to be added to the circles in preparation for the motion law.

The distances before movement are calculated first in equations (5) and (6).

$$D_x = {}_2P_x - {}_1P_x \tag{5}$$

$$D_y = {}_2P_y - {}_1P_y \tag{6}$$

where the numeric prefices indicate different circles. The radial distance between the centres is given by

$$D_r = \sqrt{D_x^2 + D_y^2} \tag{7}$$

The angle β that this line makes with the x axis gives $\sin\beta = D_y/D_r$ and $\cos\beta = D_x/D_r$.

The movements, M are then calculated from the displacements S.

$$M_x = ({}^n_2S_x - {}^n_1S_x) \tag{8}$$

$$M_y = ({}^n_2S_y - {}^n_1S_y) \tag{9}$$

The current radial distance is given by

$$G_r = \sqrt{(D_x + M_x)^2 + (D_y + M_y)^2} \tag{10}$$

Equation (11) gives the change in the gap between the circles from the initial meshing positions. G_o is the original offset, it is the sum of the two radii if the circles just touched originally and less than this if they overlapped.

$$\Delta G = G_r - G_o \tag{11}$$

It should noted at this point that ΔG will lead to the increment of consolidation force applicable from this calculation cycle. This complicates the decision regarding whether a contact is tensile, as this cannot be deduced from a tensile increment alone. This problem is overcome by keeping a total of ΔG. When this total is positive the contact is tensile.

$$^{n}G_{sum} = \; ^{n-1}G_{sum} + \Delta G \tag{12}$$

The contact forces are calculated by adding the increments to the consolidation forces already accumulated.

$$F_x = \; ^{n-1}C_x + \Delta G \times d \times \cos\beta \tag{13}$$

$$F_y = \; ^{n-1}C_y + \Delta G \times d \times \sin\beta \tag{14}$$

If $^{n}G_{sum} > c$ the contact has failed in tension by exceeding the cohesion and the contact forces are set to zero, $F_x = F_y = 0$. This completes the force displacement law for a tensile contact. The rest of the law is executed for compressive contacts, that is those where $^{n}G_{sum} < 0$.

The contact forces are converted to stresses by $\sigma_x = |F_x/r|$ and $\sigma_y = |F_y/r|$ where r is the circle radius. The contact stresses are now examined for failure by calculating the failure σ_1 consistent with the soil parameters and the σ_3 given by

the lesser of the x and y contact stresses. This procedure is described above. If the contact has failed the stresses are modified by this procedure so that they are limited by the soil strength, otherwise they are unchanged. The contact stresses are converted to forces by multiplying by the circle radius. The signs of the original forces are retained by the modified ones. The new forces are then summed to the body forces of the circles involved and replace the consolidation forces.

(15) $${}_{c}Force_x = {}_{c-1}Force_x + F_x$$

(16) $${}_{c}Force_y = {}_{c-1}Force_y + F_y$$

(17) $${}^{n+1}C_x = F_x$$

(18) $${}^{n+1}C_y = F_y$$

The motion law, which is executed for all free circles involves the calculation of the new displacements.

(19) $${}^{n+1}S_x = \left(\frac{Force_x}{m} + g_x\right) \times \delta t^2$$

(20) $${}^{n+1}S_y = \left(\frac{Force_y}{m} + g_y\right) \times \delta t^2$$

Finally the body forces are reset to zero, $Force_x = Force_y = 0$.

4.1.3.2 The traditional Distinct Element Analysis formulation

The force displacement law consists of determining the movement on the contacts and converting this to a contact force. The movement on the contact is given by equations (21) to (24).

(21) $$G_x = ({}_{2}P_x - {}_{1}P_x) + ({}^{n}_{2}S_x - {}^{n}_{1}S_x)$$

$$G_y = ({}_2P_y - {}_1P_y) + ({}_2^nS_y - {}_1^nS_y) \tag{22}$$

where the numeric prefices indicate different circles and the ^{n} indicates the nth cycle.

The radial distance between the centres is given by

$$G_r = \sqrt{G_x^2 + G_y^2} \tag{23}$$

The angle β that this line makes with the horizontal gives $\sin\beta = G_y/G_r$ and $\cos\beta = G_x/G_r$. Equation (24) gives the the change in the gap between the circles from the initial meshing positions.

$$\Delta G = G_r - G_o \tag{24}$$

Having found the movement it is now possible to calculate the contact forces.

$$F_r = k \times \Delta G \times d \tag{25}$$

$$F_x = C_x + F_r \times \cos\beta \tag{26}$$

$$F_y = C_y + F_r \times \sin\beta \tag{27}$$

If the contact is tensile, that is, the equivalent gap between the circles is now positive and greater than a small tensile fuzz used to damp transient movements, then the contact has failed in tension and the contact forces are set to zero, $F_x = F_y = 0$. If the contact is deemed to be compressive then the forces are converted to stresses by $\sigma_x = |F_x/r|$ and $\sigma_y = |F_y/r|$ where r is the circle radius.

The contact stresses are examined for failure using the same method as the consolidation formulation. The new forces are then summed to the body forces of the circles involved.

$$ {}_{c}Force_x = {}_{c-1}Force_x + F_x \tag{28} $$

$$ {}_{c}Force_y = {}_{c-1}Force_y + F_y \tag{29} $$

The motion law, which is executed for all free circles involves the calculation of the acceleration, velocity and displacement.

$$ a_x = \frac{Force_x}{m} + g_x \tag{30} $$

$$ {}^{n+1}v_x = {}^{n}v_x + a_x \times \delta t \tag{31} $$

$$ {}^{n+1}S_x = {}^{n+1}v_x \times \delta t \tag{32} $$

The equivalent equations for y have been ommitted for clarity. Finally the body forces are reset to zero, $Force_x = Force_y = 0$.

4.2 Implementation

4.2.1 The Program Memory Structure

A program memory schema is shown in Figure 4.3. This is intended to show how the graphical problem area is mapped in memory. The grid of squares is linked by pointers allowing a traverse in any direction and sequentially from SW to NE. The area at the top of the figure is the *re_area_list* and is used for those circles that move between areas in the traditional method. These circles are temporarily placed in this area and are placed into the correct areas just prior to a contact

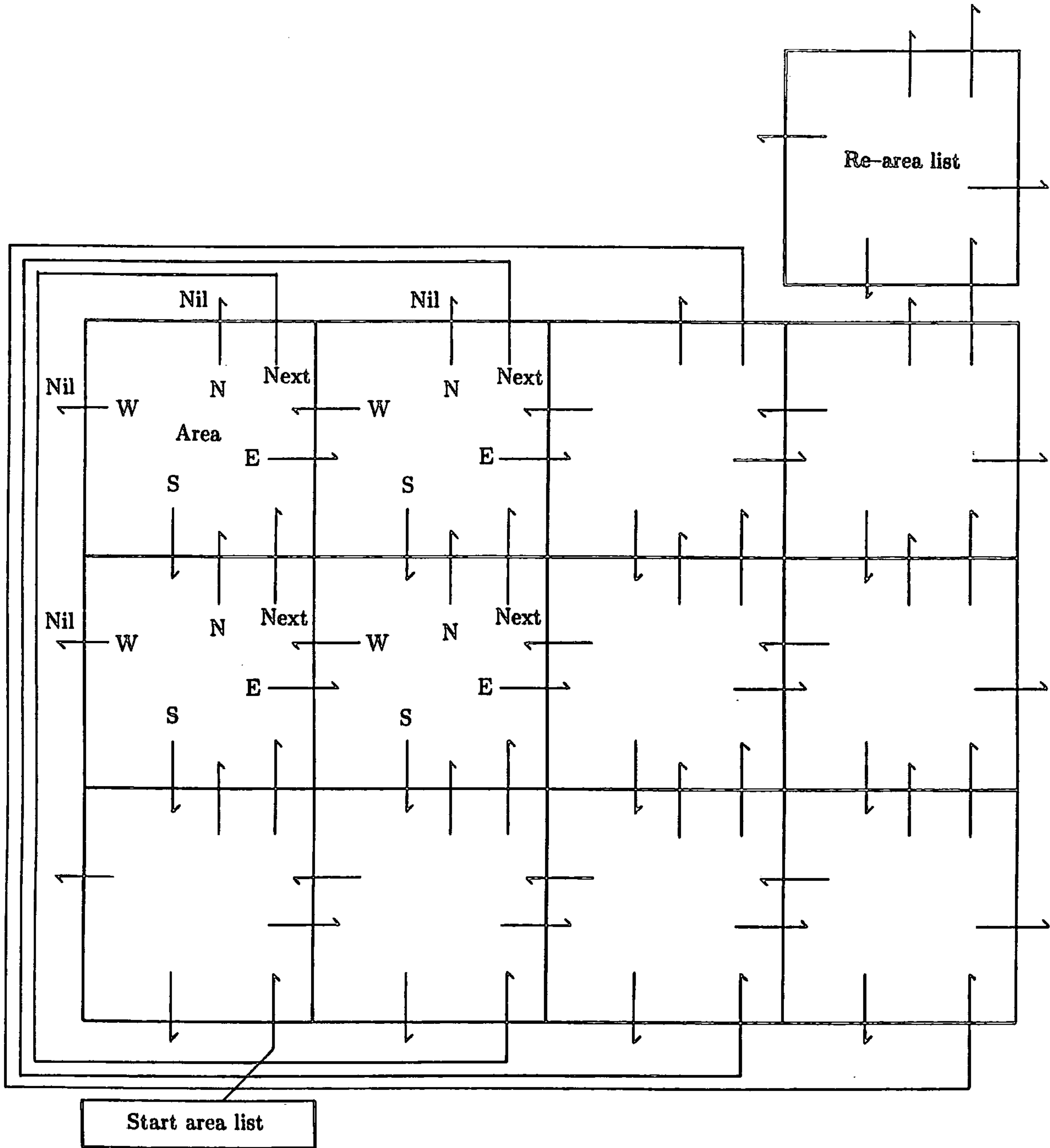


Figure 4.3 An high level view of the memory structure

update. A separate area, *spare_area* is provided for those circles that move out of the problem area altogether. These circles are then effectively hidden for ever. A Bachmann diagram showing the relationships between the areas, circles and contacts is shown in Figure 4.4.

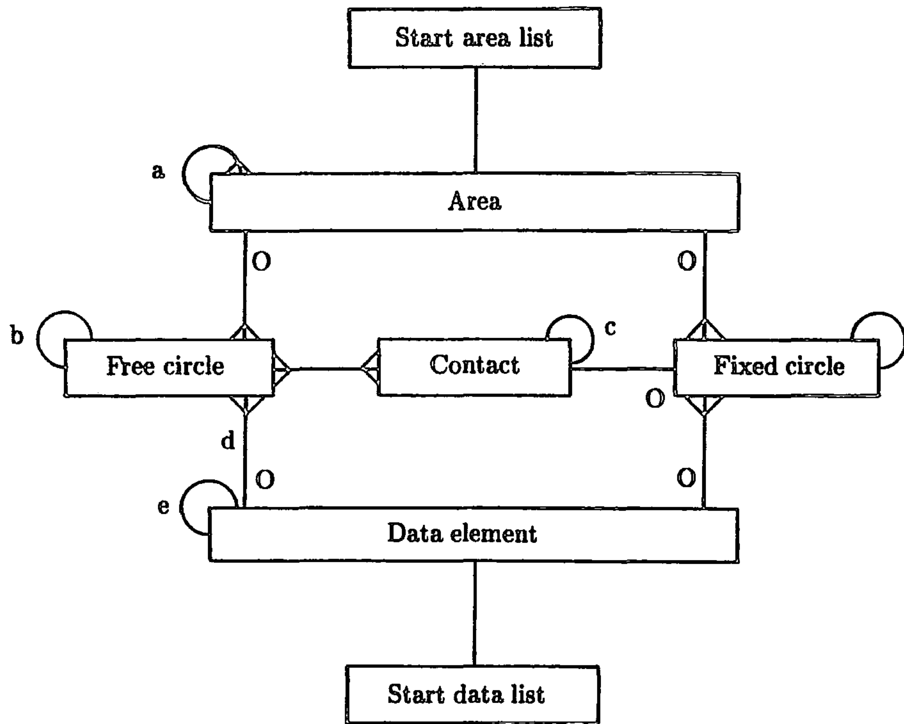
Each area has five pointers leading to the adjacent areas to the North, East, South, West and to the next sequential area. The next sequential area is the same as the area to the north except at the northern edge of the problem domain where it is the southernmost area of the adjacent eastern column. The pointers at the problem edge have NIL values. Diagonal pointers are not required as these areas may be accessed by shifting north and then east for example.

Each grid area has various regulating variables associated with it and two pointers to the circle elements. One points to the free list, that is a list of those circles able to move, and the other to the fixed list containing circles forming fixed boundaries for the problem.

The free and fixed lists are made up of the circle elements which contain three pointers, one to the next element in the list, one to the parameter data block of its type and one to the list of contacts belonging to the element. The contact list contains a pointer to the other circle involved and a pointer to the next contact for the current element.

Of the global variables in the program the pointers *sal*, *re_area_list* and *sdl* are the most important and are discussed below.

sal points to the start of the area element list, that is to the extreme south west area, and is the entry point to the main memory structure.



a – Pointers to E, W, N, S and Next areas

b – Pointers to next circle

c – Pointer to next contact

d – Pointer to element data type information

e – Pointer to next data type element

Crows feet indicate a one to many relation

Double crows feet indicate a many to many relation

O – indicates an optional relation at the end shown

Figure 4.4 A Bachmann diagram of the program memory elements

re_area_list points to the beginning of a list of circles, that, due to movement, have changed position from one area to another. These circles are stored in a separate list until a convenient break in processing and are then placed into the correct areas before a contact update.

sdl points to the start of the parameter data block list. There is one element in this list for each soil type in the problem. The element contains the physical and geotechnical data for the soil type as well as a pointer to the next element in the list. This arrangement allows the information for a particular circle type to be held in common rather than for each circle individually. Hence the circle element need only point to the relevant element in this data list.

4.2.2 Program structure

Functionally the program elements of CIRCLES and SLICES are similar, therefore, it is not intended to describe in detail the procedural elements of CIRCLES. The input and output routines and the error correction methodology are the same. Plotting, cycling, restart, parameter setting, repeat and command processing are similar in both programs. The essential differences are caused by the memory structure mapping the grid areas of the problem space and the contact updating required by the break and make of contacts permitted by the traditional Distinct Element Analysis implementation. These differences are discussed in the following sections. The source code for program CIRCLES may be found in Appendix E.

4.2.2.1 Program structure that maps the memory

The main addition to the complexity of the program structure as compared with SLICES is due to the extra effort required to traverse the element lists. To access all of the circle elements the current area pointer is set to *sal*, the start of

the area list, that is the most south westerly area. The fixed list pointer of this area makes the first circle contained here available. The fixed list can be traversed in the normal fashion by obtaining the next element from the *next* pointer. The end of the list is reached when this *next* element pointer is NIL. The free list is then traversed in this same fashion. When all of the circle elements for the area have been visited, the next area is obtained by updating the current area pointer with the value of the *next_area* pointer. The elements of this area can then be visited. Each area is traversed sequentially until the *next_area* pointer has a value of NIL, that is, after processing the *re_area_list.*

All of this processing is contained in a single procedure *do_this* which is called with the parameters of *proc_name*, *curr_area*, *single* and *lists.* This allows either the free, fixed or both element lists to be traversed for either a single or all areas. For all circles control is transferred to the procedure passed as the parameter *proc_name* to *do_this.* This technique allows each function requiring a circle traverse to have the algorithm coding replaced by a simple statement such as

do_this(motion, sal, all, both);

which will cause the procedure *motion* to be executed for all elements.

To traverse the contact list associated with a circle element the simple construction shown below is used. The beginning of the contact list is found from the *con_list* pointer which is part of the circle memory element.

```
con_ptr := el@.con_list;
while con_ptr not = NIL do
   begin
      ...
      con_ptr := con_ptr@.next_con;
   end;
```

4.2.2.2 The updating of contacts

In SLICES the basic geometrical relationships between the components could never change. The same set of contacts were sufficient to describe the system throughout the analysis. Not so with CIRCLES. Under traditional Distinct Element Analysis large scale movements need to be accommodated, together with the associated make and break of contacts during the analysis. Therefore, it is important that potential contacts are located in advance of the immediate requirement, old ones updated in the contact lists and new ones added efficiently. These are currently achieved by the following method.

A complete circle traverse is carried out as described above with both the fixed and free lists examined for each area sequentially. Each circle element, the 'home' element, is checked for contact with the circles occurring later in the free list for the area. It is not necessary to check for contact with circles occurring in the free list before the 'home' circle as any contact will already have been found. Likewise when the 'home' circle is a free type it is not necessary to check for contact with any circle which is in the fixed list of the current area.

Once the free list for the area has been checked and the 'home' circle is not near to the edge of the area, the next circle in the free list is taken as 'home' and the search for its contacts can begin. However, if the circle is close to the edge of the area, the northern, north eastern, eastern and south eastern areas in turn have both the free and fixed lists checked for contact with the 'home' circle. It should be noted that the areas in the southern, south western, western, and north western areas need not be examined as they occur sequentially before the current area, so that any contact will already have been found.

If, during the scan for contacts outlined above, a contact is found, it must be seen if the contact is already recorded. To do this the contact list for the 'home'

circle is traversed until either the contact or the end of the list is located. If the contact is not found in this list the contact list for the other circle is also traversed. If an old contact is located and the distance between the circle circumferences is less than the contact limit, nothing more is done. However if this distance is greater than this limit, the contact is destroyed, that is, removed from the contact list. A contact that is not found in either of the contact lists and the distance apart is less than the contact limit then a new contact is created in the 'home' circle contact list.

To summarise, each free circle acts as the 'home' element and each free circle later in the area and, if necessary, all circles in certain neighbouring areas are investigated for contact with it. If there is a possible contact both contact lists are scanned and housekeeping is performed.

Two main factors govern the efficiency of this algorithm. Clearly efficiency is increased when most circles are not near an area edge, which can be achieved by having areas large in comparison with the size of the circles. The reason for this is that more circles are only compared with circles from the same area. Efficiency is also increased for large number of circles by restricting the number of circles per area by increasing the number of areas. It should be noted that large numbers of areas will add an overhead to the circle access as all areas are looked at by the routine *do_this*. These two apparently conflicting factors need to be balanced to gain the optimum efficiency for any given problem.

4.2.3 Input command language

4.2.3.1 Introduction

All program tasks are controlled or defined by the Input Command Language. As shall be explained later the program requires some commands in a particular

order, but on the whole the majority of commands may be used at any time. Although the program is not designed to run interactively, it is possible with care. Normally, however, the commands should be contained in a file prior to use.

The commands may be categorised into broad sections, dealing with program control, plotting, meshing, debugging, and the setting of options and parameters. These correspond to the major procedures of the program. The commands are hierarchial, forming a tree system and follow the same rules as for SLICES.

The following sections, 4.2.3.2 to 4.2.3.8 describe the functions of the commands of each set. The symbols used in the syntax definition of the input command language is shown in Table 4.1 with the definition in Table 4.2.

Symbol	Definition
...	indicates possible repetition of the clause
[ ]	indicates an optional clause
( )	indicates a group of clauses
≺ ≻	indicates substitution by a value, which may be either a clause or literal
‘ ’	indicates a literal value
|	indicates an alternative
IS	is the definition operator

Table 4.1 Input Command Language Parsing Symbols

4.2.3.2 Control commands

The control commands form the outermost command set, all other commands are accessed through this set.

The **set** command enters the parameter procedure to allow parameters to be set up, altered or inspected.

```
task IS [≺com≻ ...] 'stop'
correction IS [≺com≻ ...]('return' |'stop')
limits IS ≺real≻ ≺real≻ ≺real≻
boxes IS ≺integer≻ ≺integer≻
reply IS 'on' | 'off'
com IS ('start' ≺start block≻) | ('restart' ≺limits≻)
    | ('plot' [≺plot command≻...]) | ('set' [≺set command≻...])
    | ('debug' [≺debug command≻...]) | ('go' ≺integer≻)
    | 'save', 'cend' | 'rend'| 'settle' | 'collapse'
    | ('repeat' ≺integer≻ [≺com≻...] 'rend')
parameter IS 'framelimit' | 'writegap' | 'interval' | 'gravity'| 'time'
property IS 'damp' | 'mass' | 'cohesion' | 'friction'
    | 'density' | 'radius' | 'stiffness'
datanumber IS ≺integer≻
oper IS '*' | '+' | '−' | '/' | '∧' | '='
set command IS (('echo' | 'echodebug' | 'cmdproc' | 'overwrite') ≺reply≻)
    | (('framelimit' | 'writegap' | 'interval') ≺integer≻ )
    | (('gravity' | 'time') ≺real≻ ) | ('cmdlist' [≺com≻ ...] 'cend')])
    | ('calculate' [(≺parameter≻ (≺oper≻ ≺real≻) | '?') ...])
    | ('calculate' [('soiltype' ≺datanumber≻
      [(≺property≻ (≺oper≻ ≺real≻) | '?') ...]) ...])
plot command IS ('initialise' ≺limits≻) | 'ballplot' | 'dotplot'
    | 'velocities' | 'displacement' | 'conplot' | 'failplot' | 'graticule'
    | 'standard' | 'page'| 'border' | 'endplot'
    | ('map' ≺map command≻) | ('zoom' ≺limits≻)
map command IS 'picture' | 'horizontal' | 'vertical'
    | 'full' | 'fullnoscales' | ('zoom' ≺limits≻)
debug command IS 'contacts' | 'energy' | 'general' | 'flagson'| 'flagsoff'
    | 'datalist' | 'blocks' | 'areas'
    |(('update' | 'motion' | 'consolidate' | 'ford' | 'cycle'
    | 'cycle' | 'rearea' | 'trace' | 'oscillate') ≺reply≻)
data IS ≺datanumber≻ ≺real≻ ≺real≻ ≺real≻
    ≺real≻ ≺real≻ ≺real≻ ≺real≻ ≺type≻
type IS  'free' | 'fixed' | 'track'
coord IS ≺real≻ ≺real≻
eoln IS (' ' ...) until the end of line is reached
mesh command IS 'relative' | 'absolute' | 'single' | 'multiple'
    | ('create' (≺coord≻ ...) ≺eoln≻)
    | 'move' | 'position' (≺coord≻) | ('angle' | ≺real≻)
meshinfo IS ('dataset' ≺data≻) ('for ' [≺mesh command≻ ...] 'endfor')
start block IS ≺heading≻ ≺limits≻ ≺boxes≻ [≺meshinfo≻ ...] 'meshend'
```

Table 4.2 Input Command Language Parsing Definition

The command **restart** causes the restart of a previous problem run. A file containing the restart information must be attached to unit 1. Within the command file the mapping information must follow.

save causes a restart file to written, it may either overwrite or append the file attached to unit 2 according to the setting of the overwrite command (a set command). This is used to save the solution to the task found so far for a large job, thus avoiding loss in the case of a system crash.

Command **start** begins a new problem. A title up to 80 characters long may follow, but the next line must contain the mapping information and then mesh information is required. Section 4.2.3.8 describes the meshing commands.

stop this causes the geometry to be plotted, a restart file to be written and the program run terminated.

The command **debug** causes the debug procedure to be entered, so that debug options can be set or general information generated.

The **plot** command causes the plot procedure to be entered, which allows requests for the manipulation of the plot format, size, and the production of the different plot types available.

go is the command that causes the calculation cycle to be entered and it must be followed by an integer, the number of cycles to be executed.

Command **repeat** is the opening statement of the repeat n commands rend loop structure. It must be followed by an integer, which is the number of times the loop is to be executed. There are certain commands for which inclusion in this structure would be pointless.

To close the repeat loop the command **rend** is used in two ways. As regards to input, it terminates input to the repeat controlling procedure and is the last statement in the repeat loop, in this case it is not a control level command. The second way in which it is used is internally, during execution of the loop, here it signifies the end of the loop so that the commands may be repeated again.

A similar command to **rend**, **cend** is used in two ways. Firstly, it terminates input to the command list structure of the set command set, and secondly it terminates execution of the command list during use. Section 3.3.2.4 describes the **set cmdlist** commands **cend** facility in detail.

The command **return** terminates interactive input during input error handling, and is described together with this facility in section 3.3.3.3.

The command **settle** causes the consolidation implementation to be used during the calculation cycle while **collapse** will cause a traditional Distinct Element Analysis implementation to be used instead.

4.2.3.3 The debug command set

To gain access to these second level commands the debug command must be entered at the control command level. This facility falls into two parts, one outputs information at the point of issue of the command, while the other assigns options which provide data during the subsequent execution of the program. All output from this routine is written to the file attached to unit 7 unless otherwise stated. The debug commands are as follows.

The **datalist** command writes out the circle parameter data block list while **areas** gives the information associated with each area. Command **blocks** produces the information for each circle while the **contacts** command writes out the contact

information. **general**, as is to be expected, produces some general information. Just as in SLICES, **flagson** sets all the debug options on, and should be used with care and **flagsoff** turns all of them off.

All of the following commands must be followed by the third level commands of either **on** or **off**, which clearly sets the option on or off.

The **update** option produces contact information as the contacts are created or destroyed. **motion** controls the production of debug output from the motion law during execution of the calculation cycle. The option **ford** controls the production of the debug output from the force displacement law during execution of the calculation cycle. The **consolidate** option produces limited information from both the motion and the force displacement law, again during execution of the calculation cycle. **cycle** this produces information from all procedures within the calculation cycle and procedure *cycle* itself. The option **trace** causes a message to be written on entering and exiting all procedures and functions. Output is written on the file attached to the unit 8. **oscillate** causes information from calculation laws, formatted for input to the program SOP, to be written onto the file attached to the unit 10. Finally the option of **rearea** causes information connected with the change of area of a circle to be written out.

4.2.3.4 The set command set

To gain access to these second level commands the command set must be issued at the control level. This set of commands falls into two groups, problem parameters such as gravity and options such as frame limit. The set commands are as follows.

When set to on **echo** enables all input commands to be echoed on the running commentary. The command must be followed by the third level commands of either **on** or **off**. The default is on.

debugecho if set to on this causes headings for the debugging information to be written in addition to the information itself.

The **overwrite** option controls the restart file output. If set, the file attached to unit 2 is emptied prior to use, otherwise the file is appended by the restart information. The default is off.

Option **cmdlist** sets up a subsidiary file and copies all command input to it until the command **cend** is entered. The execution of this secondary command file is controlled by two further set commands, **cmdproc** and **interval**. Transfer of control is passed from the file attached to the unit scards to the secondary file (always named internally as the temporary file -sass.cmd), during the execution of procedure *cycle*. The default value is null.

The option **interval** must be followed by an integer, the number of cycles to be executed between successive executions of the command list secondary command file. The default value is 100.

The **cmdproc** option must be followed by either of the commands **on** or **off**. If it is set to on, the command list secondary file is executed whenever the total cycles executed so far divided by the interval, (as set by the command **interval**), is an integer value. If set to off this facility is not used. The default value is off.

framelimit is followed by an integer. The GHOST library limits the number of frames of plot output to twenty. If this is exceeded the program will terminate. This command allows this limit to be reset. The default is 20.

writegap sets the interval of cycles between display of some of the running commentary information. The default is 100.

The **gravity** option is followed by a real number, which represents the value of gravity in the positive y direction. The default value is 0.

Option **time** is followed by a real value this sets the time step size.

The **calculate** command allows the values of some parameters and options to be modified or inspected rather than simply reset. Calculator commands are described in the following section, 4.3.2.5, and are level three commands.

4.2.3.5 The calculator command set

This set is at the third level and is accessed by the command string **set calculate**. Almost all the calculator commands have the same format, that of ≺parameter≻ ≺operator≻ ≺real≻ with the exception of the enquiry, **?** when a value is not required. Permissible parameters (the third level commands) are **interval, writegap, gravity** and **time**. The operators (fourth level commands, to be precise) are = replace, * multiply, / divide, + add, - subtract, ^ exponentiation. The **?** enquiry is also used here. The values are read in as real numbers only. For parameters which are integer in nature, conversion takes place to give an integer result. The final value of a calculation command is written to the running commentary output stream.

The command **soiltype** allows access by the calculator to the soil parameters of a particular soil type. The number of the soil type, that is the data type flag must follow. The parameter that is to be modified is the input as for a normal calculator command. The parameters that may be modified here are the damping

factor, mass, cohesion, friction, density, radius and stiffness by using the commands of **damp**, **mass**, **cohesion**, **friction**, **density**, **radius** and **stiffness**.

4.2.3.6 The plot command set

To gain access to this second level set the command plot must be issued in the control command set. As all the GHOST library routines are contained in the procedure **plot** to ease maintenance, and many plotting functions are automatically carried out by the program, it has been necessary for some of these and map commands to be issued internally. Although these internal commands are described, it may be that they will never need to be issued externally. They are **initialise**, **endplot**, and most map commands with the exception of **zoom**.

The **initialise** command sets the initial plotting parameters and turns the plot output stream on. This command is issued automatically on receipt of the **start** or **restart** commands and should not need to be used normally.

The command **ballplot** draws the circles in the current plot space, **dotplot** causes the centre points of the circles to be plotted while **velocities** will draw the current velocity vectors. The contact forces may be drawn by using the **conplot** command and a plot of the failed contacts produced by **failplot**. The command **displacement** draws the current incremental displacement vectors.

The **graticule** command produces an outline of the area limits. **standard** this produces a standard plot of the circles. The **page** command calls for a new frame, or in physical terms a new sheet of paper while **border** produces a border with the problem title and current problem time. The command of **map** will enter the tertiary level map set and enables the modification of plot formats. Finally the **endplot** command is issued internally during closedown of the program under

normal termination, it produces a frame with a slice plot and turns the plot output stream off.

4.2.3.7 The map command set

These tertiary level commands are accessed by first issuing the commands **plot map**.

The command **horizontal** sets the page format to lie along the A4 sheet of paper as in a landscape picture. The default size is (0.06,0.96,0.05,0.65) expressed in a $(xmin, xmax, ymin, ymax)$ format.

Likewise the **vertical** command sets the page format to lie down the A4 sheet as in conventional portrait picture. This is the default format, the default size is (0.15,0.75,0.06,0.96).

The command **full** sets the plotting space to the maximum permitted page size suitable for A4 paper. A border is drawn around this area together with axes scaled to the current mapping limits while **fullnoscales** does the same but does not draw scaled axes.

The **zoom** command is followed by three real numbers, $xmin$, $xmax$, and $ymin$ which form the mapping limits. $xmin$ is the minimum value of x, $xmax$ is the maximum value of x, and $ymin$ is the minimum value of y of the problem geometry to be plotted. As the plots are in fixed proportions in both landscape and portrait mode it is not necessary for the maximum y value to be provided. This command enables portions of the problem to be examined in more detail. Mapping limits are expected as part of the input after both the **start** and **restart** commands.

4.2.3.8 The mesh Command Shell

This is the only set of commands that cannot be accessed at random by a user. It is automatically entered after the issue of the level one command **start**, a further oddity is that this set can only be exited by issuing the **meshend** command. The numbers of areas in the x and y directions are expected before any further commands are entered.

dataset create a dataset type. This is followed by the various parameters governing the soil type. All of the circles that are created following this are of this type until the next **dataset** command is encountered. The parameters which must follow are, dataset number, damping coefficient, mass, cohesion, $\tan\phi$, r, radius, stiffness and dataset type. Apart from the dataset number and dataset type all of the parameters are real numbers. The dataset number is an integer and the dataset type is one of **free**, **fixed** or **track**. The dataset type of **free** indicates that the circles are to be free, conversely **fixed** indicates that the dataset is to be of fixed circles. The **track** type will allow the circles to be tracked by the **oscillation** debug command.

Once the dataset has been set up it is then necessary to create some circles. To enable sets of circle to be created in a relatively easy fashion a set of commands have been produced to govern the position of the creation point. Firstly, the command **relative** will cause the x and y values following the **create** command to be taken as the relative distances between creation points. Conversely the **absolute** command causes the values to be taken as the absolute coordinates. Furthermore these absolute or relative values in the x and y directions may be operated on by the **angle** command. This command is followed by the angle, in degrees, from the horizontal at which the circles are to be generated.

The creation position can also be controlled by the **move** and **position** commands. **move** causes the creation position to be moved relative to the current position. The x and y values for the movement must follow. The **position** command causes the creation position to be at the absolute position of x, y which again must follow.

The **single** command will allow only one circle to be created at a time, where **multiple** will allow several circles to be created with one command. The command **for** begins a for loop, it is followed by the number of circles to be created. An **endfor** will end the for loop construction. Finally, the **create** command triggers the creation of a new circle.

4.2.4 The utility files

The input command file format restrictions for CIRCLES are the same as for program SLICES and may be found in section 3.3.3.1. The task definition using the input command language has already been largely covered in section 3.3.3.2, the particular meshing information for CIRCLES is discussed in section 4.2.3.8. However, with reference to the chapter three discussion it should be noted that no factors of safety are produced. Automatic plot production occurs at the start and at the end of the task. The rules for error handling are the same as for SLICES and have been discussed in section 3.3.3.3.

The repeat, command list, restart, trace and oscillation output files all function in the same way as for SLICES. The restart file record tags are given in Table 4.3 as clearly there are some changes in the memory elements.

The various debug commands produce the messages shown in Table 4.4. Most of the formats shown need no further explanation. For reasons of brevity not all of the messages detail the quantities shown. These are now described.

Tag	Restart record type
G	the general information
c	a command list word
r	a repeat list word
D	a data parameter element
A	an area information element
F	a fixed circle element
f	a free circle element
C	a contact element
*	the end of the restart data

Table 4.3 The restart file line tags

The **rearea command** produces several messages and shows the details of the re–location process of circles that have moved out of their original area. The formats are shown for completeness but this level of debugging information is only useful if the details of the program are understood. The command **update** produces information concerned with the production of contacts. The contact creation message shows the contact gap and the positions of both of the circles involved. The other messages are self explanatory.

The debug command **ford** produces the distance between the original centres of the circles and the sine and cosine of the angle between this line and the horizontal. The current gap between the circles is then printed followed by the consolidation force and the body forces. The body forces of the element which owns the contact are quoted first. The data produced by the **consolidate** command are the limiting forces calculated by the failure law and the consolidation forces upon the contact. The **motion** command produes the data type of the element followed by the forces and displacments.

The contact information printed consists of the consolidation forces, the current offset and the positions of the two circles involved. The current circle is given last. The area data is made up of the area x and y limits, the column and row

Flag	Format
rearea	Area 999999 999999 999999
	Area 999999 999999
	ori x,y, new x,y 999999 999999 999999 999999
	x,xm,y,ym 999999 999999 999999 999999
	setup areas col number 999999
	setup areas row number 999999
updat	Victim destroyed
	Contact created 999999 999999 999999 999999 999999
	total number of contacts 99999999
ford	deltagap con_force force t_force for x then y
	99999999999ES99 99999999999ES99 99999999999ES99 99999999999ES99
	99999999 99999999 99999999 99999999 99999999 99999999 99999999
consol	99999999999ES99 99999999999ES99
motion	999 f 999999999ES99 999999999ES99 s 999999999ES99 999999999ES99
cycling	max individual disp 99999999999ES99
contact	Contact information :
	forces of contact, sibling, owner
	999999 999999 999999 999999 999999 999999 999999
area	Area data :
	xmin,xmax,ymin,ymax, upd_par
	999999 999999 999999 999999 999999 999999 999999
element	Element data :
	offs posn force velocity accleration datatype
	999999 999999 999999 999999 99999999 (occurs 7 more times) 999
data	flag damp mass c phi rho rad kn typ
	999 99999999 99999999 99999999 99999999 (occurs 4 more times)
general	xxxxxxxxxxxx task title xxxxxxxxxxxxxxx
	xareas number 999999 length 999999
	yareas number 999999 length 999999
	total number 999999
	mapping xmax 999999 ymax 999999
	plot interval 999999
	gravity x 999999 y 999999
	timing delay 999999
	totals balls 999999 fixed 999999
	cracked 999999 types 999999
	contact 999999 cycles 999999
	updates 999999 frames 999999
	plots 999999

Table 4.4 Debug output formats

numbers and the update parameter. The element data consists of the original and current positions, the final consolidation force upon the element, followed by the current force, velocity, acceleration and then the data type.

The running commentary produces various information concerning the task on to the screen. The screen positions for this are given in Table 4.5 and the specific messages occur in Table 4.6. The same codes to clear the screen and turn the cursor on and off are used as in SLICES and may be found in section 3.3.4.7.

Line	Content
1	Blank
2	The program running commentary heading
3	Blank
4	The task title
5	Blank
6	The number of cycles requested
7	The number of plot frames completed
8	The number of plot types completed
9	The number of updates completed
10	The number of cracked circles
11	The number of cycles completed
12	The command under execution
13	General messages
14	Error messages and first line of totals
15	Message requesting replacement commands
16	Prompts to user for command or parameter data
17	Input line and last line of totals
18	Blank
19	Blank
20	Messages dealing with the restart files

Table 4.5 The running commentary screen lines

2	PROGRAM CIRCLES RUNNING COMMENTARY ON :
12	Command : xxxxxxxxxxxx
12	Reading : x
13	Warning : all velocities zero
13	Warning : all contact forces zero
13	Warning : no failures : no plot
13	Warning no circles left
13	Decreasing stability 9.999999999999ES99
13	Increasing stability 9.999999999999ES99
13	A restart file has been read
13	The value is : 999999999999999
14	Current time step set to : 99.9999999999
14	! error xxxxxxxxxxxx found in routine get_command
15	Input corrected commands ... <RETURN> ...
16	Input a command please ....
14	total balls 999999 fixed 999999
15	total cracked 999999 contacts 999999
16	total cycles 999999 no.updats 999999
17	total frames 999999 plots 999999
20	A restart file has been written

Table 4.6 The running commentary messages

4.3 Validation

4.3.1 Introduction

Program CIRCLES must primarily be viewed as a development program. The main reasons for this are the exclusion of effective stress and difficulties encountered in controlling effects at the boundaries of the mesh. In this validation section it is intended to show that the program correctly consolidates contacts and models the contact failures under cohesive and frictional conditions. It was realised from the outset that different mesh types could influence the failure of a given slope geometry. The extent to which this occurred was to be left to a stage when the technique, having been shown to be applicable to soil masses, could be tuned to the 'live' situation. During this investigation it has been found that edge effects seriously mask the expected behaviour of the mesh, it is therefore intended

to discuss these effects and to show the steps that have been taken to overcome them.

4.3.2 The contact behaviour

To show that the contact behaviour is correct the case of a single contact is discussed first. Consider two circles, one above the other, with the lower one fixed. Table 4.7 shows the forces and displacements relevent to the upper mass in the 2000th calculation cycle. The information is from procedure *consolsl.* The first line shows the forces and the second line the displacements, both sets of values show the x value to be zero. The weight of the circle was 40 Newtons which corresponds to the y value of force. The small y displacement shows that the situation has reached a stable position.

```
A TWO CIRCLE TOWER
f  0.0000000000000E+00  4.0000000000000E+01
s  0.0000000000000E+00 -2.4868995751604E-14
max individual disp     2.4868995751604E-14
```

Table 4.7 Output from CIRCLES after 2000 cycles

The case of an equilateral triangle of circles, where the lower two circles are fixed, is a useful one. This is because the simplicity of the case eliminates edge effect distortions to the stresses but does allow the investigation of contact failure. The equilibrium force on each of the diagonal contacts may be shown to be $\frac{mg}{\sqrt{3}}$ acting along the contact line. The consolidation forces for equilibrium are, therefore, $\sigma_1 = mg/2$ and $\sigma_3 = mg/2\sqrt{3}$. For failure just to occur then the equivalent stresses of these forces must form the limiting values. For the case of a purely cohesive soil these limits would be generated by a value of cohesion corresponding to $\frac{mg}{4}(1-\frac{1}{\sqrt{3}})$. For a mass of 4 and gravity of 1 this simplifies further to give $c = 0.42265$. It was found by analysing this system with $c = 0.42264$ that

both contacts failed. However, failure did not occur for $c = 0.42265$, which shows the program behaviour to be correct.

4.3.3 The Mesh Edge effects

It is not possible to model a slope using a mesh consisting of a series of vertical columns of circles. The reason for this is that because gravity acts downwards there is no lateral movement of the circles to cause lateral forces. The columns are therefore uncoupled and consolidate independently. An hexagonal close packed mesh overcomes this lateral coupling problem.

The analysis of such a mesh as shown in Figure 4.5 highlights an edge effect. The figure is composed of three sections. The lowest sections shows the circle element mesh, the middle section shows the forces on the contacts and the top section shows the contacts that have failed. Failed contacts appear as lines between the dots representing the circle centres. These lines are proportional to the failure forces. It can be seen in the figure that the forces within the mesh increase from the top to the bottom and from the sides to the centre. However the forces for successive contacts on the lowermost row of contacts show an alternation between large and small forces. A large force occurring at the edge, then a small one, and so on, to the middle.

To explain this effect consider the analysis in the early stages. As the mesh begins to consolidate each successive horizontal layer of contacts becomes compressive with all of the circles above it falling due to gravity only. Consider such a layer of circles, that is where the upper contacts are neither compressive nor tensile, but where the lower contacts are compressive. In the next cycle the circles above this layer will all fall by $g\delta t^2$ and will produce compressive forces in the upper contacts. These forces will have horizontal and vertical components. In the case of all circles except the outermost ones the horizontal forces will sum to the

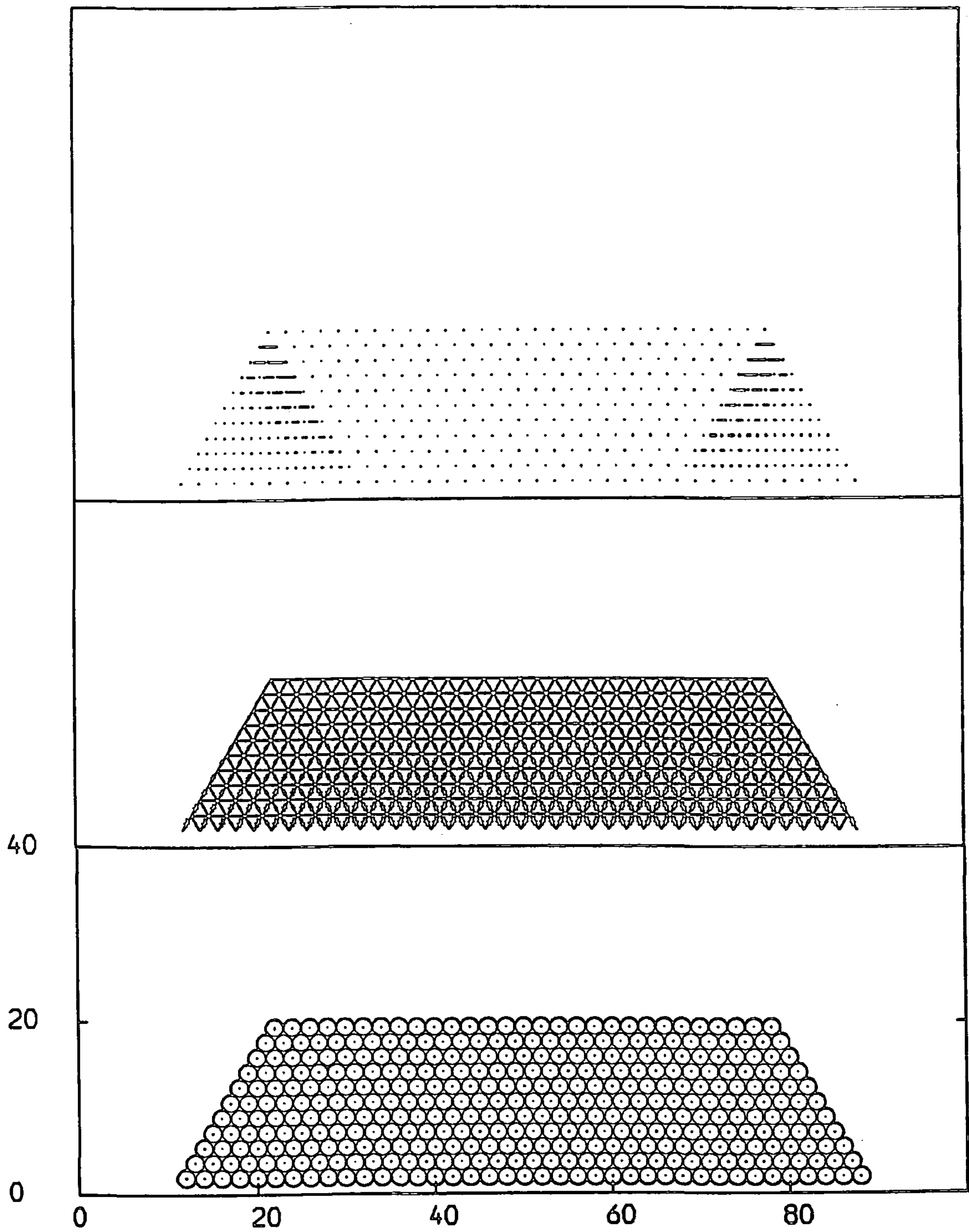


Figure 4.5 Analysis of embankment without contact correction

circle forces to give a zero lateral body force. In the case of the outermost circles the single upper contact force is not balanced due to the asymmetry of the mesh at this point and hence will have a lateral displacement in the next cycle.

Each of the outermost circles will behave in a similar fashion and will give rise to these circles acting independently from the main body. This independence gives rise to higher contact forces down this diagonal column and lower forces between this column and the next one. The low forces between these columns causes the next diagonal column to behave in a similar fashion to the first. This then leads to the generation of a series of high and low contact forces.

This effect builds up progressively in the embankment. The tensile forces are generated first at the bottom corners and then gradually up the diagonal. The tensile forces are then produced in the next diagonal, and so on, until eventually the embankment may be seen to consist of the three regions as shown in the figure. These are lefthand and righthand triangular zones of tensile horizontal contacts and a middle zone of compressive horizontal contacts. If the cohesion and friction of the soil are reduced the embankment shows compressive failure at the toes of the slopes but the tensile zones dominate and distort the failure zones so that if further contact failure occurs it happens at the top of the slope and in the middle of the embankment. The latter occurs when the soil characteristics are made to be weak to try and force a proper failure.

This edge effect problem is exaggerated by the edge circles consolidating faster than the internal ones due to the number of contacts affecting them. This is as a direct result of the equation for N_s given previously. This was not initially accounted for by the Distinct Element Analysis employed as there did not seem to be a straightforward method of dealing with it. The problem in trying to accommodate the number of contacts is in determining the number of contacts a circle has before the force displacement law is executed, for it is here that the

damping factor is used. Furthermore, where two circles are involved which have different numbers of contacts, the modified damping factor on the mutual contact can not be easily determined. Such cases occur at the edge of the mesh, where, if an average number of contacts were used, would not effectively deal with the problem.

Therefore, it was decided to counter this effect by modifying the motion law. This was achieved quite simply by dividing the circle displacement by the number of active contacts that it has. Reducing the circle displacement at this stage is equivalent to reducing it by this factor in the force displacement law to give a reduced force. It is merely done at the last stage of the cycle rather than at the first stage of the next. Applying this factor here allows the adjustment to be made correctly for each circle. This is because it is dependent upon a property of the circle rather than upon a property of a pair of circles, namely the number of active contacts that a circle is involved in.

To find this number an additional process was introduced prior to the motion law. As the displacements are greatest at the beginning of the analysis, it is at this stage that the correction needs to be the most accurate. An active contact is one that is compressive, or will become compressive in the next calculation cycle. This is achieved crudely at present. If a circle has a contact with a fixed circle the contact is assumed to be active, this will always be true if the fixed circles are placed to restrain the circles and cause consolidation. To decide if a contact is active when it is between two free circles the relative displacement is examined, if it is not zero then the contact is active. Horizontal contacts are presently considered as inactive as they are unimportant at the beginning of the analysis. The contacts are counted by scanning the contact list for each circle and the number of active contacts incremented for both the owner and the other circle involved, if the contact is found to be an active one.

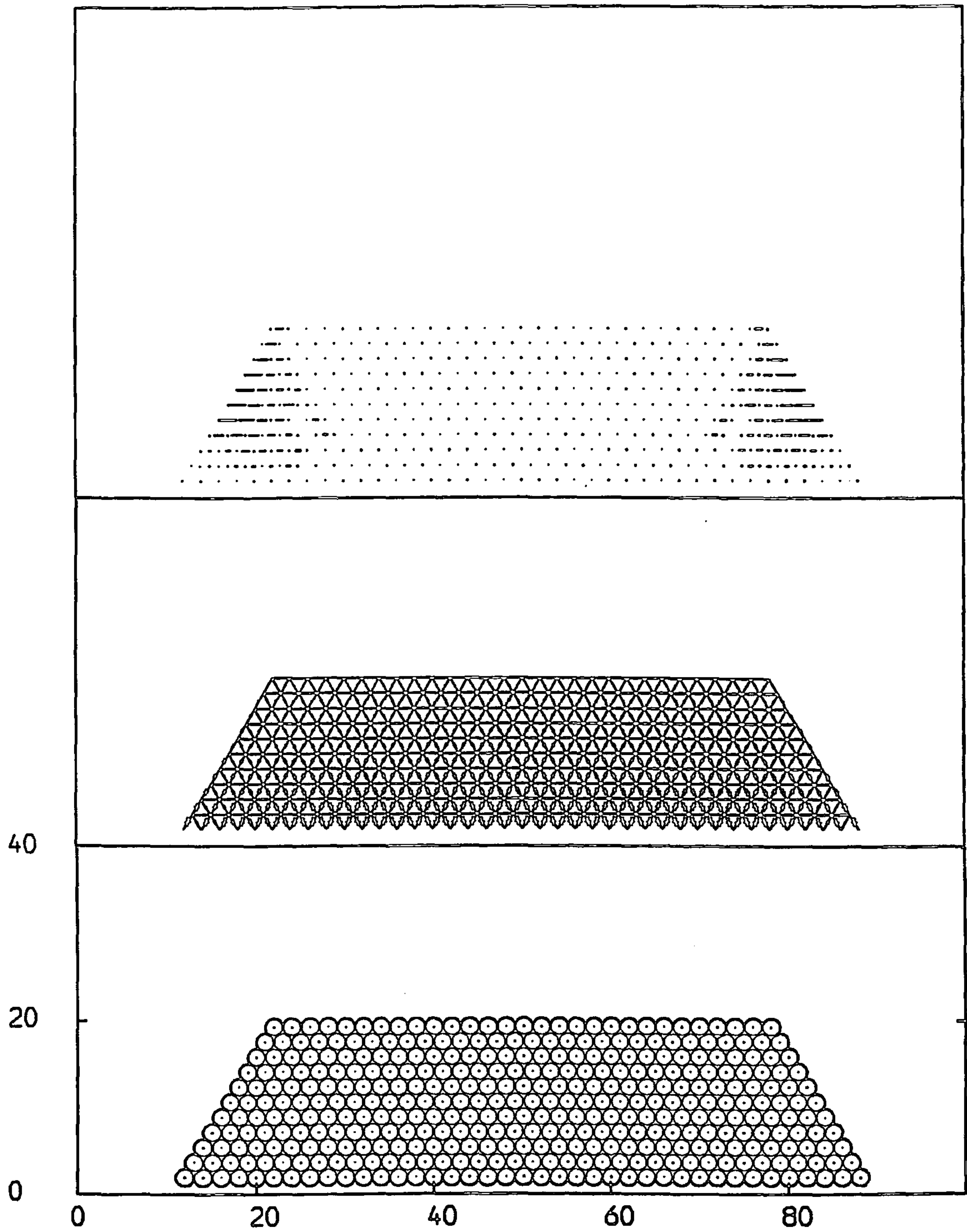


Figure 4.6 Analysis of embankment using a contact correction

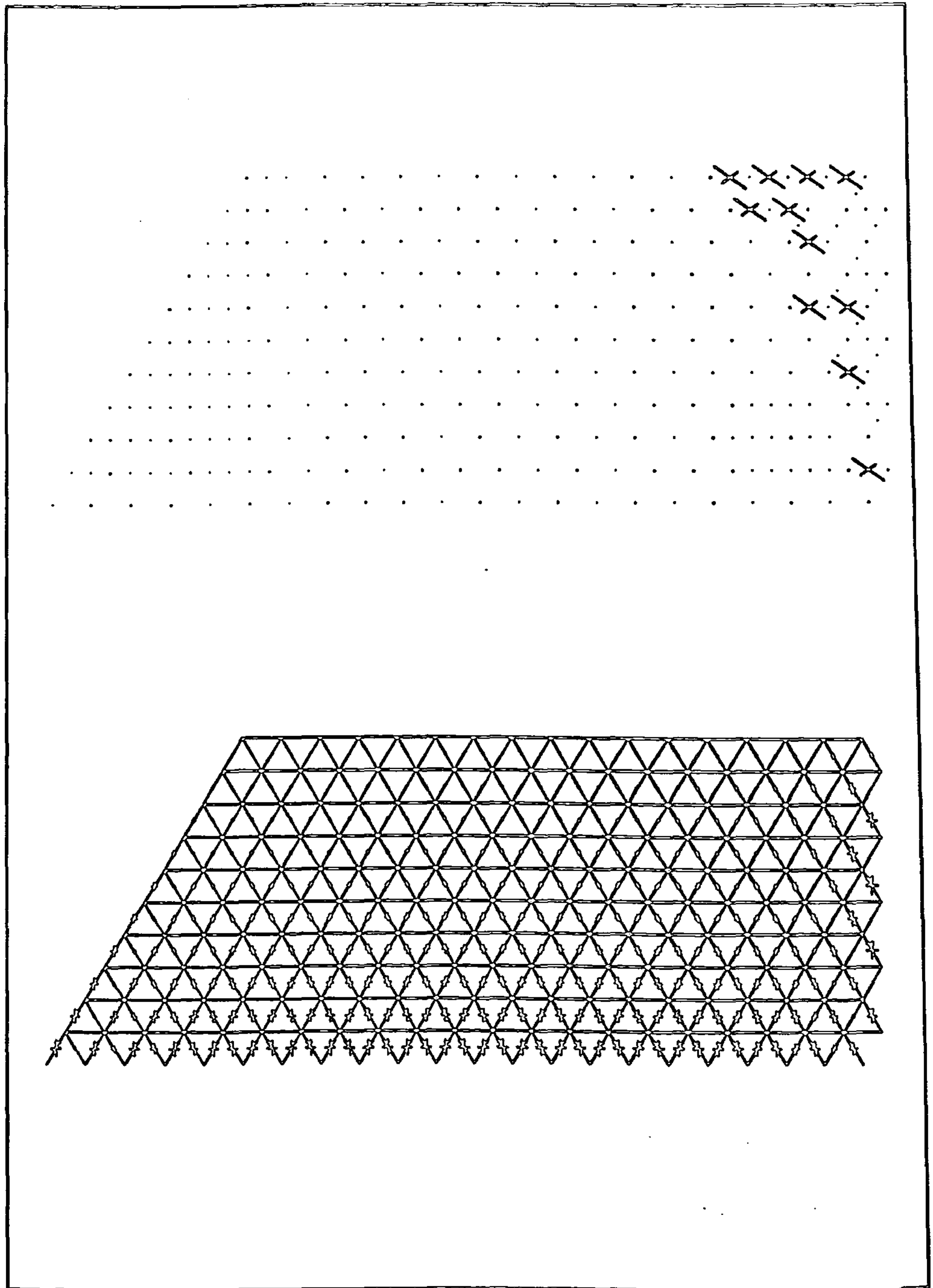


Figure 4.7 Analysis showing partial wedge failure

The effect of this correction can be seen in Figure 4.6. Here the effect is still found but in addition the outer diagonal edges failed in tension immediately with high tensile stresses. An informative analysis is that shown in Figure 4.7 where the righthand vertical slope begins to shows a wedge failure caused by sliding, as is to be expected. This at least shows some promise, however, the correction for the number of contacts, as applied currently does not have the desired effect and any soil like failure mechanisms are being obscured. It is presumed that the current damping implementation lies along the lines described, but due to time has not been elucidated.

CHAPTER 5

CONCLUSIONS

The work described here developed from an investigation of the vibrations associated with a traditional Distinct Element Analysis implementation. It was found that they could be eliminated from the system by implementing a consolidation type analysis. This consolidation technique has been described here by difference equations. The solution of these equations shows that the convergence of the system is governed by the stiffness, damping factor, gravity, contact length, number of active contacts, element mass and the time step. The solution also shows that the number of cycles required for convergence to a limiting value of force, acceleration or displacement may be calculated for simple cases.

The machine accuracy limits the propagation of effects through the system. Long hand expansion of the displacements, forces and acceleration of a simple system showed that the limit of propagation was affected largely by the time step. This led to the recommendation that the time step and stiffness are both unity so that the effects are not attenuated too quickly on passing through contacts.

To facilitate the continuing development of the Distinct Element Analysis the programs have been written in PASCAL for ease of amendment. These programs may be viewed as suitable for forming the basis for new implementations, requiring a change in the motion or force displacement laws to alter the media under investigation. This has a great advantage over FORTRAN equivalents where considerable effort is required for quite small amendments.

By including an input procedure that parses an input command language and allows interactive error correction the setting up of analysis problems is straight forward. This input method also prevents having to abort a run part way through due to a simple typing error. In conclusion, the programs written are flexible in terms of their ease of use, modification and utilisation as a base for modelling a different media.

The analysis carried out by Program SLICES is similar to the traditional method of slices and to the Rigid Block Model that formed the initial study. SLICES uses the consolidation technique developed to model soil slopes given a failure arc by dividing the slope into slices. The use of the program is therefore restricted to systems with a predetermined failure arc or to interactive use so that the least stable failure arc is found for a given slope. Currently the soil slope may not contain different layers of soil.

The validation was carried out by comparing the results from SLICES against a traditional method. Fifteen test cases have been used, five cases for each of three slopes. Each slope was tested under two cases of total stress and three cases of effective stress conditions. The last case utilised a non–linear failure criterion. Both frictional and cohesive soil types were modelled.

The testing showed that the failure conditions were most easily determined for frictional soils. The results from SLICES were similar to those from the traditional method, differing by one or two degrees in friction or kN/m^2 in cohesion. Where a difference larger than this occurred it was because a tension crack was predicted by SLICES that could not be modelled by the standard method. This shows an advantage of SLICES over the traditional method. A further advantage of SLICES is the ability to use a non–linear failure criterion. The results using this could not be compared with the traditional method but were consistent with the other SLICES results. This facility coupled with the ability to predict tension

cracks enables SLICES to give a more accurate indication of the behaviour of a slope than the traditional method.

A worthwhile enhancement to SLICES would be the incorporation of additional subcontacts on the inter–slice edges to allow for the modelling of soil masses with layers of different soil types. The contact failure laws need adjustment to reduce the rigid block behaviour of the slices and to increase the soil–like nature of the contacts. Some further work needs to be carried out in the validation of SLICES particularly in comparison with known case studies.

The development of the program CIRCLES has been constrained by time, predominantly in the later stages of validation. In Program CIRCLES the Distinct Element Analysis has been applied to soil in terms of circular areas of influence rather than as a physical model. There is no restriction upon the number of soil types modelled and a predetermined failure mechanism is not required.

The program has been shown to work adequately for simple cases but some difficulties have been encountered in applying it in general. The validation showed that edge effects caused an incorrect stress regime to be set up that masked the failure process. An attempt was made to rectify this by introducing an additional damping factor which was applied to the displacements of the circles. This factor was the reciprocal of the number of active contacts belonging to the circles, but proved to be only partially successful. However a sliding type failure was demonstrated where the edge effects seemed not to be strong enough to mask the effect.

Perhaps the most far reaching finding of the investigation into the edge effects is the impact that differing numbers of active contacts belonging to the circles can have. In terms of time this has the effect of promoting some circles ahead of others in the analysis and is rather like a 'time warp' occurring in the mesh.

Work needs to be carried out to investigate the edge effects further and a satisfactory correction procedure implemented. The starting point for this work could be an investigation into the behaviour of several different mesh configurations, for example, loosely packed, close packed and random. Two enhancements to CIRCLES to bring it in to line with SLICES would be the accommodation of effective stress, and the inclusion of a non-linear failure criterion for the contacts.

The aim of this work was to show that Distinct Element Analysis may be applied to soil masses. Unlikely though this may seem this has been achieved by Program SLICES which provides a more accurate indication of the slope behaviour than traditional methods. In view of this, despite the current edge effects shown by CIRCLES, the goal of modelling the generation of a failure zone in a soil slope is worth pursuing along these lines.

REFERENCES

Belytschko, T., Plesha, M.E., Dowding, C.H., 1983, *A Computer Method for the Stability Analysis of Caverns in Jointed Rock.* Proceedings of the International Conference on Constitutive Laws for Engineering Materials. January 1983. pp333–339

Bishop, A.W., 1955, *The Use of the Slip Circle in the Stability Analysis of Slopes.* Gèotechnique **v5** pp7–17.

Cundall, P.A., 1971, *A Computer Model for Simulating progressive, Large Scale Movements in Blocky Rock Systems.* Proceedings of the International Symposium on Rock Fracture. Nancy, France. (I.S.R.M.) Paper II–8.

Cundall, P.A., 1976, *Explicit Finite Difference Methods in Geomechanics.* A.S.C.E Engineering Conference Numerical Methods in Geomechanics. Blacks burg, Virginia. pp. 132–150.

Cundall, P.A., Strack, O.D.L., 1979, *Discrete Numerical Model for Granular Assemblages.* Geotechnique, **v29** No1 March 1979, pp47–65.

Dames and Moore., 1978, *Computer Modelling of Jointed Rock Masses.* Dames and Moore Advanced Technical Group Technical Report no N–78–4, August 1978.

Dowding, C.H., Belytschko, T., Yen, Y.J., 1983, *Dynamic Computational Analysis of Openings in Jointed Rock.* Journal of Geotechnical Engineering **v109**, pp1551–1566.

Dowding, C.H., Belytschko, T., Yen, Y.J., 1983, *A Coupled Finite–Element–Rigid Block Method for Transient Analysis of Rock Caverns.* International Journal for Numerical and Analytical methods in Geomechanics v7, pp117–127.

Fellenius, W., 1936, *Calculation of Stability of Earth Dams.* Transactions of the 2nd. Congress on Large Dams.

Garrard, G.F.G. 1984 *The Collapse of Shallow Mine Workings.* Ph. D. Thesis, University of Durham.

Goodman, R.E., 1976, *Methods of Geological Engineering in Discontinuous Rocks.* West Publishing Company, St. Paul, MN.

Grogono, P., 1980, *Programming in Pascal.* Revised Edition, Addison–Wesley. Reading, Mass.

Janbu, N., 1973, *Slope Stability Computations.* in Hirschfeld, R.C., Poulos, S.J. (Ed.) *Embankment — Dam Engineering*, Casagrande Volume, J. Wieby and Sons, pp47–86.

Lorig, L.J.A., Brady, B.H.G., 1982, *A Hybrid Discrete Element Boundary Element Method of Stress Analysis.* Issues in Rock Mechanics, Univ. California, Berkeley. pp628–636.

Lorig, L.J.A., Brady, B.H.G., 1983, *An Improved Procedure for Excavation Design in Stratified Rock.* in *Rock Mechanics — Theory — Experiment — Practice*, 24th US Rock Mechanics Symposium. Texas A and M University College Station. pp577–586.

Lorig, L.J.A., Brady, B.H.G., 1984, *A Hybrid Computational Scheme for Excavation Support Design in Jointed Rock Media.* I.S.R.M Symposium — *Design and Performance of Underground Excavations.* pp105–112. Cambridge UK.

Lorig, L.J.A., Brady, B.H.G., Cundall, P.A., 1986, *Hybrid Distinct Element — Boundary Element Analysis of Jointed Rock.* International Journal of Rock Mechanics Mining Science and Geomechanical Abstracts. **v23** No 4 August 1986 pp303–312.

Lysmer, J. and Kuhlemeyer, R.L., 1969, *Finite Dynamic Model for Infinite Media.* Journal of the Engineering Mechanics Division, ASCE, **v95**, No. EM4, August 1969, pp859–877.

Meek, J.L. and Beer, G., 1984, *A Review of Analysis Techniques for the Determination of Stresses around and Performance of Excavations in Hard Rock.* Chapter 1. pp1–10. in *Design and Performance of Underground Excavations.* I. S. R. M. /B. G. S., Cambridge.

Michigan Terminal System, Volume 2, 1981, *Public File Descriptions.* Published by the University of Michigan Computing Centre, Ann Arbor, Michigan, October 1981.

Plesha, M.E., 1986, *Mixed Time Integration for the Transient Analysis of Jointed Media.* International Journal of Numerical and Analytical Methods in Geomechanics, **v10**, No 1, January - February 1986, pp91–110.

Rouse, K.A., 1982 *A Computer Program for Modelling Jointed Rock Mass Behaviour.* M.Sc. Thesis, University of Durham.

Voegele, M., 1978, *An Interactive Graphics - Based Analysis of the Support Requirements of Excavations in Jointed Rock Masses.* Ph.D. Thesis, University of Minnesota.

Watson, C.R., 1983, *A Program Designed to Model the Behaviour of Discontinuous Rock Masses.* M.Sc. Thesis, University of Durham.

Wilkins, M.L., 1969, *Calculation of Elastic - Plastic flow.* Report UCRL–7322 Revision I, Lawrence Radiation Laboratory, University of California, Livermore.

APPENDIX A

MATHEMATICAL NOTATION

$|x|$ is the absolute value of x.

$\widehat{x}$ is the maximum value of x.

$\vee$ is the logical **or** operator.

$\wedge$ is the logical **and** operator.

$\rightarrow$ tends to.

$\Rightarrow$ implies that.

α is the angle the failure arc makes with the horizontal at the toe of a slope.

a is acceleration.

A is a controlling constant in difference equation solutions.

β is the angle between circle centres and the horizontal.

B is a controlling constant in difference equation solutions.

c is cohesion.

C is consolidation force.

d is the numerical damping factor.

D is the distance between circle centres.

D_f is the overall numerical damping factor.

ϕ is a soil parameter representing the angle of friction.

F is contact force.

$Force$ is force acting on the centroid of elements.

g is gravity.

G_o is the initial radial distance between centres.

G_r is the radial distance between centres.

ΔG is the change in the radial distance between centres.

H is dashpot damping force.

I is the number of active contacts an element has.

k is contact stiffness.

K is contact dashpot stiffness.

L is contact length.

Lim is the limiting difference value of force, stress and so on.

μ is the coefficient of friction.

m is mass.

M is element movement from the original position.

N is the number of cycles for a limiting difference to be reached.

P is the coordinate position of element centroid.

σ is stress.

σ_1 is the major principle stress.

σ_3 is the minor principle stress.

S is the increment of displacement occurring in a specific time step.

τ is shear stress.

t is total time.

θ is angle between block edge and the horizontal.

δt is the time step.

u is pore water pressure,.

v is velocity.

The following symbols are used as subscripts.

c prefixed to a quantity refers to a block contributing the corner to a contact.

e prefixed to a quantity refers to a block contributing the edge to a contact.

1 prefixing the quantity refers to a specific distinct element.

a, f, σ or s applied to Lim refers to the quantity at limiting difference.

n refers to the direction of normal movement on a contact.

r refers to the radial direction.

s refers to the direction of shear movement.

x refers to the x direction.

y refers to the y direction.

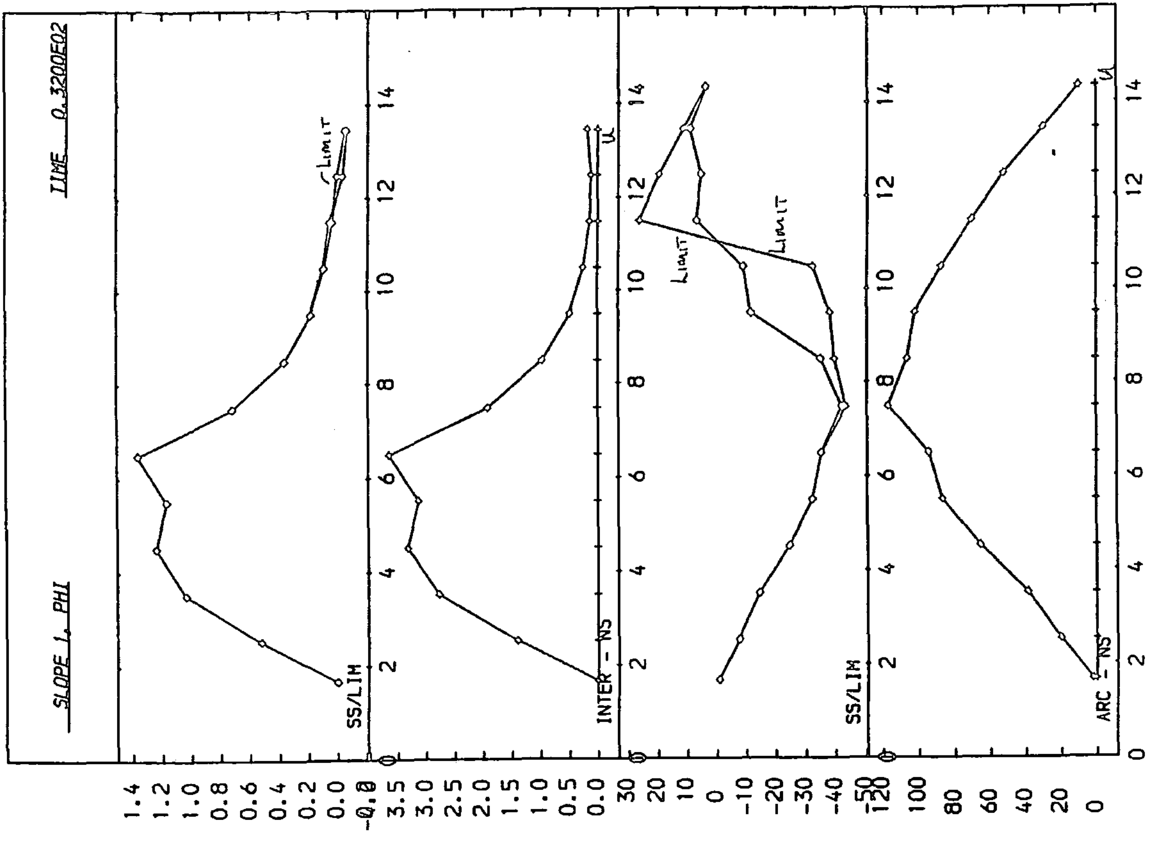
n as a preceeding superscript refers to a specific calculation cycle.

APPENDIX B

SLICE RESULTS

```
set echo off damp 0.05 0.2
start  SLOPE 1, PHI
  0 16 0
   create free  0.0 20.5 2.0 1.0 0.0 20.5 0.0 0.0 0.0  1 14.0  1 14
                                                       2 10.5  2 14
   create free  0.0 20.5 2.0 1.0 0.0 20.5 0.0 0.0 0.0  3  8.9  3 14
   create free  0.0 20.5 2.0 1.0 0.0 20.5 0.0 0.0 0.0  4  7.7  4 14
   create free  0.0 20.5 2.0 1.0 0.0 20.5 0.0 0.0 0.0  5  6.9  5 14
   create free  0.0 20.5 2.0 1.0 0.0 20.5 0.0 0.0 0.0  6  6.3  6 14
   create free  0.0 20.5 2.0 1.0 0.0 20.5 0.0 0.0 0.0  7  5.7  7 14
   create free  0.0 20.5 2.0 1.0 0.0 20.5 0.0 0.0 0.0  8  5.4  8 13
   create free  0.0 20.5 2.0 1.0 0.0 20.5 0.0 0.0 0.0  9  5.1  9 12
   create free  0.0 20.5 2.0 1.0 0.0 20.5 0.0 0.0 0.0 10  5.0 10 11
   create free  0.0 20.5 2.0 1.0 0.0 20.5 0.0 0.0 0.0 11  4.9 11  9.9
   create free  0.0 20.5 2.0 1.0 0.0 20.5 0.0 0.0 0.0 12  5.0 12  8.9
   create free  0.0 20.5 2.0 1.0 0.0 20.5 0.0 0.0 0.0 13  5.1 13  7.8
   create free  0.0 20.5 2.0 1.0 0.0 20.5 0.0 0.0 0.0 14  5.4 14  6.8
   create free  0.0 20.5 2.0 1.0 0.0 20.5 0.0 0.0 0.0 15  5.8 15  5.8
   meshend
  set damp 0.05 0.2 time 1 gravity -10 cmdproc on framelimit 100
    writegap 32 interval 32
    cmdlist plot standard set calc writegap * 2 interval * 2 cend
    echo on
  go 16383  stop
```

Table 3.14 Input commands for result set 1



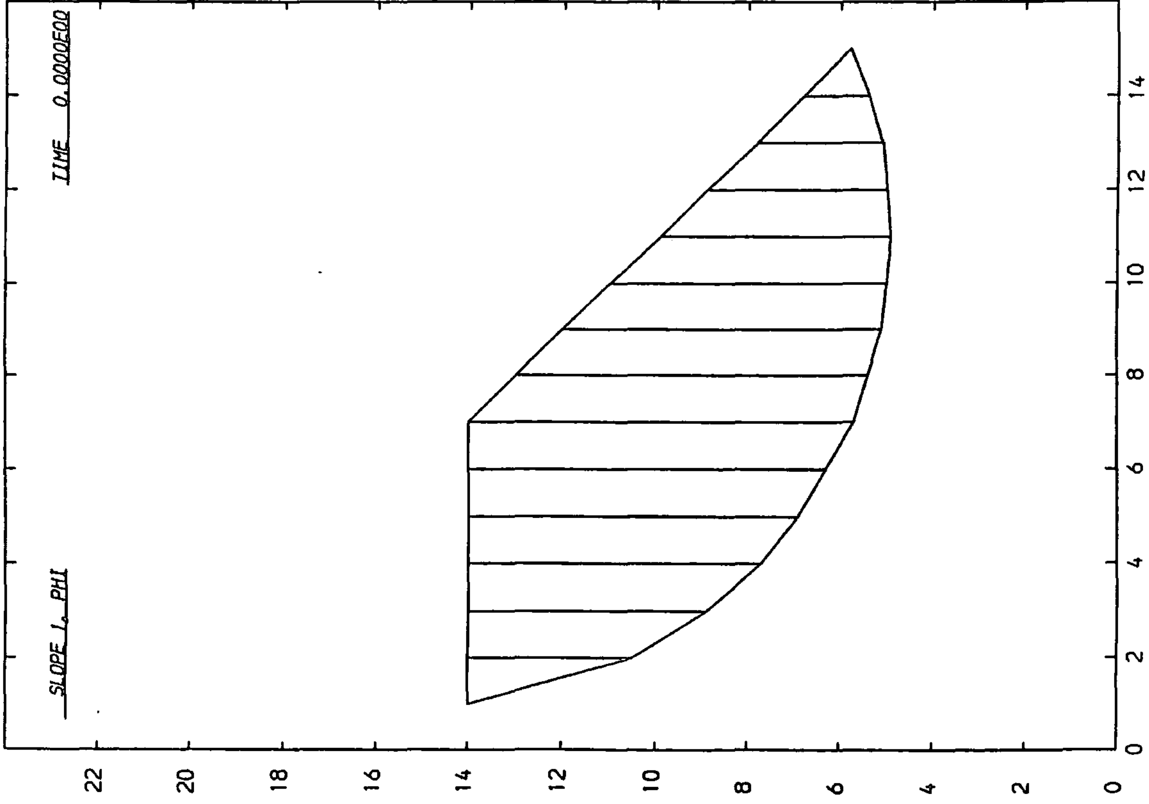
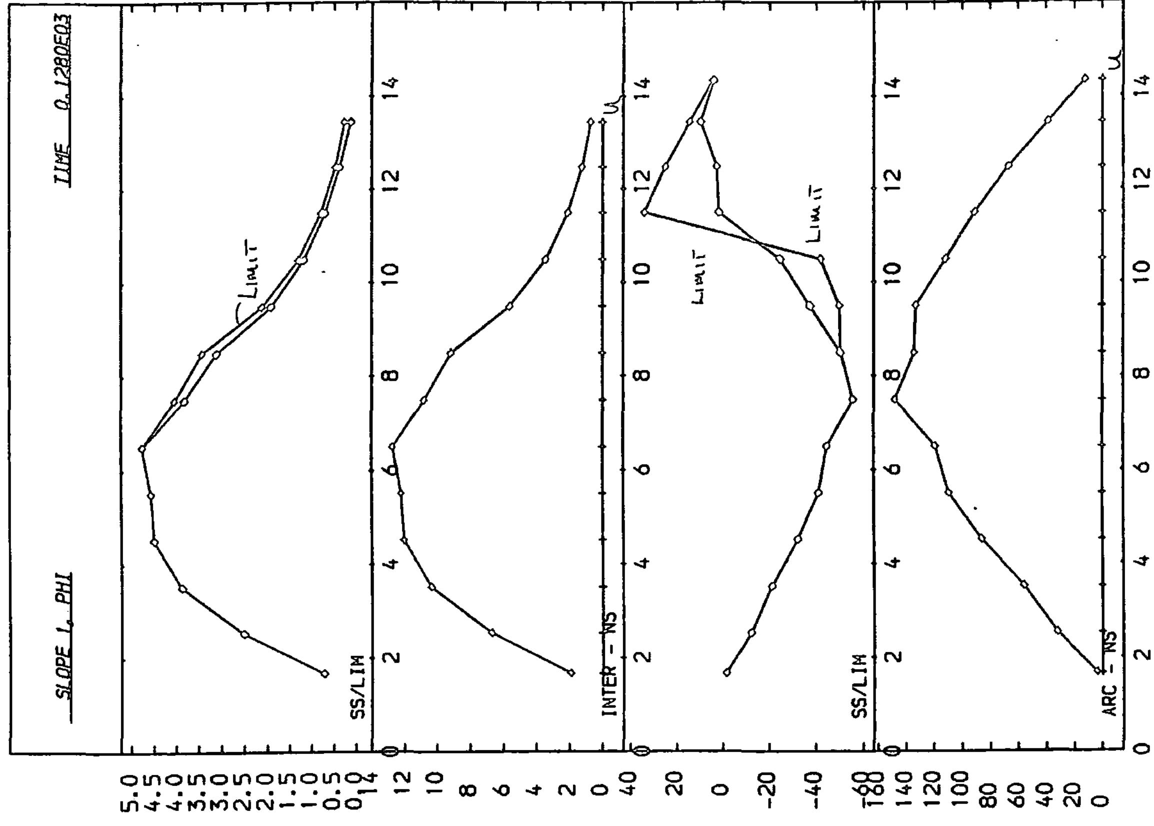


Figure 3.6 Stress profiles for result set 1



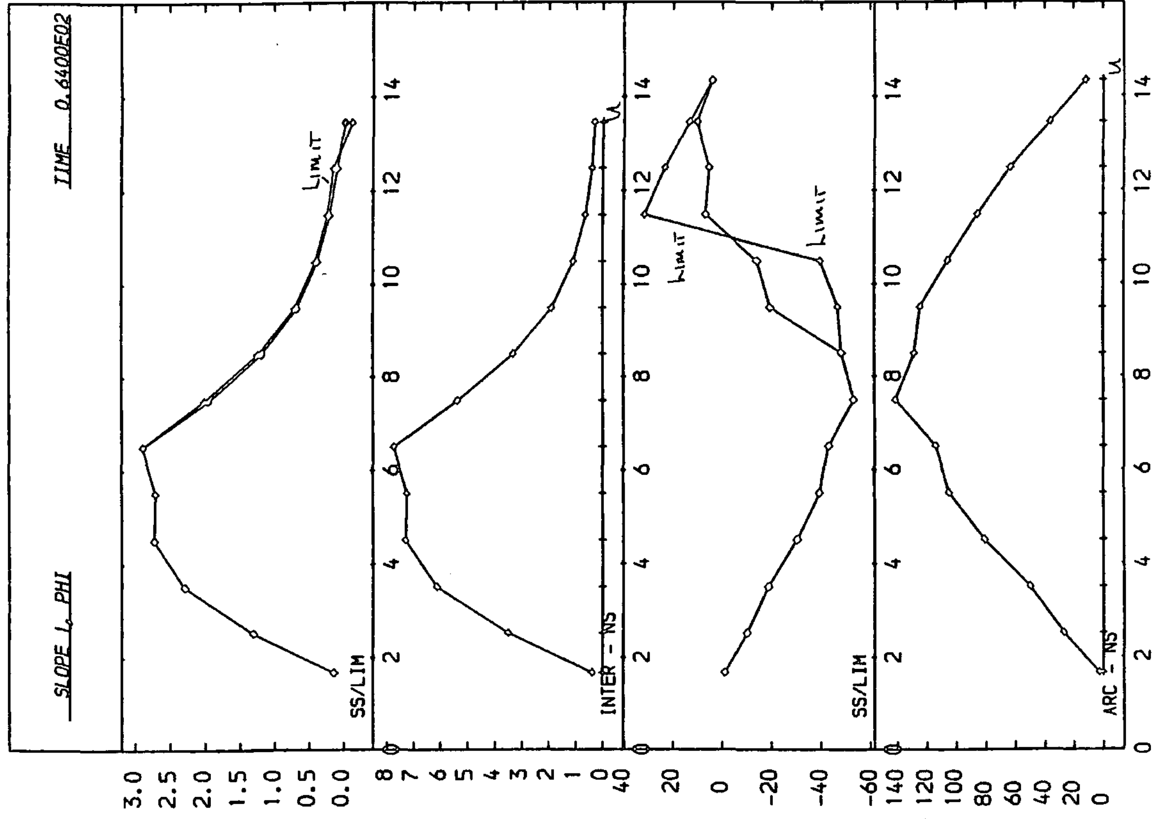
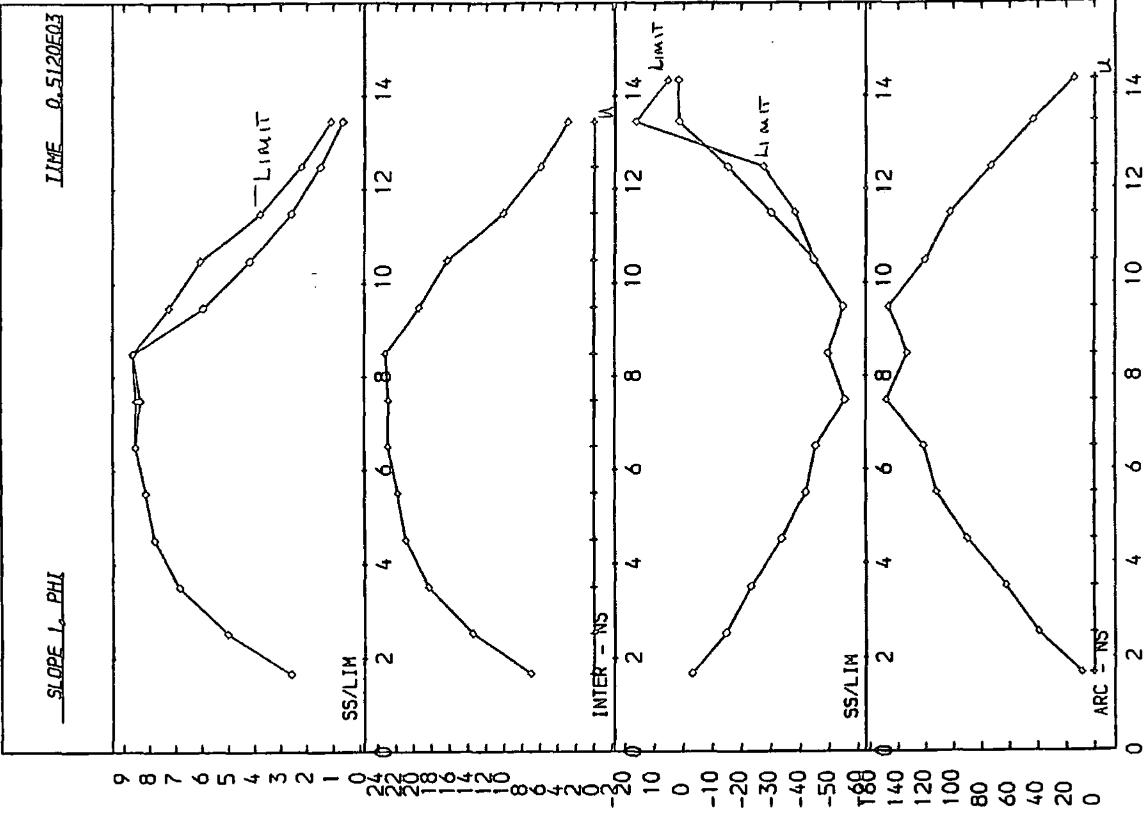


Figure 3.6 Stress profiles for result set 1 (continued)



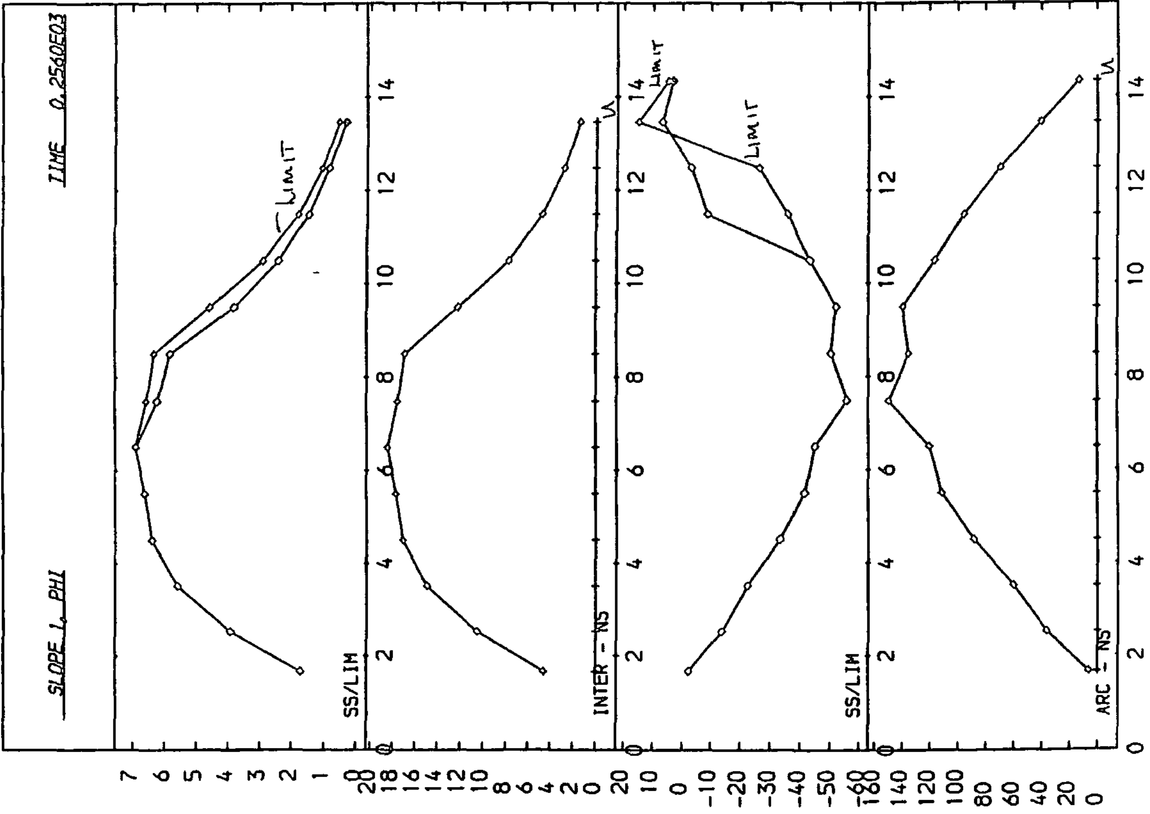
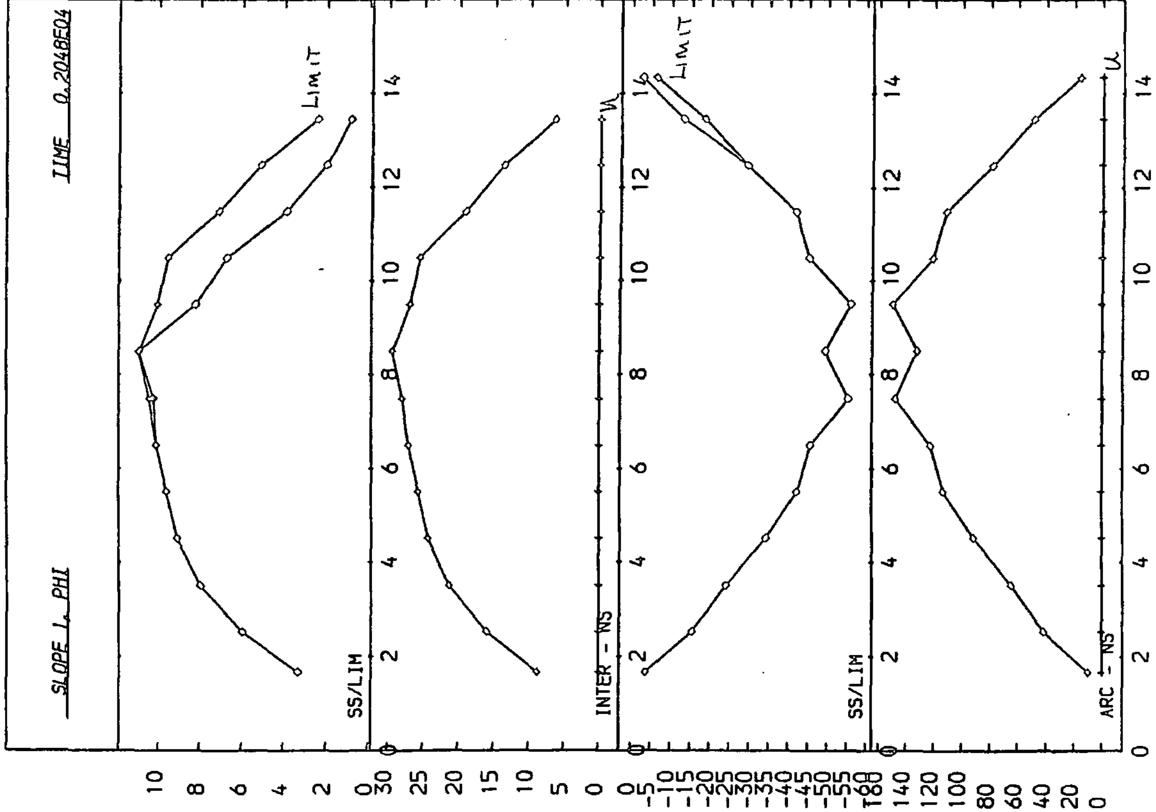


Figure 3.6 Stress profiles for result set 1 (continued)



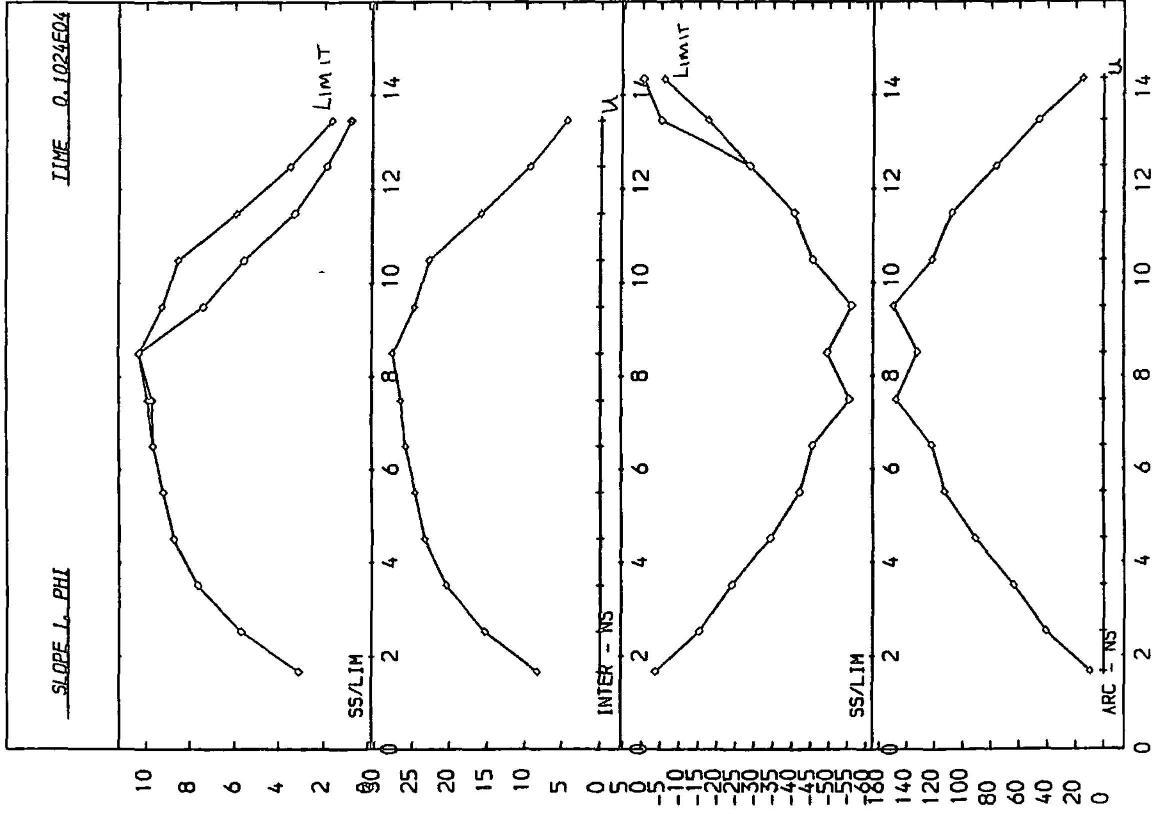
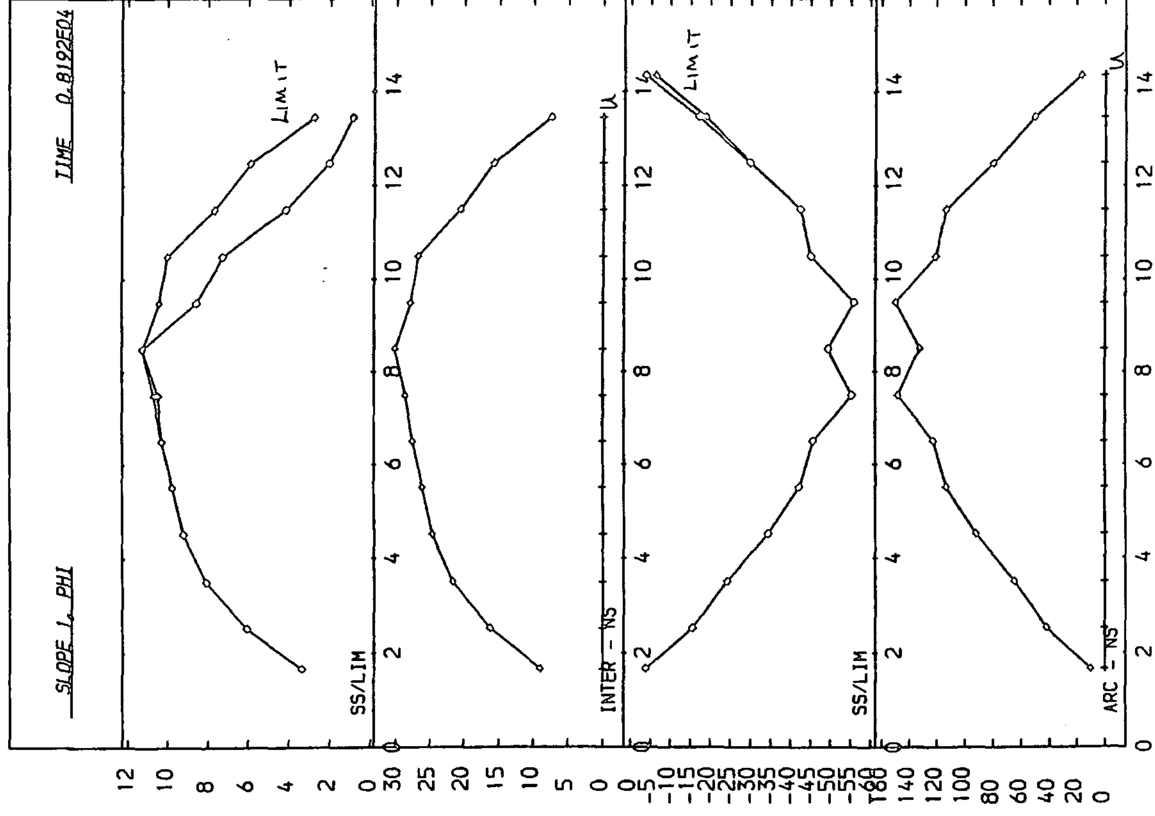


Figure 3.6 Stress profiles for result set 1 (continued)



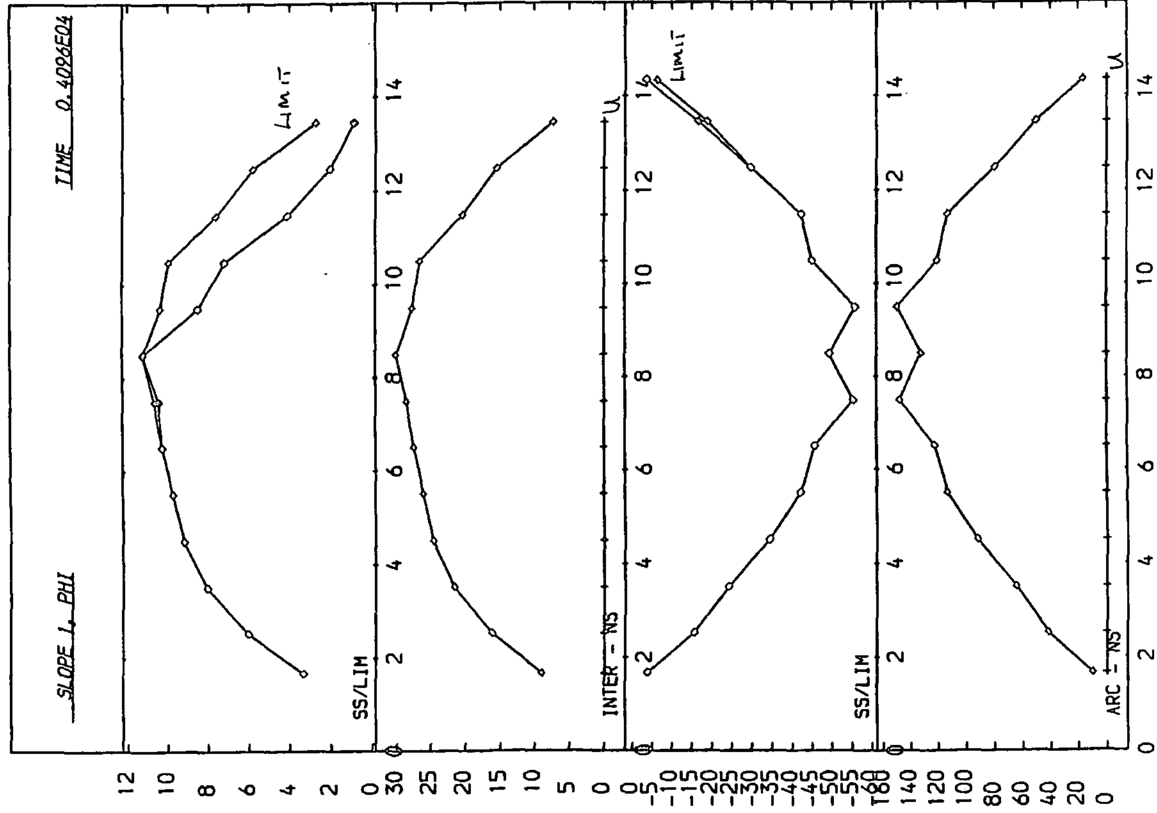


Figure 3.6 Stress profiles for result set 1 (continued)

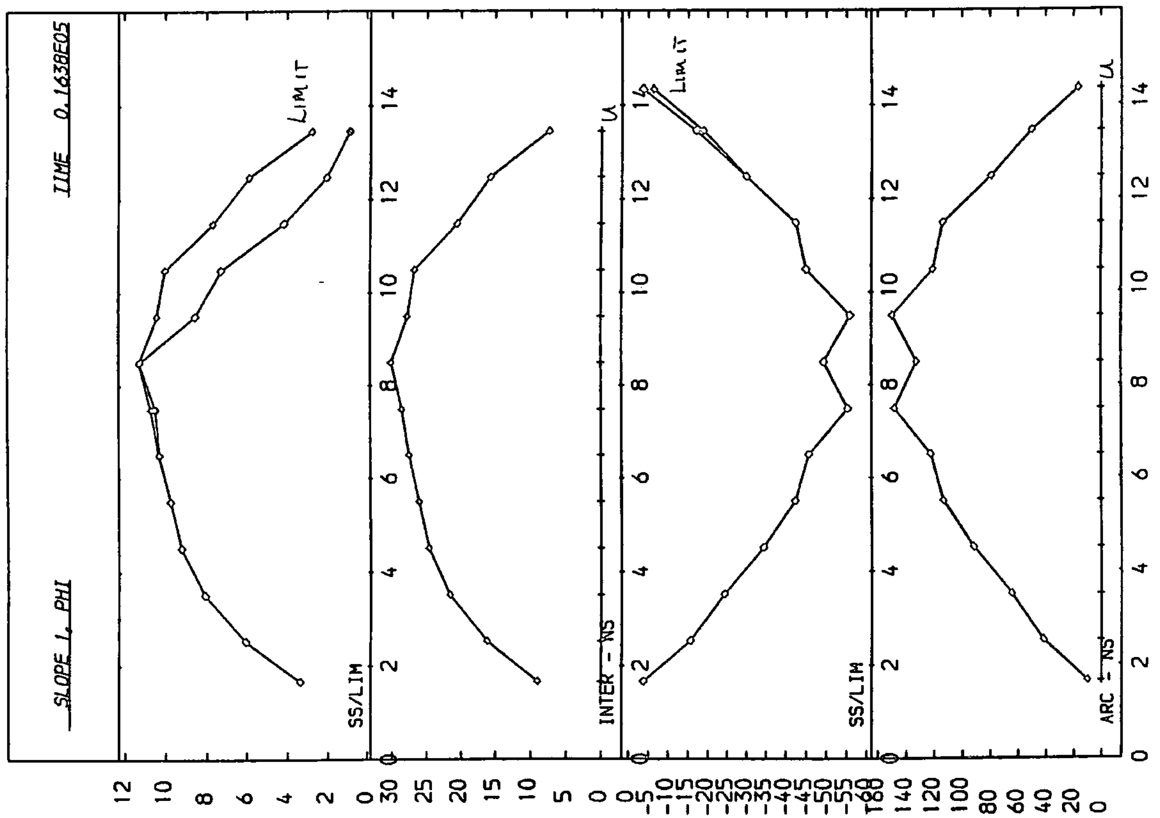
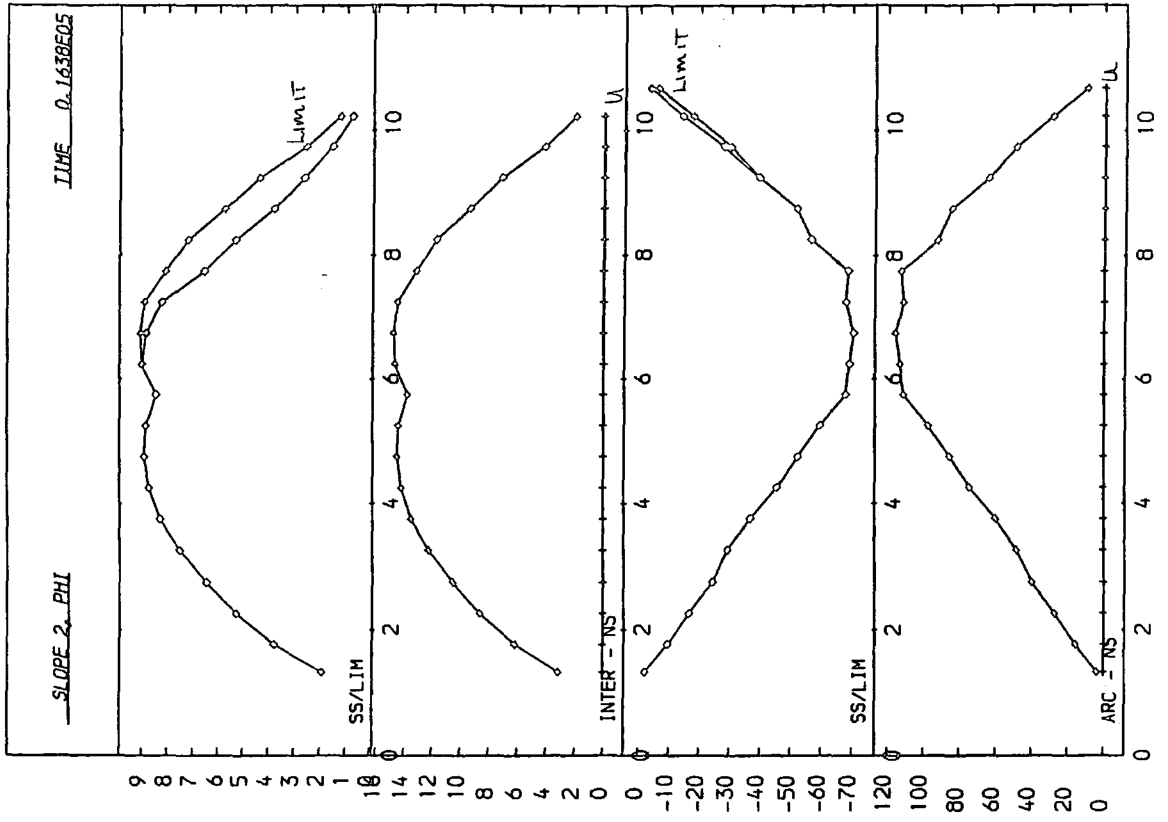


Figure 3.6 Stress profiles for result set 1 (continued)

```
set echo off damp 0.05 0.2
start  SLOPE 2, PHI
  0 12 0
   create free  0.0 31.5 2.0 1.0 0.0 31.5 0.0 0.0 0.0  1.0 14.0   1.0 14
                                                       1.5 11.8   1.5 14
   create free  0.0 31.5 2.0 1.0 0.0 31.5 0.0 0.0 0.0  2.0 10.6   2.0 14
   create free  0.0 31.5 2.0 1.0 0.0 31.5 0.0 0.0 0.0  2.5  9.7   2.5 14
   create free  0.0 31.5 2.0 1.0 0.0 31.5 0.0 0.0 0.0  3.0  9.0   3.0 14
   create free  0.0 31.5 2.0 1.0 0.0 31.5 0.0 0.0 0.0  3.5  8.35  3.5 14
   create free  0.0 31.5 2.0 1.0 0.0 31.5 0.0 0.0 0.0  4.0  7.8   4.0 14
   create free  0.0 31.5 2.0 1.0 0.0 31.5 0.0 0.0 0.0  4.5  7.35  4.5 14
   create free  0.0 31.5 2.0 1.0 0.0 31.5 0.0 0.0 0.0  5.0  6.95  5.0 14
   create free  0.0 31.5 2.0 1.0 0.0 31.5 0.0 0.0 0.0  5.5  6.6   5.5 14
   create free  0.0 31.5 2.0 1.0 0.0 31.5 0.0 0.0 0.0  6.0  6.3   6.0 14
   create free  0.0 31.5 2.0 1.0 0.0 31.5 0.0 0.0 0.0  6.5  6.025 6.5 13
   create free  0.0 31.5 2.0 1.0 0.0 31.5 0.0 0.0 0.0  7.0  5.8   7.0 12.1
   create free  0.0 31.5 2.0 1.0 0.0 31.5 0.0 0.0 0.0  7.5  5.6   7.5 11.2
   create free  0.0 31.5 2.0 1.0 0.0 31.5 0.0 0.0 0.0  8.0  5.45  8.0 10.3
   create free  0.0 31.5 2.0 1.0 0.0 31.5 0.0 0.0 0.0  8.5  5.3   8.5  9.5
   create free  0.0 31.5 2.0 1.0 0.0 31.5 0.0 0.0 0.0  9.0  5.2   9.0  8.6
   create free  0.0 31.5 2.0 1.0 0.0 31.5 0.0 0.0 0.0  9.5  5.1   9.5  7.7
   create free  0.0 31.5 2.0 1.0 0.0 31.5 0.0 0.0 0.0 10.0  5.05  10.0 6.8
   create free  0.0 31.5 2.0 1.0 0.0 31.5 0.0 0.0 0.0 10.5  5.0   10.5 5.85
   create free  0.0 31.5 2.0 1.0 0.0 31.5 0.0 0.0 0.0 11.0  5.0   11   5.0
   meshend
  set damp 0.05 0.2 time 1 gravity -10 cmdproc on framelimit 100
    writegap 32 interval 32
    cmdlist plot standard set calc writegap * 2 interval * 2 cend
    echo on
  plot page go 16383  stop
```

Table 3.15 Input commands for result set 2



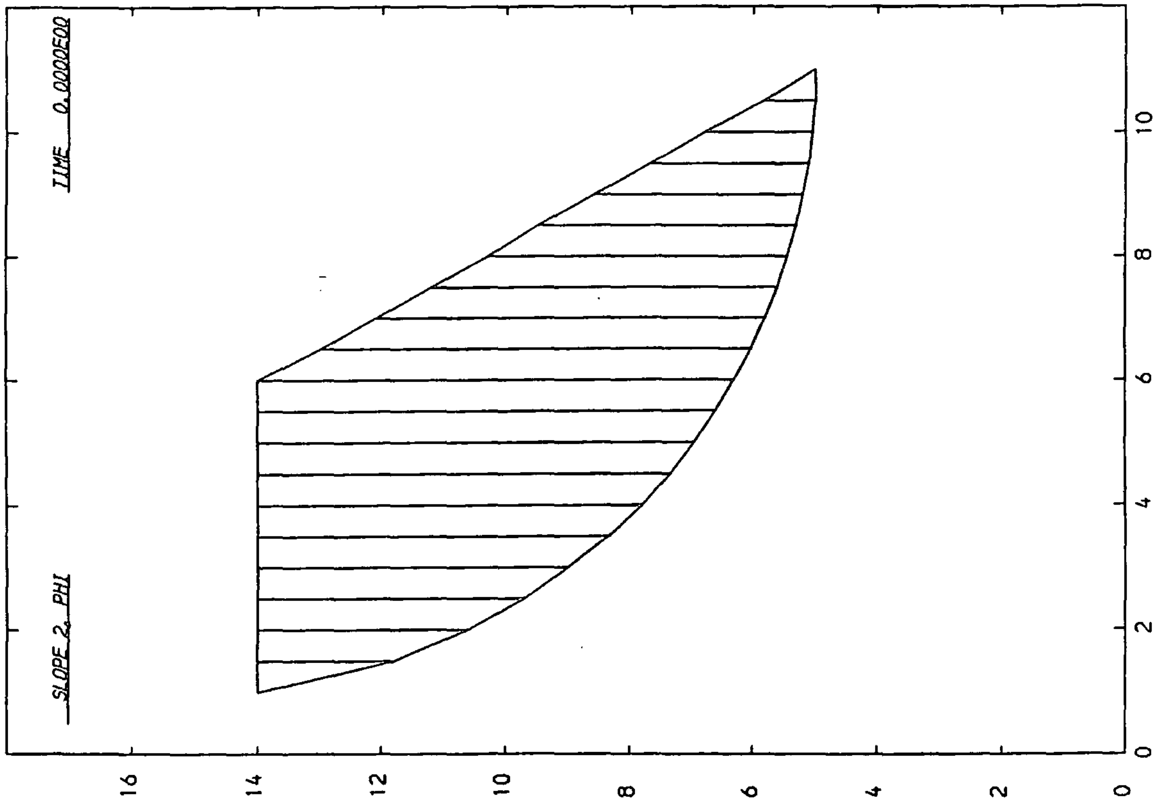
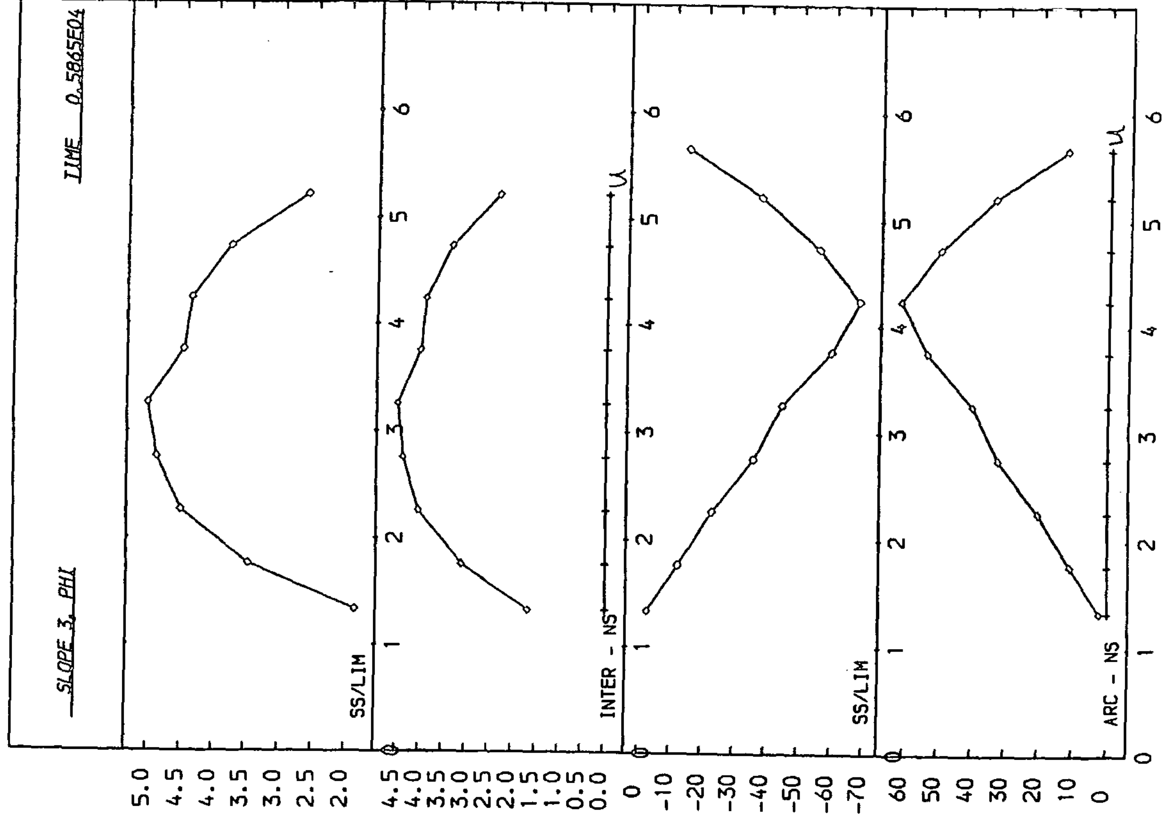


Figure 3.7 Stress profiles for result set 2

```
set echo off damp 0.05 0.2
start  SLOPE 3, PHI
  0 7  0
   create track  0.0 48.0 2.0 1.0 0.0 48.0 0.0 0.0 0.0  1.0 14.0  1.0 14
                                                        1.5 11.8  1.5 14
   create track  0.0 48.0 2.0 1.0 0.0 48.0 0.0 0.0 0.0  2.0 10.6  2.0 14
   create track  0.0 48.0 2.0 1.0 0.0 48.0 0.0 0.0 0.0  2.5  9.7  2.5 14
   create track  0.0 48.0 2.0 1.0 0.0 48.0 0.0 0.0 0.0  3.0  9.0  3.0 14
   create track  0.0 48.0 2.0 1.0 0.0 48.0 0.0 0.0 0.0  3.5  8.35 3.5 14
   create track  0.0 48.0 2.0 1.0 0.0 48.0 0.0 0.0 0.0  4.0  7.8  4.0 14
   create track  0.0 48.0 2.0 1.0 0.0 48.0 0.0 0.0 0.0  4.5  7.35 4.5 12
   create track  0.0 48.0 2.0 1.0 0.0 48.0 0.0 0.0 0.0  5.0  6.95 5.0 10
   create track  0.0 48.0 2.0 1.0 0.0 48.0 0.0 0.0 0.0  5.5  6.6  5.5  8
   create track  0.0 48.0 2.0 1.0 0.0 48.0 0.0 0.0 0.0  6.0  6.3  6.0  6.3
   meshend
  set damp 0.05 0.2 time 1 gravity -10 cmdproc on framelimit 100
    writegap 32 interval 32
    cmdlist plot standard set calc writegap * 2 interval * 2 cend
    echo on debug oscil on
  plot page go 16383  stop
```

Table 3.16 Input commands for result set 3



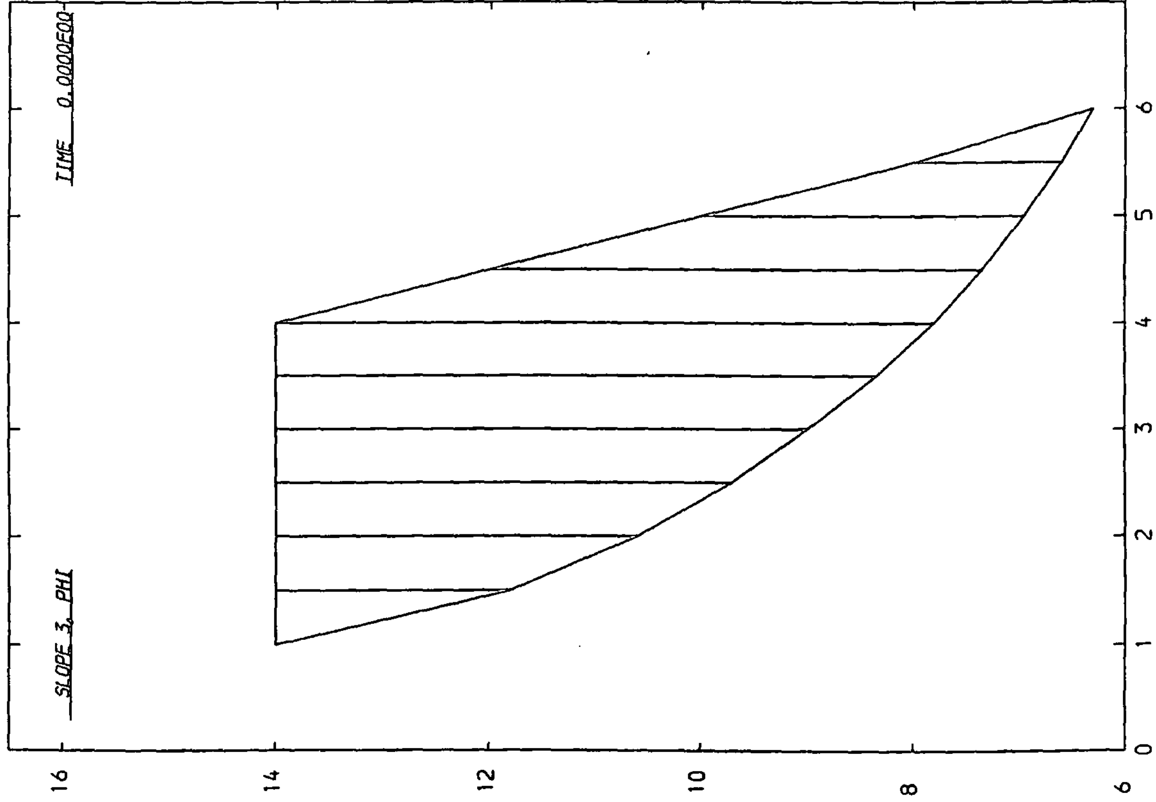
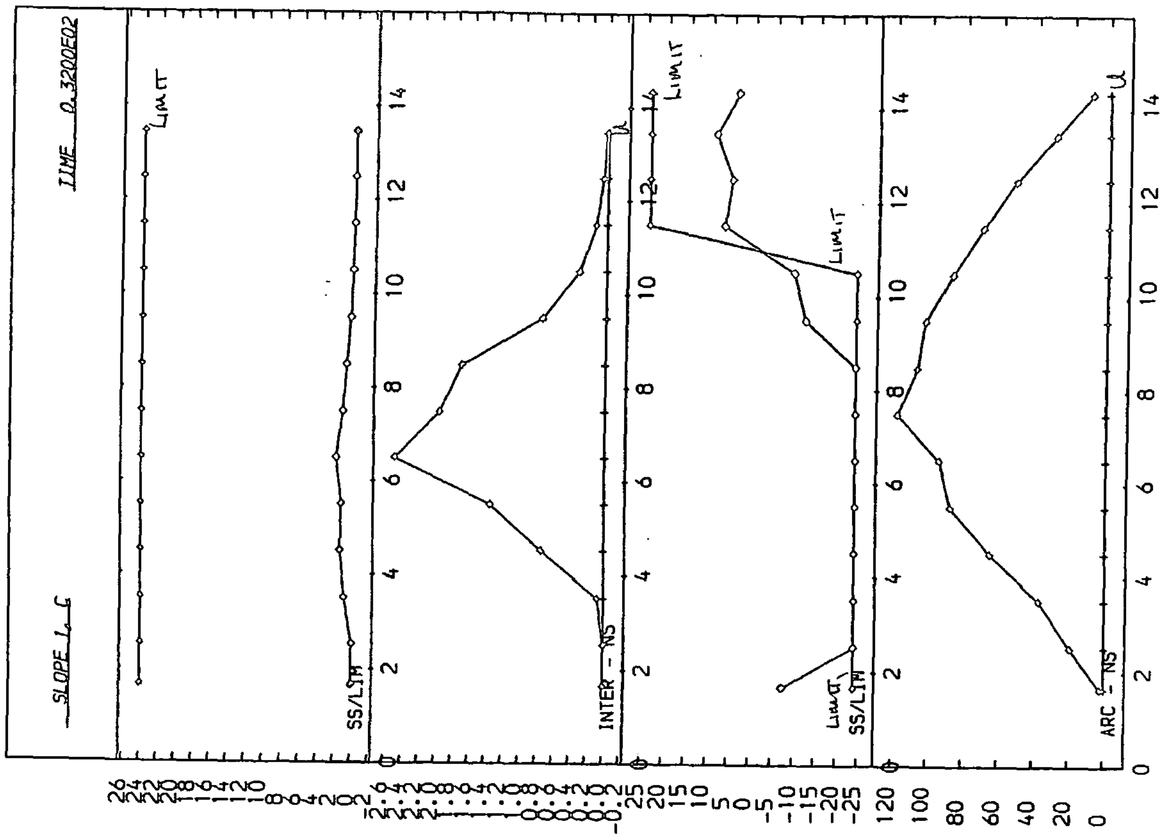


Figure 3.8 Stress profiles for result set 3

```
set echo off damp 0.05 0.2  debug update on
start  SLOPE 1, C
  0 16 0
   create free 24.0 0 2.0 1.0 24.0 0 0.0 0.0 0.0 1 14.0  1 14
                                                 2 10.5  2 14
   create free 24.0 0 2.0 1.0 24.0 0 0.0 0.0 0.0 3  8.9  3 14
   create free 24.0 0 2.0 1.0 24.0 0 0.0 0.0 0.0 4  7.7  4 14
   create free 24.0 0 2.0 1.0 24.0 0 0.0 0.0 0.0 5  6.9  5 14
   create free 24.0 0 2.0 1.0 24.0 0 0.0 0.0 0.0 6  6.3  6 14
   create free 24.0 0 2.0 1.0 24.0 0 0.0 0.0 0.0 7  5.7  7 14
   create free 24.0 0 2.0 1.0 24.0 0 0.0 0.0 0.0 8  5.4  8 13
   create free 24.0 0 2.0 1.0 24.0 0 0.0 0.0 0.0 9  5.1  9 12
   create free 24.0 0 2.0 1.0 24.0 0 0.0 0.0 0.0 10 5.0 10 11
   create free 24.0 0 2.0 1.0 24.0 0 0.0 0.0 0.0 11 4.9 11  9.9
   create free 24.0 0 2.0 1.0 24.0 0 0.0 0.0 0.0 12 5.0 12  8.9
   create free 24.0 0 2.0 1.0 24.0 0 0.0 0.0 0.0 13 5.1 13  7.8
   create free 24.0 0 2.0 1.0 24.0 0 0.0 0.0 0.0 14 5.4 14  6.8
   create free 24.0 0 2.0 1.0 24.0 0 0.0 0.0 0.0 15 5.8 15  5.8
   meshend
  set time 1 gravity -10 cmdproc on framelimit 100
  debug slices contacts general stop
    writegap 32 interval 32
    cmdlist plot standard set calc writegap * 2 interval * 2 cend
    echo on
  go 16383  stop
```

Table 3.17 Input commands for result set 4



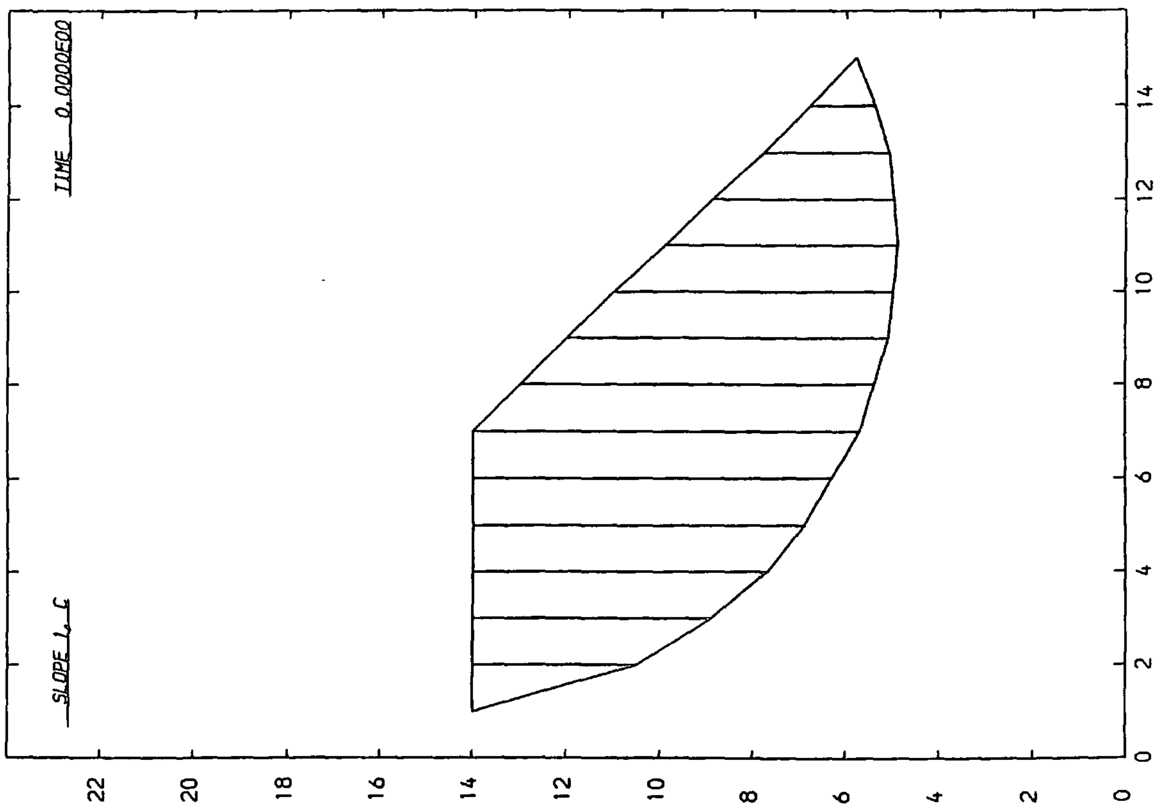
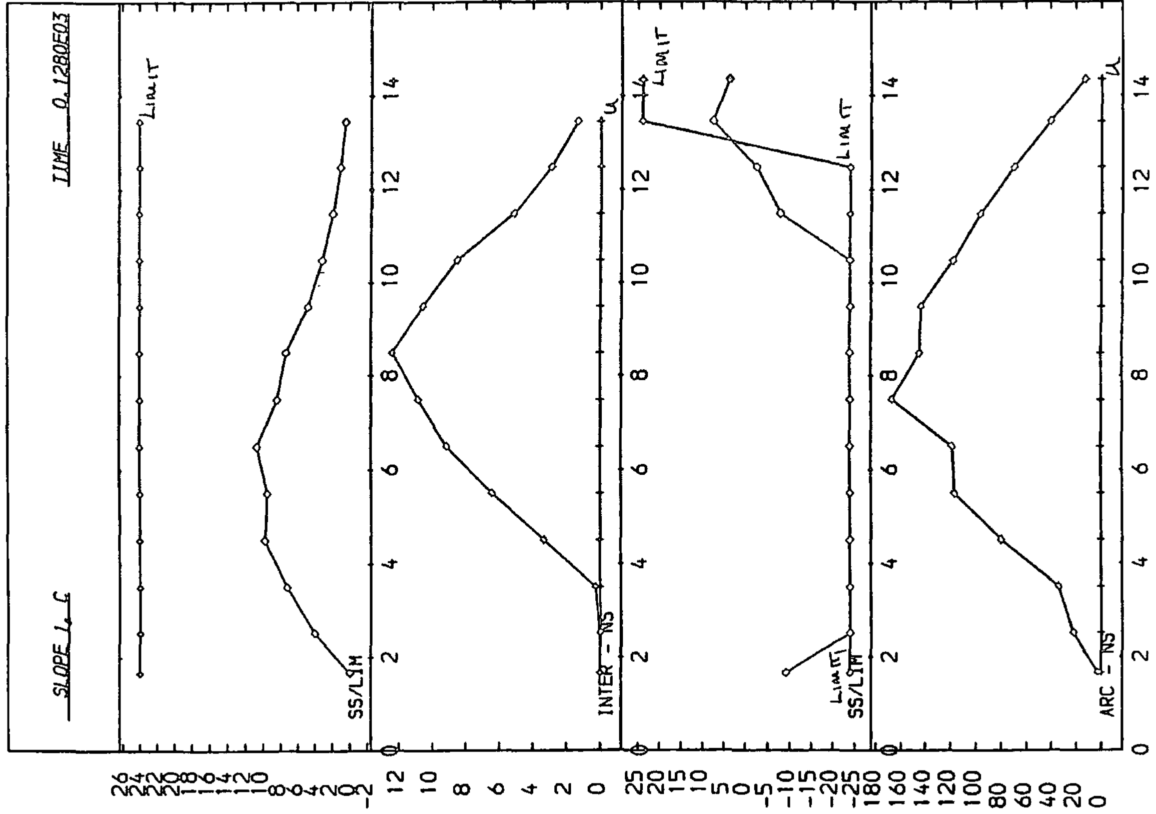


Figure 3.9 Stress profiles for result set 4



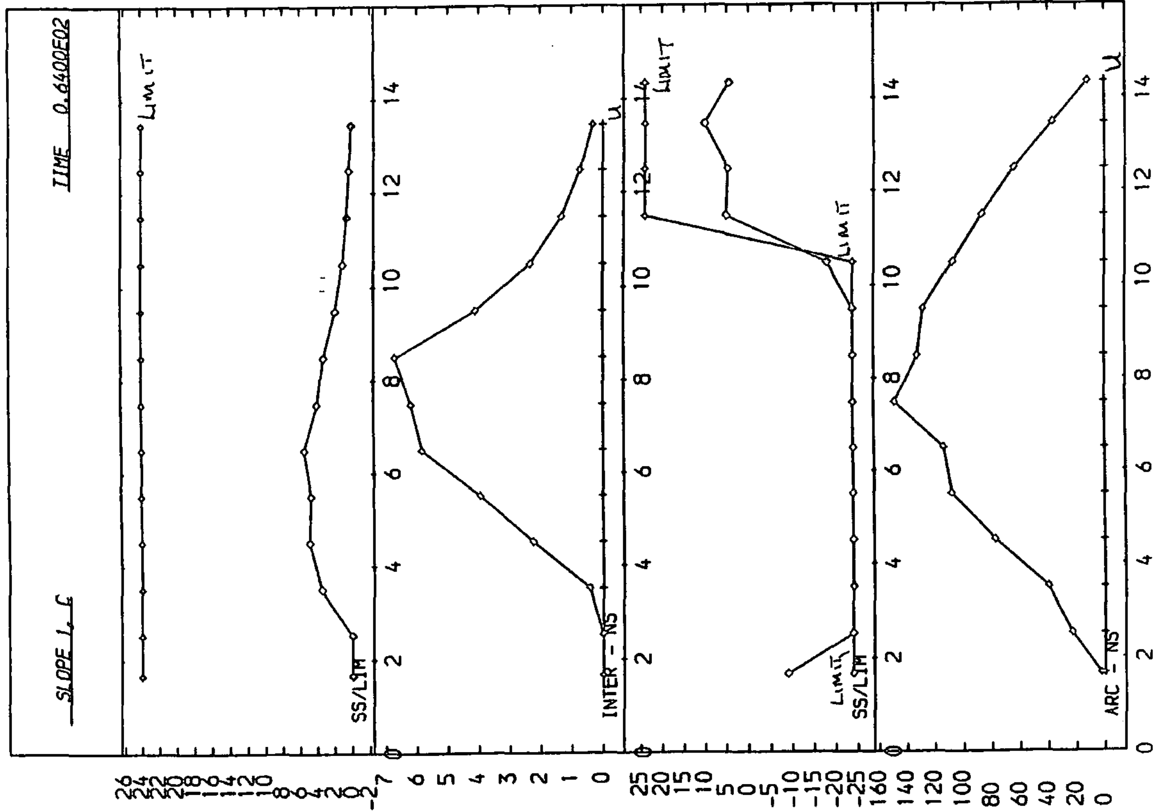
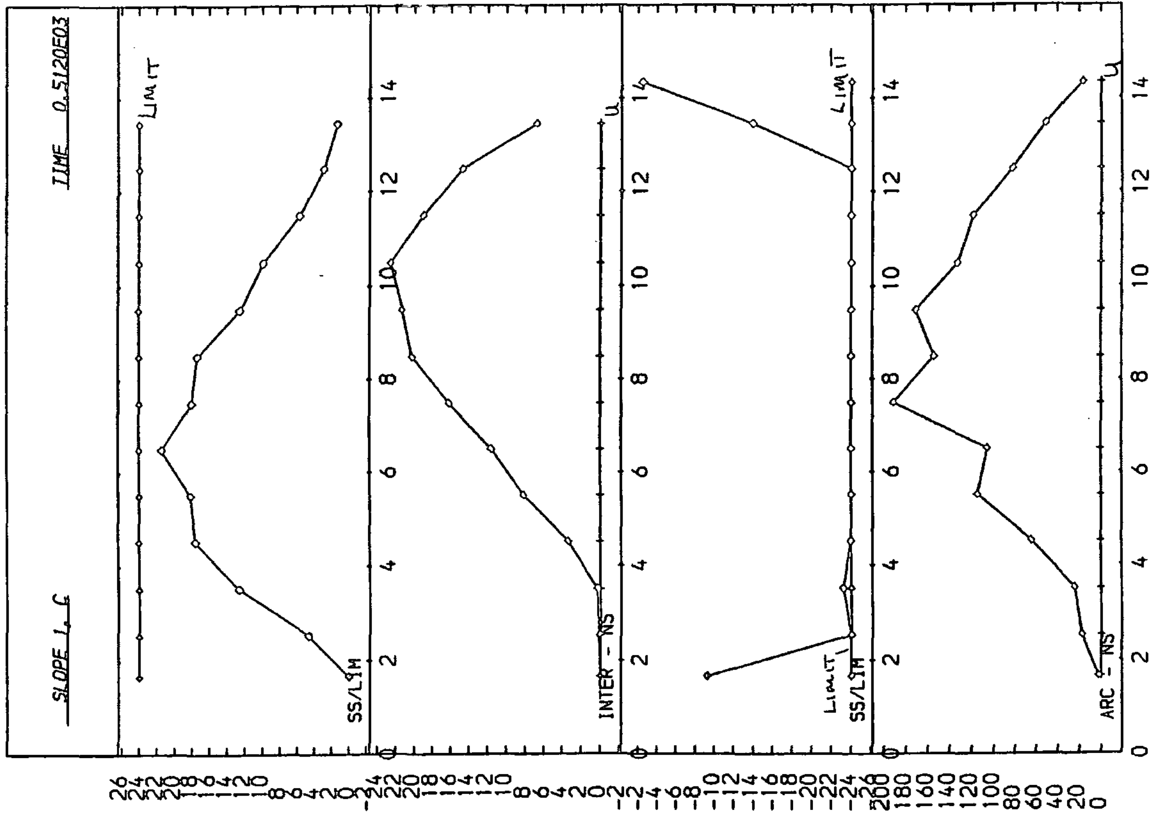


Figure 3.9 Stress profiles for result set 4 (continued)



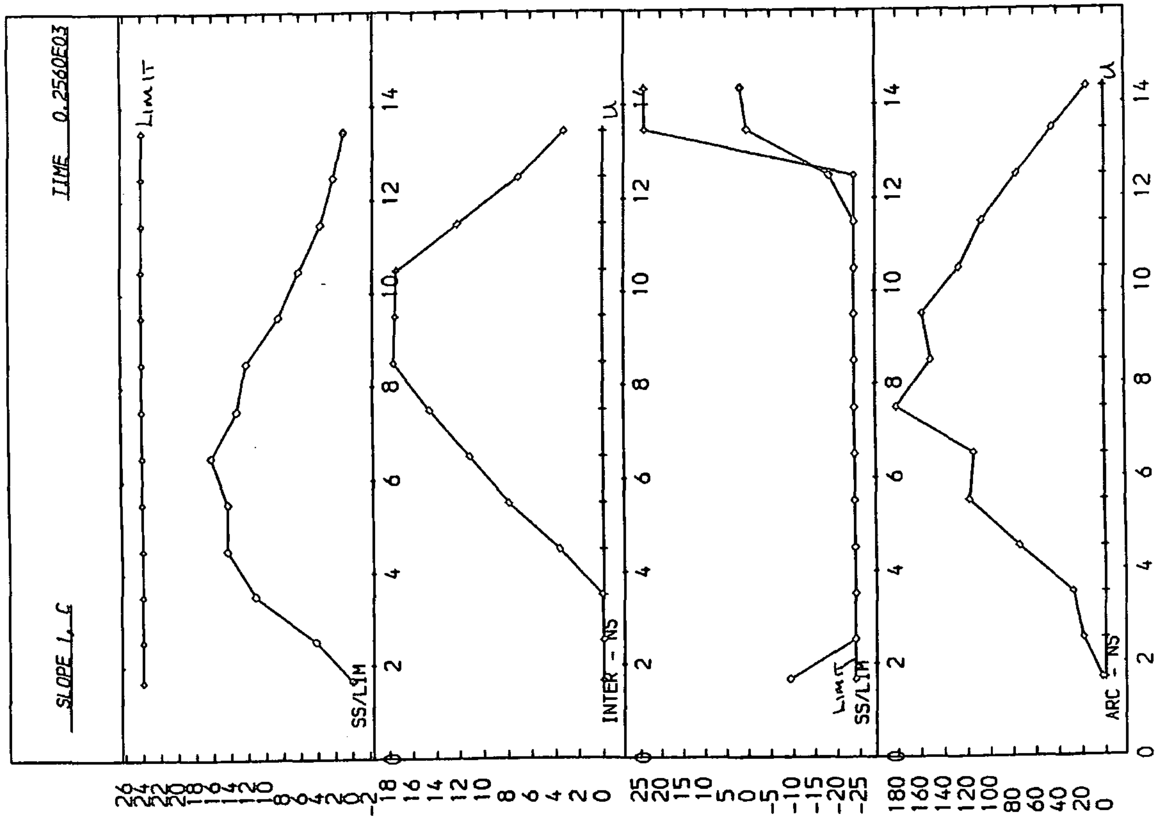
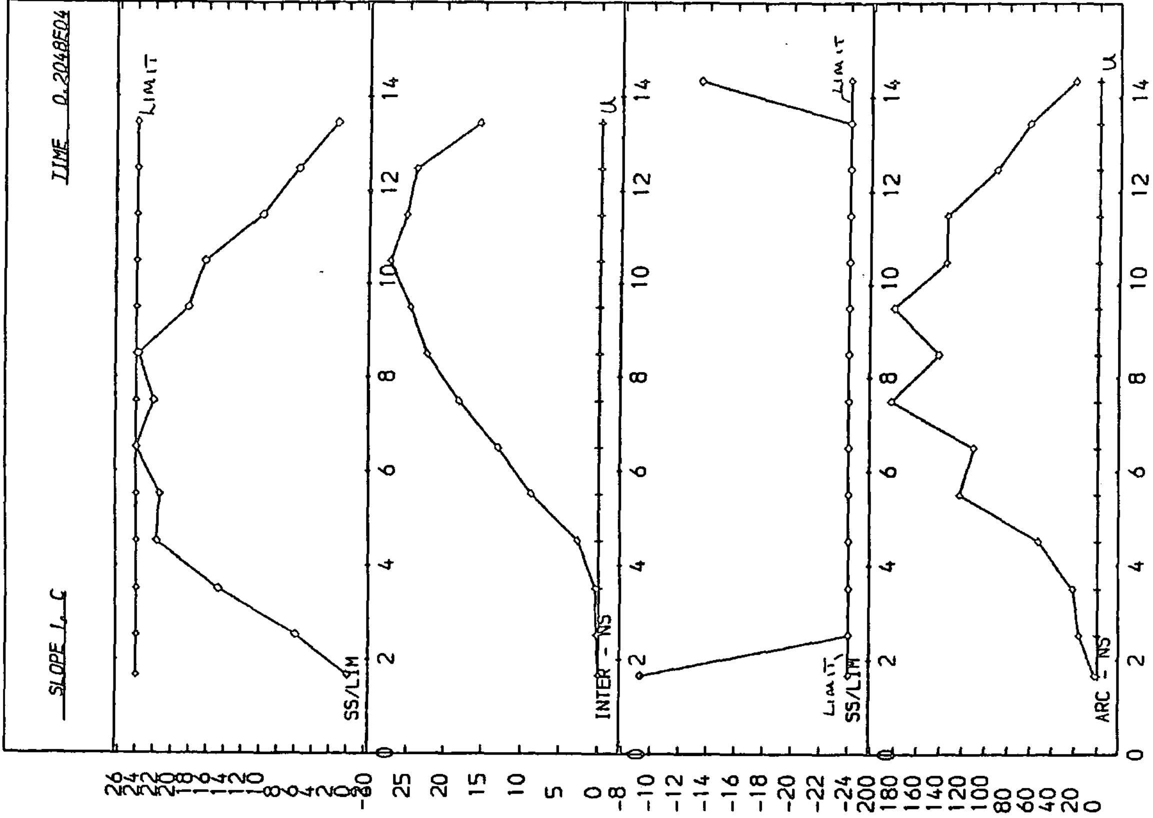


Figure 3.9 Stress profiles for result set 4 (continued)



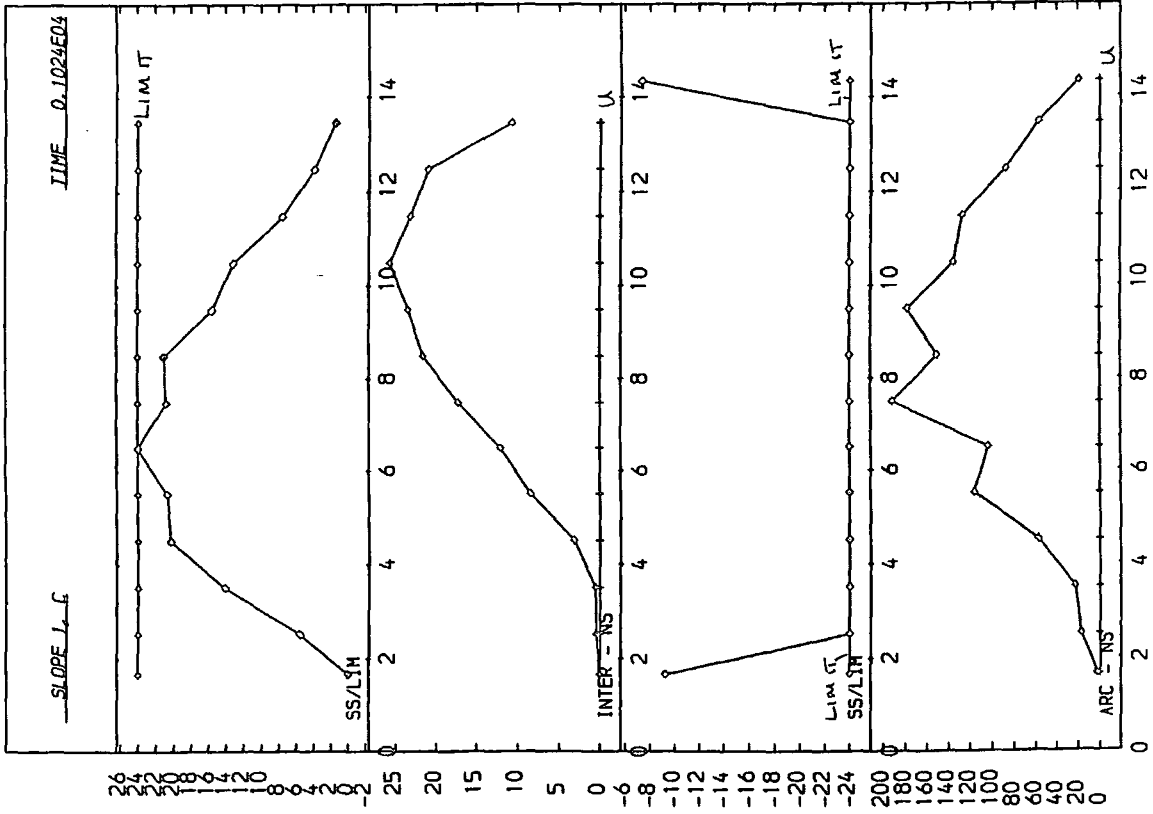
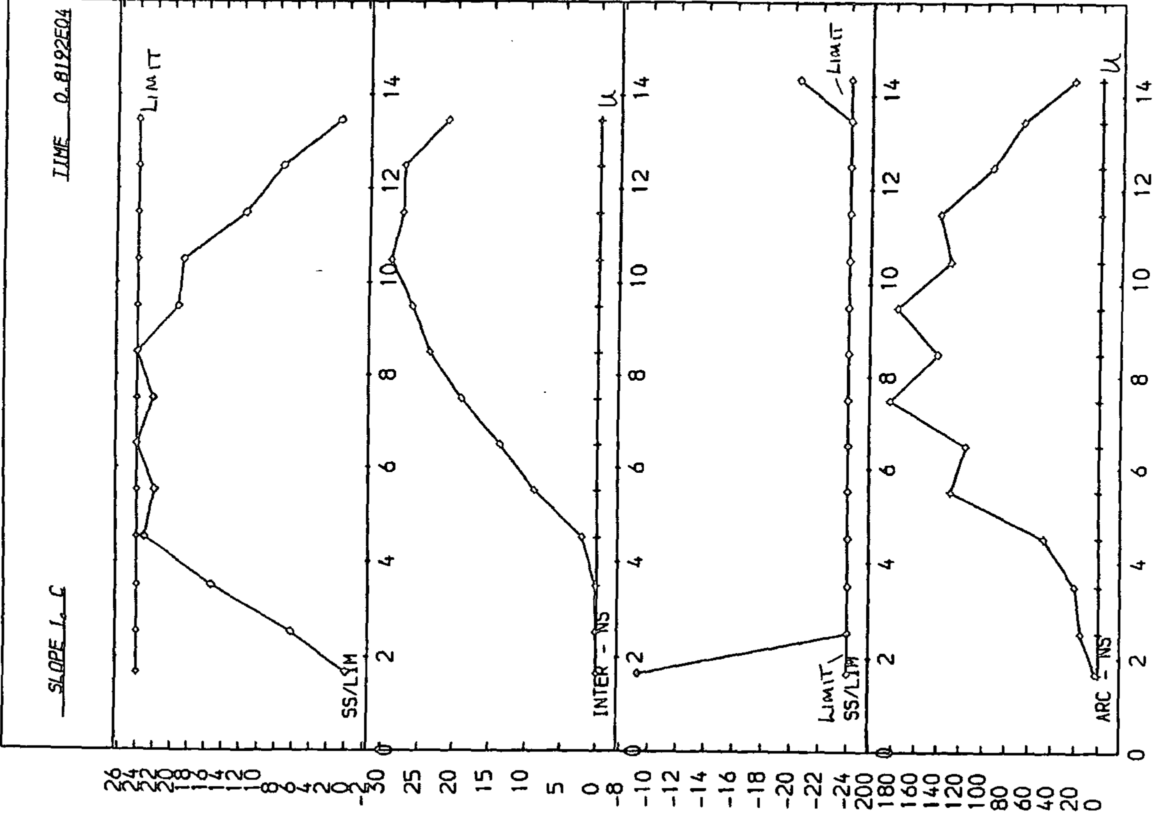


Figure 3.9 Stress profiles for result set 4 (continued)



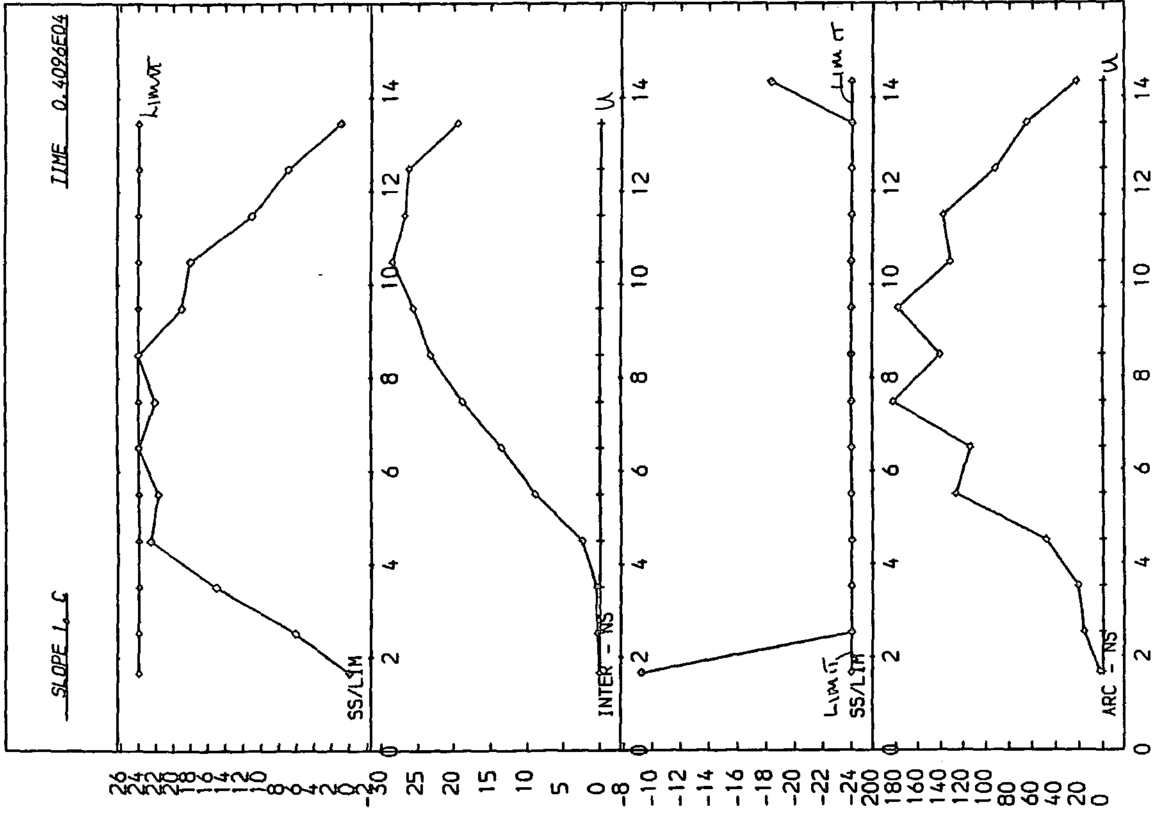


Figure 3.9 Stress profiles for result set 4 (continued)

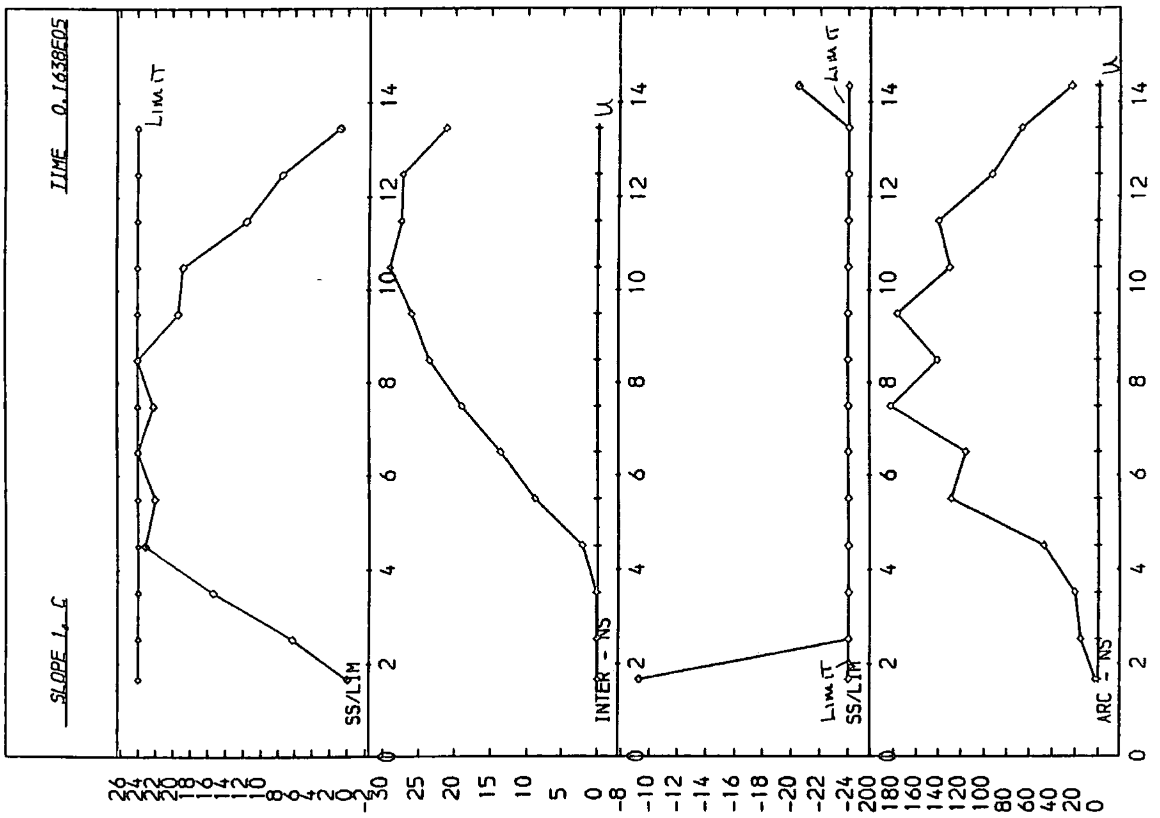
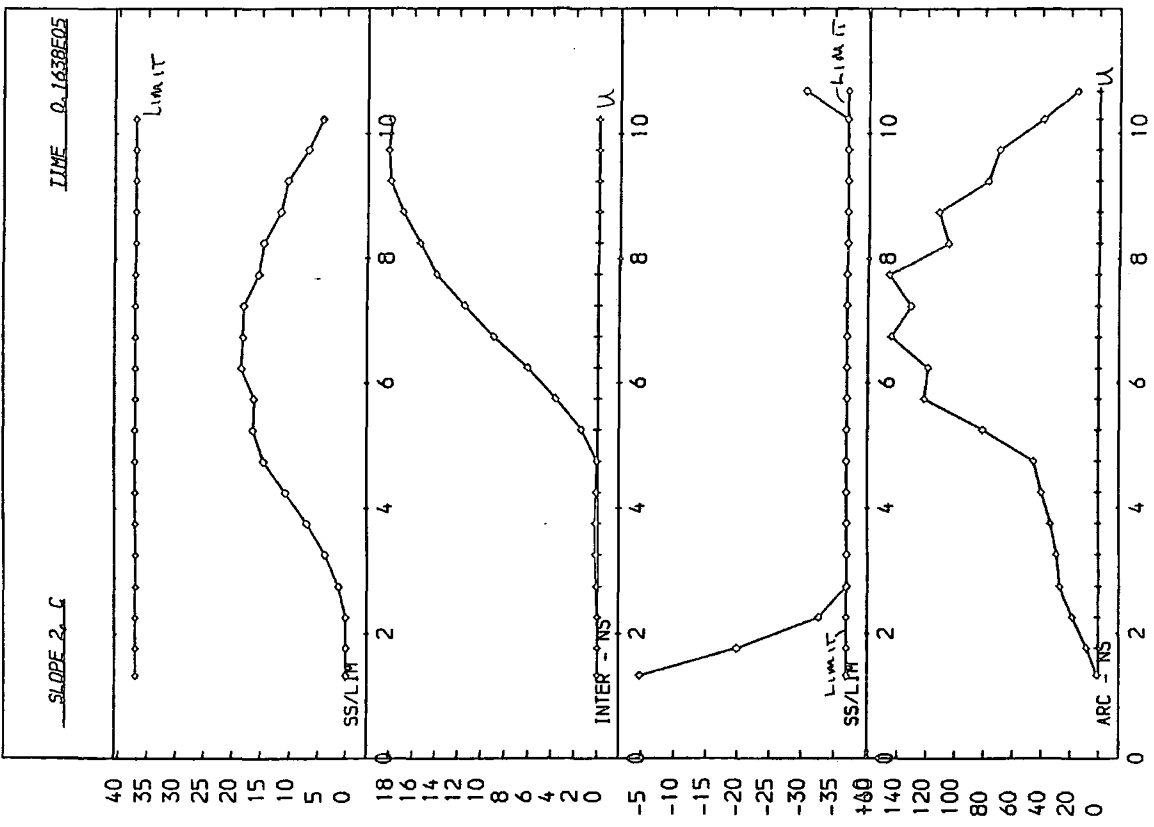


Figure 3.9 Stress profiles for result set 4 (continued)

```
set echo off damp 0.025 0.1
start  SLOPE 2, C
  0 12 0
   create free 37.0 0.0 2.0 1.0 37.0 0.0 0.0 0.0 0.0  1.0 14.0   1.0 14
                                                      1.5 11.8   1.5 14
   create free 37.0 0.0 2.0 1.0 37.0 0.0 0.0 0.0 0.0  2.0 10.6   2.0 14
   create free 37.0 0.0 2.0 1.0 37.0 0.0 0.0 0.0 0.0  2.5  9.7   2.5 14
   create free 37.0 0.0 2.0 1.0 37.0 0.0 0.0 0.0 0.0  3.0  9.0   3.0 14
   create free 37.0 0.0 2.0 1.0 37.0 0.0 0.0 0.0 0.0  3.5  8.35  3.5 14
   create free 37.0 0.0 2.0 1.0 37.0 0.0 0.0 0.0 0.0  4.0  7.8   4.0 14
   create free 37.0 0.0 2.0 1.0 37.0 0.0 0.0 0.0 0.0  4.5  7.35  4.5 14
   create free 37.0 0.0 2.0 1.0 37.0 0.0 0.0 0.0 0.0  5.0  6.95  5.0 14
   create free 37.0 0.0 2.0 1.0 37.0 0.0 0.0 0.0 0.0  5.5  6.6   5.5 14
   create free 37.0 0.0 2.0 1.0 37.0 0.0 0.0 0.0 0.0  6.0  6.3   6.0 14
   create free 37.0 0.0 2.0 1.0 37.0 0.0 0.0 0.0 0.0  6.5  6.025 6.5 13
   create free 37.0 0.0 2.0 1.0 37.0 0.0 0.0 0.0 0.0  7.0  5.8   7.0 12.1
   create free 37.0 0.0 2.0 1.0 37.0 0.0 0.0 0.0 0.0  7.5  5.6   7.5 11.2
   create free 37.0 0.0 2.0 1.0 37.0 0.0 0.0 0.0 0.0  8.0  5.45  8.0 10.3
   create free 37.0 0.0 2.0 1.0 37.0 0.0 0.0 0.0 0.0  8.5  5.3   8.5  9.5
   create free 37.0 0.0 2.0 1.0 37.0 0.0 0.0 0.0 0.0  9.0  5.2   9.0  8.6
   create free 37.0 0.0 2.0 1.0 37.0 0.0 0.0 0.0 0.0  9.5  5.1   9.5  7.7
   create free 37.0 0.0 2.0 1.0 37.0 0.0 0.0 0.0 0.0 10.0  5.05 10.0  6.8
   create free 37.0 0.0 2.0 1.0 37.0 0.0 0.0 0.0 0.0 10.5  5.0  10.5  5.85
   create free 37.0 0.0 2.0 1.0 37.0 0.0 0.0 0.0 0.0 11.0  5.0  11.0  5.0
   meshend
  set time 1 gravity -10 cmdproc off framelimit 100
    writegap 2000 interval 32
    cmdlist plot standard set calc writegap * 2 interval * 2 cend
    echo on
  plot page go 32383  stop
```

Table 3.18 Input commands for result set 5



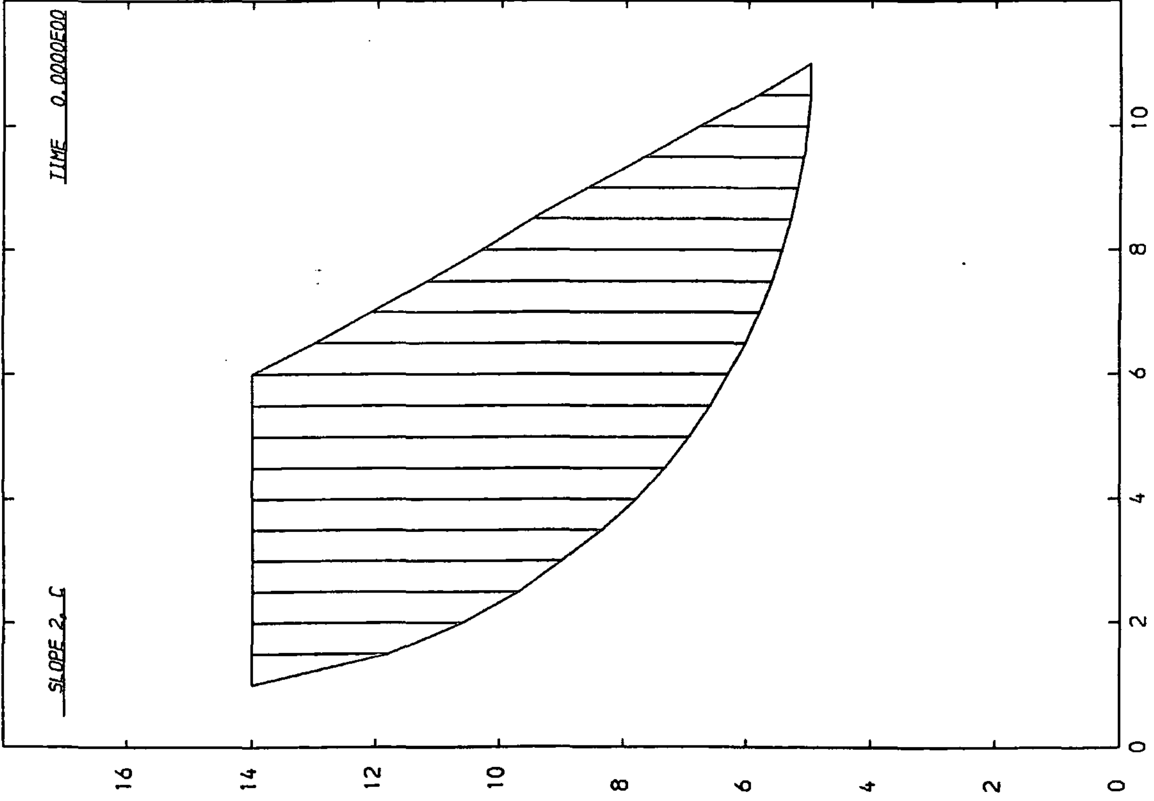
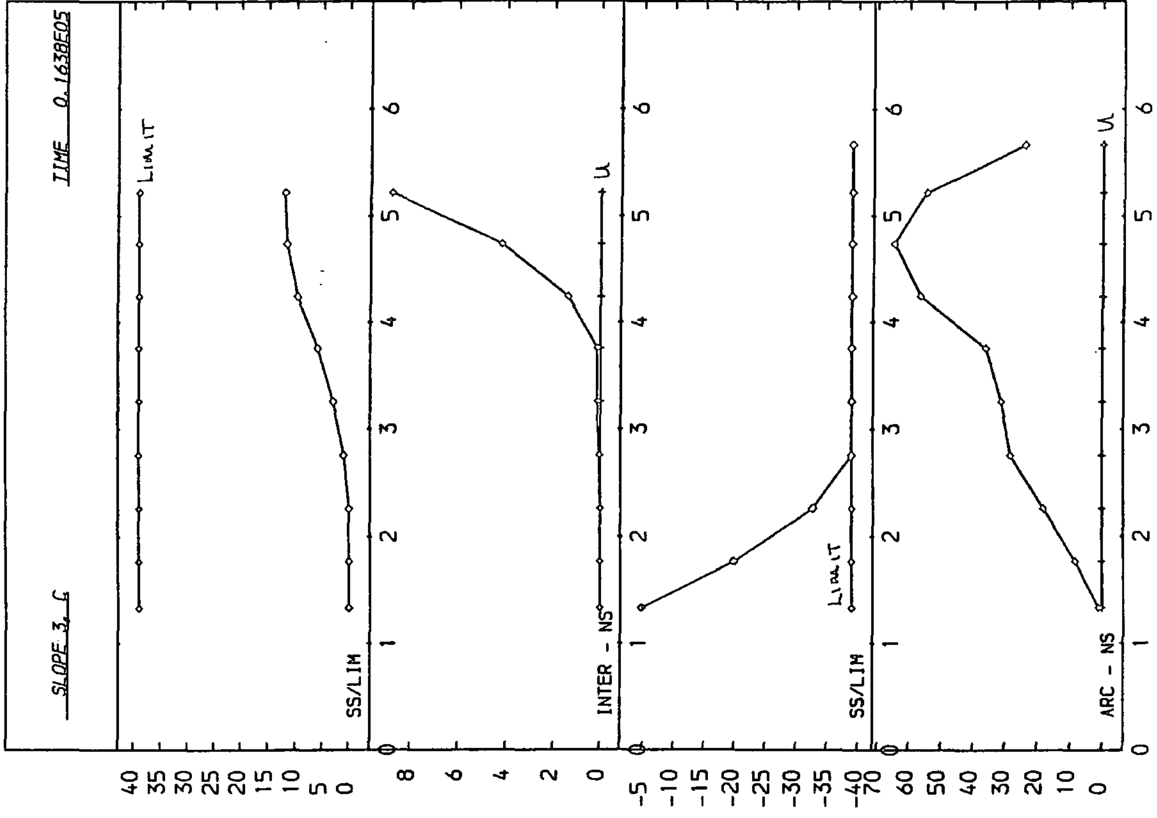


Figure 3.10 Stress profiles for result set 5

```
set echo off damp 0.05 0.2
start  SLOPE 3, C
  0 7  0
   create track 39 0.0 2.0 1.0 39 0.0 0.0 0.0 0.0  1.0 14.0  1.0 14
                                                   1.5 11.8  1.5 14
   create track 39 0.0 2.0 1.0 39 0.0 0.0 0.0 0.0  2.0 10.6  2.0 14
   create track 39 0.0 2.0 1.0 39 0.0 0.0 0.0 0.0  2.5  9.7  2.5 14
   create track 39 0.0 2.0 1.0 39 0.0 0.0 0.0 0.0  3.0  9.0  3.0 14
   create track 39 0.0 2.0 1.0 39 0.0 0.0 0.0 0.0  3.5  8.35 3.5 14
   create track 39 0.0 2.0 1.0 39 0.0 0.0 0.0 0.0  4.0  7.8  4.0 14
   create track 39 0.0 2.0 1.0 39 0.0 0.0 0.0 0.0  4.5  7.35 4.5 12
   create track 39 0.0 2.0 1.0 39 0.0 0.0 0.0 0.0  5.0  6.95 5.0 10
   create track 39 0.0 2.0 1.0 39 0.0 0.0 0.0 0.0  5.5  6.6  5.5  8
   create track 39 0.0 2.0 1.0 39 0.0 0.0 0.0 0.0  6.0  6.3  6.0  6.3
   meshend
  set damp 0.05 0.2 time 1 gravity -10 cmdproc on framelimit 100
    writegap 32 interval 32
    cmdlist  plot standard  set calc writegap * 2 interval * 2 cend
    echo on debug osc on
  plot page go 4000 stop
```

Table 3.19 Input commands for result set 6



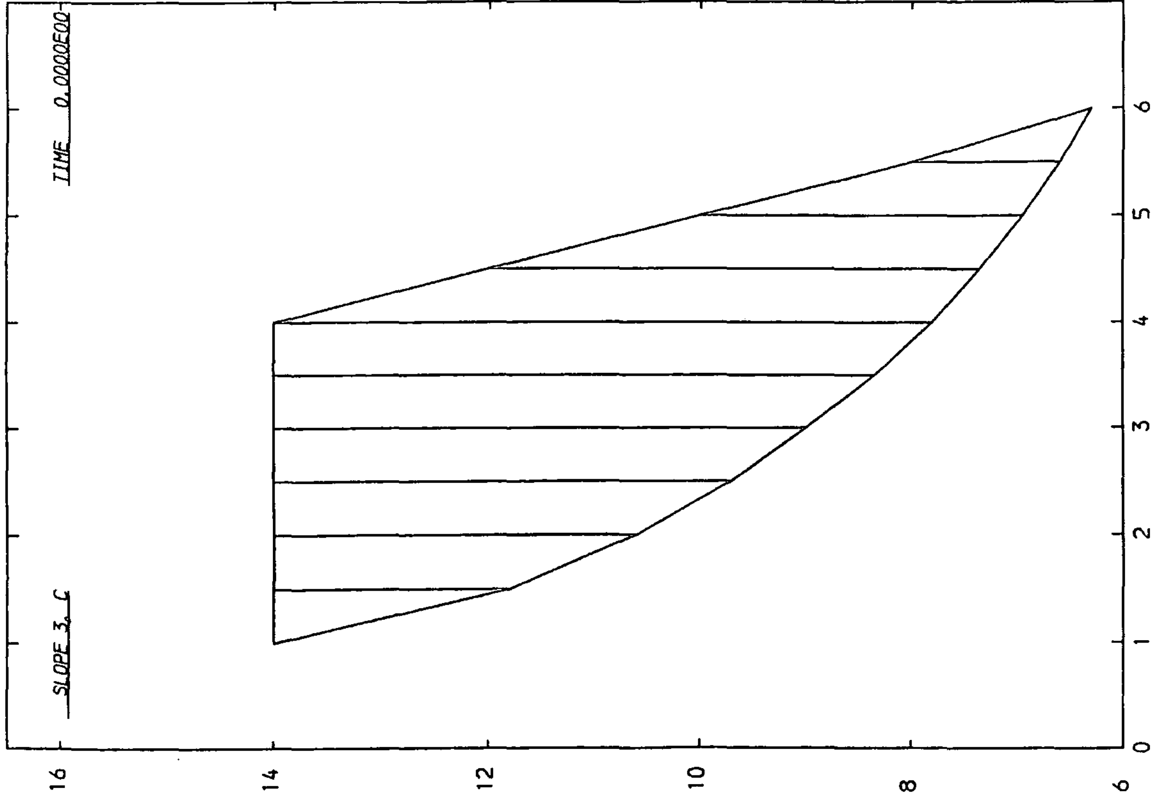
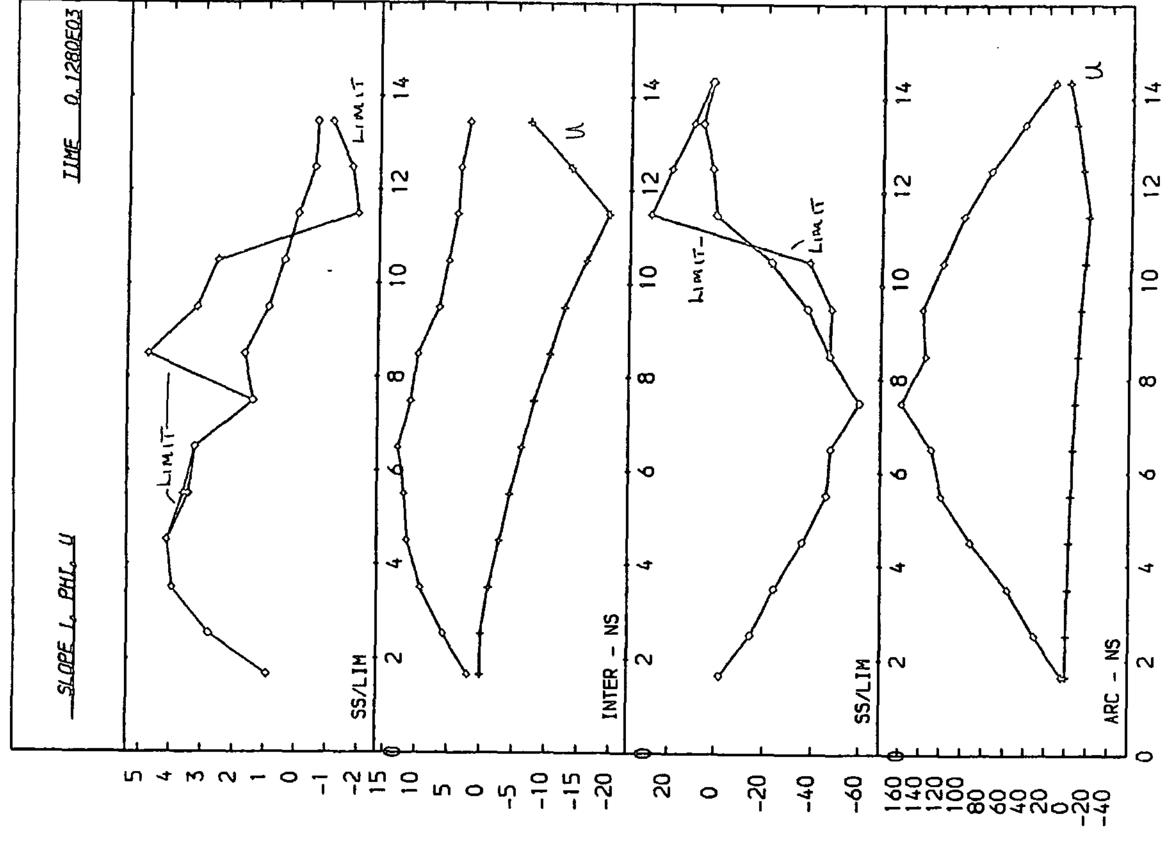


Figure 3.11 Stress profiles for result set 6

```
set echo off damp 0.00125 0.005 gravity -10
start  SLOPE 1, PHI, U
  0 16 0
   create free  0.0 25.0 2.0 1.0 0.0 25.0 0.0  0.0  0.23  1 14.0  1 14
                                                          2 10.5  2 14
   create free  0.0 25.0 2.0 1.0 0.0 25.0 0.0  0.0  0.23  3  8.9  3 14
   create free  0.0 25.0 2.0 1.0 0.0 25.0 0.6  0.6  0.23  4  7.7  4 14
   create free  0.0 25.0 2.0 1.0 0.0 25.0 1.6  1.0  0.23  5  6.9  5 14
   create free  0.0 25.0 2.0 1.0 0.0 25.0 2.3  1.3  0.23  6  6.3  6 14
   create free  0.0 25.0 2.0 1.0 0.0 25.0 2.9  1.6  0.23  7  5.7  7 14
   create free  0.0 25.0 2.0 1.0 0.0 25.0 3.35 1.75 0.23  8  5.4  8 13
   create free  0.0 25.0 2.0 1.0 0.0 25.0 3.65 1.9  0.23  9  5.1  9 12
   create free  0.0 25.0 2.0 1.0 0.0 25.0 3.85 1.95 0.23 10  5.0 10 11
   create free  0.0 25.0 2.0 1.0 0.0 25.0 3.95 2.0  0.23 11  4.9 11  9.9
   create free  0.0 25.0 2.0 1.0 0.0 25.0 3.95 1.95 0.23 12  5.0 12  8.9
   create free  0.0 25.0 2.0 1.0 0.0 25.0 3.3  1.35 0.23 13  5.1 13  7.8
   create free  0.0 25.0 2.0 1.0 0.0 25.0 2.05 0.7  0.23 14  5.4 14  6.8
   create free  0.0 25.0 2.0 1.0 0.0 25.0 0.7  0.0  0.23 15  5.8 15  5.8
   meshend
  set time 1 gravity -10 cmdproc on framelimit 100
    writegap 1000 interval 1000
    cmdlist plot standard cend
    echo on
  plot page go 40000  stop
```

Table 3.20 Input commands for result set 7



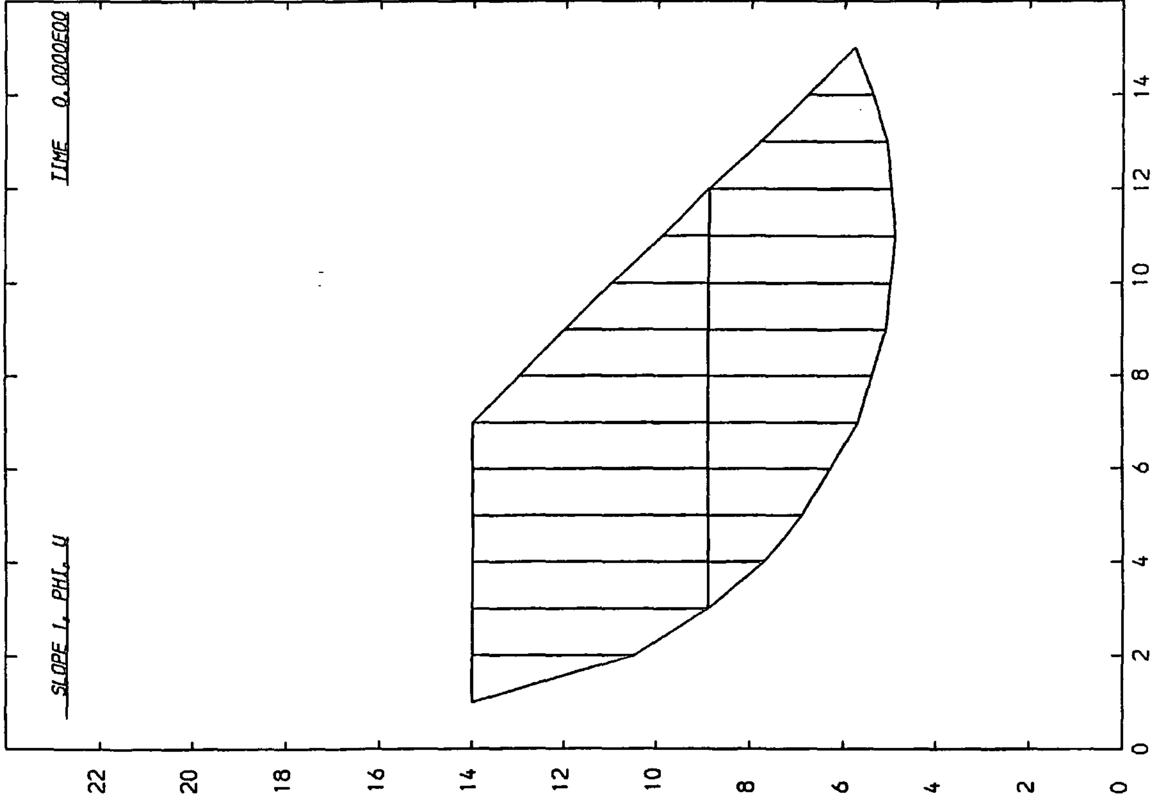
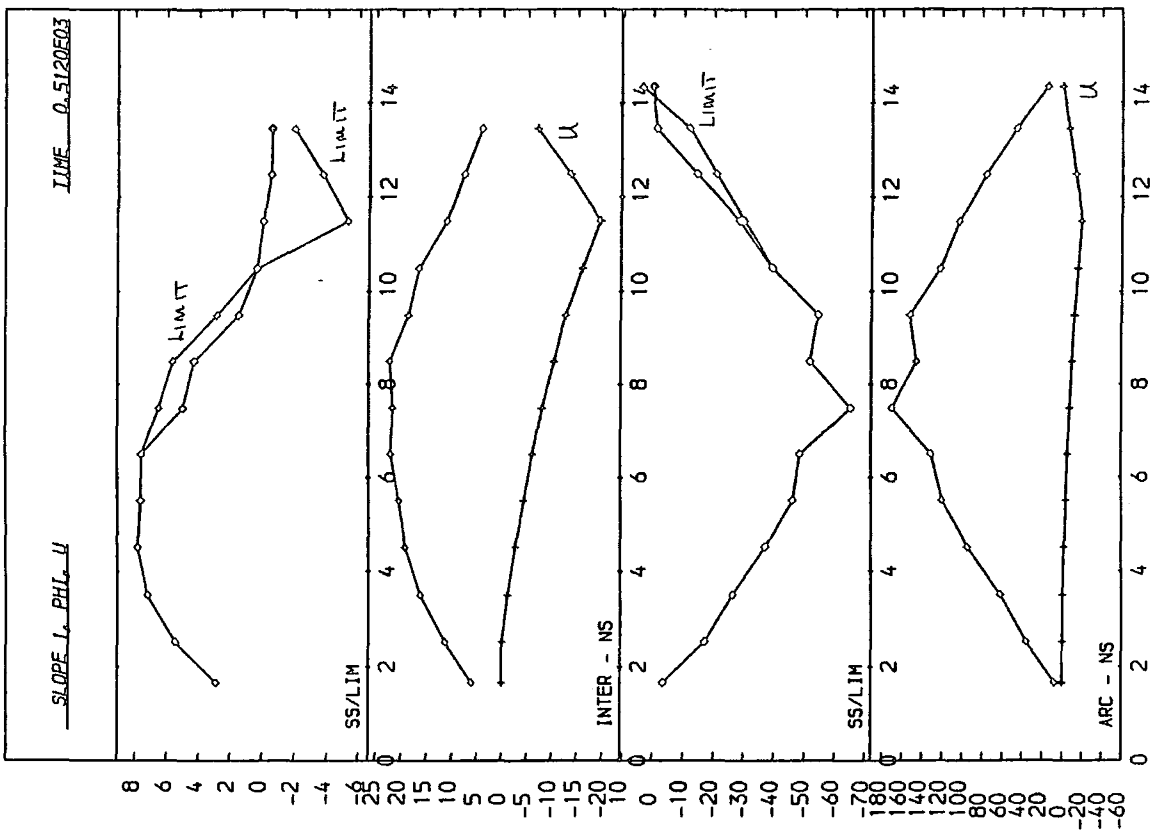


Figure 3.12 Stress profiles for result set 7



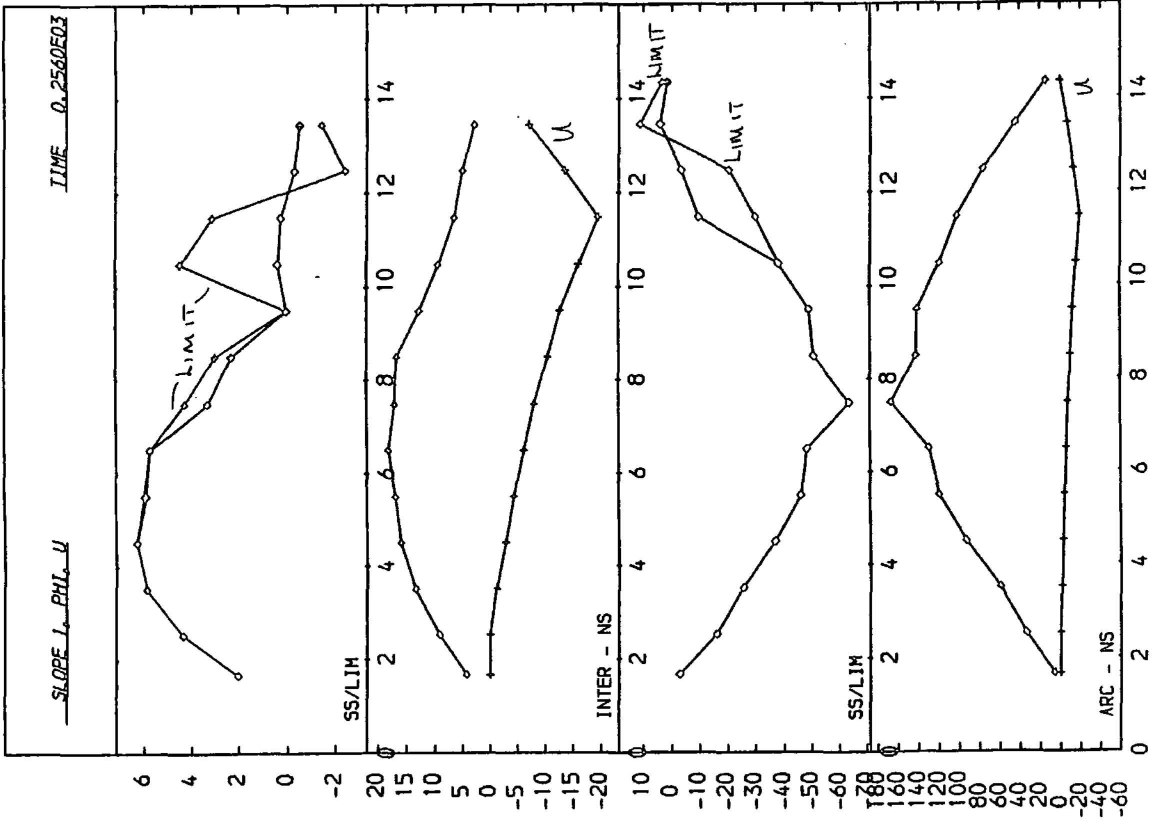
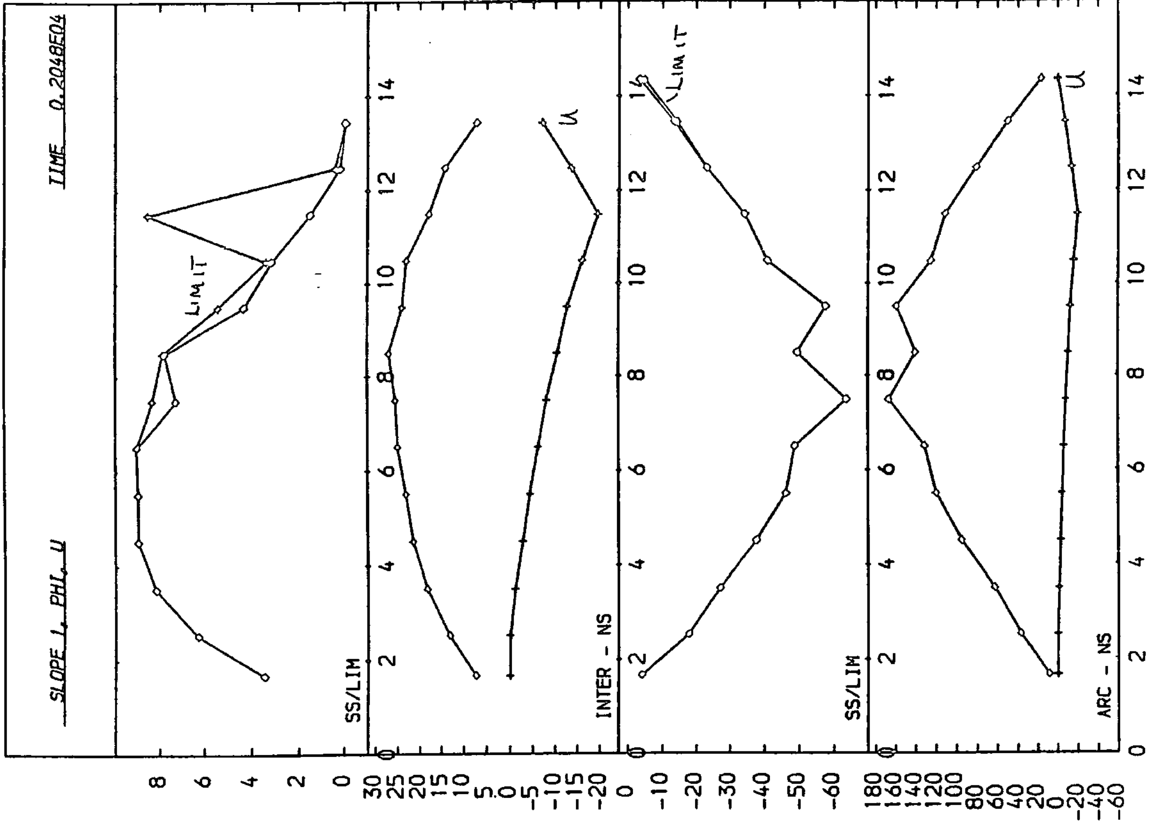


Figure 3.12 Stress profiles for result set 7 (continued)



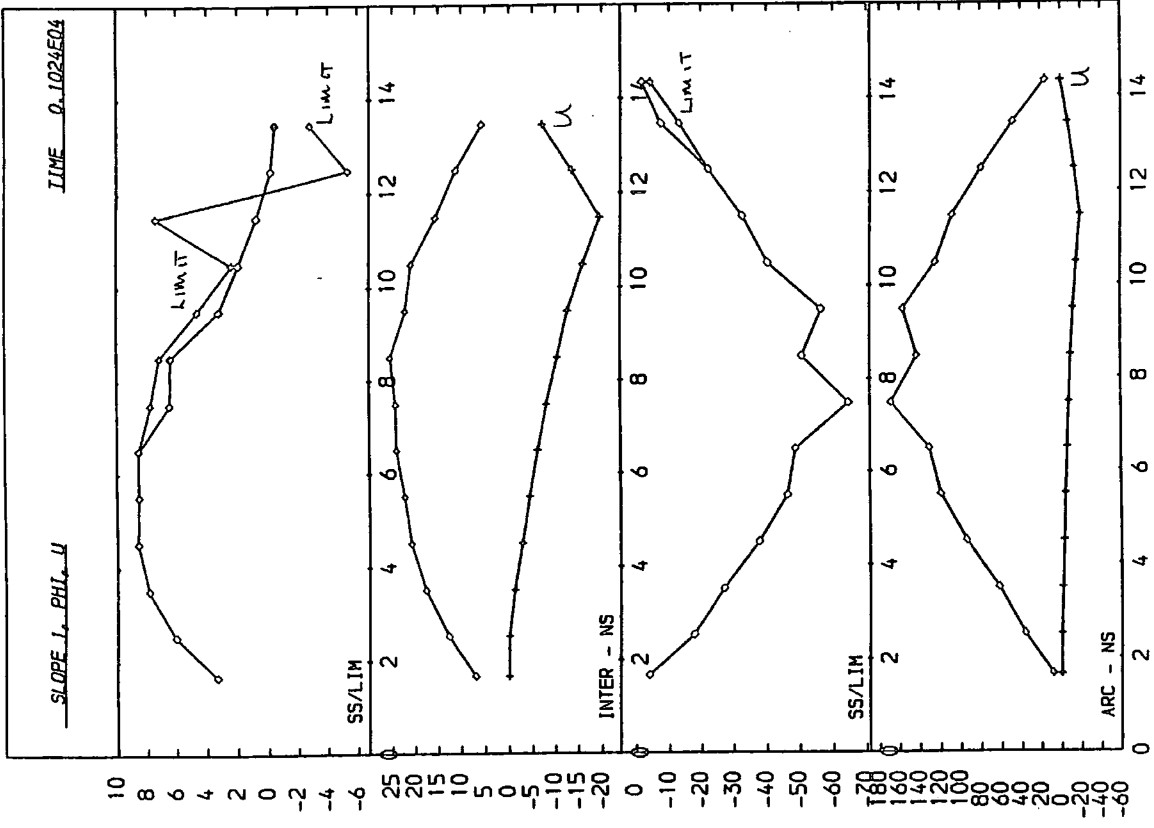
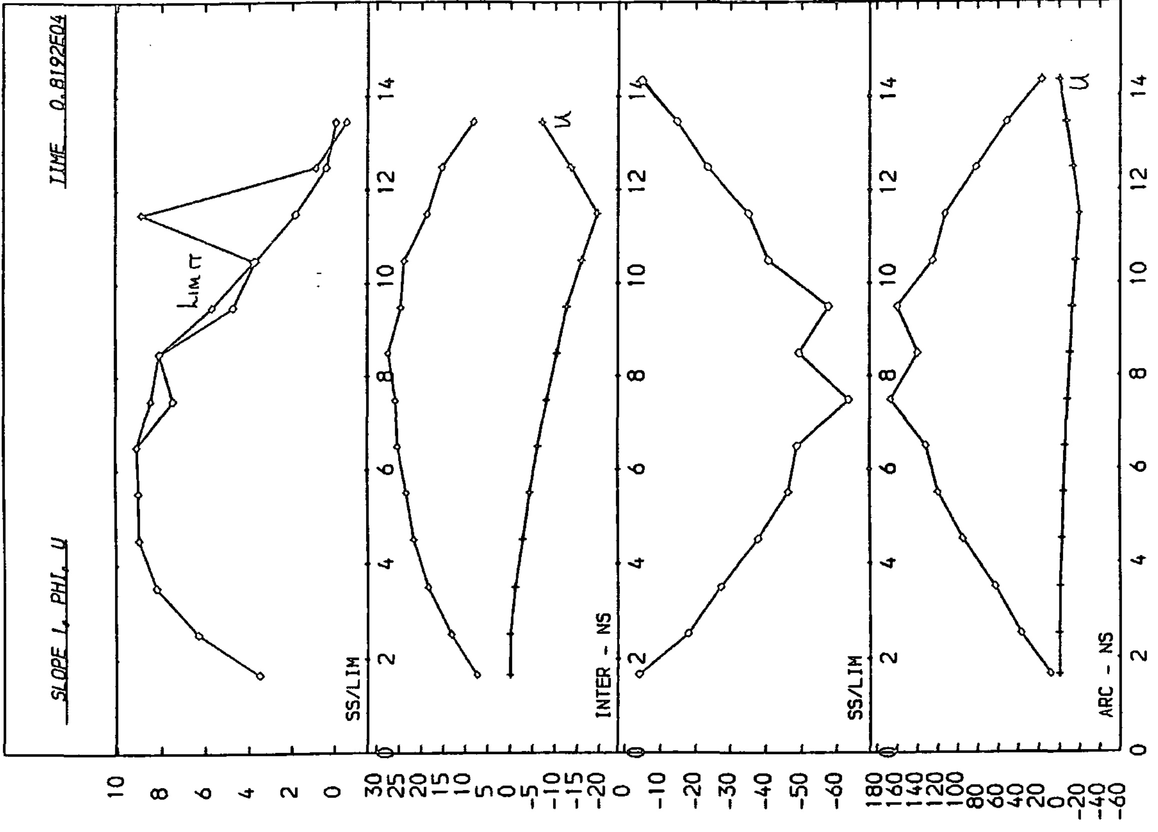


Figure 3.12 Stress profiles for result set 7 (continued)



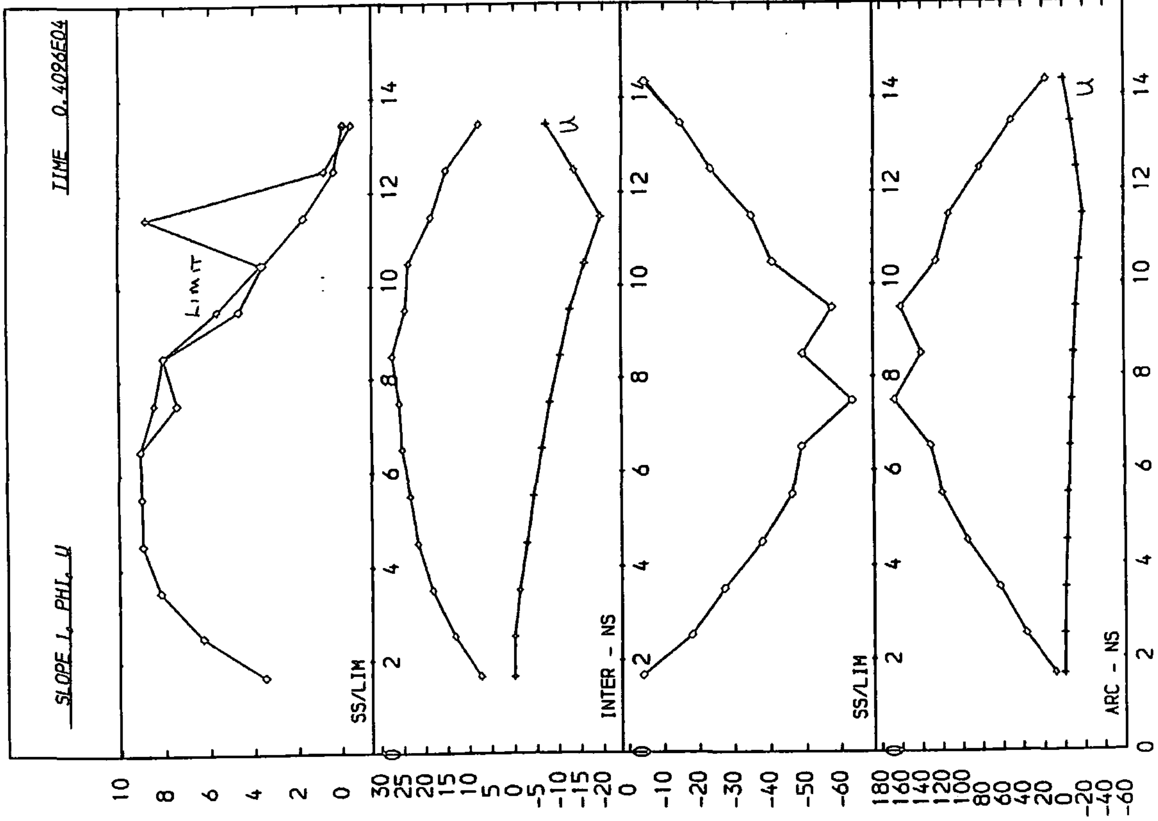


Figure 3.12 Stress profiles for result set 7 (continued)

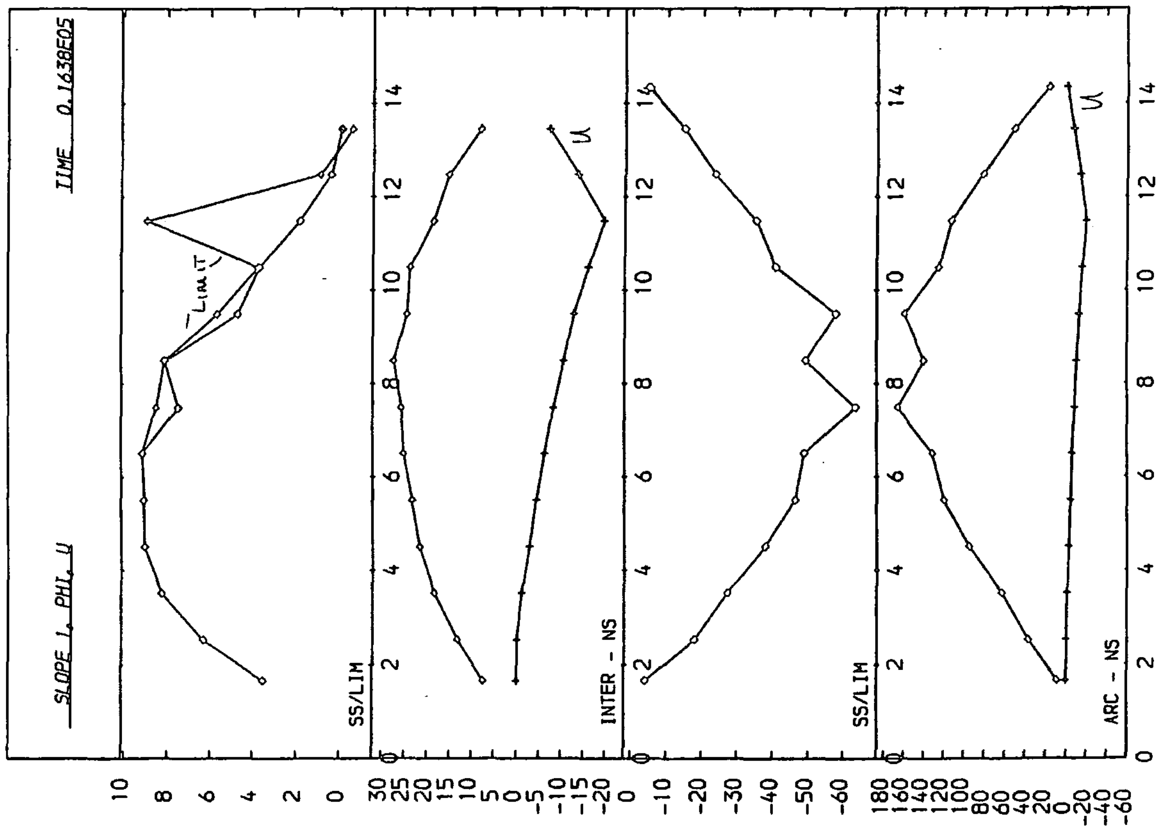
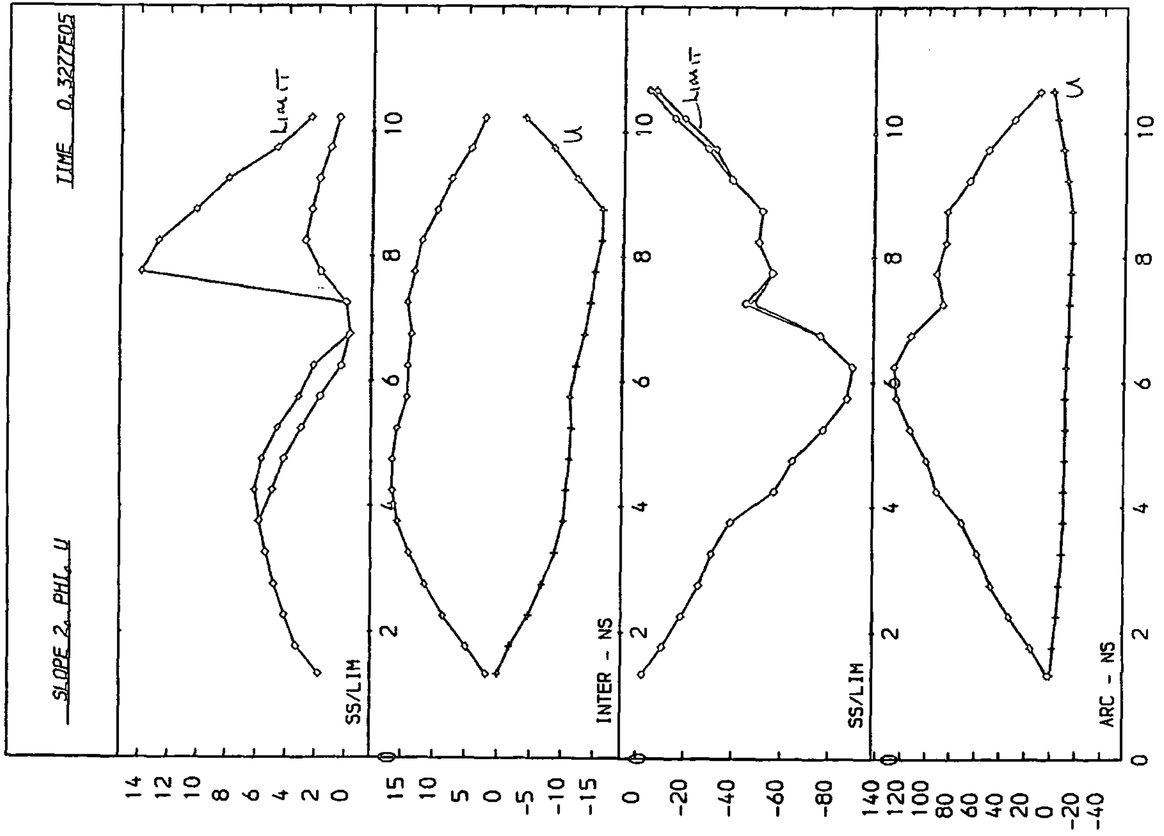


Figure 3.12 Stress profiles for result set 7 (continued)

```
set echo off damp 0.00125  0.005  gravity -10
start  SLOPE 2, PHI, U
  0 12 0
   create free 0.0 46.5 2.0 1.0 0.0 46.5 0.0   0.0   0.23  1.0 14.0   1.0 14
                                                           1.5 11.8   1.5 14
   create free 0.0 46.5 2.0 1.0 0.0 46.5 0.55  0.55  0.23  2.0 10.6   2.0 14
   create free 0.0 46.5 2.0 1.0 0.0 46.5 1.55  1.0   0.23  2.5  9.7   2.5 14
   create free 0.0 46.5 2.0 1.0 0.0 46.5 2.3   1.3   0.23  3.0  9.0   3.0 14
   create free 0.0 46.5 2.0 1.0 0.0 46.5 2.875 1.575 0.23  3.5  8.35  3.5 14
   create free 0.0 46.5 2.0 1.0 0.0 46.5 3.35  1.775 0.23  4.0  7.8   4.0 14
   create free 0.0 46.5 2.0 1.0 0.0 46.5 3.65  1.875 0.23  4.5  7.35  4.5 14
   create free 0.0 46.5 2.0 1.0 0.0 46.5 3.85  1.975 0.23  5.0  6.95  5.0 14
   create free 0.0 46.5 2.0 1.0 0.0 46.5 4.025 2.05  0.23  5.5  6.6   5.5 14
   create free 0.0 46.5 2.0 1.0 0.0 46.5 4.125 2.075 0.23  6.0  6.3   6.0 14
   create free 0.0 46.5 2.0 1.0 0.0 46.5 4.125 2.05  0.23  6.5  6.025 6.5 13
   create free 0.0 46.5 2.0 1.0 0.0 46.5 4.1   2.05  0.23  7.0  5.8   7.0 12.1
   create free 0.0 46.5 2.0 1.0 0.0 46.5 4.05  2.0   0.23  7.5  5.6   7.5 11.2
   create free 0.0 46.5 2.0 1.0 0.0 46.5 3.9   1.9   0.23  8.0  5.45  8.0 10.3
   create free 0.0 46.5 2.0 1.0 0.0 46.5 3.725 1.825 0.23  8.5  5.3   8.5  9.5
   create free 0.0 46.5 2.0 1.0 0.0 46.5 3.475 1.65  0.23  9.0  5.2   9.0  8.6
   create free 0.0 46.5 2.0 1.0 0.0 46.5 2.9   1.25  0.23  9.5  5.1   9.5  7.7
   create free 0.0 46.5 2.0 1.0 0.0 46.5 2.1   0.85  0.23 10.0  5.05 10.0  6.8
   create free 0.0 46.5 2.0 1.0 0.0 46.5 1.25  0.4   0.23 10.5  5.0  10.5  5.85
   create free 0.0 46.5 2.0 1.0 0.0 46.5 0.4   0.0   0.23 11.0  5.0  11.0  5.0
   meshend
  set time 1 gravity -10 cmdproc on framelimit 100
    writegap 128  interval 128
    cmdlist plot standard set calc writegap * 2 interval * 2 cend
    echo on
  plot page go 32767  stop
```

Table 3.21 Input commands for result set 8



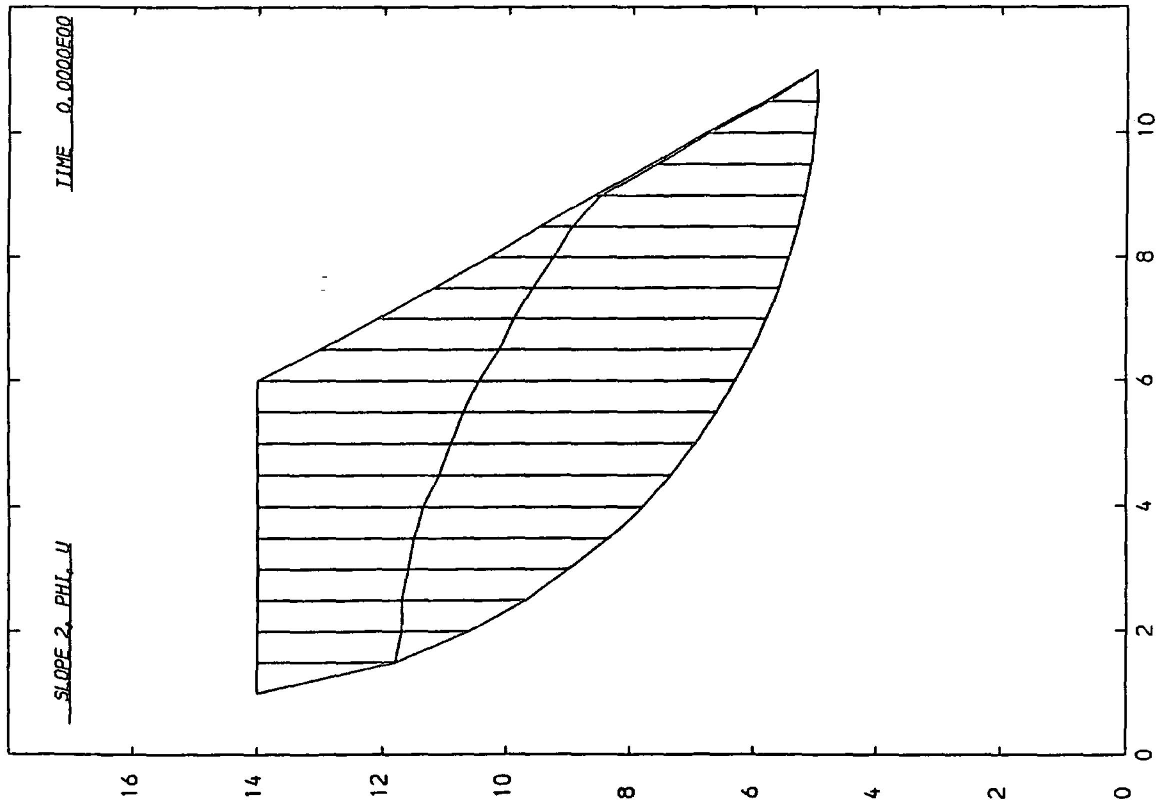
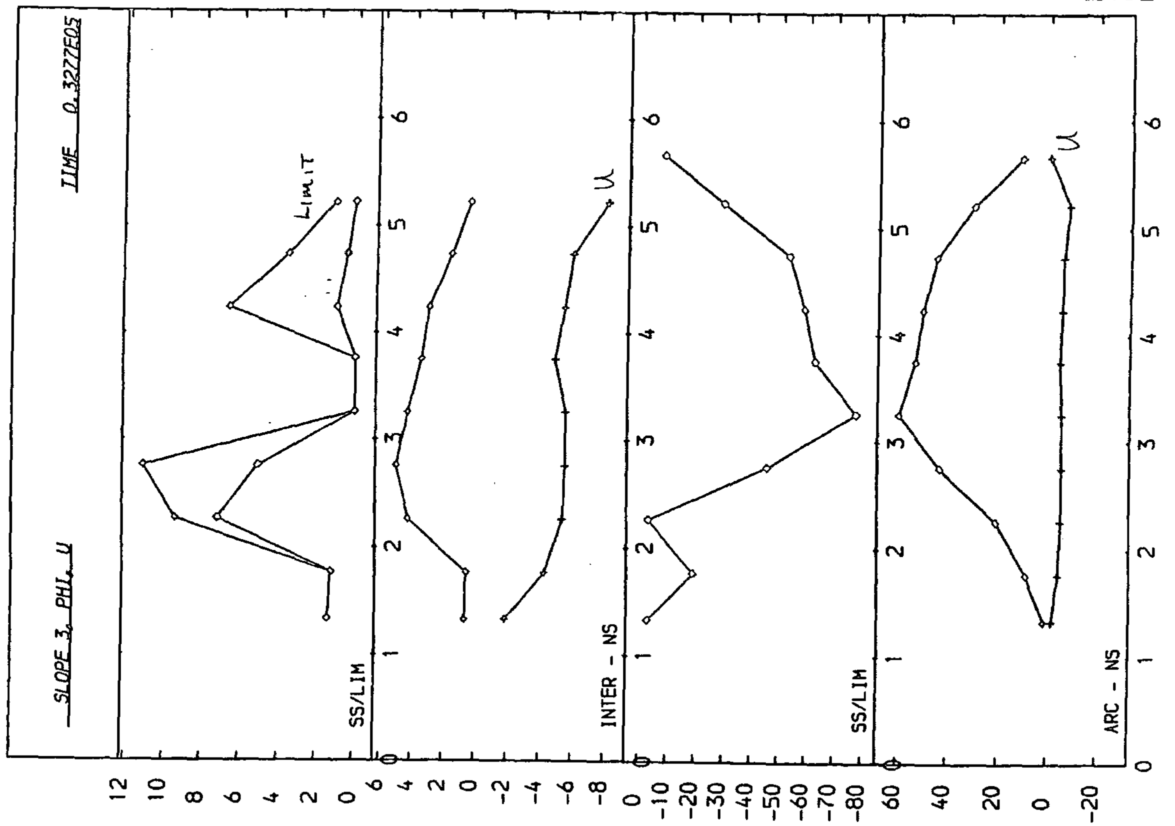


Figure 3.13 Stress profiles for result set 8

```
set echo off damp 0.05 0.05 gravity -10
set damp 0.00125 0.005 gravity -10
start  SLOPE 3, PHI, U
  0 7  0
   create free  0.0 69.5 2.0 1.0 0.0 69.5 0.225 0.45  0.23 1.0 14.0  1.0 14
                                                           1.5 11.8  1.5 14
   create free  0.0 69.5 2.0 1.0 0.0 69.5 1.3   0.85  0.23 2.0 10.6  2.0 14
   create free  0.0 69.5 2.0 1.0 0.0 69.5 1.925 1.075 0.23 2.5  9.7  2.5 14
   create free  0.0 69.5 2.0 1.0 0.0 69.5 2.25  1.175 0.23 3.0  9.0  3.0 14
   create free  0.0 69.5 2.0 1.0 0.0 69.5 2.425 1.25  0.23 3.5  8.35 3.5 14
   create free  0.0 69.5 2.0 1.0 0.0 69.5 2.475 1.225 0.23 4.0  7.8  4.0 14
   create free  0.0 69.5 2.0 1.0 0.0 69.5 2.35  1.125 0.23 4.5  7.35 4.5 12
   create free  0.0 69.5 2.0 1.0 0.0 69.5 2.075 0.95  0.23 5.0  6.95 5.0 10
   create free  0.0 69.5 2.0 1.0 0.0 69.5 1.7   0.75  0.23 5.5  6.6  5.5  8
   create free  0.0 69.5 2.0 1.0 0.0 69.5 0.75  0.0   0.23 6.0  6.3  6.0  6.3
  set time 1 cmdproc on framelimit 100
    writegap 128  interval 128
    cmdlist  plot standard  set calc writegap * 2 interval * 2 cend
    echo on debug oscil off
  plot page go 32767  stop
```

Table 3.22 Input commands for result set 9



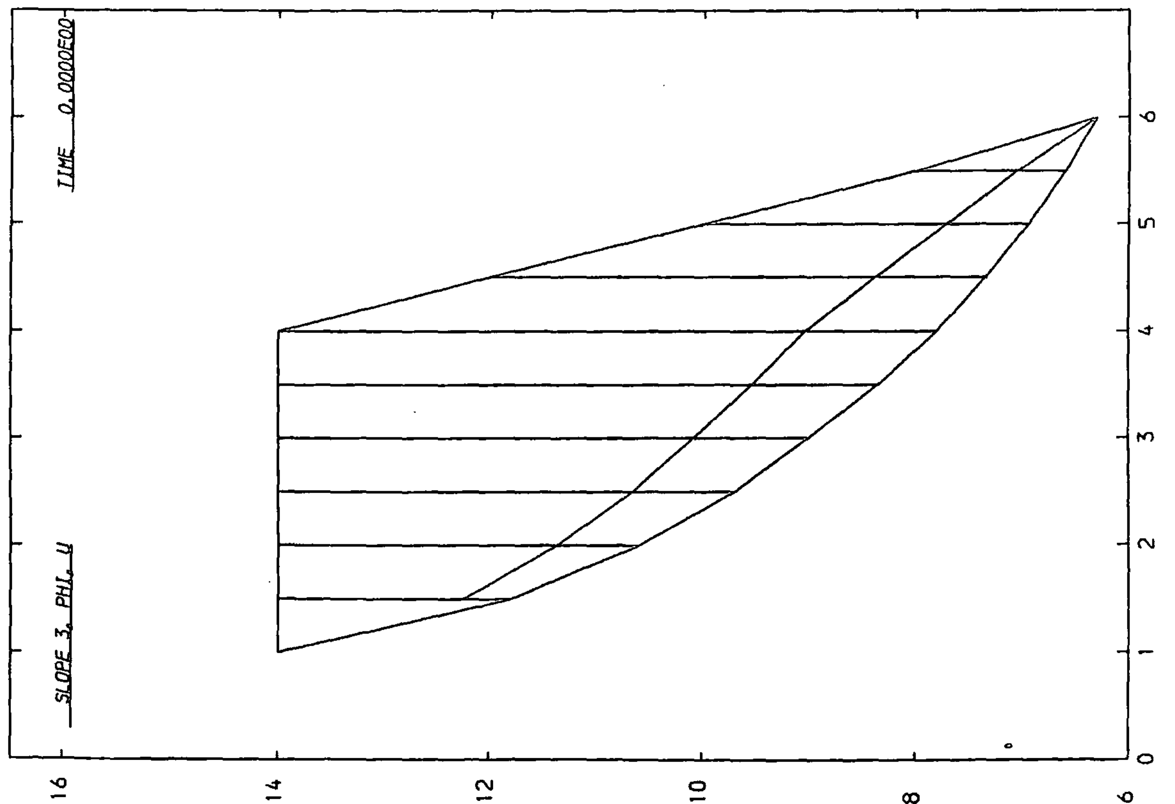
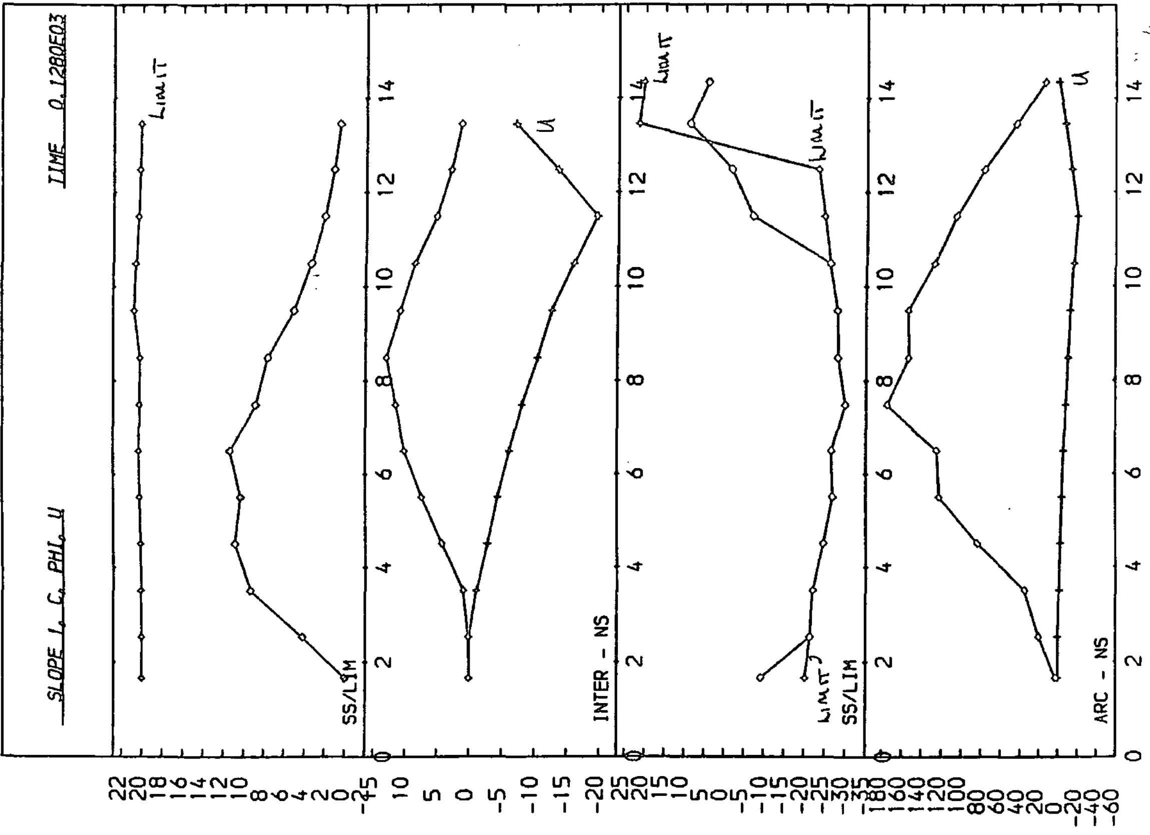


Figure 3.14 Stress profiles for result set 9

```
set echo off  debug update on
set echo off damp 0.005 0.02 gravity -10 debug update on
start  SLOPE 1, C, PHI, U
  0 16 0
   create free  20 5.0 2.0 1.0 20 5.0 0.0  0.0  0.23  1 14.0  1 14
                                                     2 10.5  2 14
   create free  20 5.0 2.0 1.0 20 5.0 0.0  0.0  0.23  3  8.9  3 14
   create free  20 5.0 2.0 1.0 20 5.0 0.6  0.6  0.23  4  7.7  4 14
   create free  20 5.0 2.0 1.0 20 5.0 1.6  1.0  0.23  5  6.9  5 14
   create free  20 5.0 2.0 1.0 20 5.0 2.3  1.3  0.23  6  6.3  6 14
   create free  20 5.0 2.0 1.0 20 5.0 2.9  1.6  0.23  7  5.7  7 14
   create free  20 5.0 2.0 1.0 20 5.0 3.35 1.75 0.23  8  5.4  8 13
   create free  20 5.0 2.0 1.0 20 5.0 3.65 1.9  0.23  9  5.1  9 12
   create free  20 5.0 2.0 1.0 20 5.0 3.85 1.95 0.23 10  5.0 10 11
   create free  20 5.0 2.0 1.0 20 5.0 3.95 2.0  0.23 11  4.9 11  9.9
   create free  20 5.0 2.0 1.0 20 5.0 3.95 1.95 0.23 12  5.0 12  8.9
   create free  20 5.0 2.0 1.0 20 5.0 3.3  1.35 0.23 13  5.1 13  7.8
   create free  20 5.0 2.0 1.0 20 5.0 2.05 0.7  0.23 14  5.4 14  6.8
   create free  20 5.0 2.0 1.0 20 5.0 0.7  0.0  0.23 15  5.8 15  5.8
   meshend
  set time 1 gravity -10 cmdproc on framelimit 100
    writegap 128  interval 128
    cmdlist plot standard set calc writegap * 2 interval * 2 cend
    echo on
   go 32383  stop
```

Table 3.23 Input commands for result set 10 and 13



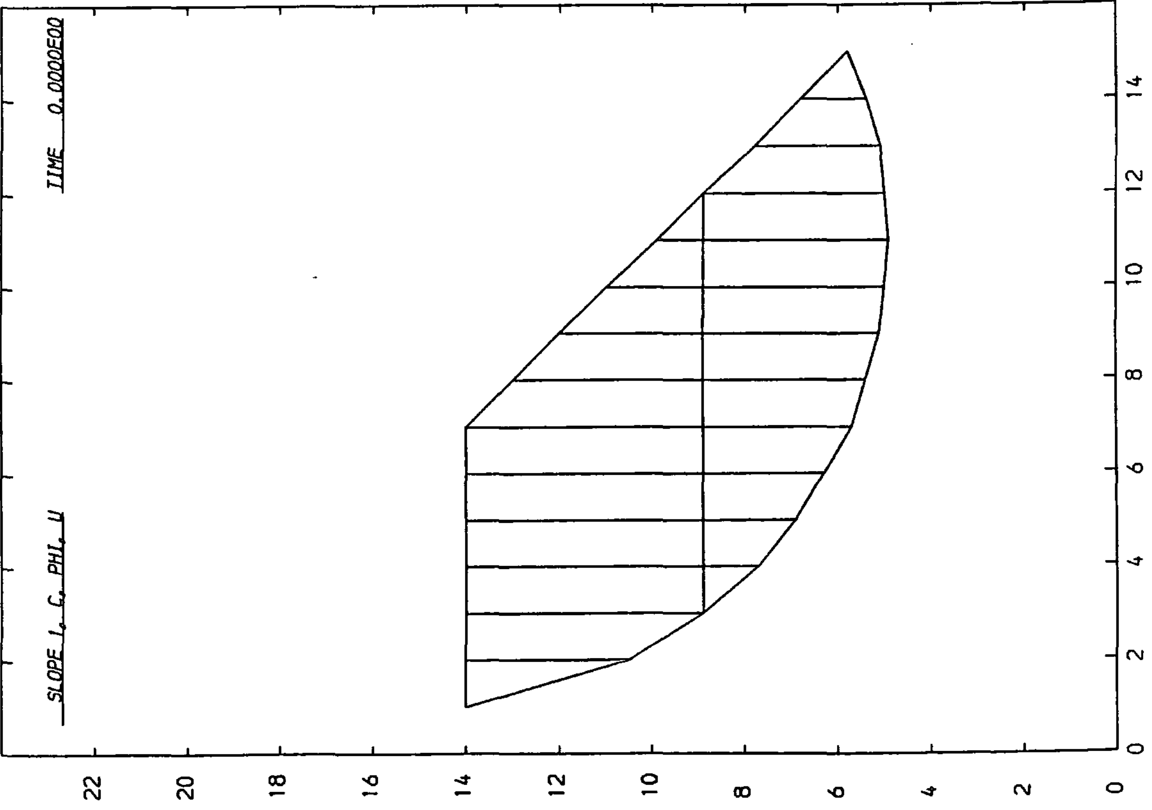
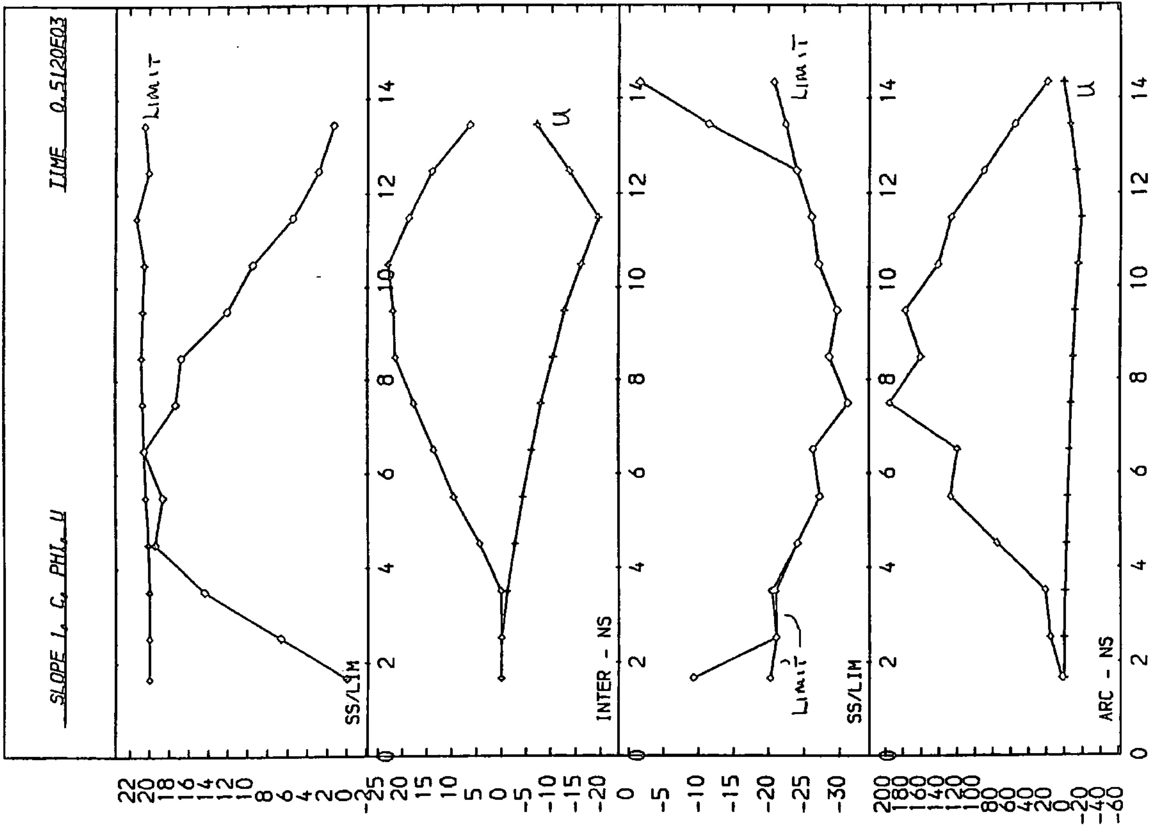


Figure 3.15 Stress profiles for result set 10



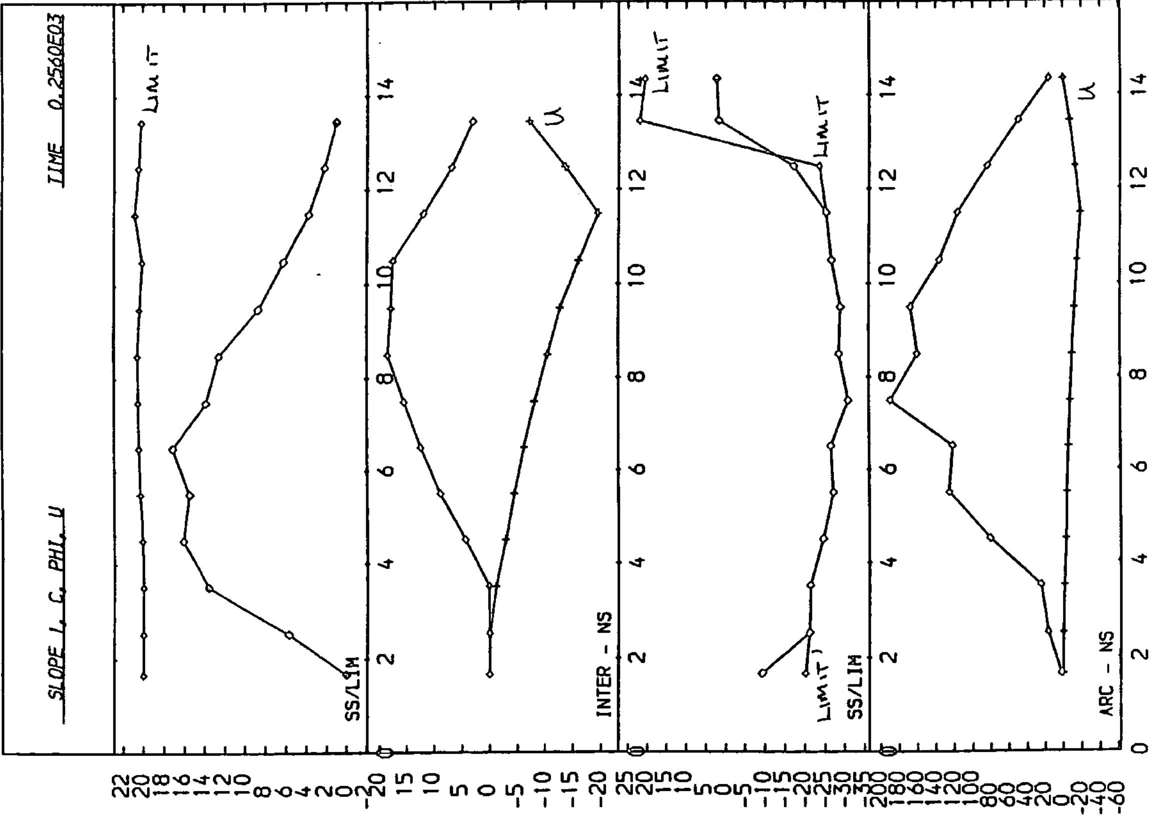
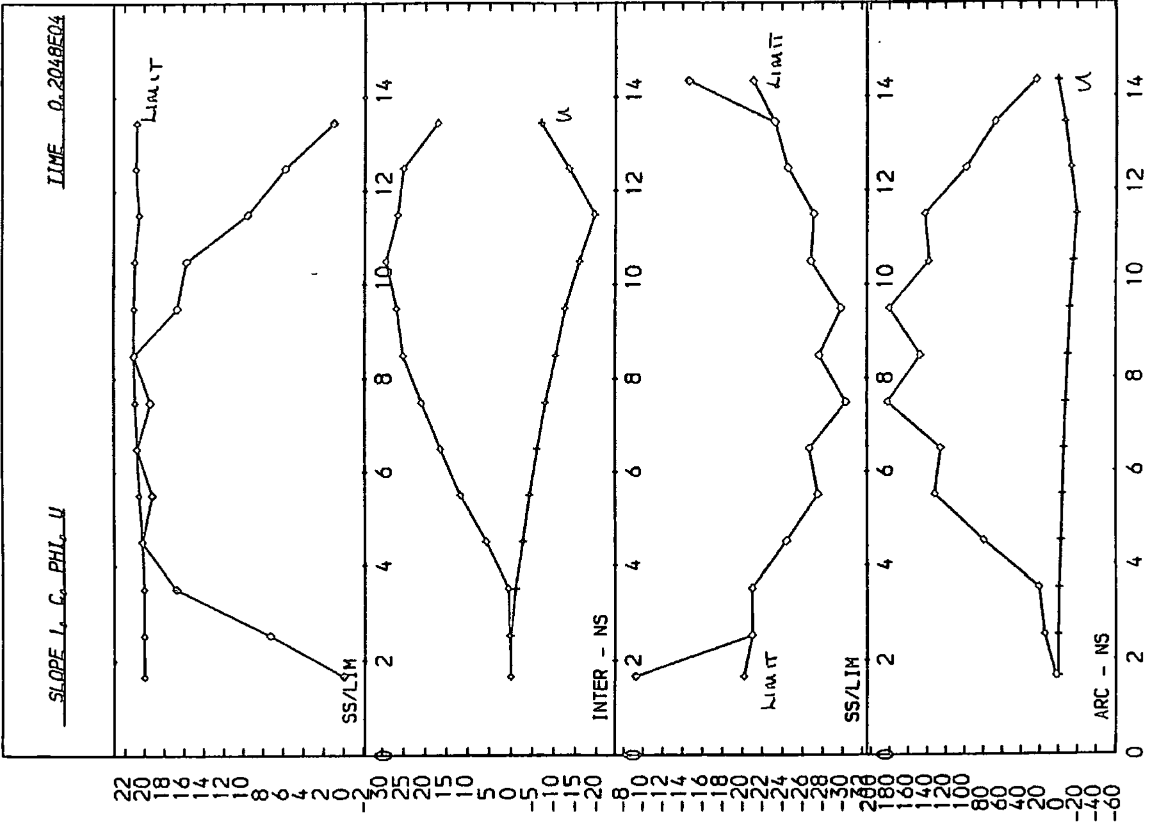


Figure 3.15 Stress profiles for result set 10 (continued)



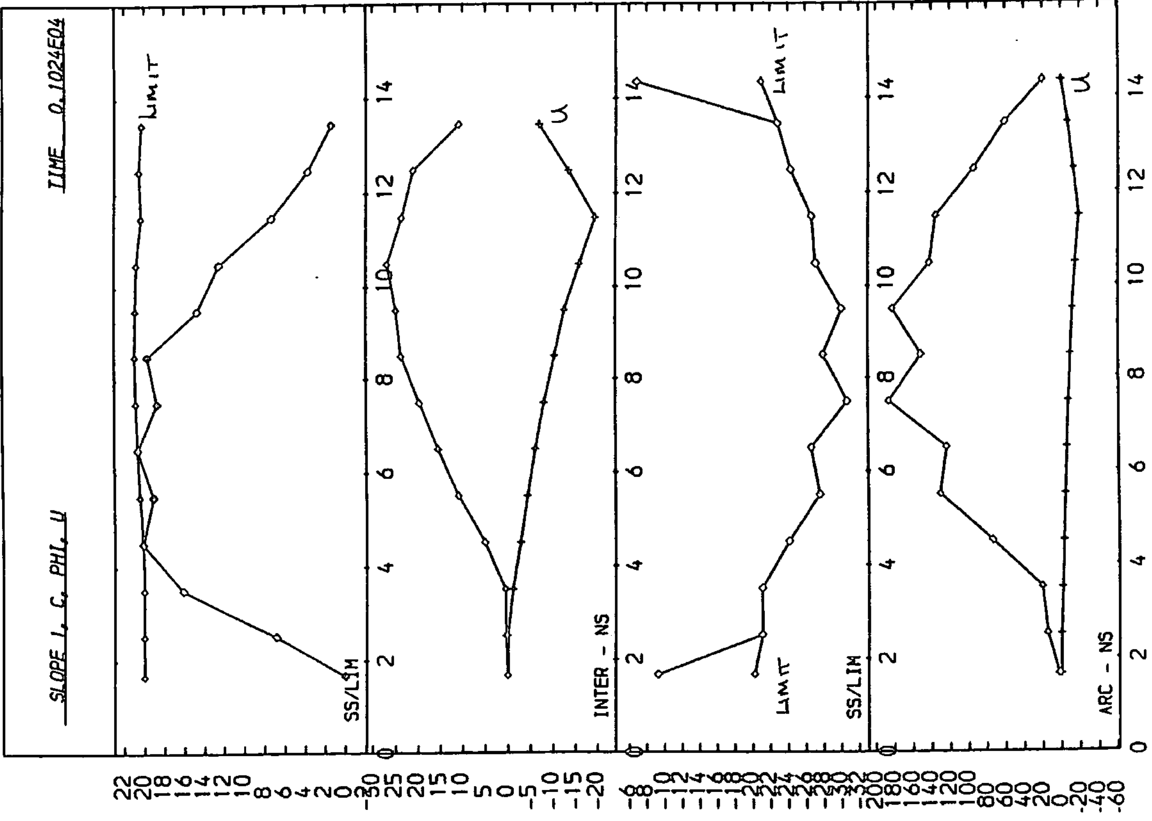
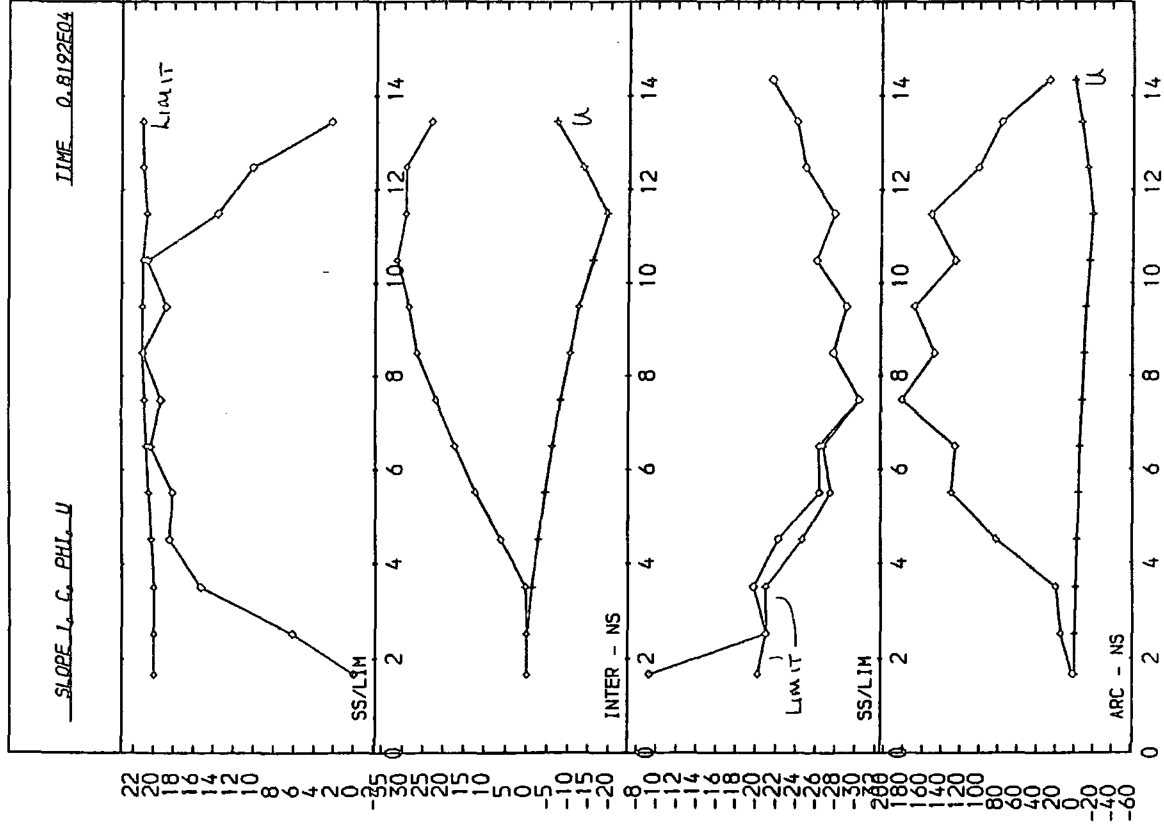


Figure 3.15 Stress profiles for result set 10 (continued)



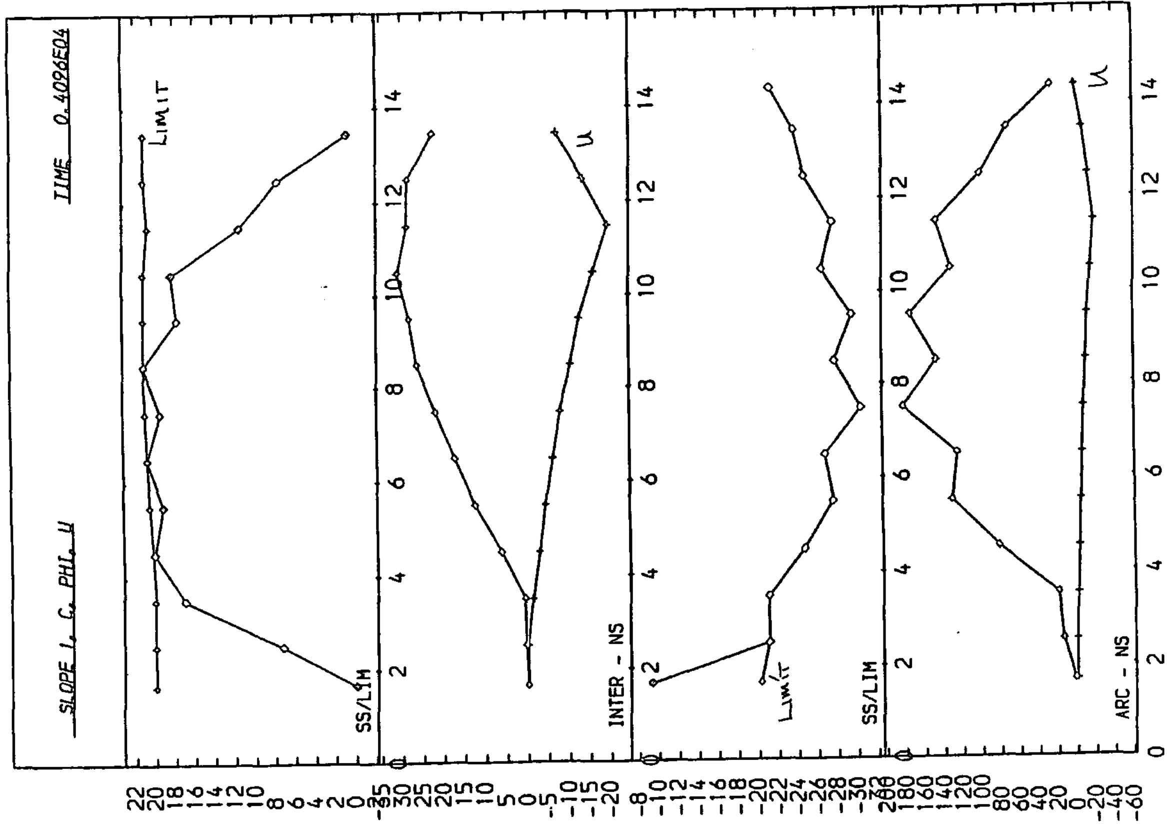


Figure 3.15 Stress profiles for result set 10 (continued)

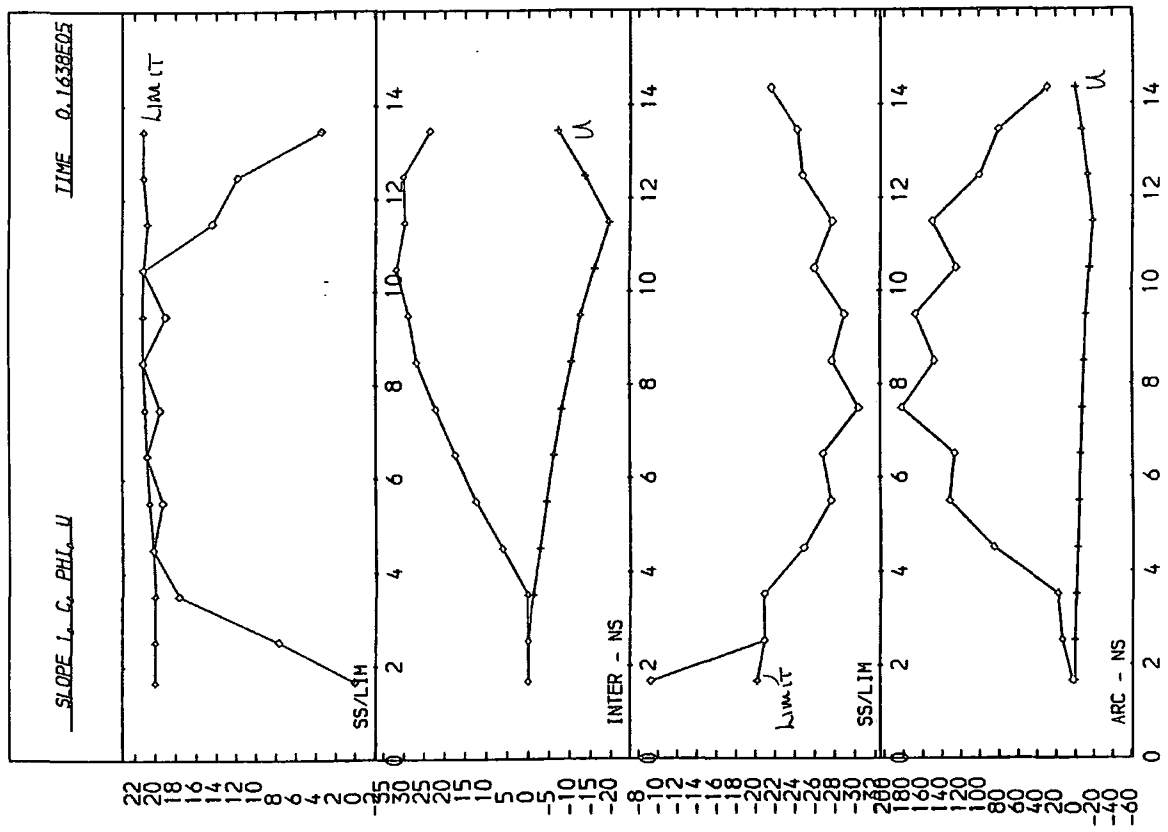


Figure 3.15 Stress profiles for result set 10 (continued)

```
debug update on
start  SLOPE 2, C, PHI, U
  0 12 0
   create free 20.0 21.5 2.0 1.0 20.0 21.5 0.0   0.0   0.23  1.0 14.0   1.0 14
                                                              1.5 11.8   1.5 14
   create free 20.0 21.5 2.0 1.0 20.0 21.5 0.55  0.55  0.23  2.0 10.6   2.0 14
   create free 20.0 21.5 2.0 1.0 20.0 21.5 1.55  1.0   0.23  2.5  9.7   2.5 14
   create free 20.0 21.5 2.0 1.0 20.0 21.5 2.3   1.3   0.23  3.0  9.0   3.0 14
   create free 20.0 21.5 2.0 1.0 20.0 21.5 2.875 1.575 0.23  3.5  8.35  3.5 14
   create free 20.0 21.5 2.0 1.0 20.0 21.5 3.35  1.775 0.23  4.0  7.8   4.0 14
   create free 20.0 21.5 2.0 1.0 20.0 21.5 3.65  1.875 0.23  4.5  7.35  4.5 14
   create free 20.0 21.5 2.0 1.0 20.0 21.5 3.85  1.975 0.23  5.0  6.95  5.0 14
   create free 20.0 21.5 2.0 1.0 20.0 21.5 4.025 2.05  0.23  5.5  6.6   5.5 14
   create free 20.0 21.5 2.0 1.0 20.0 21.5 4.125 2.075 0.23  6.0  6.3   6.0 14
   create free 20.0 21.5 2.0 1.0 20.0 21.5 4.125 2.05  0.23  6.5  6.025 6.5 13
   create free 20.0 21.5 2.0 1.0 20.0 21.5 4.1   2.05  0.23  7.0  5.8   7.0 12.1
   create free 20.0 21.5 2.0 1.0 20.0 21.5 4.05  2.0   0.23  7.5  5.6   7.5 11.2
   create free 20.0 21.5 2.0 1.0 20.0 21.5 3.9   1.9   0.23  8.0  5.45  8.0 10.3
   create free 20.0 21.5 2.0 1.0 20.0 21.5 3.725 1.825 0.23  8.5  5.3   8.5  9.5
   create free 20.0 21.5 2.0 1.0 20.0 21.5 3.475 1.65  0.23  9.0  5.2   9.0  8.6
   create free 20.0 21.5 2.0 1.0 20.0 21.5 2.9   1.25  0.23  9.5  5.1   9.5  7.7
   create free 20.0 21.5 2.0 1.0 20.0 21.5 2.1   0.85  0.23 10.0  5.05 10.0  6.8
   create free 20.0 21.5 2.0 1.0 20.0 21.5 1.25  0.4   0.23 10.5  5.0  10.5  5.85
   create free 20.0 21.5 2.0 1.0 20.0 21.5 0.4   0.0   0.23 11.0  5.0  11.0  5.0
   meshend
  set time 1 gravity -10 cmdproc off framelimit 100
    writegap 5000 interval 128
    cmdlist  plot standard  set calc writegap * 2 interval * 2 cend
    echo on
  plot page go 32767  stop
```

Table 3.24 Input commands for result set 11 and 14

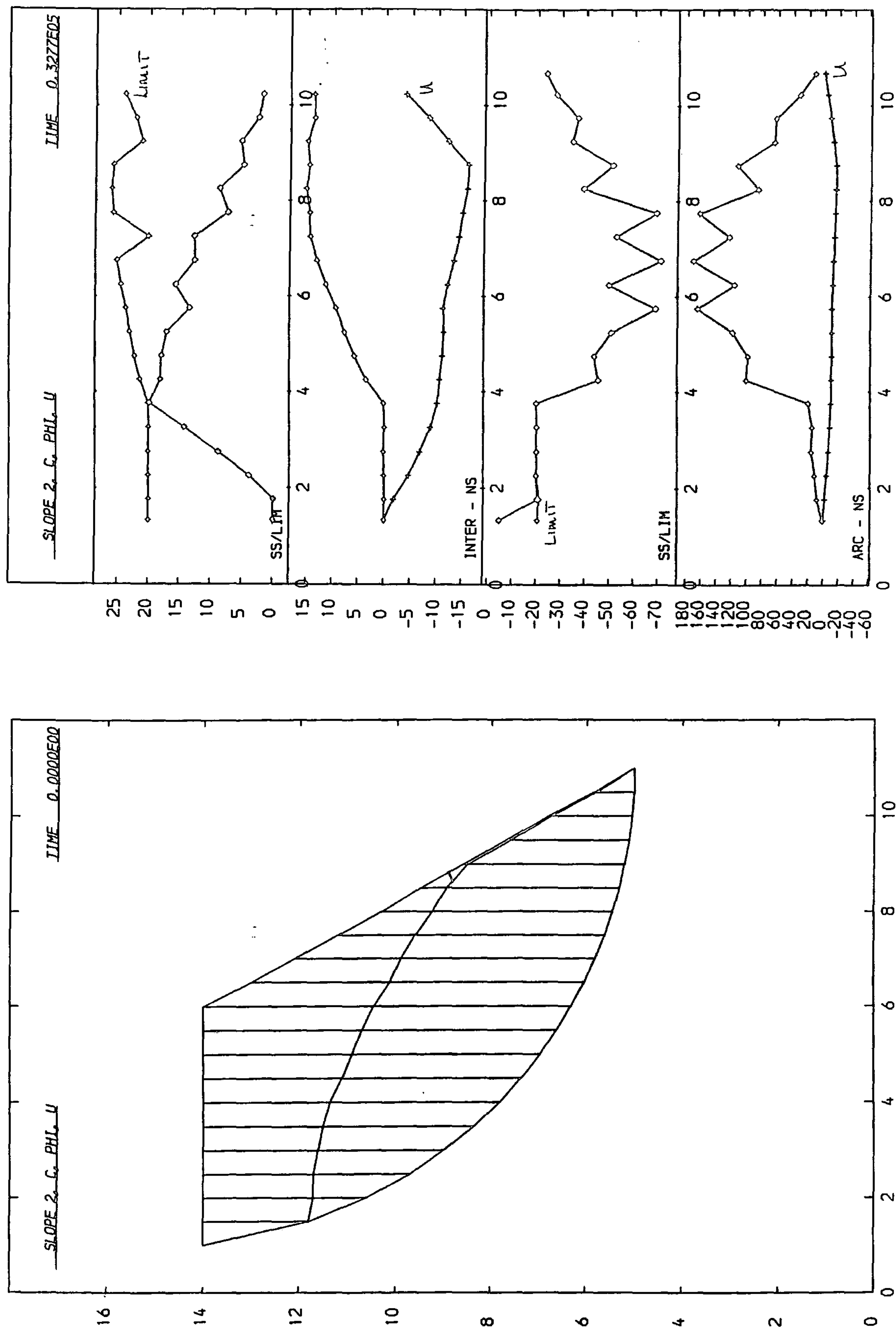
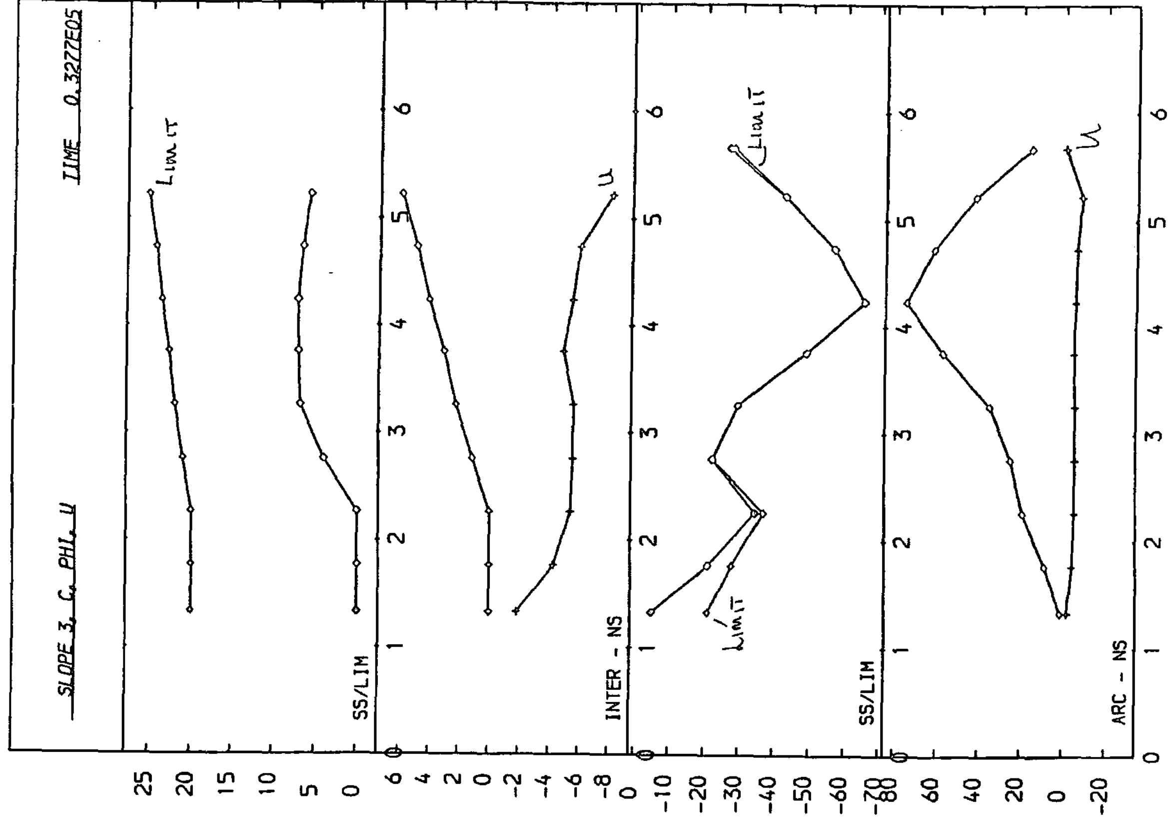


Figure 3.16 Stress profiles for result set 11

```
set echo off  debug update on
set damp 0.00125 0.005 gravity -10
start  SLOPE 3, C, PHI, U
  0  7 0
   create free 20.0 46 2.0 1.0 20.0 46 0.225 0.45  0.23 1.0 14.0  1.0 14
                                                        1.5 11.8  1.5 14
   create free 20.0 46 2.0 1.0 20.0 46 1.3   0.85  0.23 2.0 10.6  2.0 14
   create free 20.0 46 2.0 1.0 20.0 46 1.925 1.075 0.23 2.5  9.7  2.5 14
   create free 20.0 46 2.0 1.0 20.0 46 2.25  1.175 0.23 3.0  9.0  3.0 14
   create free 20.0 46 2.0 1.0 20.0 46 2.425 1.25  0.23 3.5  8.35 3.5 14
   create free 20.0 46 2.0 1.0 20.0 46 2.475 1.225 0.23 4.0  7.8  4.0 14
   create free 20.0 46 2.0 1.0 20.0 46 2.35  1.125 0.23 4.5  7.35 4.5 12
   create free 20.0 46 2.0 1.0 20.0 46 2.075 0.95  0.23 5.0  6.95 5.0 10
   create free 20.0 46 2.0 1.0 20.0 46 1.7   0.75  0.23 5.5  6.6  5.5  8
   create free 20.0 46 2.0 1.0 20.0 46 0.75  0.0   0.23 6.0  6.3  6.0  6.3
   meshend
  set time 1 cmdproc off framelimit 100
    writegap 2000  interval 2500
    cmdlist plot standard set calc writegap * 2 interval * 2 cend
    echo on
  plot page go 40000  stop
```

Table 3.25 Input commands for result set 12 and 15



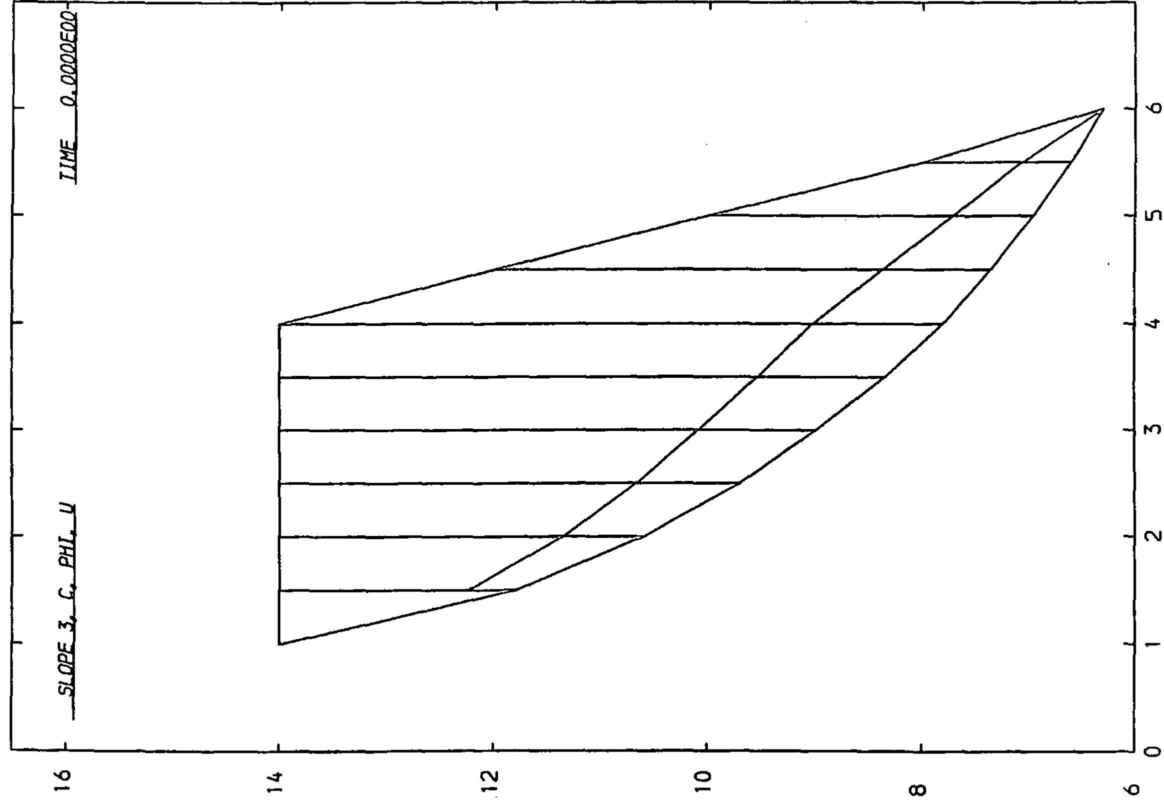
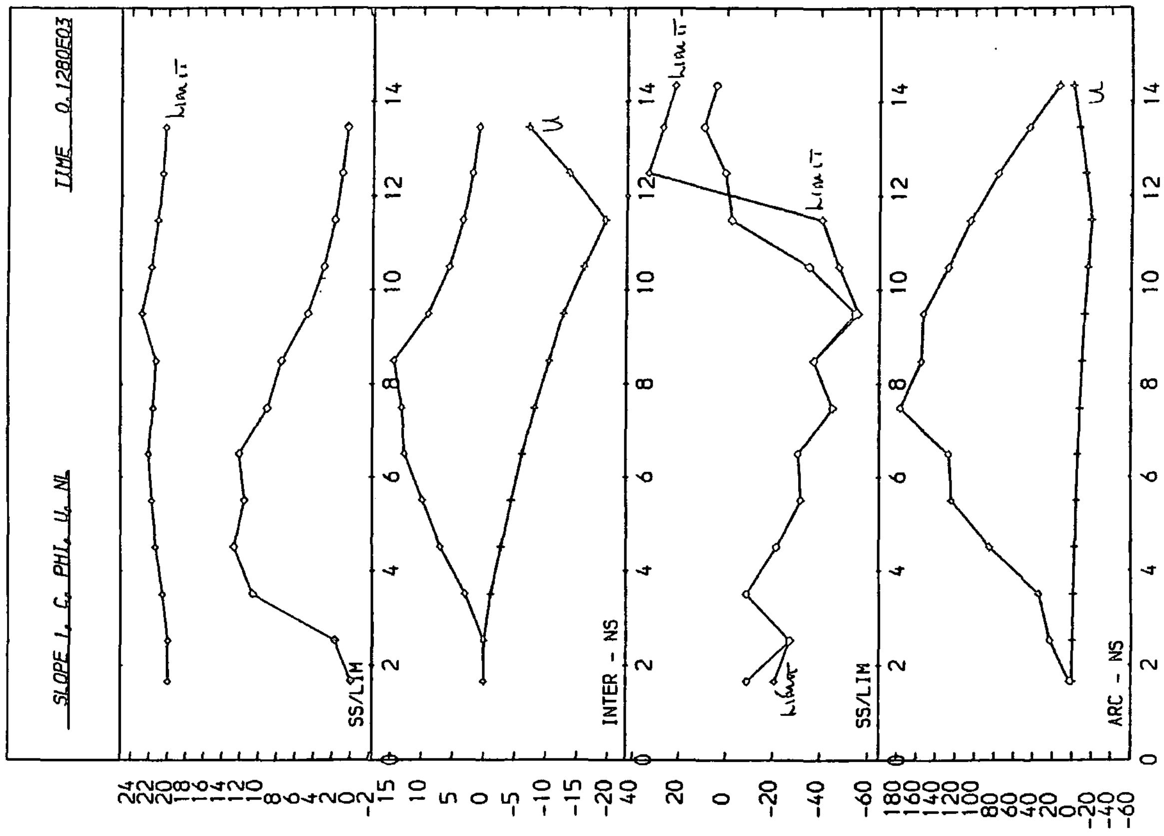


Figure 3.17 Stress profiles for result set 12



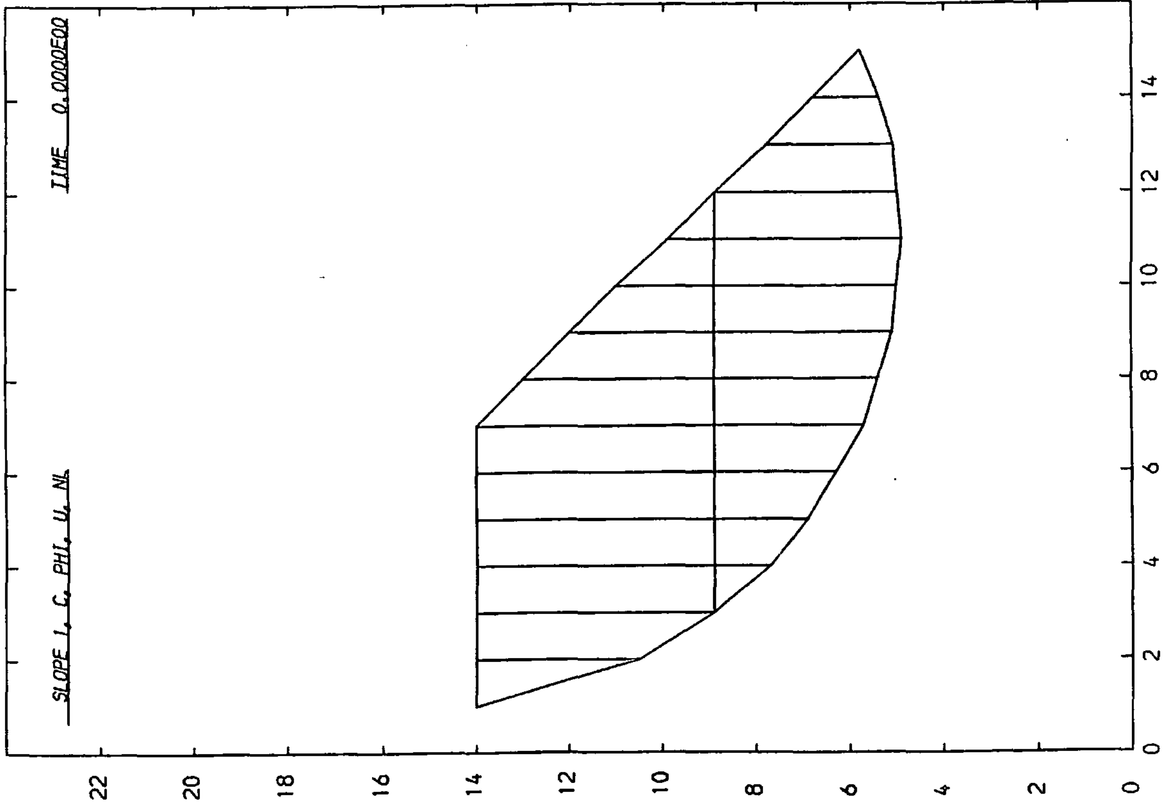
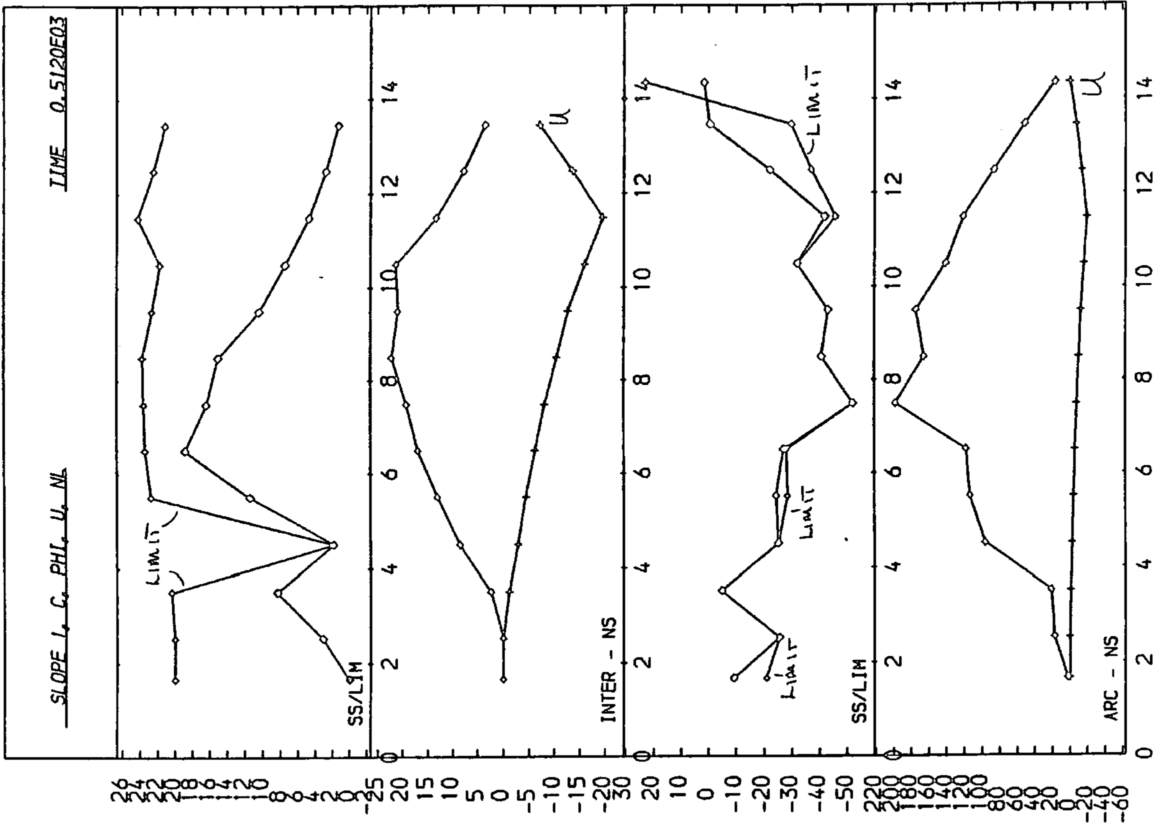


Figure 3.18 Stress profiles for result set 13



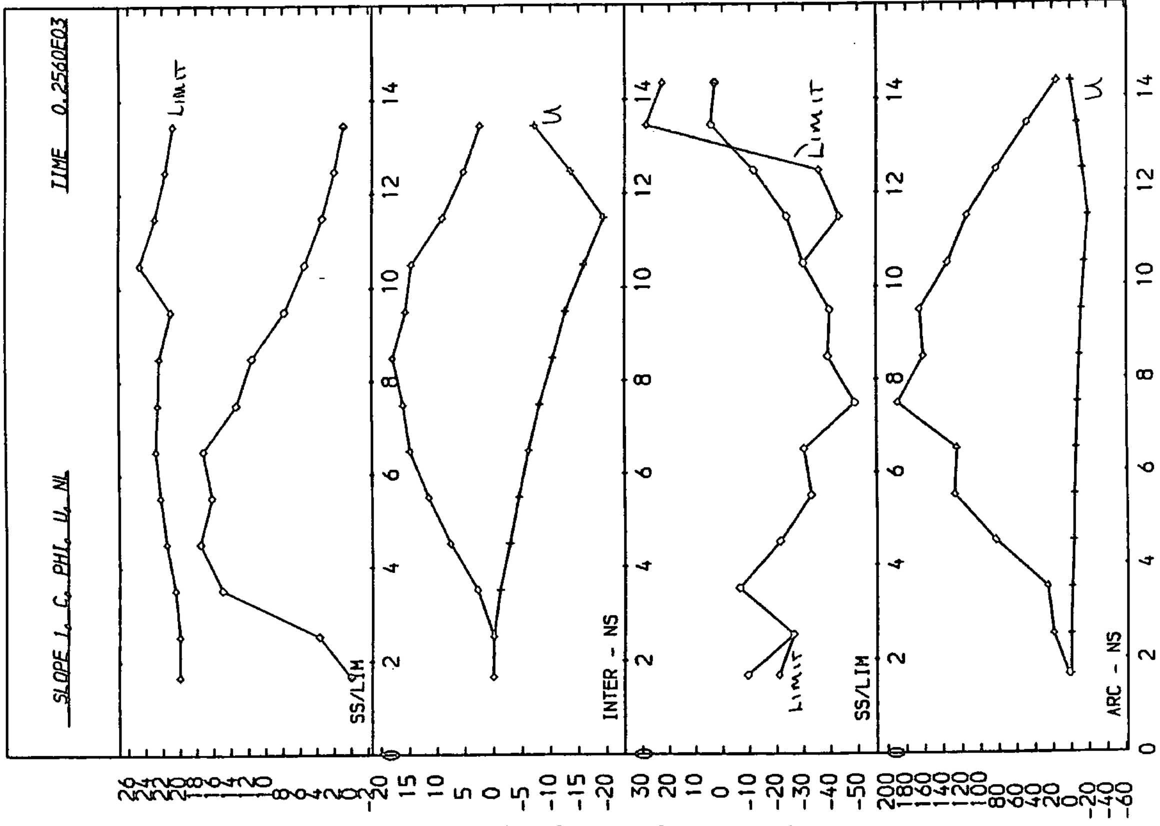
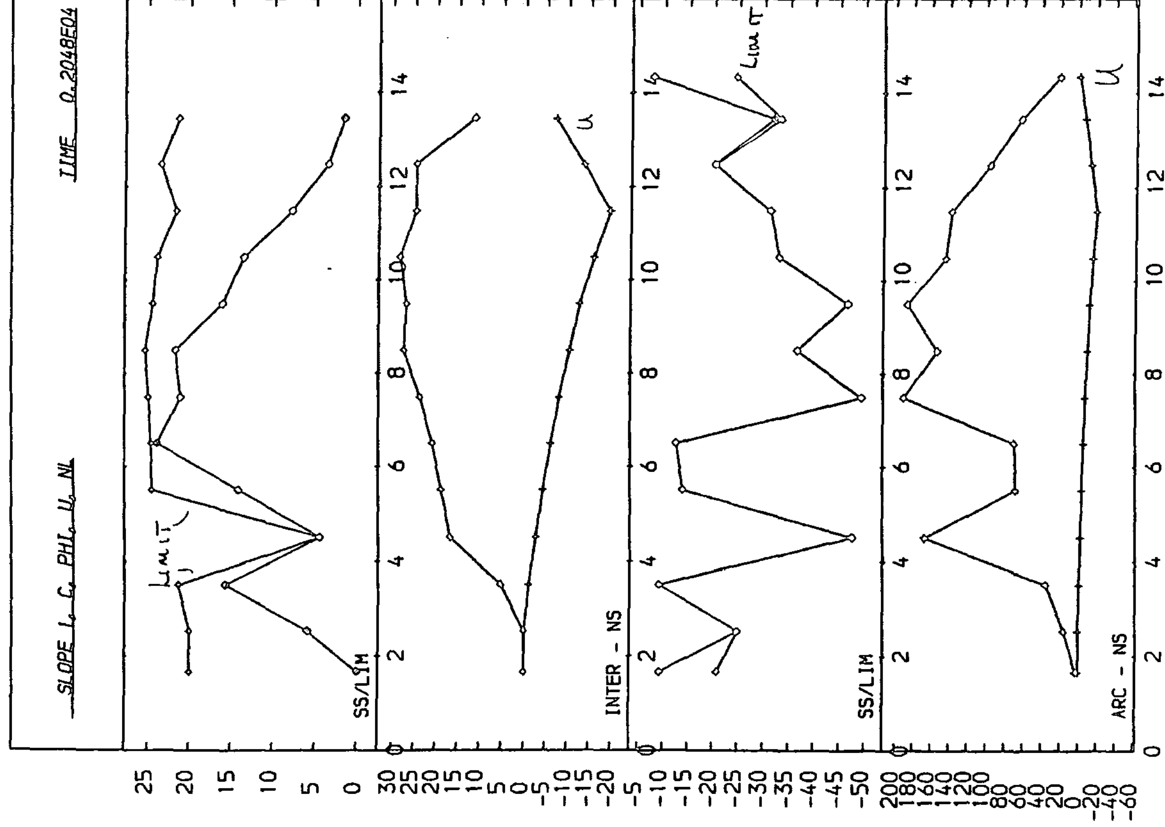


Figure 3.18 Stress profiles for result set 13 (continued)



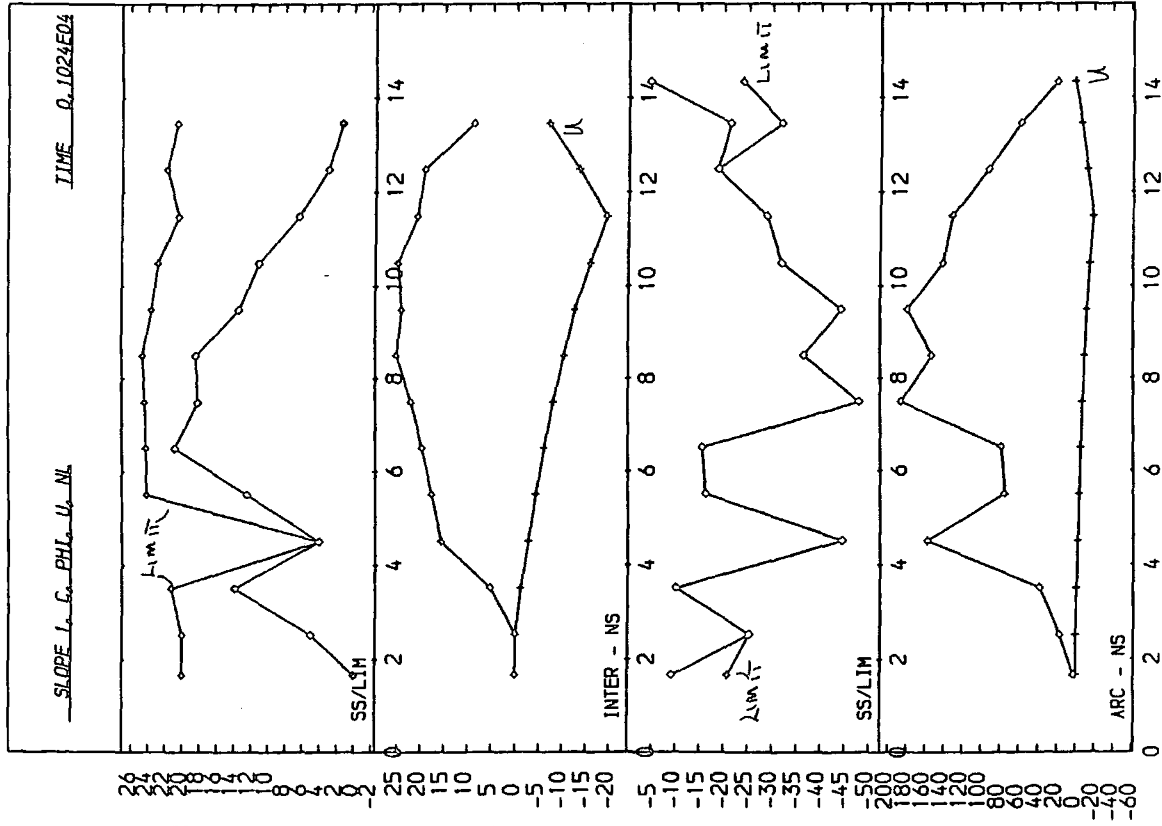
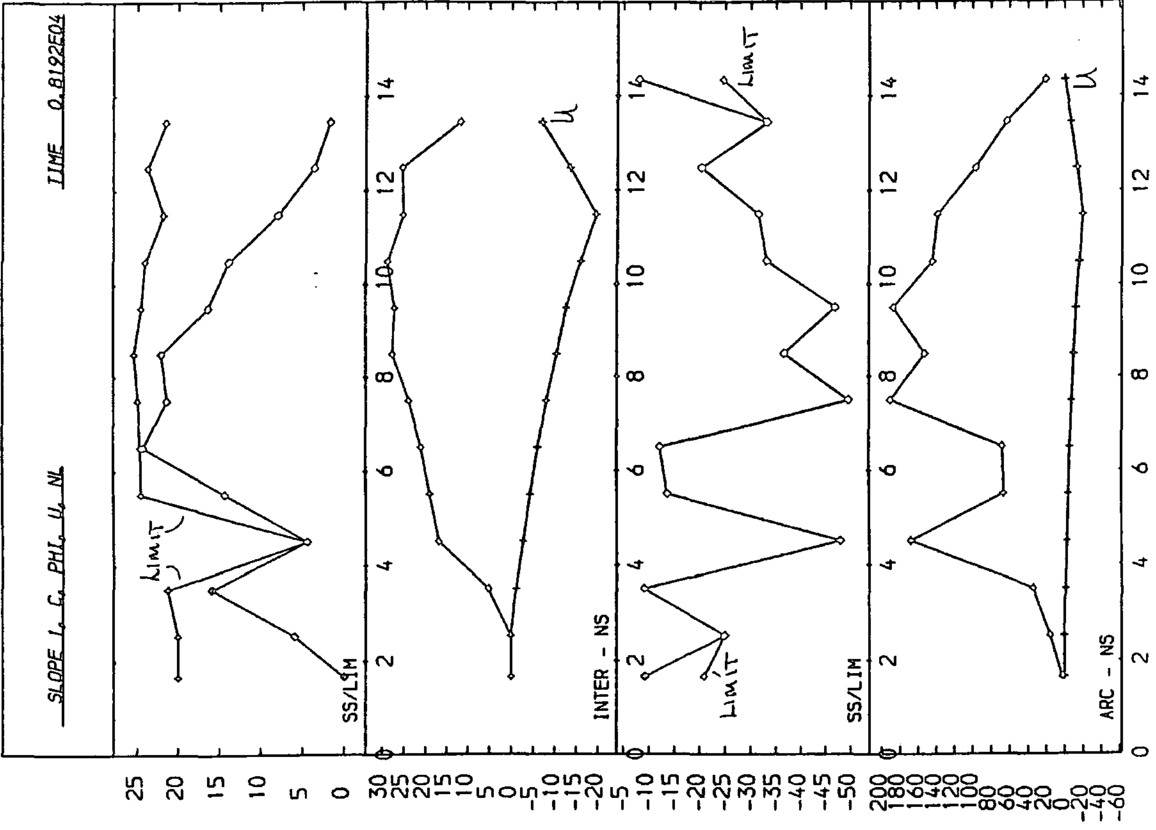


Figure 3.18 Stress profiles for result set 13 (continued)



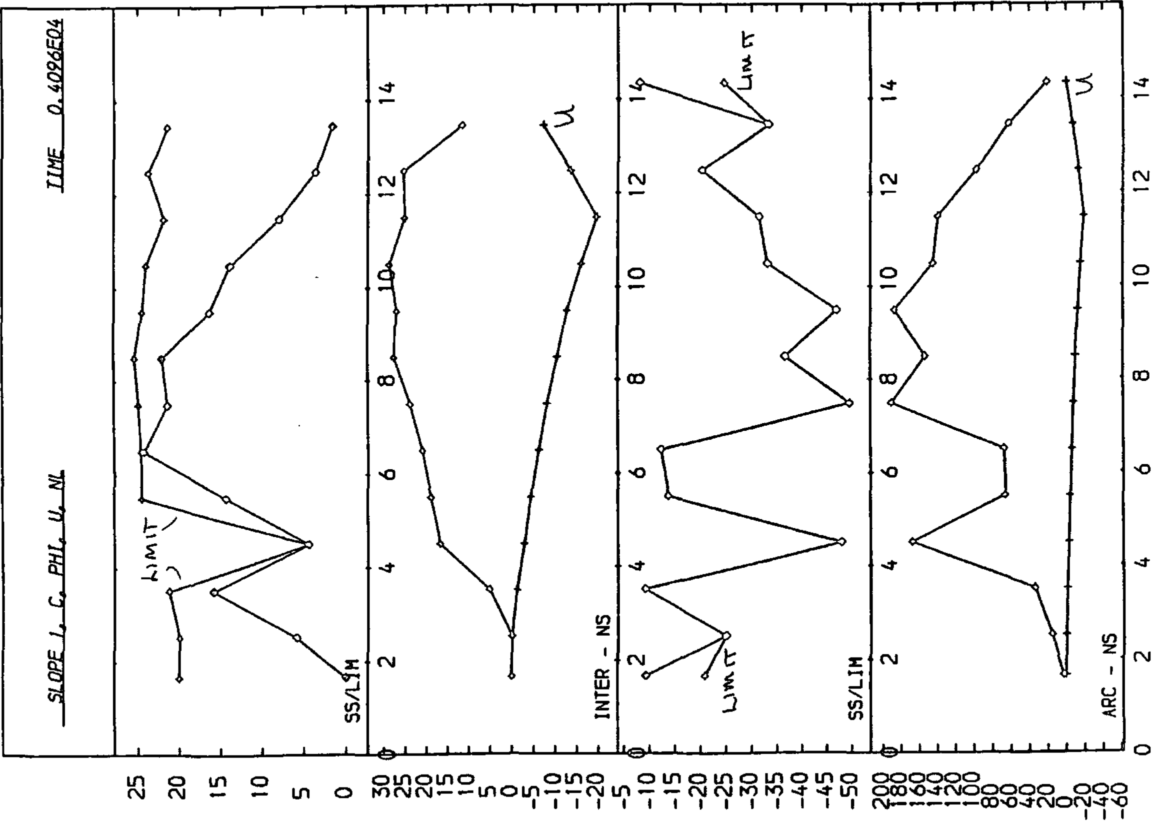


Figure 3.18 Stress profiles for result set 13 (continued)

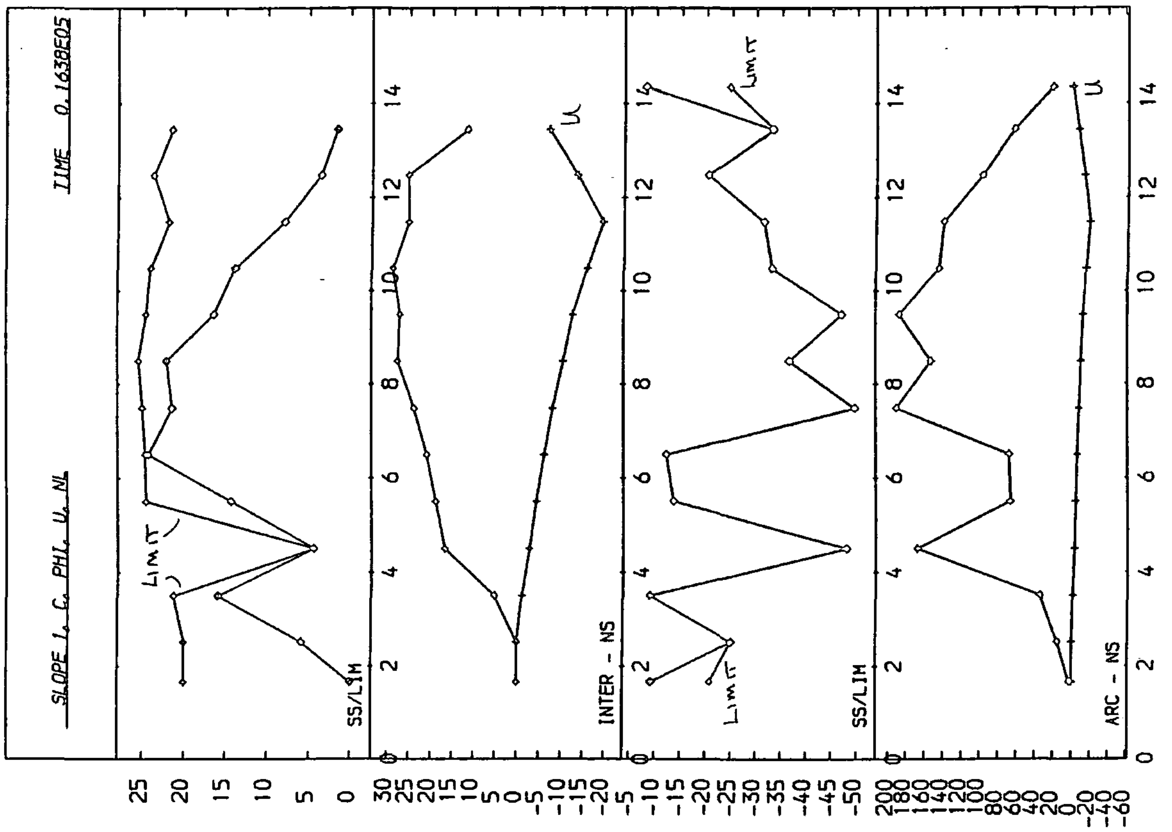
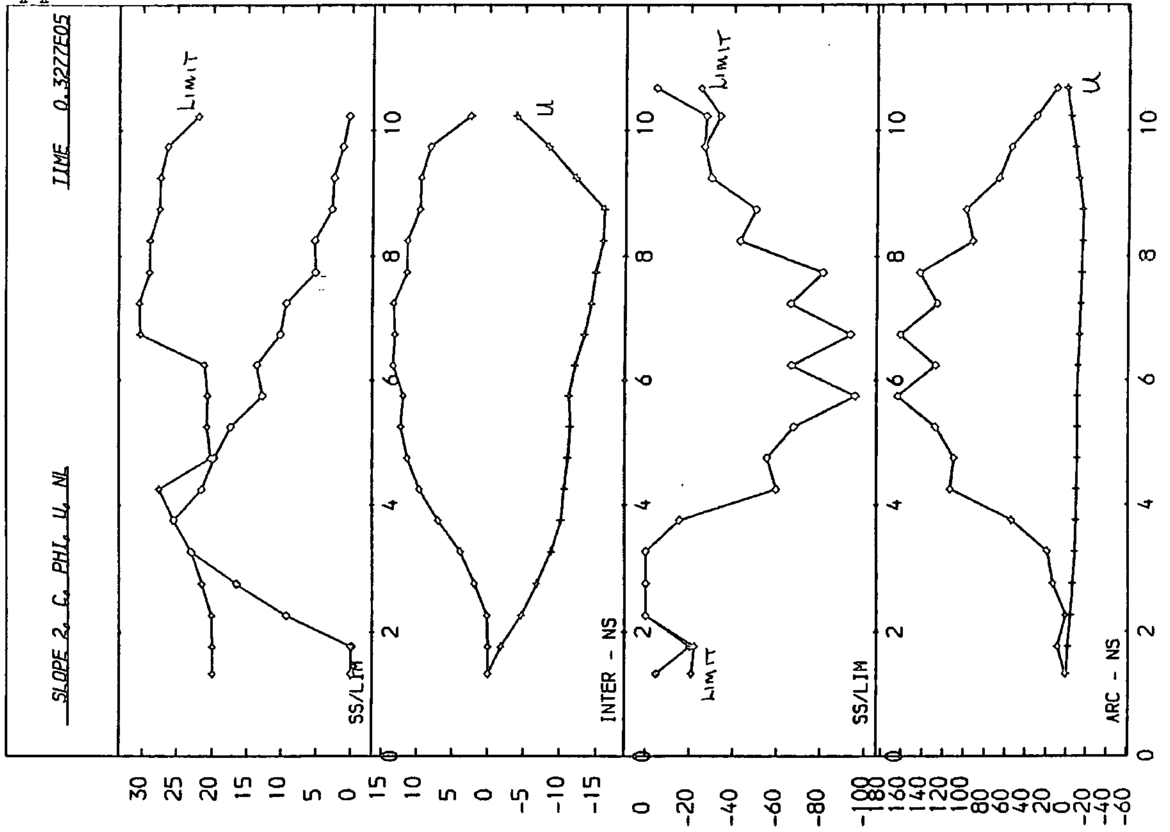


Figure 3.18 Stress profiles for result set 13 (continued)



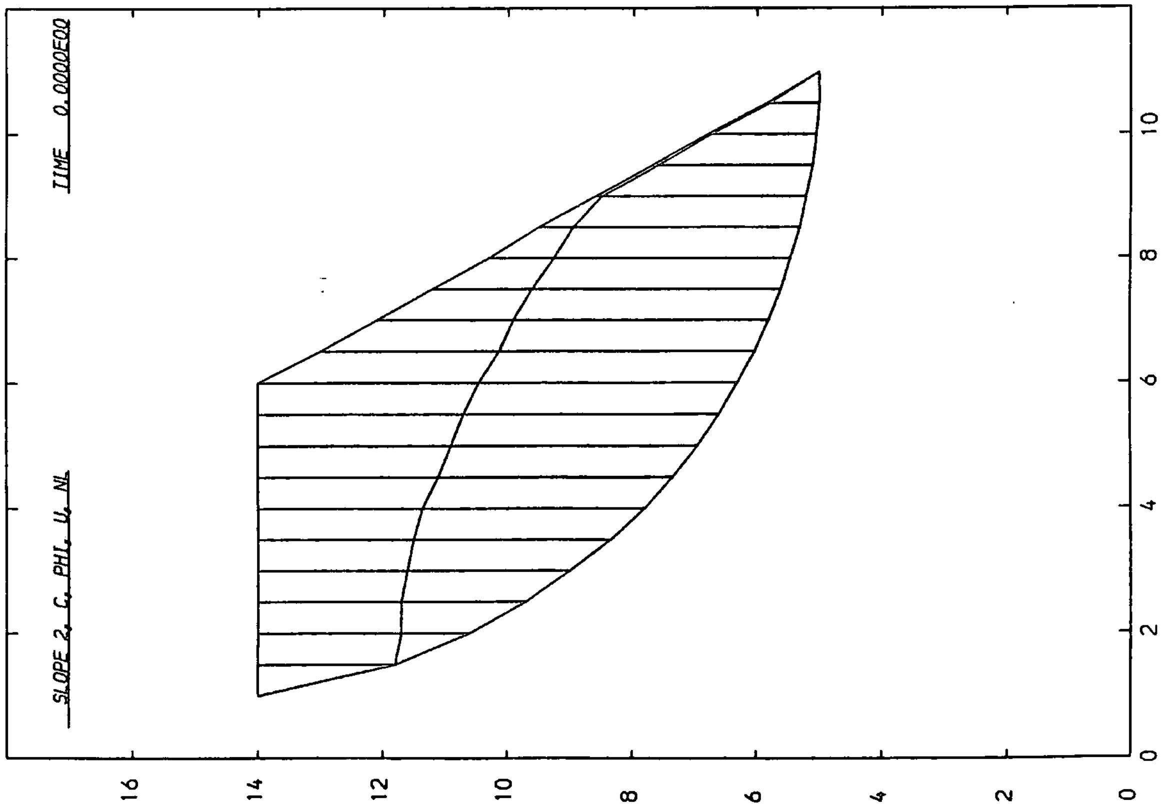
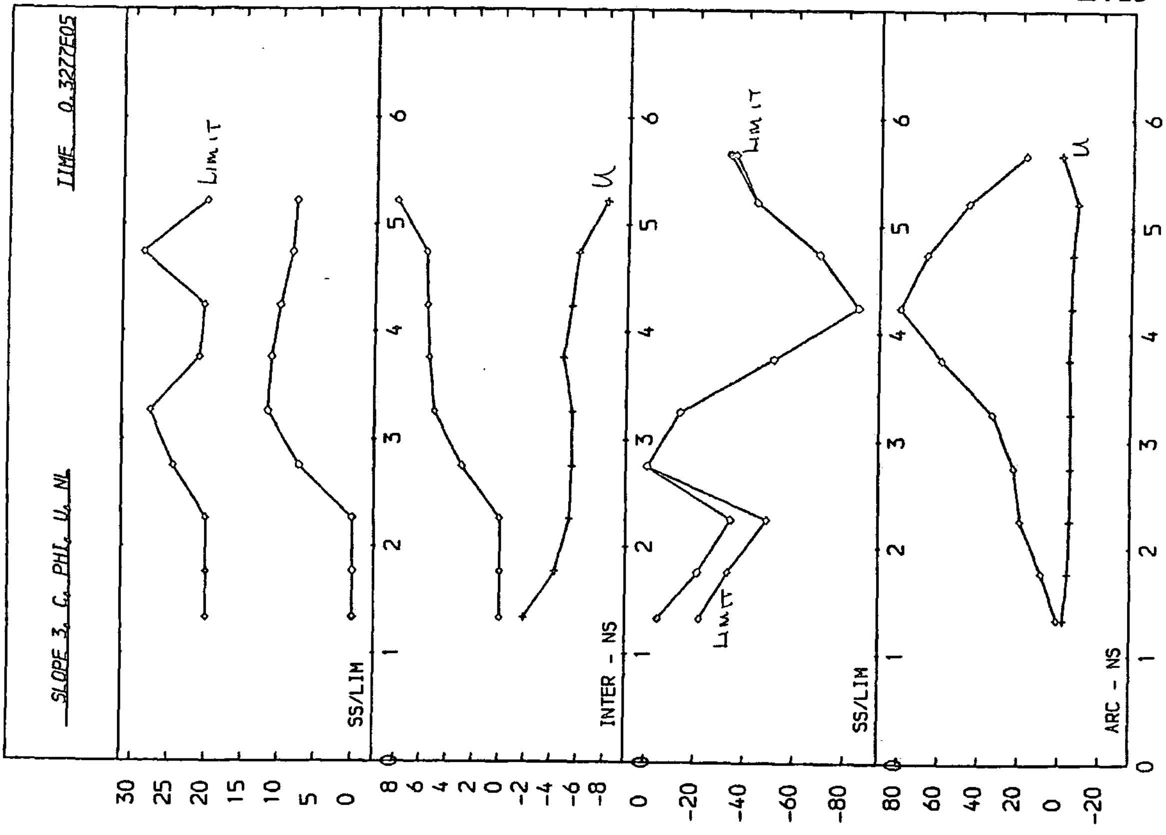


Figure 3.19 Stress profiles for result set 14



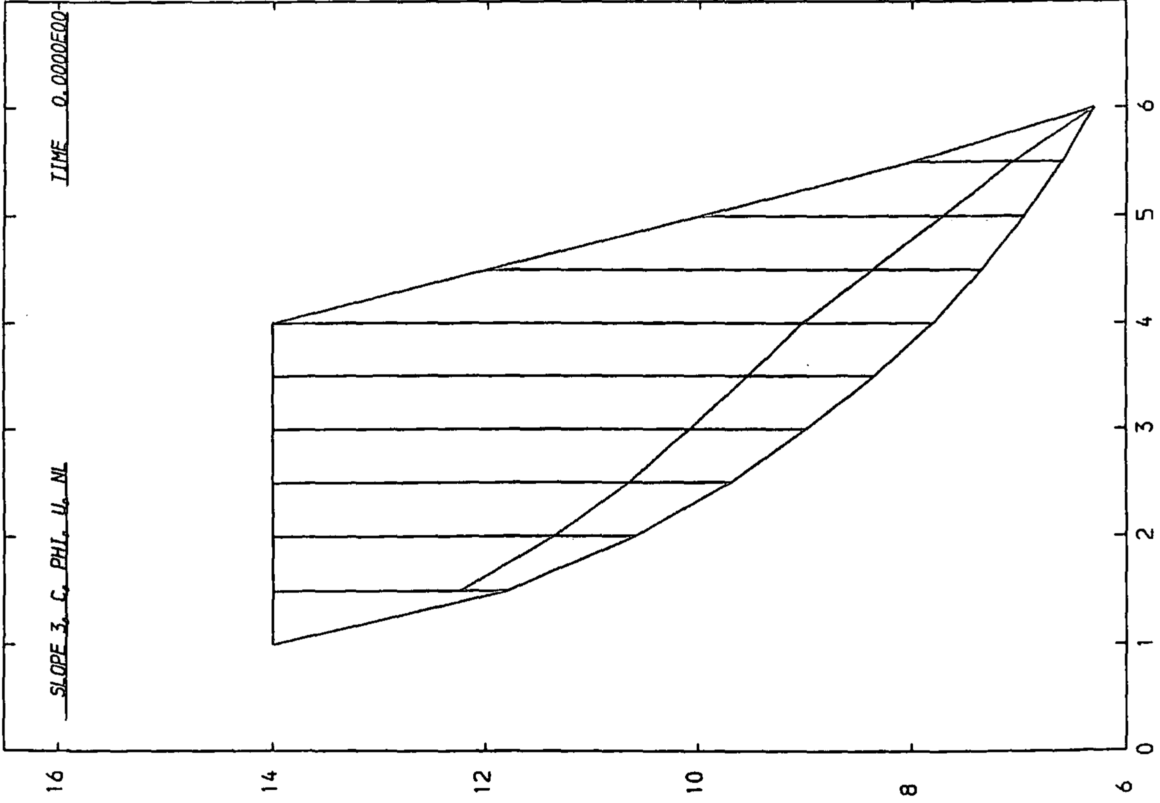


Figure 3.20 Stress profiles for result set 15

APPENDIX C
PROGRAM SLICES
STRUCTURE CHARTS

Appendix C is a series of structure charts for the procedures and functions that make up program SLICES. The structure charts consist of various boxes linked by lines. The boxes represent logical units of code and the lines represent the flow of control from one part of the program to another.

The rounded boxes are descriptive, indicating the start and end of the charts. They may also be used to show logical processes that have not been broken down in to their constituent parts. In this respect they may be viewed as comments.

The rectangular boxes that are double sided indicate a call to another procedure and hence a flow to another chart. The normal boxes however, represent logical processes within the confines of the current chart.

The flow of program control is always from the top to the bottom of the chart. The current chart is exited when the bottom is reached. Control then falls back to a previous chart. Flow, within the chart is along the joining lines. At a junction of lines, flow continues by turning left. On reaching the end of a branch the flow returns to the last junction and continues along the righthand branch. The case and loop structures are, however special constructs.

Loop structures are shown by long loops eminating from a control box. The control box indicates the termination condition. Flow continues around the loop in a clockwise direction. Case structures are shown by a contol box with a series of diamond decision symbols. The decision diamonds are normally associated with

two boxes. The one to the left on the chart is the case condition. If this condition is met then program flow continues by branching left, as normal. If the condition is not met then flow continues down the chart.

Some explanatory text is associated with each chart, which is organised in the same order as the procedure headers occur in the program. This causes some local procedures to appear later here than in the source. The charts for procedure headers that have the *forward* directive are placed in the correct logical sequence, that is with the later definitions.

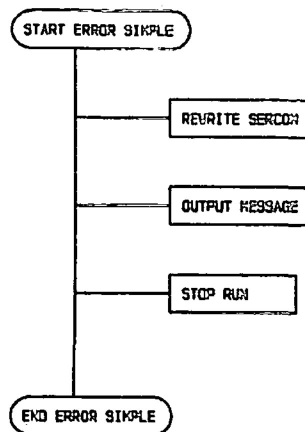


Figure C.1 Chart for procedure *error_simple*

PROCEDURE error_simple(ob, caller: string(40)); This is a global procedure and immediately halts the program after the production of an explanatory message. This is only called when an irretrievable situation occurs.

PROCEDURE word_scan(var cmds_in : text; var word:string(12)); This global procedure is called from repeater, parameters and get_command. It reads a set of consecutive non–blank characters to form a word of maximum length 12, and passes it back to the calling procedure in *word.* It reads from the file device unit buffer given in *cmds_in* and, if reading from a terminal prompts the user for a command. This procedure looks after end of line conditions and skips all blanks between words and all comments by calling the local procedures *skipblks* and *skipcomment.*

PROCEDURE skipblks(VAR ch : string(1)); A local procedure to *word_scan*, this simply reads characters until a non–blank is encountered, this is returned in ch.

PROCEDURE skipcomment(VAR ch : string(1)); A local procedure to *word_scan*, this reads characters until the end comment symbol '}' is encountered. This will recursively call itself on encountering another comment symbol '{'.

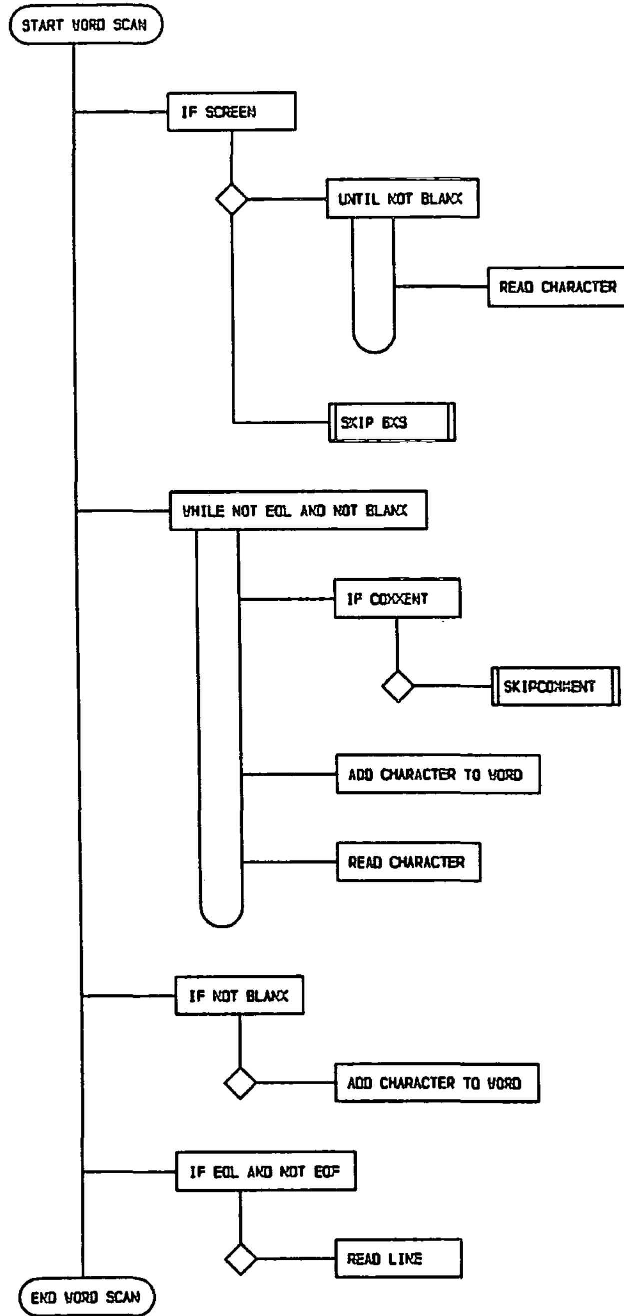


Figure C.2 Chart for procedure *word_scan*

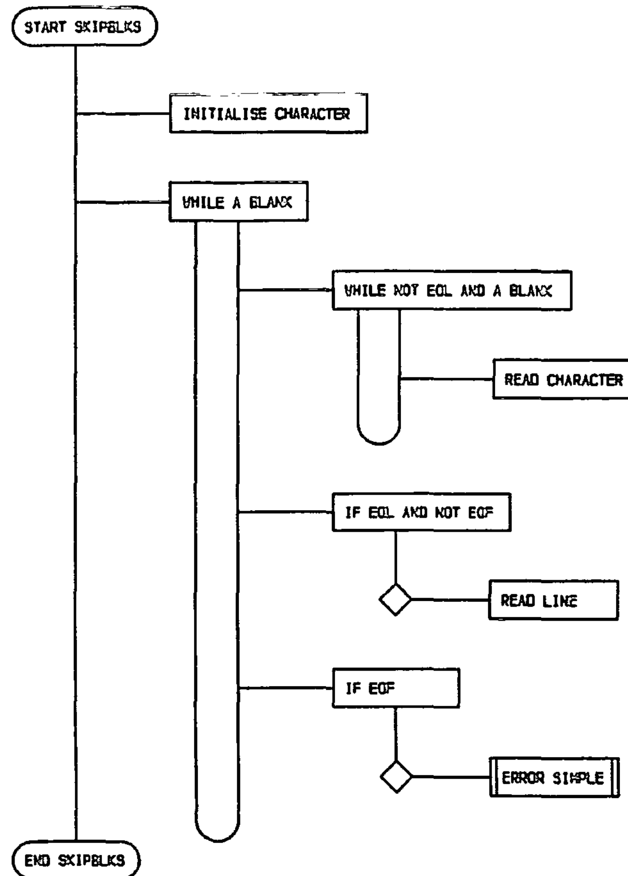


Figure C.3 Chart for procedure *skipblks*

PROCEDURE start_shut(Var cmd_i : text; starting : start_type); FORWARD; Procedure start_shut is defined later, but is headed here as it is called from procedure *trapper.*

PROCEDURE control(var cmd_i : text); FORWARD; Procedure *control* is defined later, but must be headed before reference can be made to it.

PROCEDURE trapper; This is called from procedures *cycle* and *get_command*, and is executed when an attention interrupt is passed by the system to the program.

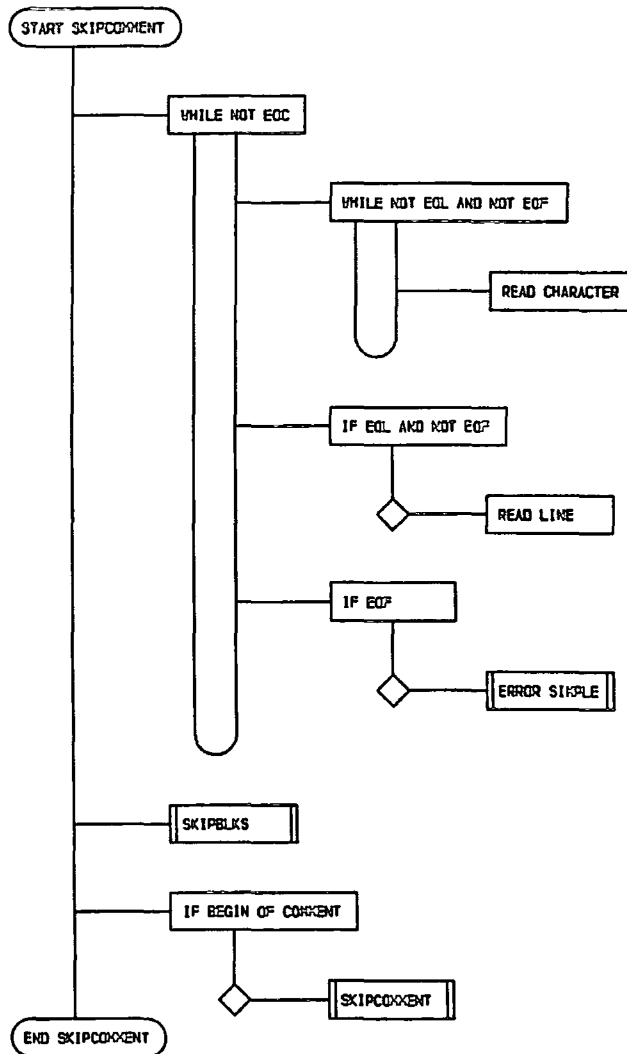


Figure C.4 Chart for procedure *skipcomment*

The user is prompted for confirmation before the run is terminated by calling *start_shut.*

PROCEDURE get_command(caller : call_type; var quiter : boolean; var retcom : com_type; intcall : string(12); cmds_ig : text);

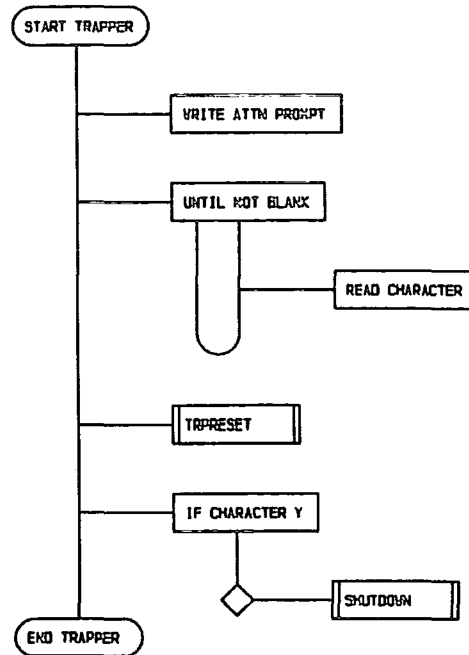


Figure C.5 Chart for procedure *trapper*

This procedure is called with five parameters. *caller* designates which command set is valid. *quiter* is defined on exit and determines if the calling procedure should exit or continue with another command from the same set. *retcom*, on exit, contains the valid command scaler value. *intcall*, on entry, is either a null or contains the literal value of the internal command to be executed. *cmds_ig* is the file device unit buffer pointer from which input should be read.

This is the heart of the ICL parser, it is called under three conditions, internally, externally and recursively on error.

Under internal use the parameter *intcall* is set to the command. Under normal use the procedure *word_scan* is used to obtain a word from the file device unit buffer pointer in *cmds_ig*. The word obtained (or *intcall*) is checked against a string of all the valid commands and the resulting command type (including null if the word is in error) is compared to those which are relevent to the calling procedure. There

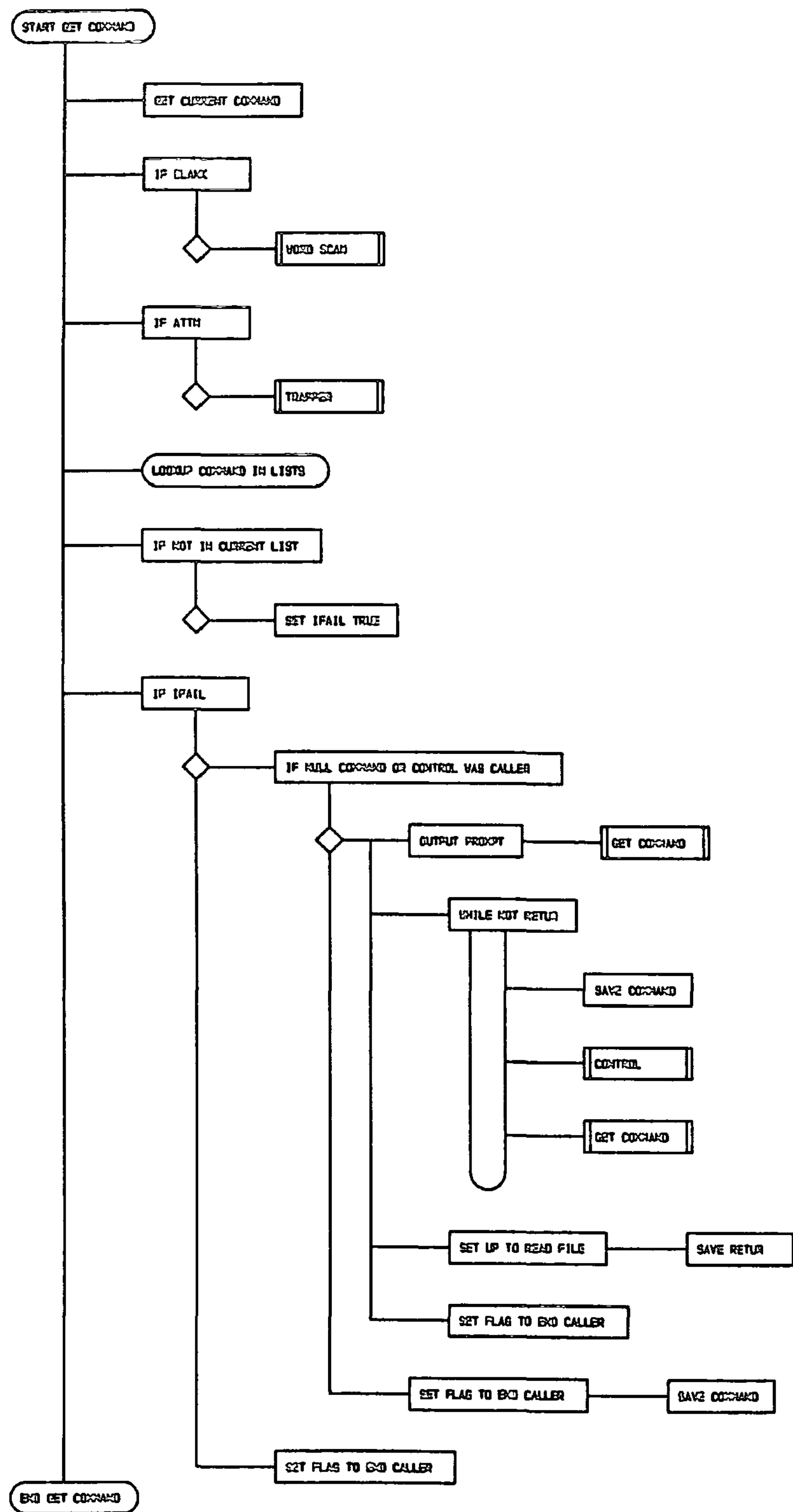


Figure C.6 Chart for procedure *get_command*

are three possible results, a relevent valid command, an irrelevent valid command and lastly one which is invalid.

In the first of these results, the routine exits with *quiter* set to true onlyif the command was internal. In the second, *quiter* is set and the word is placed into the variable *gi.nextword*, to be processed the next time *get_command* is called normally. In case three, the user is notified and *get_command* called (direct recursion) with the error communication file device unit buffer to provide a replacement. If a replacement is received, procedure *control* is called (indirect recursion) and input continues from the error communication. If further errors occur then more levels of recursion take place. The error condition is terminated by `return`, when the procedures fall back with this in the variable *gi.nextword* so that it is continuously processed until the first invocation of *get_command* is exited. At this point exit is made back to procedure *control* with *quiter* set to false to prevent program termination.

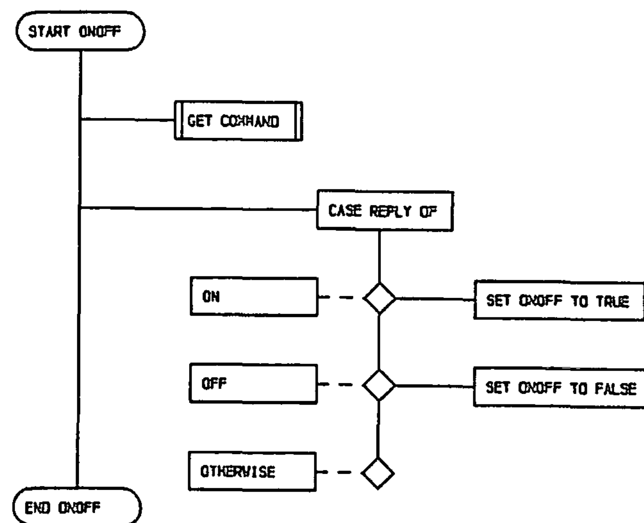


Figure C.7 Chart for function *onoff*

FUNCTION onoff(var cmd_i : text) : boolean; This global function calls *get_command*. The two possible commands are on, which causes this to return true, and *off* which causes this to return false.

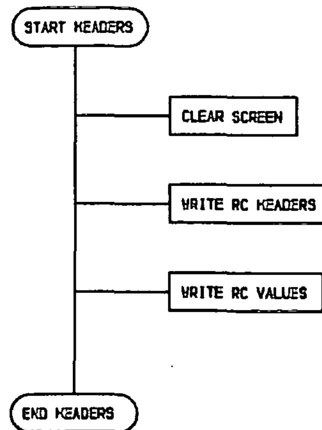


Figure C.8 Chart for procedure *headers*

PROCEDURE headers; This global procedure initiates the headings for the running commentary.

PROCEDURE factors_of_safety(el : ptr_type); This global procedure calculates the factors of safety for the base contacts of each of the slices. These values are written to the file attached to unit 7, the debug output file.

FUNCTION sign(val,donor:real) : real; This function receives two real values, and returns the value of the first with the sign of the other.

PROCEDURE initialise_globals; This is executed only once and sets the values of all the global variables to zero, default or nil values.

All of the preceeding modules are global in scope, that is they may be called from anywhere within the program. There is one restriction to this, they may only

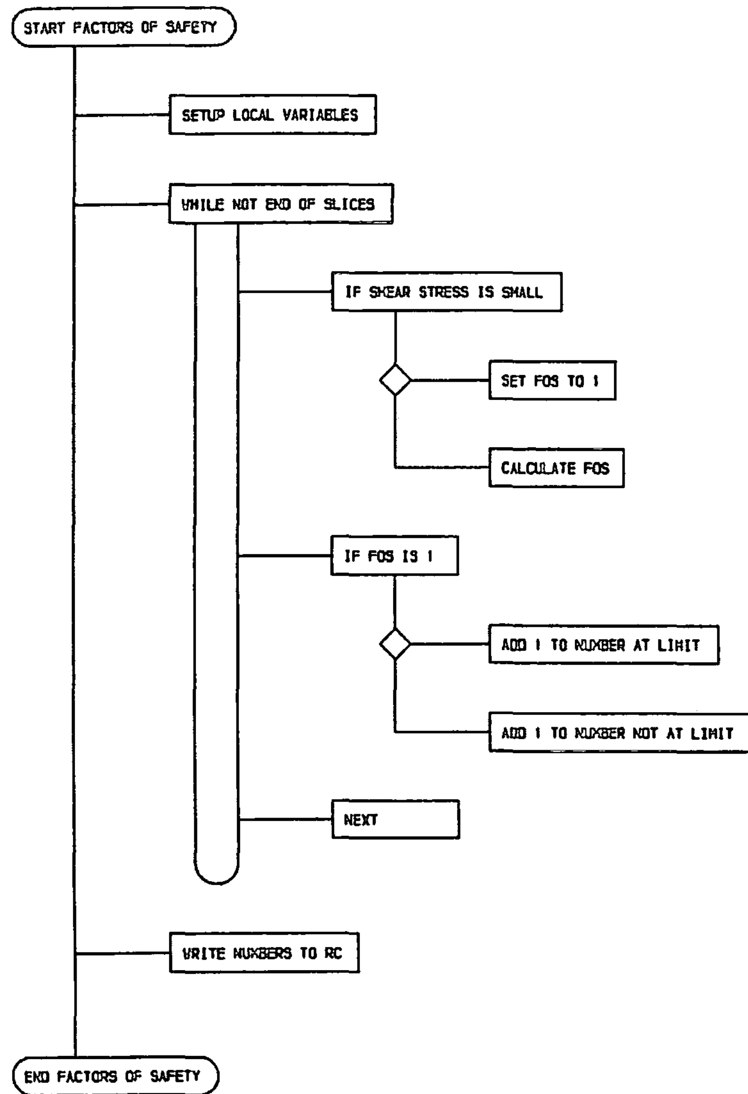


Figure C.9 Chart for procedure *factors_of_safety*

be called after they have been defined. In the cases of *control* and *start_shut* the *FORWARD* directive on the headings indicate that the definitions are provided later in the source code. The remaining modules are only called from *control*.

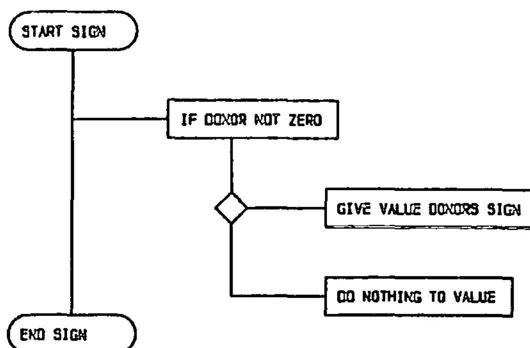


Figure C.10 Chart for function *sign*

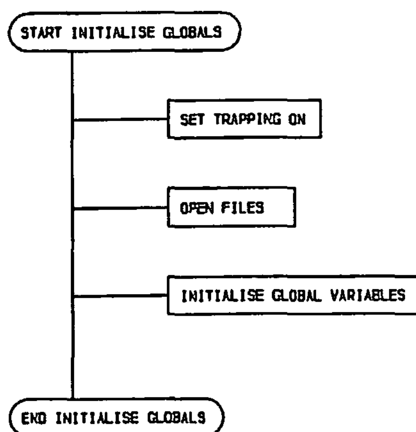


Figure C.11 Chart for procedure *initialise*

PROCEDURE plots(var cmd_i : text; plot_command : string(12)); This routine contains the calls to the *ghost library subroutines. The structure of this routine is simple and consists of a repeat loop. Essentially two processes are carried out in the loop. Firstly *get_command* is called, and then a case statement causes the relevent command to be executed. It is within the case statement that the local procedures are called. The repeat loop is exited when the value passed back from *get_command* in *plotquit* is true.

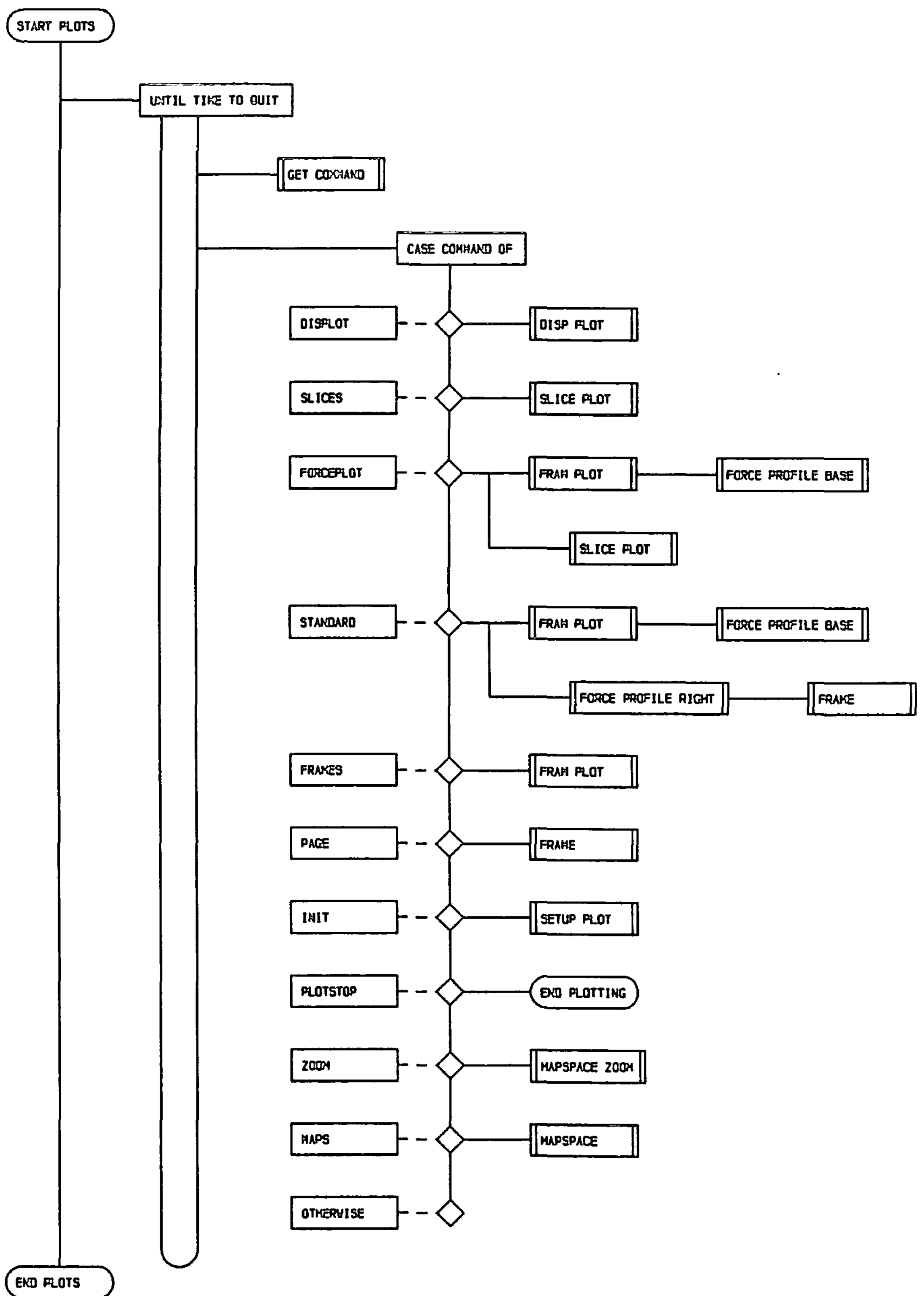


Figure C.12 Chart for procedure *plots*

During internal command processing *plots* is called with *plot_command* set to a literal value and this is passed directly to *get_command* in the parameter list.

PROCEDURE map_space(var cmd_i : text; sp_comst : string(12)); This is a local procedure to procedure *plots*. *map_space* has the same structure as *plots*, except that it contains two case statements. The first manipulates the plot space and the second the mapping onto this plot space. The repeat condition is dependent upon the value of *mapquit*, which is passed back by *get_command.*

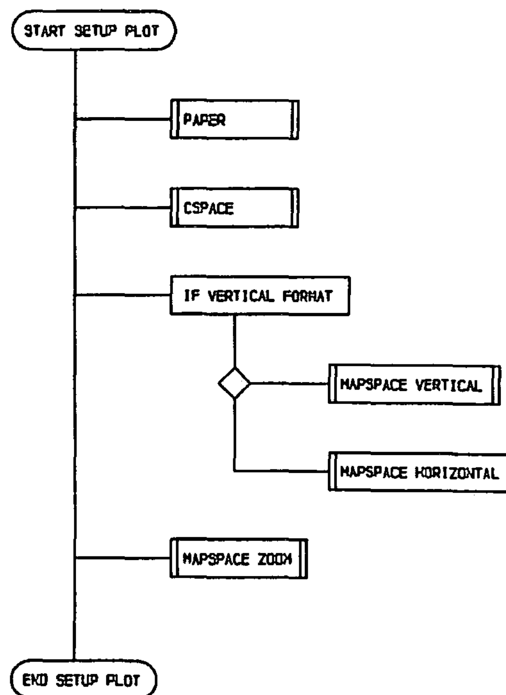


Figure C.14 Chart for procedure *setup_plot*

PROCEDURE setup_plot; This is a local procedure to procedure *plots*. The intial format is set up and the plot output stream turned on.

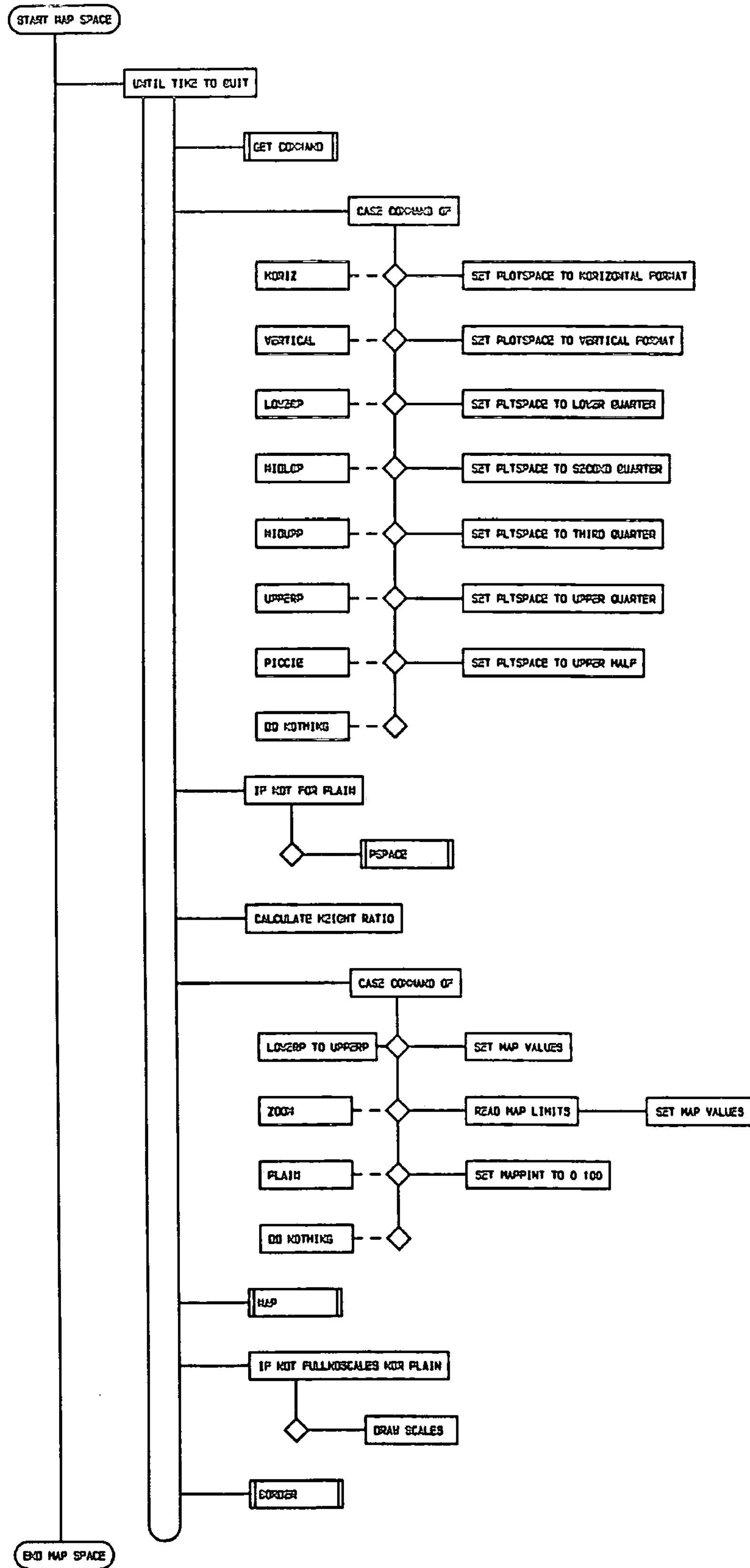


Figure C.13 Chart for procedure *map_space*

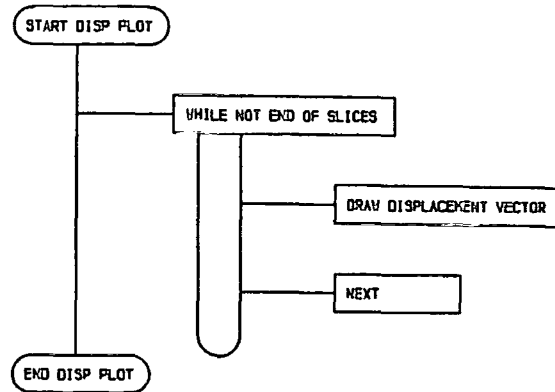


Figure C.15 Chart for procedure *disp_plot*

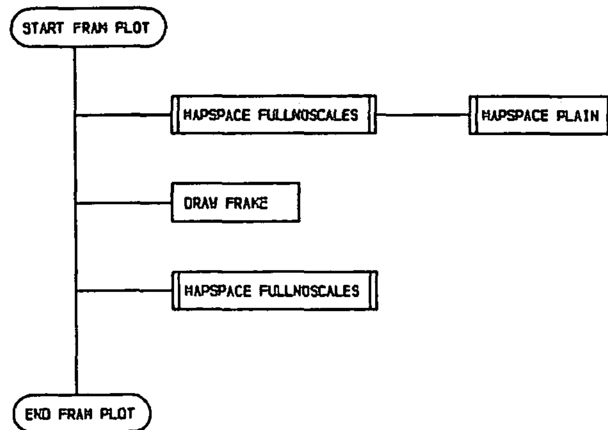


Figure C.16 Chart for procedure *fram_plot*

PROCEDURE disp_plot(el : ptr_type); This is a local procedure to procedure *plots.* A plot of the slice body displacements is produced.

PROCEDURE fram_plot; This is a local procedure to procedure *plots* and produces a frame around the main plot space.

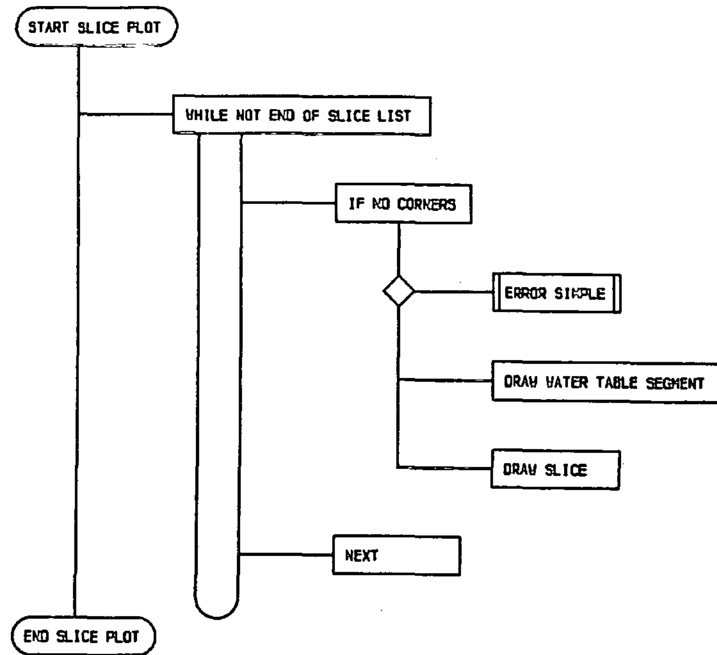


Figure C.17 Chart for procedure *slice_plot*

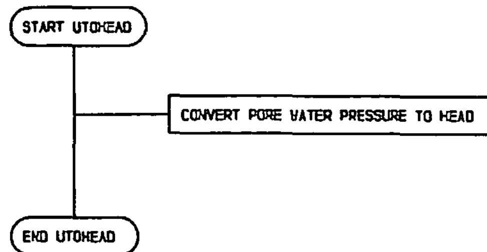


Figure C.18 Chart for function *utohead*

PROCEDURE slice_plot(el : ptr_type); This local procedure to procedure *plots* causes the slice geometry to be plotted.

FUNCTION utohead(el : ptr_type); real; This local function to procedure *slice_plot* caonverts the pore water pressure to an equivalent height for plotting of the water table.

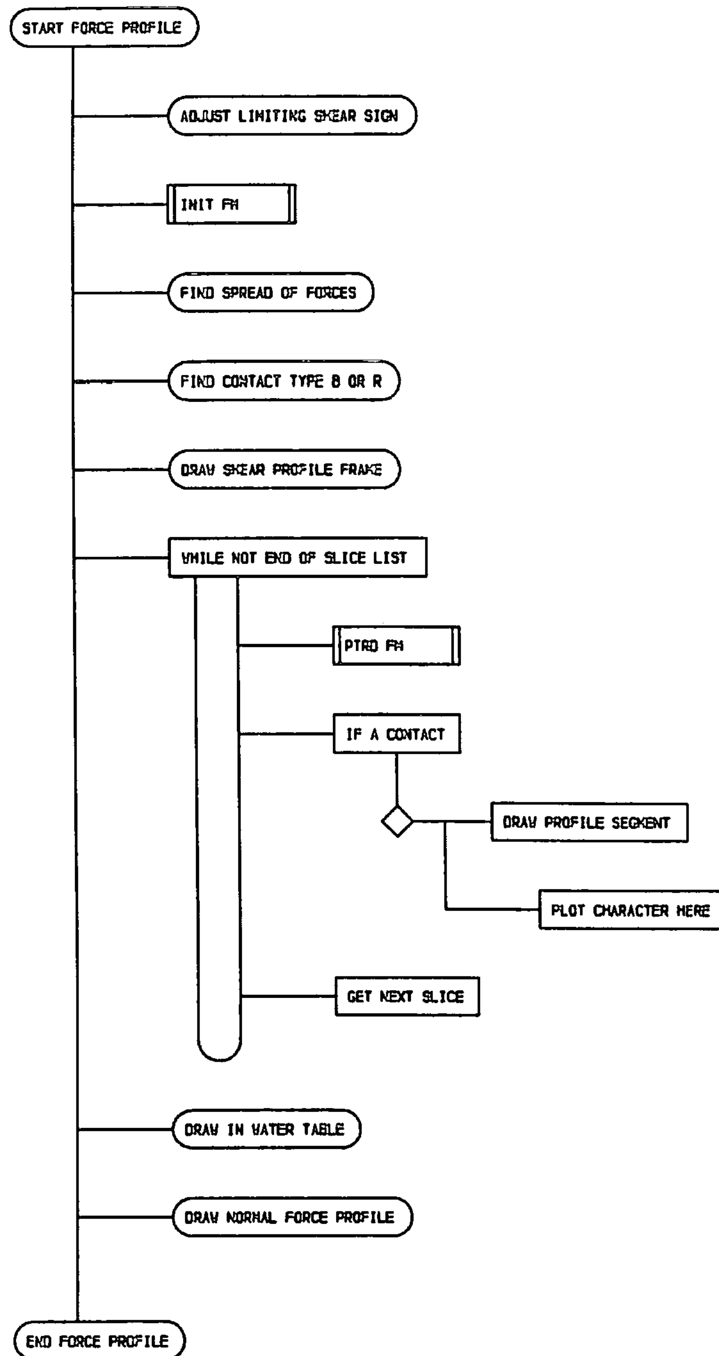


Figure C.19 Chart for procedure *force_profile*

PROCEDURE force_profile(ele : ptr_type; dire : dir_of_contacts); This is a local procedure to procedure *plots*. *force_profile* produces formats by calling *map_space* internally and plots the stress profiles for either the base or the side contacts according to the value of *dire* being either *based* or *righthand.*

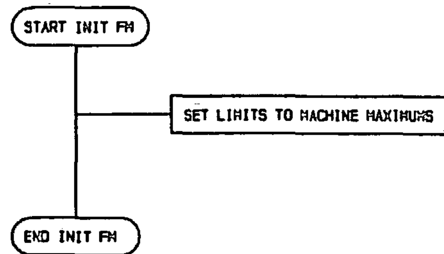


Figure C.20 Chart for procedure *init_fm*

PROCEDURE init_fm; A simple procedure for initialising stress mapping values before finding maxmimum and minimum values for scaling the stress profiles. It is local to *force_profile.*

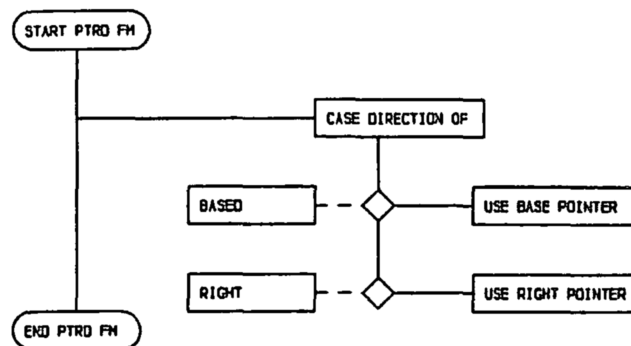


Figure C.21 Chart for function *ptrd_fm*

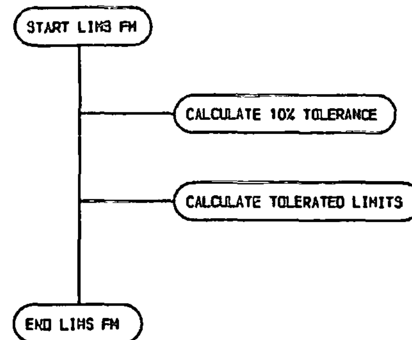


Figure C.22 Chart for procedure *lims_fm*

FUNCTION ptrd_fm(elem : ptr_type; dire : dir_of_contacts) : con_ptr; ptrd_fm is local to *force_profile* and finds the contact pointer for an element depending on whether side or base contacts are being plotted.

PROCEDURE lims_fm(VAR miny, maxy : real); This adds a ten percent margin to the stress mapping values and is local to *force_profile.*

PROCEDURE cycle(var cmd_i : text); This procedure controls the calculation sequence. The structure of *cycles* consists of a small block dealing with reading in the number of cycles to be executed, followed by a while loop to execute them. The while loop terminates either when the requested cycles are complete or when it is pointless to go further. The statements within this loop fall into four blocks. The first block calls *fordsl*, *consolsl* and increments the loop counters. The second, an if structure, determines if a running commentary update is due, executes this and calls *factors_of_safety.* The third, another if block, determines if command list processing is due. If it is, a repeat structure is entered which continually calls control until *gi.cmdend* is set by *cend. control* is called with the file device unit buffer belonging to the command list secondary command file. The fourth block moniters the behaviour of the maximum slice displacement. If this value is $< 10^{-14}$ or $> 10^{6}$ or has stayed almost constant for 100 cycles then the while loop will terminate. After execution of the while loop the maximum displacement

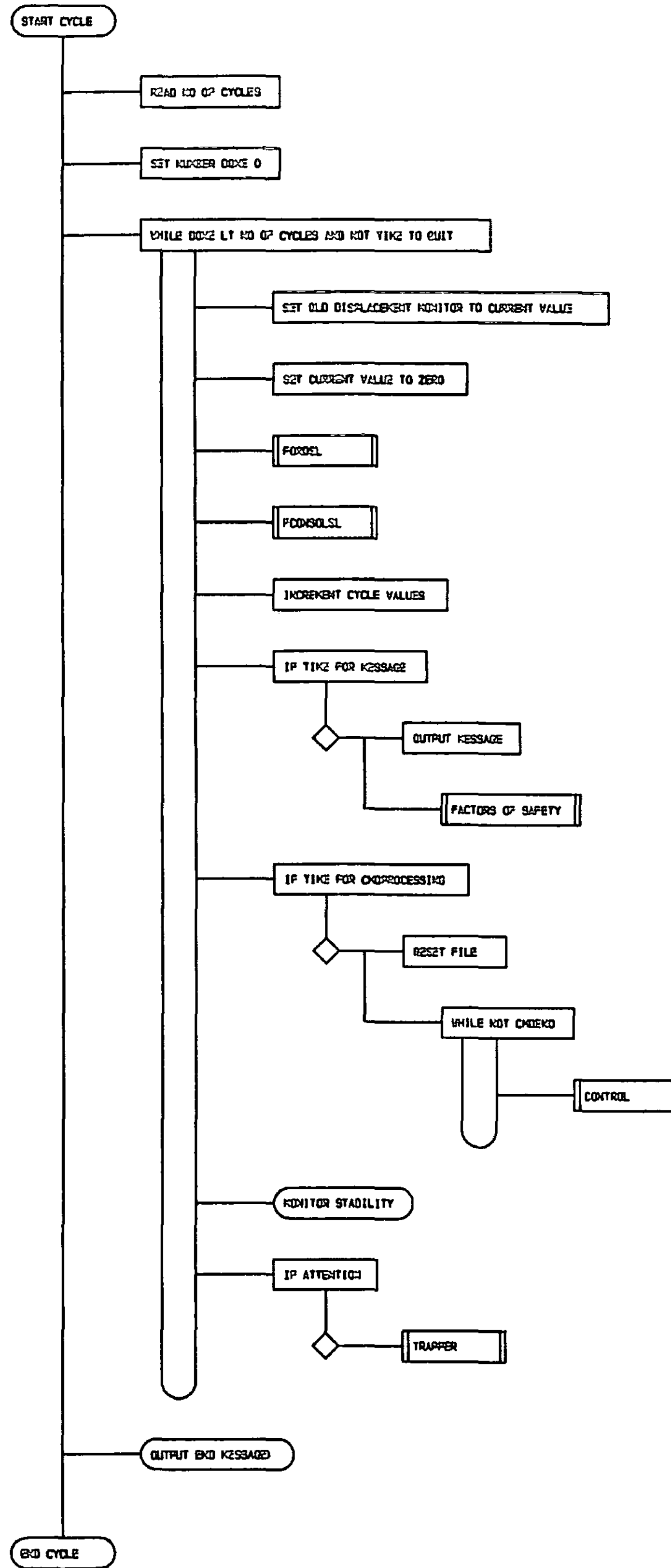


Figure C.23 Chart for procedure *cycle*

state is reviewed and a message written to the running commentary for each of the three value cases outlined above.

PROCEDURE fordsl(el : ptr_type); As one of the two main calculation procedures this is local to *cycles*. The force displacement law is defined here. This is executed for each contact by combining a while and for loop to traverse the slice list and the contacts for each element.

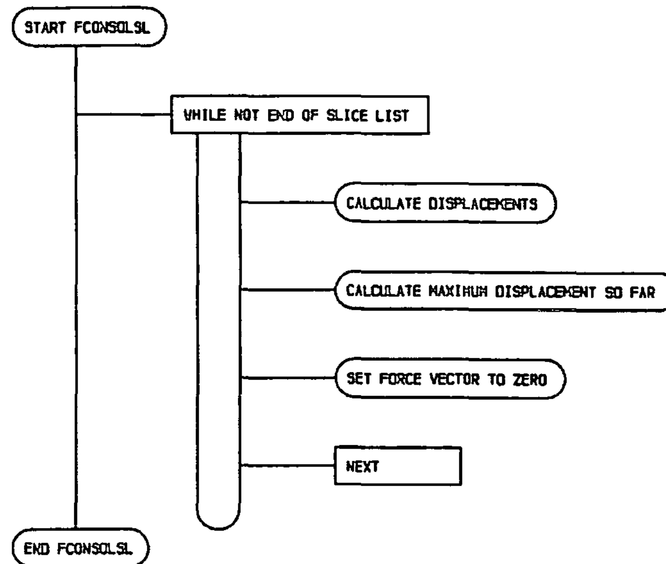


Figure C.25 Chart for procedure *fconsolsl*

PROCEDURE fconsolsl(el : ptr_type); The second of the two calculation procedures, this defines the motion law for the slices. The slice list is traversed using a while loop so that the law is executed for each slice.

PROCEDURE start_shut(var cmd_i : text; starting : start_type); This procedure controls starting and stopping procedures and looks after meshing, contact creation and reading to and from the restart files. The procedure consists of a single case

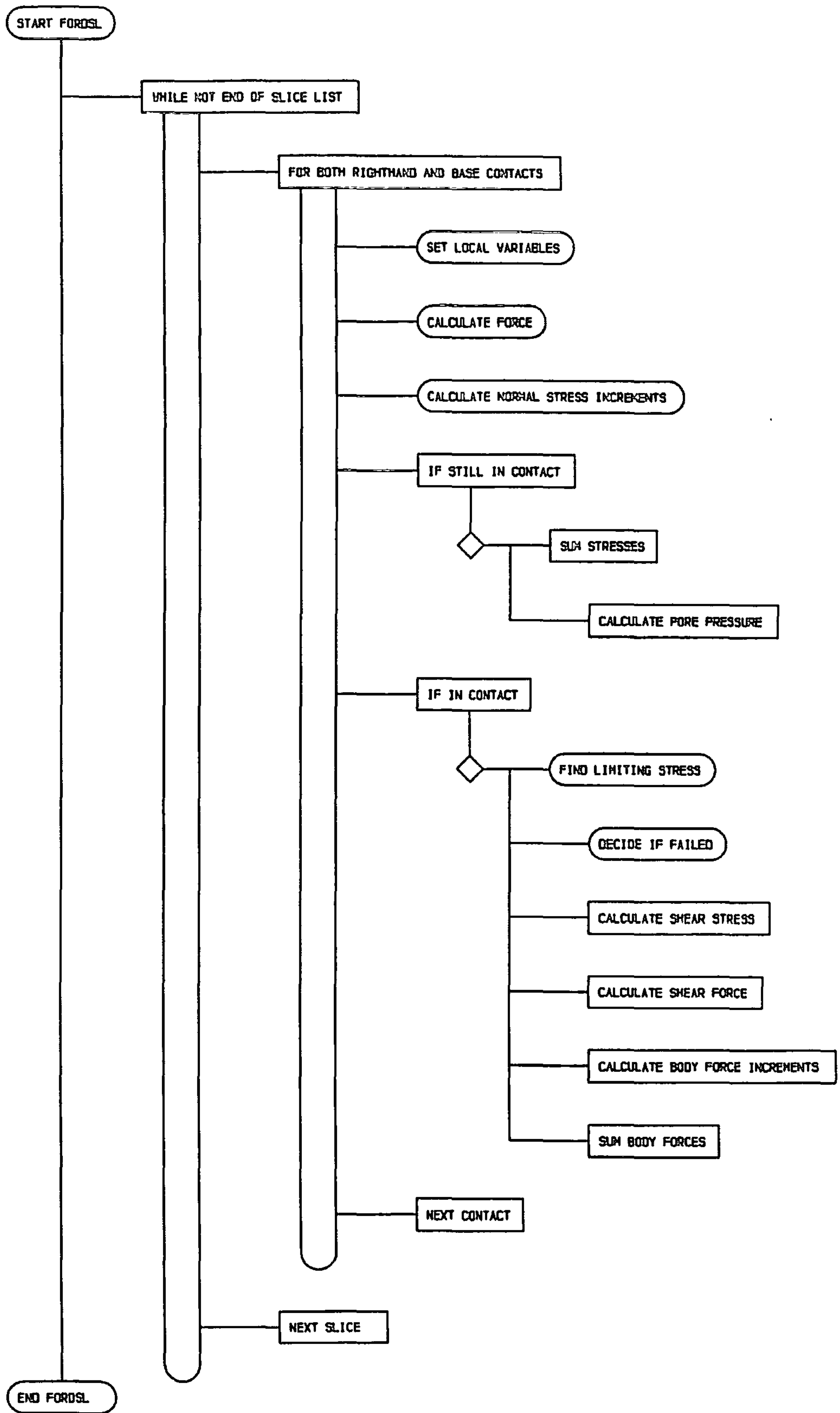


Figure C.24 Chart for procedure *fordsl*

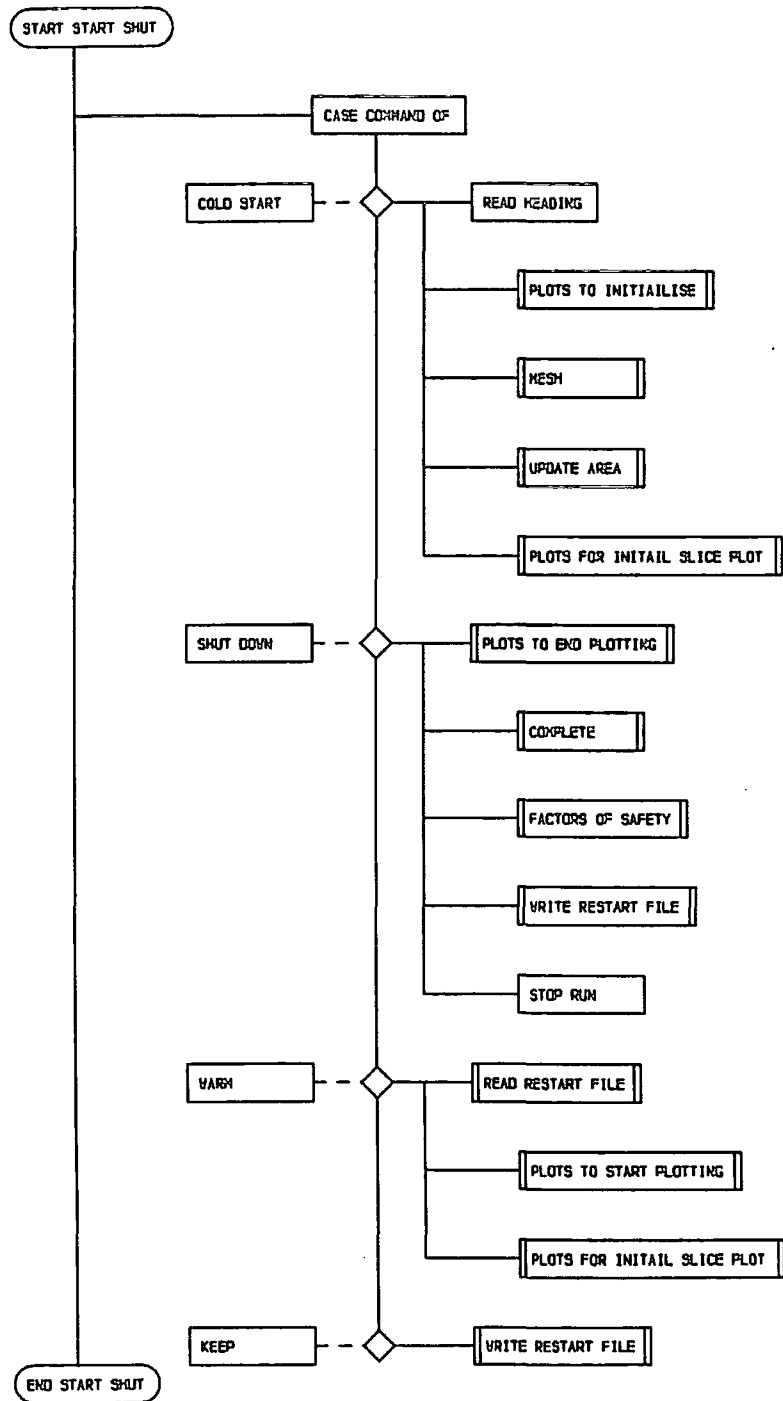


Figure C.26 Chart for procedure *start_shut*

statement with four parts, one for a new run, stopping, restarting, and updating a restart file. The procedure *plots* is referred to several times with internal type calls.

PROCEDURE update_area(el : ptr_type); This is local to *start_shut* and is responsible for setting up the contacts between the slices as well as with the platen. The slice list is traversed using a while construct.

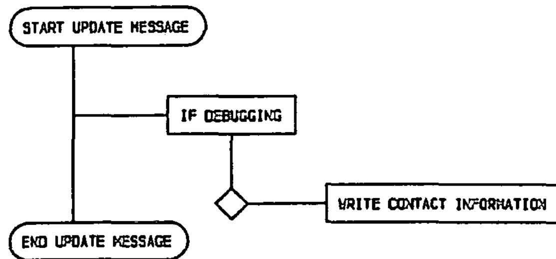


Figure C.28 Chart for procedure *update_message*

PROCEDURE update_message(direction : hed_type; el : ptr_type; cont : con_ptr); update_message is local to *update_area* and generates debug contact information for each contact as it is created during an initial or restarted run.

PROCEDURE cold_contact(el : ptr_type direction : hed_type; cont : con_ptr); cold_contact initialises the contact parameter for a new contact during a cold, that is, an initial run.

PROCEDURE get_apex(var base, oater : corn_ptr; number : integer); This procedure creates the corner doubly linked rings, the number indicates the number of corners to be created. The procedure not only produces complete rings for the slices but can splice corners into such a ring at any time. This latter facility is exploited in creating the platen corner ring. This procedure is local to *start_shut* and is called from *cre_slices* and *cre_platen.*

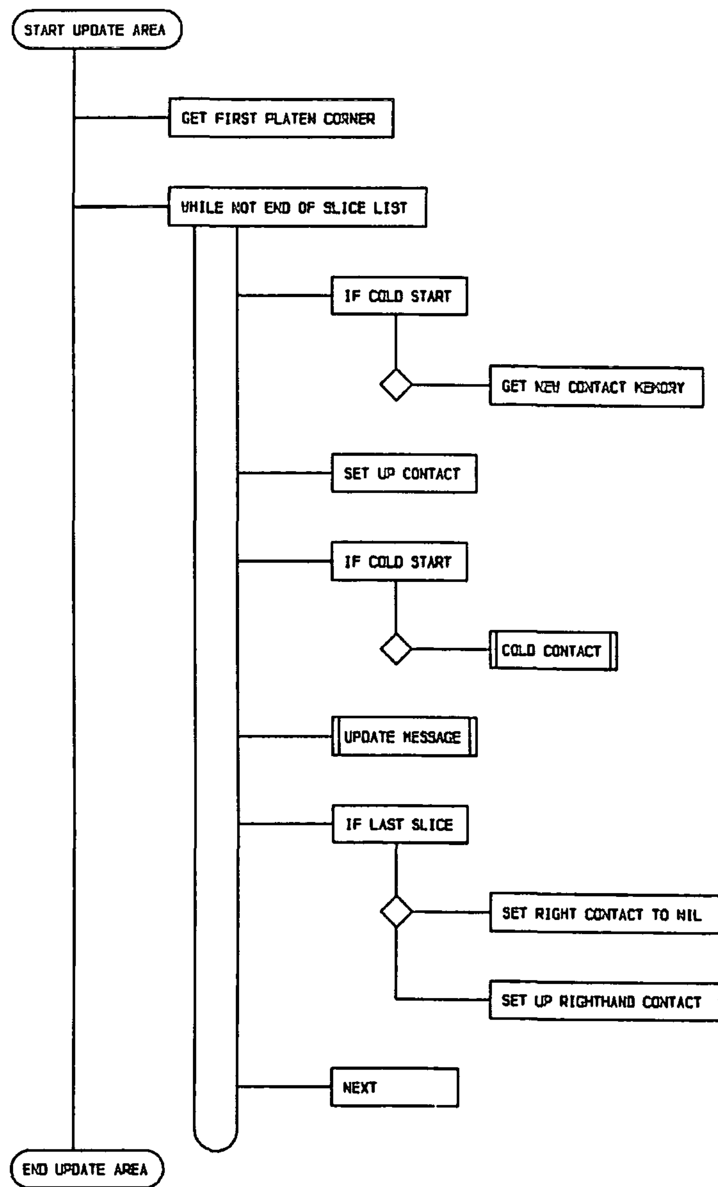


Figure C.27 Chart for procedure *update_area*

PROCEDURE mesh; mesh executes the command input dealing with the creation of slices. The structure is that of *plots*, except that the repeat loop is exited

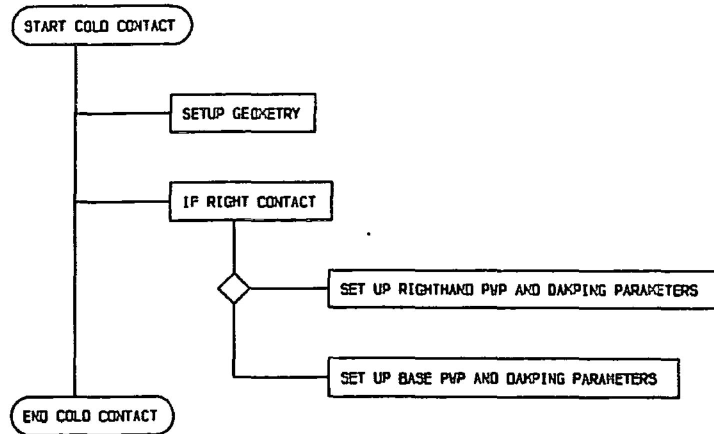


Figure C.29 Chart for procedure *cold_contact*

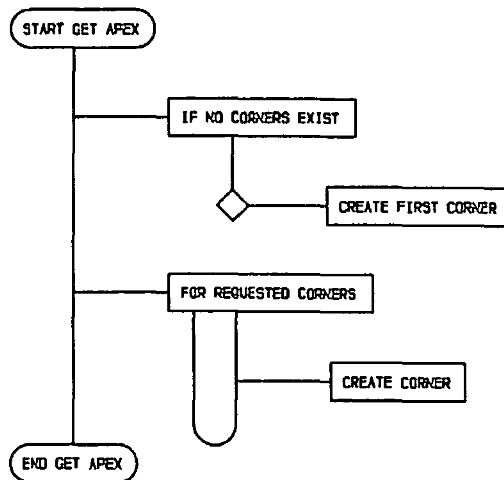


Figure C.30 Chart for procedure *get_apex*

when *meshquit* is explicity set by the user by the **meshend** command. The case statement has two options, one for creating and one for quiting. This procedure calls *cre_slice*.

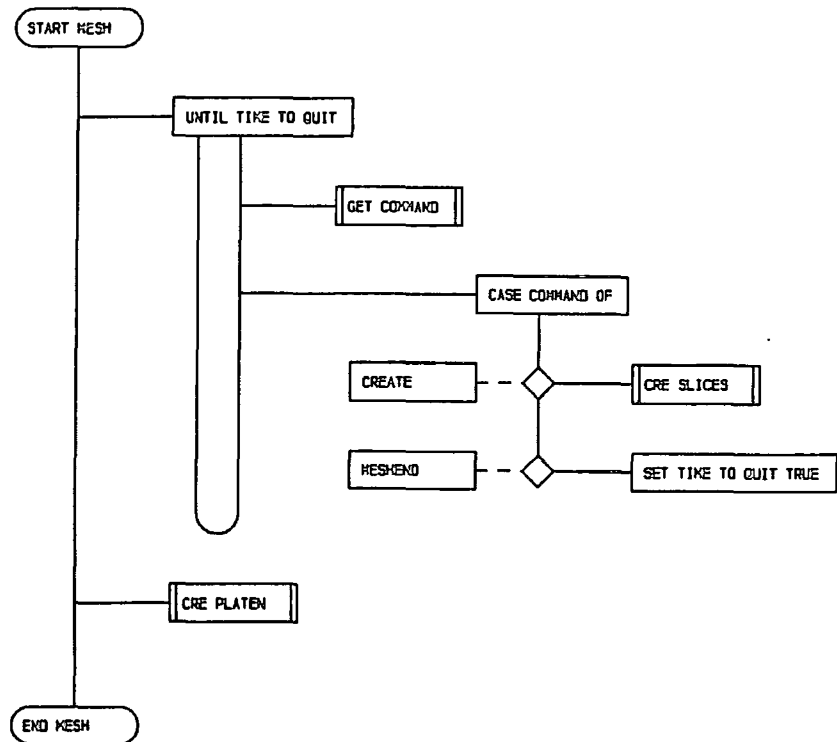


Figure C.31 Chart for procedure *mesh*

PROCEDURE cre_platen(el : ptr_type); This local procedure to *mesh* creates the platen, it is called after all the slices have been created. For each slice in the slice list it adds corners to the platen corner ring by calling *get_apex.*

PROCEDURE cre_slices; get_command is called to ascertain the slice type and a case statement with two options creates the slice according to the type. The element values are initailised to zero and the corner rings created. Once this has been done centres of gravity and masses are found by traversing the corners.

PROCEDURE read_restart_file; This procedure reads a restart file, it is local to *start_shut* and calls no other procedures. The structure is simple, the repeat and command list files are emptied and the restart file is set to the beginning. Following

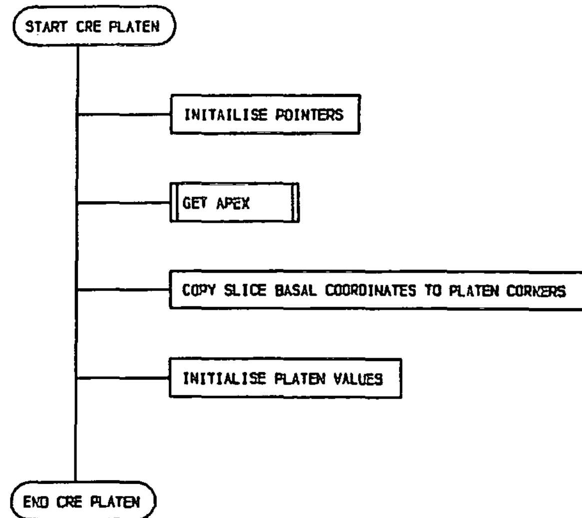


Figure C.32 Chart for procedure *cre_platen*

this a while loop executes until the end of the restart file is reached. Within the loop, a record is read from the file and a case statement option reinterpretes the buffer contents according to the tag field on the buffer record type.

PROCEDURE write_restart_file; This procedure writes a restart file and is local to *start_shut.* The restart file is produced in a standard manner. Firstly five sets of general information are moved to the buffer and written. Then, each line of the command list file and the repeat structure file is set up and written. Following this *write_r_el* is called twice, once for the slice list and then for the platen, and finally the restart file is finished with an end of file message.

PROCEDURE write_r_el(el : ptr_type; c:char); This local procedure to *write_restart_file* recieves the base of an element list in the parameter *el.* This list is traversed and the information for each element is written to the restart file. Following the element data, the righthand and base contact information is written, and lastly the corner rings are traversed and the coordinates added to the file.

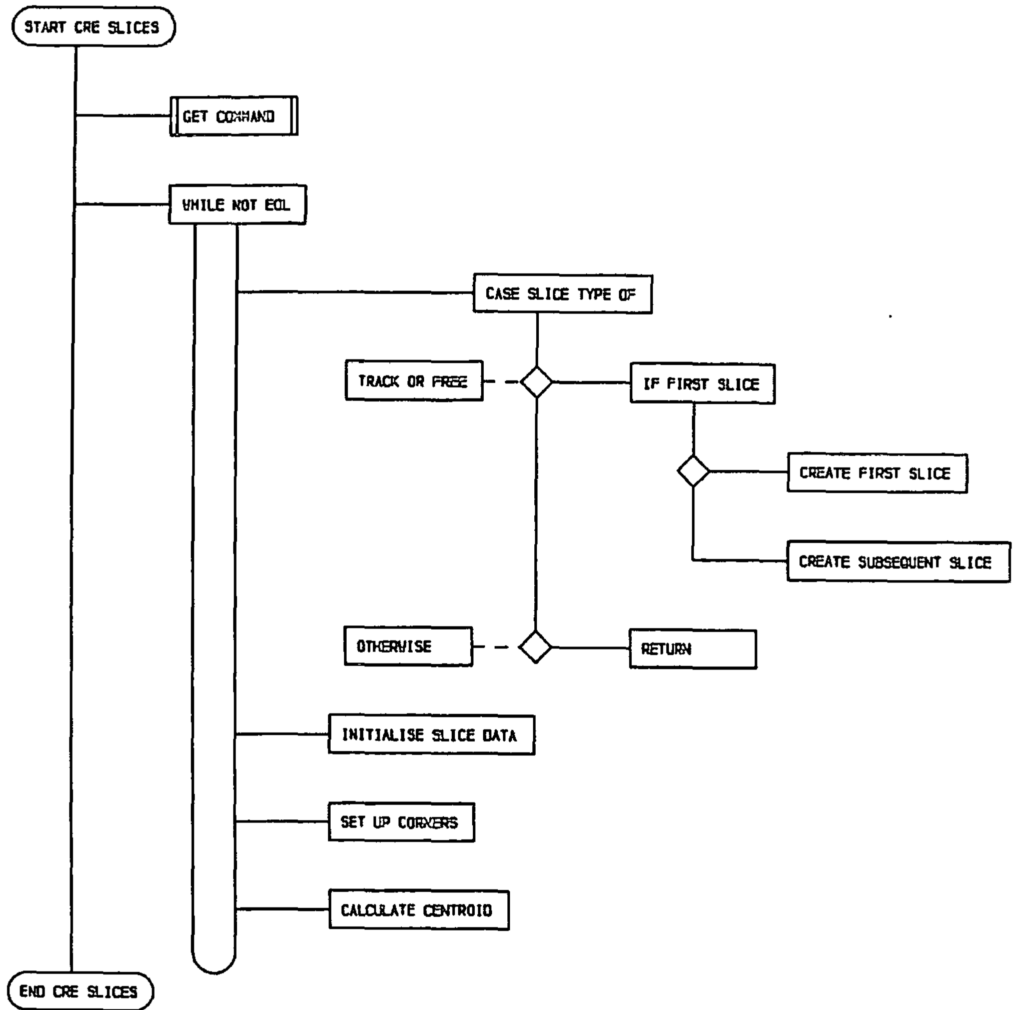


Figure C.33 Chart for procedure *cre_slices*

PROCEDURE complete; This causes some general information to be written to the running commentary.

PROCEDURE debug_slice(var cmd_i : text); This procedure produces or arranges for the production of debugging information. The structure is the same as for *plots*, a repeat containing a case statement. There are twelve case options, mirroring the

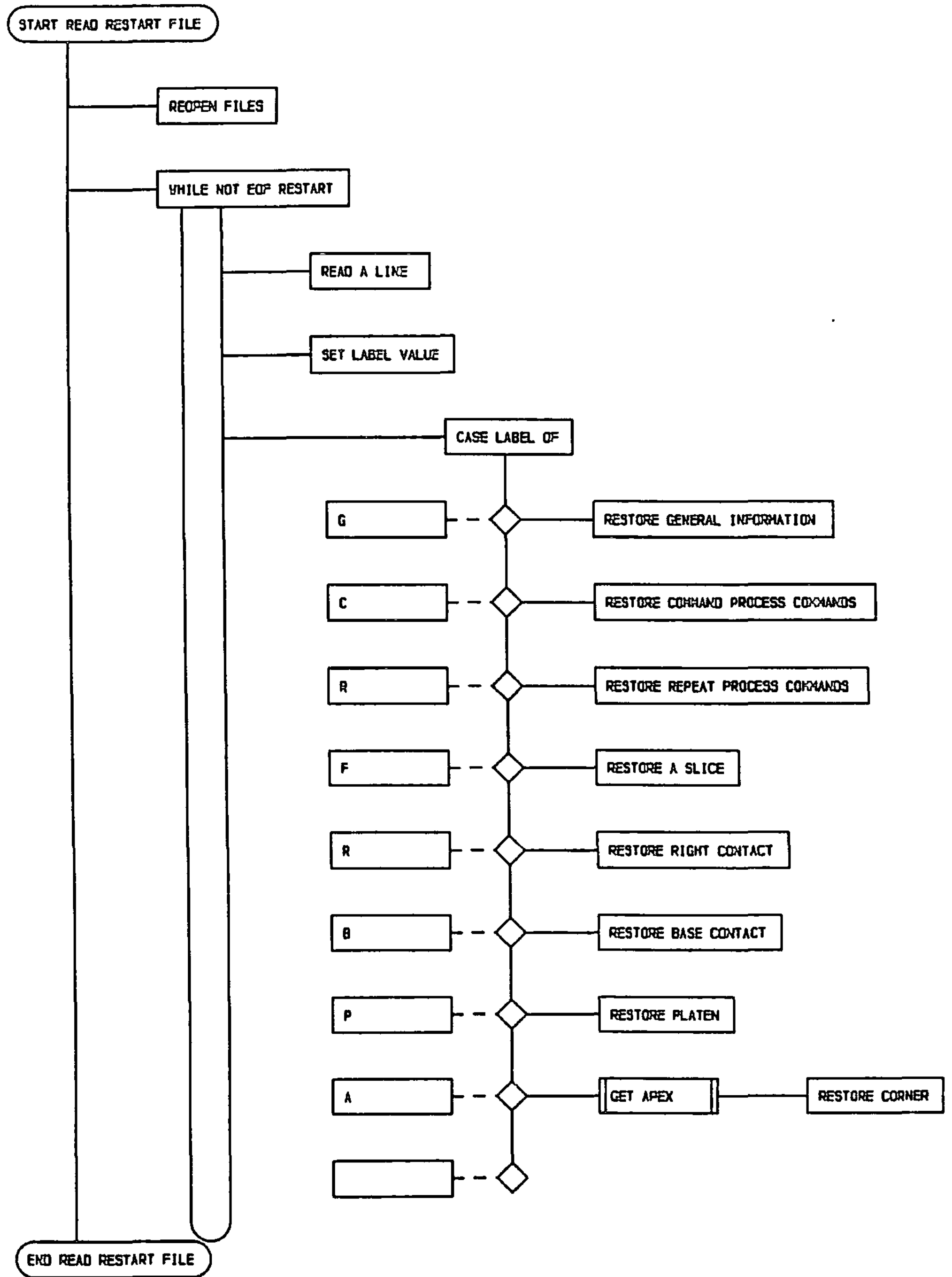


Figure C.34 Chart for procedure *read_restart_file*

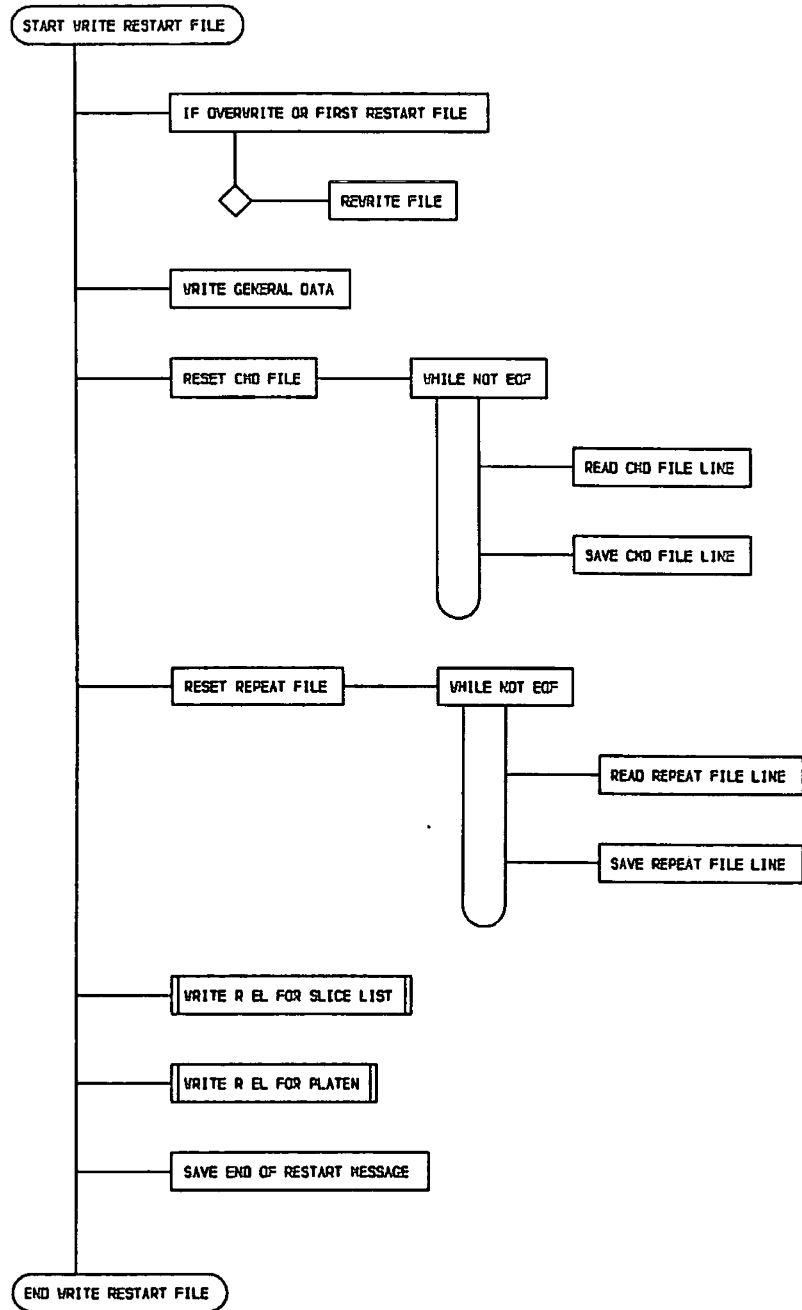


Figure C.35 Chart for procedure *write_restart_file*

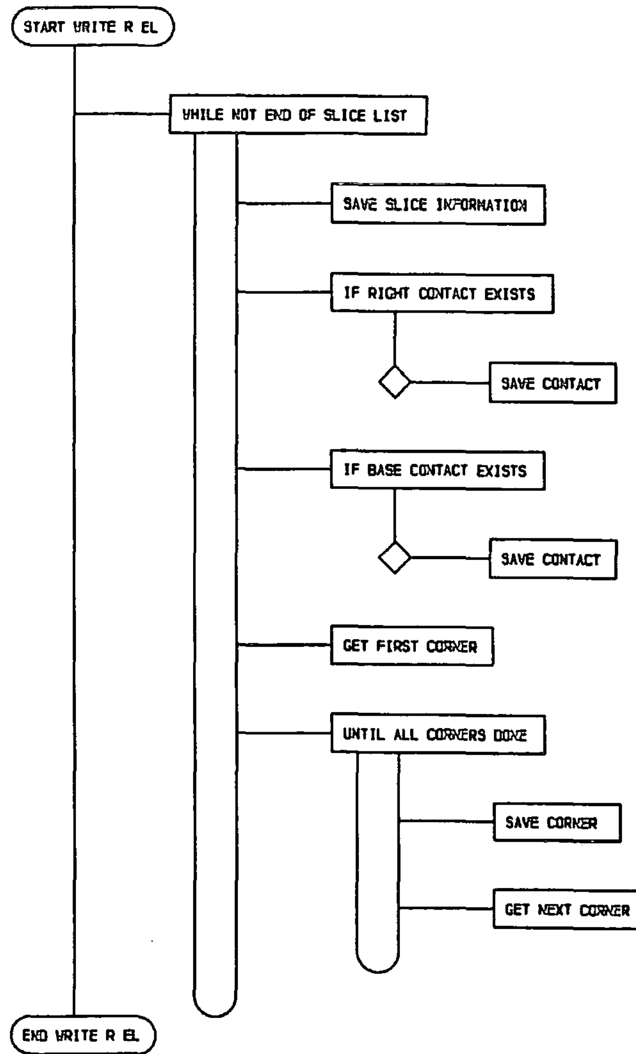


Figure C.36 Chart for procedure *write_r_el*

twelve debug commands. Reference is made to two local procedures, *write_con* and *write_sli* for the production of information.

PROCEDURE write_con(el :ptr_type); This is local to *debug_slice* and produces information for each contact, *wr_con* is called for each contact by traversing the data structure.

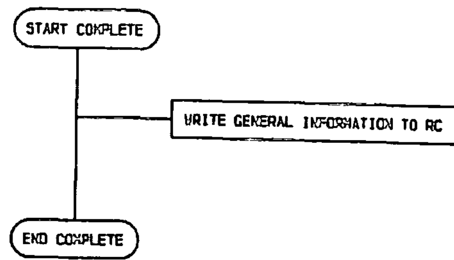


Figure C.37 Chart for procedure *complete*

PROCEDURE wr_con(el : ptr_type; con : con_ptr); This is local to *write_con* and writes out the contact information for a single contact.

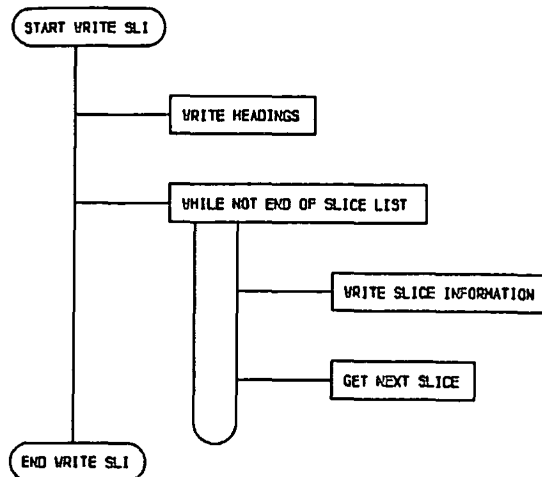


Figure C.41 Chart for procedure *write_sli*

PROCEDURE write_sli(el :ptr_type); This is local to *debug_slice* and produces slice data for each slice.

PROCEDURE parameters(var cmd_i : text); Parameters deals with the execution of the set commands. The structure is the same as *plots*, the case statement

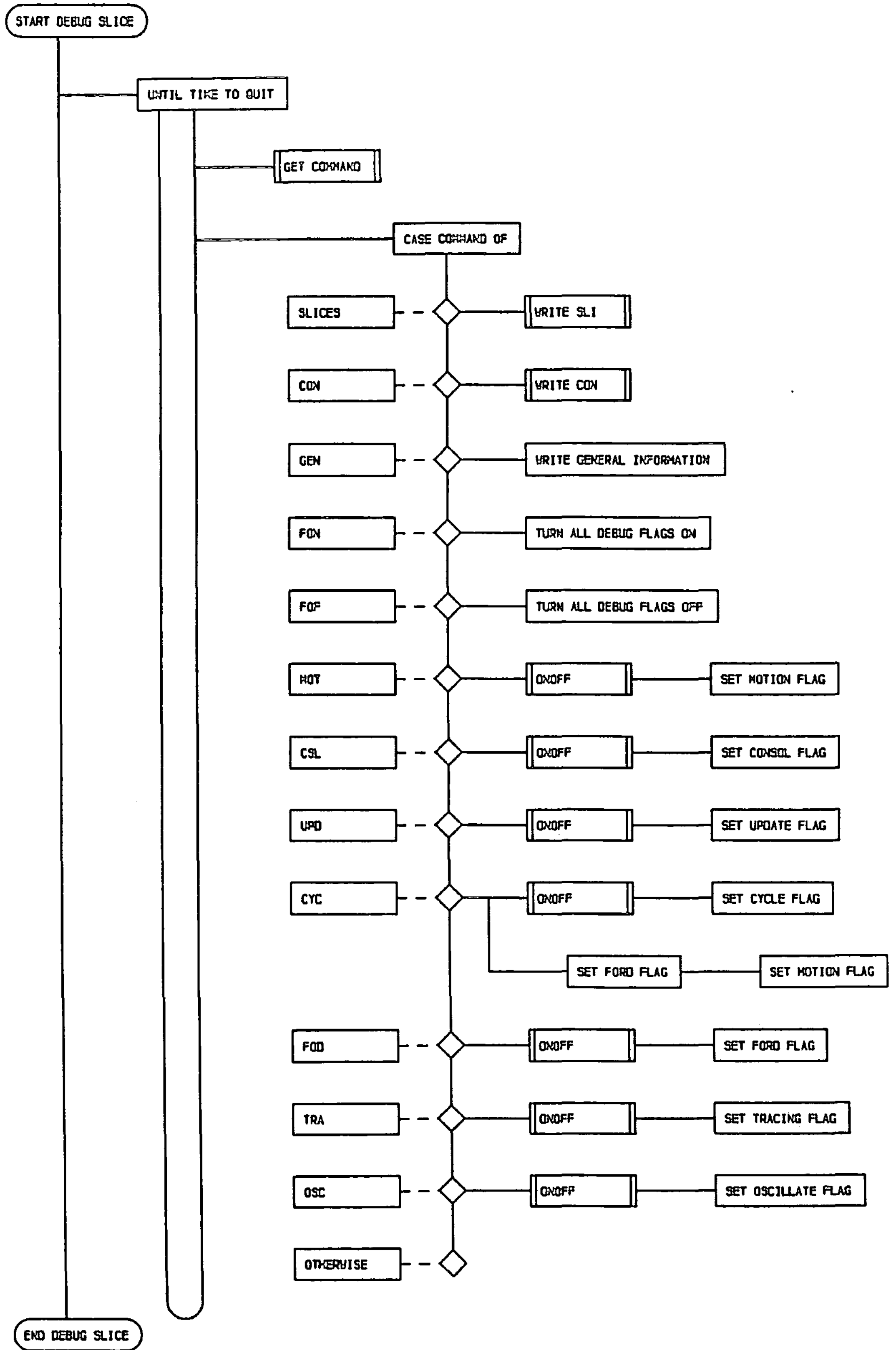


Figure C.38 Chart for procedure *debug_slice*

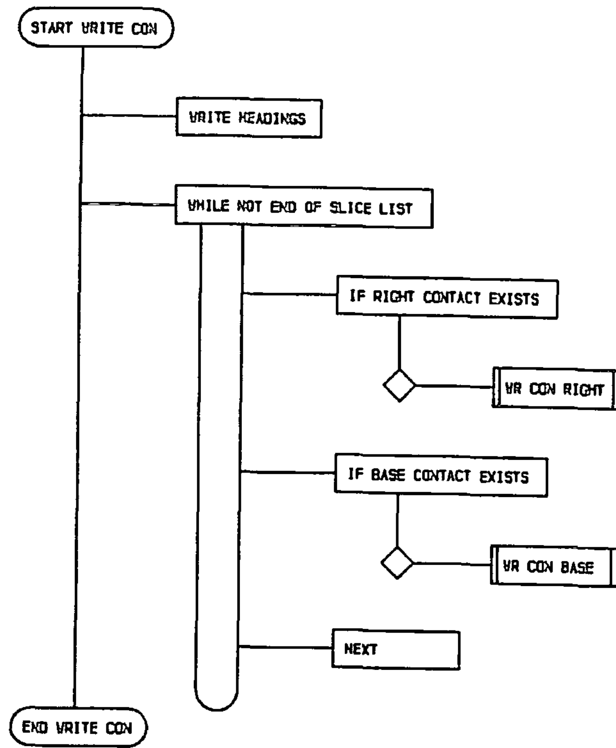


Figure C.39 Chart for procedure *write_con*

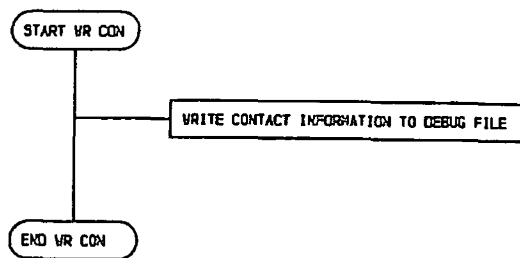


Figure C.40 Chart for procedure *wr_con*

containing twelve options. Most of these involve the prompting for, and reading in of parameter values. One option, the calculator refers to the procedure *calculator*.

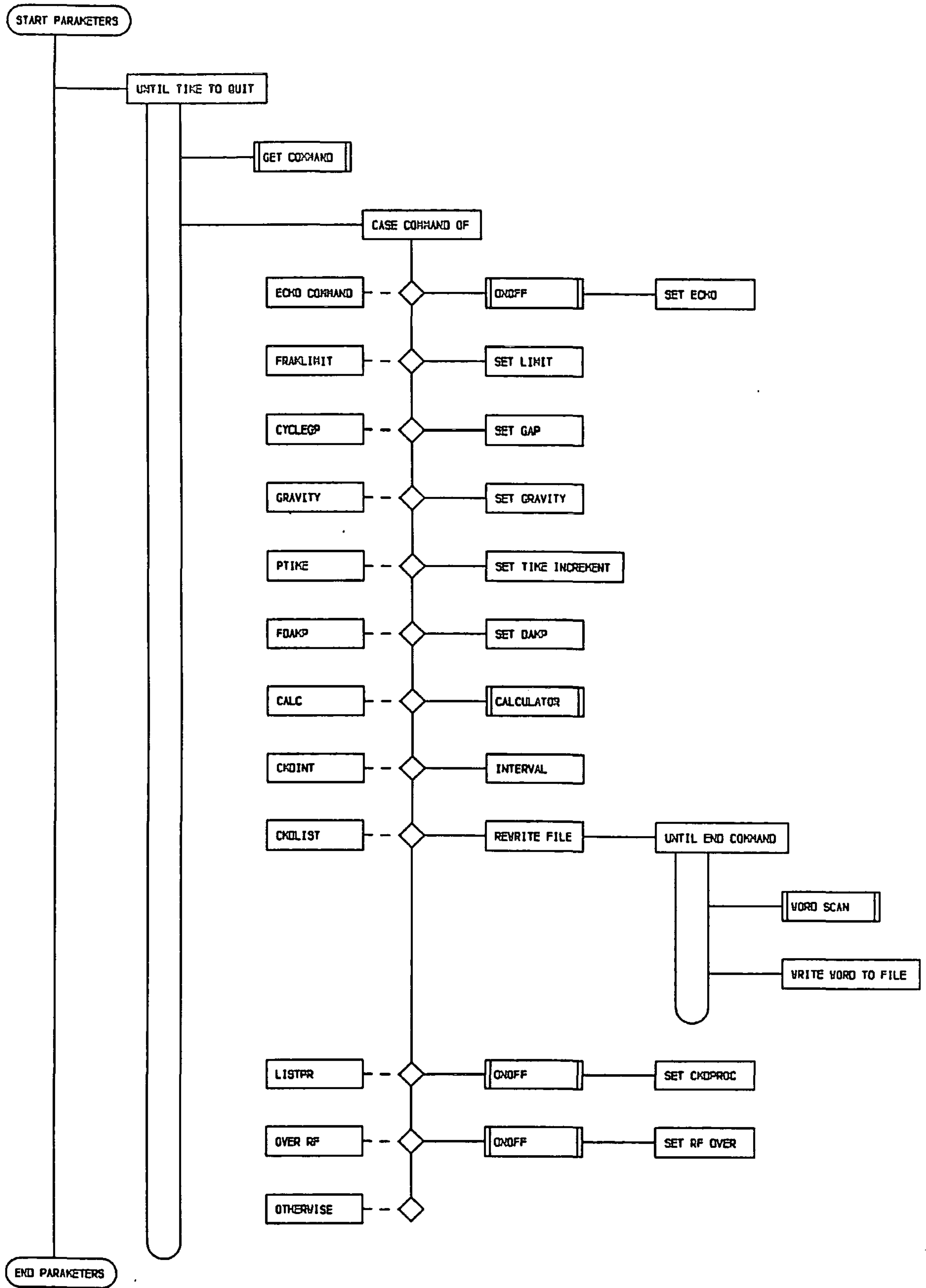


Figure C.42 Chart for procedure *parameters*

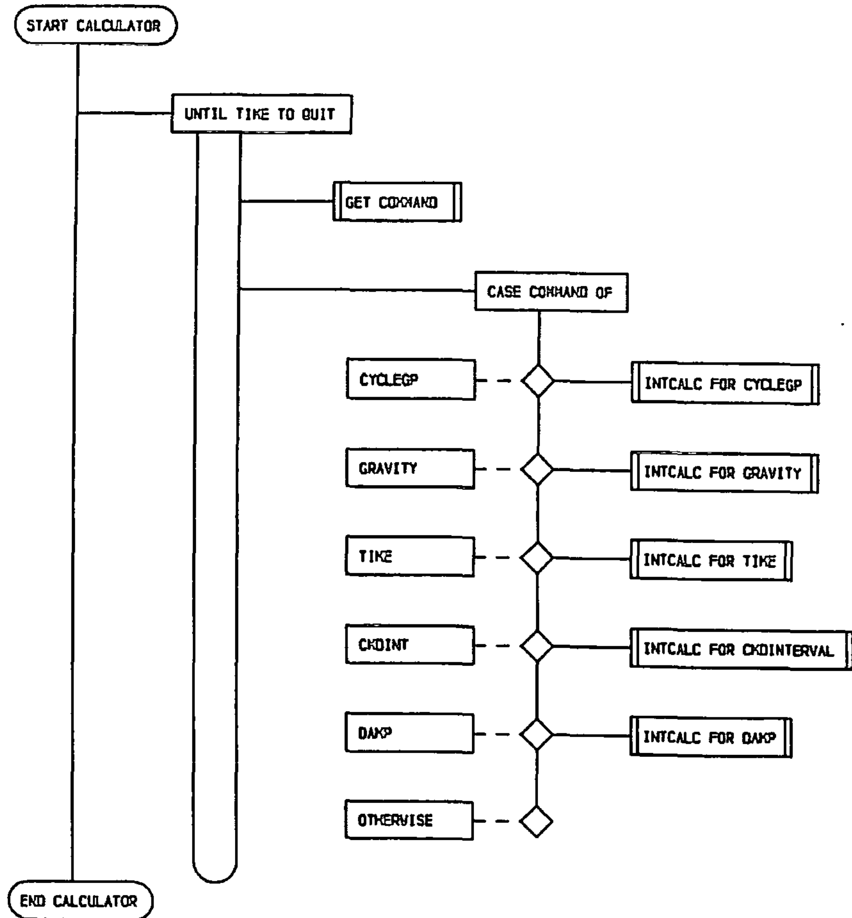


Figure C.43 Chart for procedure *calculator*

PROCEDURE calculator; This is local to *parameters* and enables several of the problem parameters to be altered by calculation. The structure is that of *plots* with a case statement of six options refering to the function *intcalc*, which performs the calculation. On return from *intcalc* the new value is placed in the variable to be changed.

FUNCTION intcalc(op : real): real; This is local to the procedure *calculator* and recieves one value, the operand, the variable to be altered. *get_command* is called

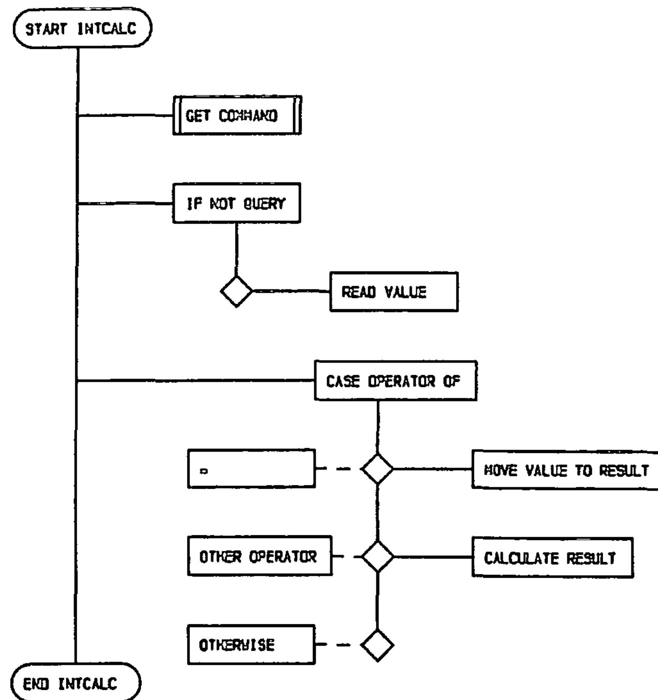


Figure C.44 Chart for function *intcalc*

to obtain the operator and a case option executed accordingly. The result of the simple calculation is written to the running commentary and then returned to the calculator.

PROCEDURE repeater(var cmd_i : text); This procedure is called from control during the execution of the repeat command. Initially the file -sass.rep is emptied and the number of repeats read. A repeat loop calling *get_command* is used to copy the input from the primary source to the secondary. The repeat facility is invoked by means of executing a for loop the number of times requested. Within this, the variable *gi.reptend* is set to false, and *control* is called from within a second repeat loop until *gi.reptend* is true. The file device unit buffer for the secondary command source is passed to *control* on invocation. *gi.reptend* is set to true on encountering rend in procedure *control.*

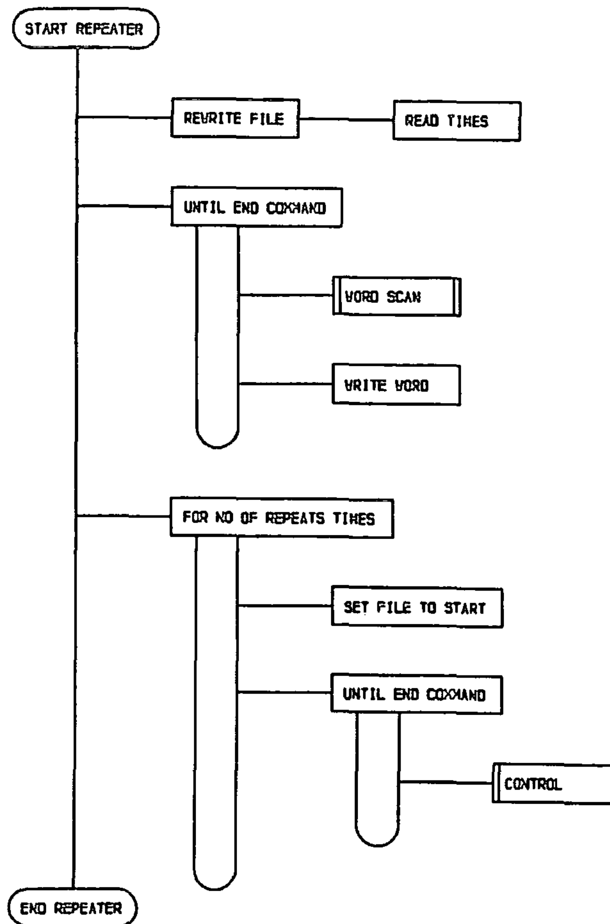


Figure C.45 Chart for procedure *repeater*

PROCEDURE control; The structure consists of a call to *get_command* followed by a case statement. Each case option refers to a permissible level one command.

PROGRAM SLICES Initialisation of the program variables is carried out first by calling *initialise_globals*, this sets all global variables to zero or default values. *headers* is called next to initiate the structure of the running commentary. The outermost control structure of the program then follows. This is a repeat loop that calls the procedure *control.* The loop termination condition can never be

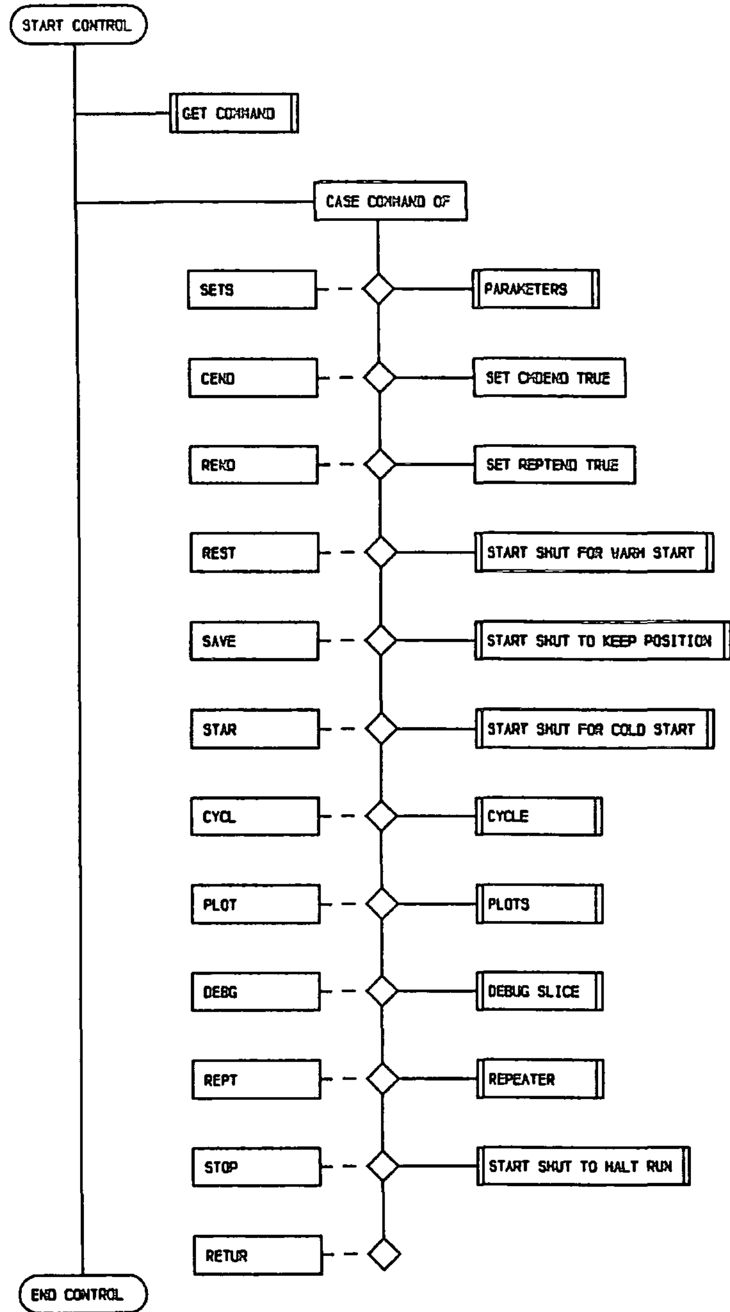


Figure C.46 Chart for procedure *control*

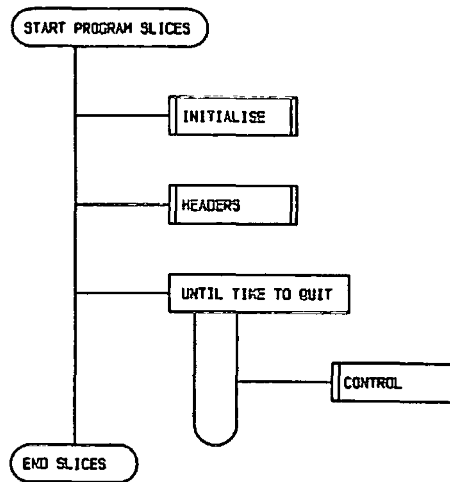


Figure C.47 Chart for program *SLICES*

true, so this is a repeat forever construct. The program, however, does closedown in procedure *start_shut* or *error_simple*. Procedure *control* is called with the file device unit buffer pointer for the primary input command file.

APPENDIX D

PROGRAM SLICES

```
program slices(debug_o, sercom);
%include est8:u.ghost.lib
%include trap

const
  led_pos = 2;
  tit_pos = 4;
  req_pos = 6;
  fra_pos = 7;
  plo_pos = 8;
  cyc_pos = 9;
  com_pos = 11;
  mes_pos = 12;
  fos_pos = 13;
  err_pos = 14;
  tot_pos = 17;
  fil_pos = 20;
  pro_pos = 16;
  pos_str = ' ';
  clearoff = ' .0';
  curson = ' .1';
  maxcycle = 1000000;
  commands = {onoffer}
  'null       on        off       ' ||      {onoffer}
  'bottom     lowermiddle uppermiddle top       picture    ' ||
  'horizontal vertical   plain     full      fullnoscales' ||
  'zoom       ' ||       {map}
  'initialise slices     displacementforces    standard    ' ||
  'page       border     map       endplot    ' || {plot}
  'free       track      ' ||      {create}
  'meshend    create     ' ||      {mesh}
  '=          *          /          +          -          ' ||
  '^          ?          ' ||      {operater}
  'echo       cmdproc    overwrite  framelimit writegap   ' ||
  'interval   cmdlist    gravity    damp       time       ' ||
  'calculate  ' ||       {set}
  'contacts   general    flagson    flagsoff   update     ' ||
  'motion     consolidate ford      cycle      trace      ' ||
  'oscillate  ' ||       {debug}
  'set        cend       rend       restart    save       ' ||
  'start      go         plot       repeat     debug      ' ||
  'stop       return     ';        {control}

type
  com_type = (null, on, off, lowerp, midlop, midupp, upperp, piccie, horiz,
          vertic, plain, whole, fnosc, zoom, init, slices, displot,
          forceplot, standard, page, frames, maps, plotstop, free, track,
          meshend, create, equal, mult, divid, plus, minus, power, enquiry,
          echo, listpr, over_rf, framlim, cyclegp, cmdint, cmdlist, gravity,
```

```
          fdamp, ptime, calc, con, gen, fon, fof, upd, mot, csl, fod, cyc,
          tra, osc, sets, cend, rend, rest, save, star, cycl, plot, rept,
          debg, stop, retur);
call_type = (errorer, onoffer, mapper, plotter, mesher, creater, operter,
          calcter, paramer, debuger, contler);
start_type = (cold, warm, shutdown, keep);
dir_of_contacts = (righthand, based);
ptr_type = @element_type;
con_ptr = @con_type;
corn_ptr = @corn_type;
vector_type = record
             x, y: real
           end;
coord_type = record
           xc, yc: real
          end;
con_type = record
          consol: record
                 ns, ss, lims, pp: real
                end;
          dampf, sine, cose, con_len: real;
          failed: boolean;
          corn, edge: corn_ptr;
          other: ptr_type;
         end;
corn_type = record
           c: coord_type;
           cw, aw: corn_ptr
          end;
element_type = record
             posn: coord_type;
             force, s: vector_type;
             data: record
                   phi, mass, sidec, cohes, sphi, rho, k, pwp, spwp, e:
                     real;
                   sliceno: integer;
                   typ: free .. track;
                  end;
             contacts: record
                     right, base: con_ptr
                    end;
             next: ptr_type;
             apexes: corn_ptr;
            end;
cycle_type = 0 .. maxcycle;
hed_type = string(80);
grid_type = record
           xmin, xmax, ymin, ymax: real
          end;
gen_info_type = record
              heading: hed_type;
              tstep: real;
              nextword: string(12);
              reptend, cmdend: boolean;
              motioning, consoling, updating, cycling, fording, tracing,
                oscing: boolean
             end;
option_type = record
             plot_lims: grid_type;
             vert: boolean;
```

```
                meshtbs: record
                         xb, yb, xt, yt: real
                       end;
                grav: vector_type;
                damp, damps: real;
                cyclegap, cycle_interval: cycle_type;
                cmdprocessing, echo, rf_over: boolean
              end;
  sum_type = record
            sc, scold, scsofar: real
          end;
  totals_type = record
                cycles, restarts, slices, cons, pics, pages: integer
              end;

const
  nilv = vector_type(0, 0);
  nilc = coord_type(0, 0);
  nilhed = ' ';
  nilgrid = grid_type(0, 0, 0, 0);

var
  repts_i, cycmd_i, oscil_o, debug_o, trace_o, sercom: text;
  rf_first, quit, qdum, screen: boolean;
  gi: gen_info_type;
  opt: option_type;
  sum: sum_type;
  total: totals_type;
  plspace, plot_space, force_map: grid_type;
  platen, slice_list, eolist: ptr_type;
  apex, platapex: corn_ptr;
{****************************************** BEGIN GLOBAL ROUTINES }


procedure error_simple(ob, caller: string(40));

  begin
    if gi.tracing
      then
        writeln(trace_o, 'Entered procedure ERROR_SIMPLE');
    rewrite(sercom, 'UNIT=11');
    writeln(sercom, ' Error ', '''', ob, '''', ' found in routine ', caller,
        '    ');
    halt;
    end {error_simple};


procedure word_scan(var cmds_in: text; var word: string(12));

  const
    blank = ' ';

  var
    ch: string(1);


  procedure skipblks(var ch: string(1));

    begin
```

```
    if gi.tracing
      then
        writeln(trace_o, 'Entered procedure SKIPBLKS');
    ch := ' ';
    while ch = blank do begin
      while (~ eoln(cmds_in)) AND (ch = blank) do
        read(cmds_in, ch);
      if (eoln(cmds_in)) AND (~ eof(cmds_in))
        then
          readln(cmds_in);
      if eof(cmds_in)
        then
          error_simple(' End of file causes return to mts', 'skipblks');
      end;
    if gi.tracing
      then
        writeln(trace_o, '  EXIT procedure SKIPBLKS');
    end {skipblks};


procedure skipcomment(var ch: string(1));

  begin
    if gi.tracing
      then
        writeln(trace_o, 'Entered procedure SKIPCOMMENT');
    while ch ~ = '}' do begin
      while (~ eoln(cmds_in)) AND (ch ~ = '}') do
        read(cmds_in, ch);
      if (eoln(cmds_in)) AND (~ eof(cmds_in))
        then
          readln(cmds_in);
      if eof(cmds_in)
        then
          error_simple('end of file causes return to mts', 'skipcomment');
      end;
    skipblks(ch);
    if ch = '{'
      then
        skipcomment(ch);
    if gi.tracing
      then
        writeln(trace_o, '  EXIT procedure SKIPCOMMENT');
    end {skipcomment};


begin {word_scan}
  if gi.tracing
    then
      writeln(trace_o, 'Entered procedure WORD_SCAN');
  word := '';
  ch := '';
  if screen
    then begin
      writeln(output, substr(pos_str, 1, err_pos + 2),
          '     Input a command please ...........    ');
      reset(cmds_in, 'UNIT=11,INTERACTIVE');
      repeat
        read(cmds_in, ch)
```

```
        until ch ~ = '';
      end
    else
      skipblks(ch);
  while (~ eoln(cmds_in)) AND (ch ~ = blank) do begin
    if ch = '{'
      then
        skipcomment(ch);
    word := word || ch;
    read(cmds_in, ch);
    end;
  if ch ~ = blank
    then
      word := word || ch;
  if opt.echo
    then
      writeln(output, substr(pos_str, 1, com_pos), ' Command : ', word,
          '      ');
  if (eoln(cmds_in)) AND (~ eof(cmds_in))
    then
      readln(cmds_in);
  if gi.tracing
    then
      writeln(trace_o, '  EXIT procedure WORD_SCAN    ', word);
  end {word_scan};


procedure start_shut(var cmd_i: text; starting: start_type);
  forward;


procedure control(var cmd_i: text);
  forward;


procedure trapper;

  var
    ch: char;

  begin
    if gi.tracing
      then
        writeln(trace_o, 'Entered procedure TRAPPER');
    writeln(output, substr(pos_str, 1, err_pos),
        ' Attn! : Do you want to stop ?');
    reset(sercom, 'UNIT=11');
    repeat
      read(sercom, ch);
      until (ch ~ = ' ');
    trpreset;
    if ch = 'y'
      then
        start_shut(input, shutdown);
    writeln(output, substr(pos_str, 1, err_pos),
        '                               ');
    if gi.tracing
      then
        writeln(trace_o, '  EXIT procedure TRAPPER');
```

```
  end {trapper};


procedure get_command(caller: call_type; var quiter: boolean; var retcom:
  com_type; intcall: string(12); var cmds_ig: text);

 const
  last = 816;

 var
  ifail: boolean;
  beg, loca, indes: 0 .. 1200;
  this_com: string(12);

 begin
  if gi.tracing
   then
    writeln(trace_o, 'Entered procedure GET_COMMAND  ', intcall);
  this_com := intcall;
  if this_com = ''
   then begin
    this_com := gi.nextword;
    gi.nextword := '';
    end;
  if this_com = ''
   then
    word_scan(cmds_ig, this_com);
  if trap
   then
    trapper;
  beg := 1;
  repeat
   indes := index(substr(commands, beg, last - beg + 1), this_com);
   loca := indes + beg - 1;
   if loca MOD 12 = 1
    then
     indes := 0
    else
     beg := loca + 1;
   until (indes = 0) OR (last - beg < 12);
  retcom := com_type(loca DIV 12);
  case caller of
   errorer:
    ifail := NOT (retcom IN on .. retur );
   onoffer:
    ifail := NOT (retcom IN on .. off );
   mapper:
    ifail := NOT (retcom IN lowerp .. zoom );
   plotter:
    ifail := NOT (retcom IN init .. plotstop, zoom );
   mesher:
    ifail := NOT (retcom IN free .. track );
   creater:
    ifail := NOT (retcom IN meshend .. create );
   operter:
    ifail := NOT (retcom IN equal .. enquiry );
   calcter:
    ifail := NOT (retcom IN cyclegp, cmdint, gravity .. ptime );
   paramer:
    ifail := NOT (retcom IN echo .. calc );
```

```
    debuger:
      ifail := NOT (retcom IN con .. osc, slices );
    contler:
      ifail := NOT (retcom IN sets .. retur );
    end;
  if ifail
    then begin {some thing's wrong}
      if (retcom = null) OR (caller = contler)
        then begin {invalid command}
          screen := true;
          writeln(output, substr(pos_str, 1, err_pos), ' Error ', '''',
             this_com, '''', ' found in routine ', 'get_command        ');
          writeln(output, 'Input corrected commands ... <RETURN> ...');
          get_command(errorer, ifail, retcom, '', sercom);
          while retcom ~ = retur do begin
            gi.nextword := substr(commands, ord(retcom) * 12 + 1, 12);
            control(sercom); {control returns with nextword = return|keyword}
            get_command(errorer, ifail, retcom, gi.nextword, sercom);
            end;
          screen := false;
          gi.nextword := 'return';
          quiter := false;
          end
        else begin {valid command wrong caller}
          quiter := true;
          gi.nextword := this_com;
          end;
      end
    else
      quiter := intcall ~ = ''; {alls ok}
  if gi.tracing
    then
      writeln(trace_o, '  EXIT procedure GET_COMMAND');
  end {get_command};


function onoff(var cmd_i: text): boolean;

 var
  onof: com_type;

 begin
  if gi.tracing
    then
      writeln(trace_o, 'Entered procedure ONOFF');
  get_command(onoffer, qdum, onof, '', cmd_i);
  case onof of
    on:
      onoff := true;
    off:
      onoff := false;
    otherwise;
    end;
  if gi.tracing
    then
      writeln(trace_o, '  EXIT procedure ONOFF');
  end {onoff};
```

```
procedure headers;

  begin
    writeln(output, clearoff);
    writeln(output, substr(pos_str, 1, led_pos),
       ' PROGRAM SLICES RUNNING COMMENTARY ON : ');
    writeln(output, substr(pos_str, 1, tit_pos), ' ', gi.heading);
    writeln(output, substr(pos_str, 1, req_pos), '     0  cycles requested');
    writeln(output, ' ', total.pages: 6, '  frames plotted');
    writeln(output, ' ', total.pics: 6, '  plots types drawn');
    writeln(output, ' ', total.cycles: 6, '  cycles and still counting!');
    end {headers};


procedure factors_of_safety(el: ptr_type);

  var
    fos: real;
    atlim, natlim: integer;

  begin
    atlim := 0;
    natlim := 0;
    writeln(debug_o, total.cycles: 6,
       'Slice no  FOS     shear    normal    limit     pwp    targu');
    while el ~ = nil do
      with el@, contacts.base@, consol do begin
        if abs(ss) < 1e-20
          then
            fos := 1
          else
            fos := abs(lims / ss);
        writeln(debug_o, '      ', data.sliceno: 10, fos: 10, ss: 10, ns: 10,
           lims: 10, pp: 10, data.pwp: 10);
        if fos < 1.0005
          then
            atlim := atlim + 1
          else
            natlim := natlim + 1;
        el := next;
        end;
    writeln(output, substr(pos_str, 1, fos_pos), '  Number slices at limit ',
       atlim: 4, ' not at limit ', natlim: 4);
    end {factors_of_safety};


function sign(val, donor: real): real;

  begin
    if donor ~ = 0
      then
        sign := abs(val * donor) / donor
      else
        sign := val;
    end {sign};


procedure initialise_globals;
```

```
 begin
  trapon;
  rewrite(debug_o, 'UNIT=7');
  rewrite(trace_o, 'UNIT=8');
  rewrite(oscil_o, 'UNIT=10');
  rewrite(cycmd_i, 'FILE=-sass.cmd.i');
  quit := false;
  rf_first := true;
  with gi, plspace, opt, total, opt.meshtbs do begin
    reptend := false;
    cmdend := false;
    tstep := 0;
    heading := nilhed;
    nextword := '';
    motioning := false;
    updating := fal~e;
    cycling := false;
    fording := false;
    oscing := false;
    tracing := false;
    consoling := false;
    plspace := nilgrid;
    force_map := nilgrid;
    plot_space := nilgrid;
    plot_lims := nilgrid;
    xb := 0;
    yb := 0;
    xt := 0;
    yt := 0;
    vert := true;
    grav.x := 0;
    grav.y := 1;
    damp := 0;
    cyclegap := 100;
    cycle_interval := maxcycle;
    echo := true;
    rf_over := true;
    cmdprocessing := false;
    sum.sc := 1E70;
    sum.scold := 0.0;
    sum.scsofar := 0;
    slices := 0;
    cons := 0;
    cycles := 0;
    restarts := 0;
    pics := 0;
    pages := 0;
    platen := nil;
    slice_list := nil;
    apex := nil;
    platapex := nil;
    end
  end {initialise_globals};
{****************************************  END GLOBALS  }
{****************************************  BEGIN PLOTS  }


procedure plots(var cmd_i: text; plot_command: string(12));
```

```
var
  plotcom: com_type;
  plotquit, writing: boolean;


procedure map_space(var cmd_i: text; sp_comst: string(12));

  const
    paph_space = grid_type(0.06, 0.96, 0.05, 0.65);
    papv_space = grid_type(0.15, 0.75, 0.06, 0.96);

  var
    quartht, htratio: real;
    sp_com: com_type;
    map_sp, plt_sp: grid_type;
    mapquit: boolean;

  begin
    if gi.tracing
      then
        writeln(trace_o, 'Entered procedure MAP_SPACE');

    repeat
      get_command(mapper, mapquit, sp_com, sp_comst, cmd_i);
      with plt_sp do begin
        plt_sp := plot_space;
        quartht := 0.9 * (ymax - ymin) / 4;
        case sp_com of
          horiz: begin
            opt.vert := false;
            plot_space := paph_space;
            plt_sp := plot_space;
            quartht := 0.9 * (ymax - ymin) / 4;
            end;
          vertic: begin
            opt.vert := true;
            plot_space := papv_space;
            plt_sp := plot_space;
            quartht := 0.9 * (ymax - ymin) / 4;
            end;
          lowerp:
            ymax := ymin + quartht;
          midlop: begin
            ymin := ymin + quartht;
            ymax := ymin + quartht
            end;
          midupp: begin
            ymin := ymin + 2 * quartht;
            ymax := ymin + quartht
            end;
          upperp: begin
            ymin := ymin + 3 * quartht;
            ymax := ymin + quartht
            end;
          piccie: begin
            ymin := ymin + 2 * quartht;
            ymax := ymin + 2 * quartht
            end;
          otherwise;
          end;
```

```
    if sp_com ~ = plain
      then
        pspace(xmin, xmax, ymin, ymax);
    if ymax - ymin < 1E-20
      then
        htratio := 1
      else
        htratio := (xmax - xmin) / (ymax - ymin);
    end;

  with map_sp do begin
    map_sp := plspace;
    case sp_com of
      lowerp, midlop, midupp, upperp: begin
        ymin := force_map.ymin;
        ymax := force_map.ymax;
        end;
      piccie, horiz, vertic, whole, fnosc:
        ymax := ymin + (xmax - xmin) / htratio;
      zoom:
        with plspace do begin
          if screen
            then
              writeln(output, substr(pos_str, 1, pro_pos),
                  ' Enter xmin, xmax, and ymin ...');
          read(cmd_i, xmin, xmax, ymin);
          ymax := ymin + (xmax - xmin) / htratio;
          map_sp := plspace;
          end;
      plain: begin
        map_sp := nilgrid;
        xmax := 100;
        ymax := 100;
        end;
      otherwise;
      end;
    ctrmag(10);
    map(xmin, xmax, ymin, ymax);
    end;
  if (sp_com ~ = fnosc) AND (sp_com ~ = plain)
    then
      scales;
  border;
  until mapquit;
if gi.tracing
  then
    writeln(trace_o, '  EXIT procedure MAP_SPACE');
end {map_space};


procedure setup_plot;
{   sets up  plotting parameters }
{   suitable for a4 size paper / laser printer}
{   called from either start or restar }
{   end of line }

  begin
    if gi.tracing
      then
```

```
      writeln(trace_o, 'Entered procedure SETUP_PLOT');
    writeln(output, substr(pos_str, 1, mes_pos - 1));
    paper(1);
    cspace(0.00, 1.00, 0.00, 1.00);
    if opt.vert
     then
      map_space(cmd_i, 'vertical')
     else
      map_space(cmd_i, 'horizontal');
    map_space(cmd_i, 'zoom');
    blkpen;
    if gi.tracing
     then
      writeln(trace_o, '  EXIT procedure SETUP_PLOT');
    end {setup_plot};


 procedure disp_plot(el: ptr_type);
{   plot of displacements, called from plot, end of line }

   begin
    if gi.tracing
     then
      writeln(trace_o, 'Entered procedure DISP_PLOT');
    while el ~ = nil do
     with el@ do begin
      gpoint(posn.xc, posn.yc);
      join(posn.xc + s.x, posn.yc + s.y);
      el := next;
      end;
    if gi.tracing
     then
      writeln(trace_o, '  EXIT procedure DISP_PLOT');
    end {disp_plot};


 procedure fram_plot;

   var
    time, yline: real;

   begin
    if gi.tracing
     then
      writeln(trace_o, 'Entered procedure FRAM_PLOT');
    with plspace do begin
     map_space(cmd_i, 'fullnoscales');
     map_space(cmd_i, 'plain');
     time := gi.tstep * total.cycles;
     undlin(1);
     italic(1);
     plotcs(5, 95, gi.heading, length(gi.heading));
     pcsend(85, 95, 'TIME  ', 6);
     plotne(88, 95, time, 4);
     italic(0);
     undlin(0);
     map_space(cmd_i, 'fullnoscales');
     end;
    if gi.tracing
```

```
      then
        writeln(trace_o, '  EXIT procedure FRAM_PLOT');
    end {fram_plot};


  procedure slice_plot(el: ptr_type);
{ plot a snapshot of the geometry, called from plot, end of line, plot a slice }


    function utohead(el: ptr_type): real;

      begin
        if el@.contacts.right ~ = nil
          then
            utohead := sqrt(abs(2 * el@.data.spwp * el@.contacts.right@.con_len
                / opt.grav.y))
          else
            utohead := 0;
        end {utohead};


    begin {slice_plot}
      if gi.tracing
        then
          writeln(trace_o, 'Entered procedure SLICE_PLOT');
      if (el ~ = nil) AND (el@.apexes ~ = nil)
        then
          with el@.apexes@.aw@ do
            positn(c.xc, c.yc + utohead(el));
      while el ~ = nil do
        with el@ do begin
          if apexes = nil
            then
              error_simple('no corners in slice', 'slice_plot')
            else
              with apexes@.aw@, contacts do begin
                apex := apexes;
                join(c.xc, c.yc + utohead(el));
                positn(apex@.c.xc, apex@.c.yc);
                repeat
                  apex := apex@.cw;
                  with apex@ do
                    join(c.xc, c.yc)
                  until apex = apexes;
                positn(c.xc, c.yc + utohead(el));
                end;
          el := el@.next;
          end;
      if gi.tracing
        then
          writeln(trace_o, '  EXIT procedure SLICE_PLOT');
      end {slice_plot};


  procedure force_profile(ele: ptr_type; dire: dir_of_contacts);

    var
      el: ptr_type;
      condir: con_ptr;
```

```
procedure init_fm;

  begin
    el := ele;
    force_map.ymax := - maxreal;
    force_map.ymin := maxreal;
    end {init_fm};


procedure ptrd_fm(elem: ptr_type; dire: dir_of_contacts): con_ptr;

  begin
    case dire of
      based:
        ptrd_fm := elem@.contacts.base;
      righthand:
        ptrd_fm := elem@.contacts.right;
      end;
    end {ptrd_fm};


procedure lims_fm(var miny, maxy: real);

  var
    tenpercent: real;

  begin
    tenpercent := (maxy - miny) / 10;
    if tenpercent = 0
      then
        tenpercent := maxy / 10;
    maxy := maxy + tenpercent;
    miny := miny - tenpercent;
    end {lims_fm};


begin {force_profile}
  if gi.tracing
    then
      writeln(trace_o, 'Entered procedure FORCE_PROFILE');
  with force_map do begin

    el := ele;
    while el ~ = nil do begin
      condir := ptrd_fm(el, dire);
      if condir ~ = nil
        then
          with condir@ do
            consol.lims := sign(consol.lims, consol.ss);
      el := el@.next;
      end;

    init_fm;
    while el ~ = nil do begin
      condir := ptrd_fm(el, dire);
      if condir ~ = nil
        then
```

```
          with condir@.consol do
            case dire of
              righthand: begin
                ymax := max(- el@.data.spwp, ymax, ns);
                ymin := min(- el@.data.spwp, ymin, ns);
                end;
              based: begin
                ymax := max(- el@.data.pwp, ymax, ns);
                ymin := min(- el@.data.pwp, ymin, ns);
                end;
              end;
      el := el@.next;
      end;
    el := ele;
    condir := ptrd_fm(el, dire);
    if condir ~ = nil
      then begin
        lims_fm(ymin, ymax);
        case dire of
          based: begin
            map_space(cmd_i, 'bottom');
            map_space(cmd_i, 'plain');
            ctrset(1);
            plotcs(5, 5, 'ARC - NS', 8);
            map_space(cmd_i, 'bottom');
            end;
          righthand: begin
            map_space(cmd_i, 'uppermiddle');
            map_space(cmd_i, 'plain');
            ctrset(1);
            plotcs(5, 5, 'INTER - NS', 10);
            map_space(cmd_i, 'uppermiddle');
            end;
          end;
        ctrset(4);
        positn(el@.posn.xc, condir@.consol.ns);
        while el ~ = nil do
          with el@.posn do begin
            condir := ptrd_fm(el, dire);
            if condir ~ = nil
              then
                with condir@.consol do begin
                  join(xc, ns);
                  plotnc(xc, ns, 45)
                  end;
            el := el@.next
            end;

        el := ele;
        case dire of
          righthand:
            positn(el@.posn.xc, - el@.data.spwp);
          based:
            positn(el@.posn.xc, - el@.data.pwp);
          end;

        while el ~ = nil do
          with el@.posn do begin
            condir := ptrd_fm(el, dire);
            if condir ~ = nil
```

```
          then
            with condir@.consol do begin
              case dire of
                righthand: begin
                  join(xc, - el@.data.spwp);
                  plotnc(xc, - el@.data.spwp, 43);
                  end;
                based: begin
                  join(xc, - el@.data.spwp);
                  plotnc(xc, - el@.data.spwp, 43);
                  end;
                end;
              end;
        el := el@.next
        end;
    end;

  init_fm;
  while el ~ = nil do begin
    condir := ptrd_fm(el, dire);
    if condir ~ = nil
      then
        with condir@.consol do begin
          if lims > ymax
            then
              ymax := lims;
          if lims < ymin
            then
              ymin := lims;
          if ss > ymax
            then
              ymax := ss;
          if ss < ymin
            then
              ymin := ss;
          end;
    el := el@.next;
    end;

  el := ele;
  ctrset(4);
  condir := ptrd_fm(el, dire);
  if condir ~ = nil
    then begin
      lims_fm(ymin, ymax);
      if dire = based
        then
          map_space(cmd_i, 'lowermiddle')
        else
          map_space(cmd_i, 'top');
      positn(el@.posn.xc, condir@.consol.ss);
      while el ~ = nil do
        with el@.posn do begin
          condir := ptrd_fm(el, dire);
          if condir ~ = nil
            then
              with condir@.consol do begin
                join(xc, ss);
                plotnc(xc, ss, 53)
                end;
```

```
            el := el@.next
            end;
          end;

      el := ele;
      condir := ptrd_fm(el, dire);
      if condir ~ = nil
        then begin
          positn(el@.posn.xc, condir@.consol.lims);
          while el ~ = nil do
            with el@.posn do begin
              condir := ptrd_fm(el, dire);
              if condir ~ = nil
                then
                  with condir@.consol do begin
                    join(xc, lims);
                    plotnc(xc, lims, 45)
                    end;
              el := el@.next
              end;
          ctrset(1);
          map_space(cmd_i, 'plain');
          plotcs(5, 5, 'SS/LIM', 6);
          end;
      end;
    if gi.tracing
      then
        writeln(trace_o, '  EXIT procedure FORCE_PROFILE');
    end {force_profile};


begin {plots}
  if gi.tracing
    then
      writeln(trace_o, 'Entered procedure PLOTS');
  repeat
    get_command(plotter, plotquit, plotcom, plot_command, cmd_i);
    if plotcom IN slices .. standard
      then
        total.pics := total.pics + 1;
    if opt.echo
      then
        writeln(substr(pos_str, 1, err_pos - 1));
    case plotcom of
      displot:
        disp_plot(slice_list);
      slices:
        slice_plot(slice_list);
      forceplot: begin
        fram_plot;
        force_profile(slice_list, based);
        map_space(cmd_i, 'picture');
        slice_plot(slice_list);
        frame;
        total.pages := total.pages + 1;
        end;
      standard: begin
        ;
        fram_plot;
```

```
        force_profile(slice_list, based);
        force_profile(slice_list, righthand);
        frame;
        total.pages := total.pages + 1;
        end;
      frames:
        fram_plot;
      page: begin
        frame;
        total.pages := total.pages + 1;
        end;
      init:
        setup_plot;
      plotstop: begin
        plots(cmd_i, 'standard');
        map_space(cmd_i, 'full');
        fram_plot;
        slice_plot(slice_list);
        grend;
        total.pics := total.pics + 1;
        total.pages := total.pages + 1;
        end;
      zoom:
        map_space(cmd_i, 'zoom');
      maps:
        map_space(cmd_i, '');
      otherwise;
      end;
    if opt.echo
      then
        writeln(output, substr(pos_str, 1, fra_pos), total.pages: 8, substr(
           pos_str, 1, plo_pos), total.pics: 8);
    until plotquit;
  if gi.tracing
    then
      writeln(trace_o, ' EXIT procedure PLOTS');
  end {plots};
{**************************************** END PLOTS }

{**************************************** BEGIN CYCLES }


procedure cycle(var cmd_i: text);

 var
   cycles, no_of_cycles, outcounter, cycle_lim: cycle_type;


 procedure fordsl(el: ptr_type);
{ treats edge contacts as one contact }
{ force displacement law for single block }
{ called from cycle, end of line }

   var
     in_contact: boolean;
     Fn, Fs, nf, sf: real;
     contdir: dir_of_contacts;
     condir: con_ptr;
     bodyfinc: vector_type;
```

```
    nsinc, u, coh, fhi: real;

  begin
    if gi.tracing
      then
        writeln(trace_o, 'Entered procedure FORDSL');
    while el ~ = nil do
      with el@ do begin
        for contdir := righthand to based do begin
          case contdir of
            righthand: begin
              condir := contacts.right;
              if condir ~ = nil
                then begin
                  u := data.spwp;
                  if condir@.failed
                    then
                      data.sidec := data.sidec * 0.85;
                  coh := data.sidec;
                  fhi := data.sphi;
                  end;
              end;
            based: begin
              condir := contacts.base;
              u := data.pwp;
              if condir@.failed
                then
                  data.cohes := data.cohes * 0.85;
              coh := data.cohes;
              fhi := data.phi;
              end;
            end;
          if condir = nil
            then
              continue;
          with condir@, other@.s do begin
            Fn := ((x - s.x) * sine - (y - s.y) * cose) * data.k;
            Fs := - ((y - s.y) * sine + (x - s.x) * cose) * data.k;
            if gi.fording
              then
                writeln(debug_o, 'Fn,Fs,sin,cos,l', Fn: 9, Fs: 9, sine: 9,
                    cose: 9, con_len: 9, s.x: 9, s.y: 9, x: 9, y: 9);
            nsinc := dampf * Fn;
            in_contact := false;
            if consol.ns > - nsinc {total stress}
              then begin
                consol.ss := consol.ss + dampf * Fs; {f/length}
                consol.ns := consol.ns + nsinc;      {f/length}
                in_contact := true;
                consol.pp := max(consol.pp + 0.001 * u, u);
                end;
            end;

          with condir@, consol, other@.force do
            if in_contact
              then begin
                lims := coh + max((ns + pp) * fhi, 0);
                nf := ns * con_len;
                failed := (failed) OR ((pp = u) AND (abs(ss) > abs(lims)));
```

```
            ss := sign(min(abs(ss), lims), ss);
            sf := ss * con_len;
            bodyfinc.x := sf * cose - nf * sine;
            bodyfinc.y := sf * sine + nf * cose;
            force.x := force.x - bodyfinc.x;
            force.y := force.y - bodyfinc.y;
            if contdir = righthand
              then begin
                x := x + bodyfinc.x;
                y := y + bodyfinc.y;
                end;
            if (gi.fording) OR (gi.consoling)
              then
                writeln(debug_o, 'ss,ns,lims,pp,nf,sf', ss: 8, ns: 8, lims
                    : 8, pp: 8, nf: 8, sf: 8);
            if gi.fording
              then
                writeln(debug_o, 'bforces', force.x: 9, force.y: 9);
            end
        end;
      el := next
      end;
  if gi.tracing
    then
      writeln(trace_o, '  EXIT procedure FORDSL');
  end {fordsl};


procedure fconsolsl(el: ptr_type);

  begin
  if gi.tracing
    then
      writeln(trace_o, 'Entered procedure FCONSOLXY');
  while el ~ = nil do
    with el@, el@.data do begin
      s.x := force.x / mass * sqr(gi.tstep);
      s.y := (force.y / mass + opt.grav.y) * sqr(gi.tstep);
      sum.sc := max(abs(s.x), abs(s.y), sum.sc);
      if gi.motioning
        then
          writeln(debug_o, 'disp ', s.x: 9, s.y: 9);
      if (gi.oscing) AND (data.typ = track)
        then begin
          with contacts.base@.consol do
            write(oscil_o, data.sliceno: 4, total.cycles: 6, s.x, s.y, ss,
                ns, lims);
          if contacts.right ~ = nil
            then
              with contacts.right@.consol do
                writeln(oscil_o, ss, ns, lims)
            else
              writeln(oscil_o);
          end;
      force := nilv;
      el := next;
      end;
  if gi.tracing
    then
```

```
      writeln(trace_o, ' EXIT procedure FCONSOLXY');
    end {fconsolsl};

 var
   cyclequit: boolean;


 begin {cycle}
   if gi.tracing
     then
       writeln(trace_o, 'Entered procedure CYCLES');
   if screen
     then
       writeln(output, substr(pos_str, 1, pro_pos),
           ' Enter no of cycles required...');
   read(cmd_i, no_of_cycles);
   if opt.echo
     then
       writeln(output, substr(pos_str, 1, req_pos), no_of_cycles: 8);

   cycles := 0;
   while (cycles < no_of_cycles) AND (~ cyclequit) do begin
     sum.scold := sum.sc;
     sum.sc := 0;
     fordsl(slice_list);
     fconsolsl(slice_list);
     total.cycles := total.cycles + 1;
     cycles := cycles + 1;
     if total.cycles MOD opt.cyclegap = 0
       then begin
         if opt.echo
           then begin
             writeln(output, substr(pos_str, 1, cyc_pos), total.cycles: 8);
             if sum.scold < sum.sc
               then
                 writeln(output, substr(pos_str, 1, mes_pos),
                     ' Decreasing stability ', sum.sc)
               else
                 writeln(output, substr(pos_str, 1, mes_pos),
                     ' Increasing stability ', sum.sc);
             end;
         factors_of_safety(slice_list);
         end;
     if (opt.cmdprocessing) AND (total.cycles MOD opt.cycle_interval = 0)
       then begin
         reset(cycmd_i, 'FILE=-sass.cmd.i');
         gi.cmdend := false;
         while ~ gi.cmdend do
           control(cycmd_i);
         if opt.echo
           then
             writeln(output, substr(pos_str, 1, req_pos), no_of_cycles: 8);
         end;
     if abs(sum.sc / sum.scold - 1) < 1e-13
       then
         sum.scsofar := sum.scsofar + 1
       else
         sum.scsofar := 0;
     cyclequit := (sum.sc < 1e-14) OR (sum.scsofar = 100) OR (sum.sc > 1e6);
```

```
    if trap
      then
        trapper;
    if gi.cycling
      then
        writeln(debug_o, 'max individual disp ', sum.sc);
    end;
  if sum.sc < 1e-14
    then
      writeln(output, substr(pos_str, 1, mes_pos),
          ' Stability has been gained ', sum.sc);
  if sum.scsofar = 100
    then
      writeln(output, substr(pos_str, 1, mes_pos),
          ' Constant sliding now occurring ', sum.sc);
  if sum.sc > 1e6
    then
      writeln(output, substr(pos_str, 1, mes_pos),
          ' This is numerically unstable ', sum.sc);
  if gi.tracing
    then
      writeln(trace_o, '  EXIT procedure CYCLES');
  end {cycle};
{**************************************** END CYCLE }

{**************************************** BEGIN START  }


procedure start_shut;
{   initialises the run, called from control, initialisation modules }

 type
  lhed_type = string(300);
  records = (rvec, rcoo, rcon, rele, rgri, rgen, ropt, rtot, rsum, rhed, bool)
      ;
  buffertype = record
               tag: char;
               case records of
                 rgen: (gen_info_rep: gen_info_type);
                 rvec: (vector_rep: vector_type);
                 rcoo: (coord_rep: coord_type);
                 rcon: (con_rep: con_type);
                 rele: (element_rep: element_type);
                 rgri: (grid_rep: grid_type);
                 ropt: (option_rep: option_type);
                 rtot: (totals_rep: totals_type);
                 rsum: (sum_rep: sum_type);
                 rhed: (hed_rep: hed_type);
                 bool: (null_rep: lhed_type);
             end;

 const
  nullrep = '                                                          ' ||
      '                                                          ' ||
      '                                                          ' ||
      '                                                          ' ||
      '                                                          ';

 var
```

```
  rest_o, rest_i: file of buffertype;
  buffer: buffertype;
  new_slice: ptr_type;


procedure update_area(el: ptr_type);


  procedure update_message(direction: hed_type; el: ptr_type; cont: con_ptr);

    begin
      if gi.updating
        then
          with el@, cont@ do begin
            write(debug_o, ' ', direction: 5, ' Contact created edge, corn ');
            writeln(debug_o, data.sliceno: 6, other@.data.sliceno: 6);
            writeln(debug_o, ' edge x,y ', edge@.c.xc: 6, edge@.c.yc: 6);
            writeln(debug_o, ' corn x,y ', corn@.c.xc: 6, corn@.c.yc: 6);
            writeln(debug_o, ' sin, cos ', sine: 6, cose: 6);
            writeln(debug_o, ' len, dam ', con_len: 6, dampf: 6);
            if direction = 'RIGHT'
              then
                writeln(debug_o, ' pwp, wt ', data.spwp: 6, opt.grav.y * data.
                    mass: 6)
              else
                writeln(debug_o, ' pwp, wt ', data.pwp: 6, opt.grav.y * data.
                    mass: 6)
            end;
      end {update_message};


  procedure cold_contact(el: ptr_type; direction: hed_type; cont: con_ptr);

    var
      dif: vector_type;

    begin
      with el@, cont@ do begin
        with edge@ do begin
          dif.x := cw@.c.xc - c.xc;
          dif.y := cw@.c.yc - c.yc;
          end;
        failed := false;
        con_len := sqrt(sqr(dif.x) + sqr(dif.y));
        sine := dif.y / con_len;
        cose := dif.x / con_len;
        consol.ss := 0;
        consol.ns := 0;
        consol.pp := 0;
        consol.lims := 0;
        if direction = 'RIGHT'
          then begin
            data.spwp := opt.grav.y * sqr(data.spwp) * 2 / con_len;
            if data.mass > other@.data.mass
              then
                dampf := data.mass * opt.damps / con_len
              else
                dampf := other@.data.mass * opt.damps / con_len;
            end
          else begin
```

```
        data.pwp := opt.grav.y * data.pwp;
        if data.mass > other@.data.mass
          then
            dampf := data.mass * opt.damp / con_len
          else
            dampf := other@.data.mass * opt.damp / con_len;
        end;
      end;
    end {cold_contact};

  begin {update_area}
    if gi.tracing
      then
        writeln(trace_o, 'Entered procedure UPDATE_AREA');

    platapex := platen@.apexes;
    while el ~ = nil do
      with el@ do begin
        if starting = cold
          then
            new(contacts.base);
        with contacts.base@ do begin
          other := platen;
          edge := apexes@.aw;
          corn := platapex@.cw;
          if starting = cold
            then
              cold_contact(el, ' BASE', el@.contacts.base);
          update_message(' BASE', el, el@.contacts.base);
          end;
        if next = nil
          then
            contacts.right := nil
          else begin
            if starting = cold
              then
                new(contacts.right);
            with contacts.right@ do begin
              other := next;
              edge := apexes@.cw@.cw;
              corn := next@.apexes@.cw;
              if starting = cold
                then
                  cold_contact(el, ' BASE', el@.contacts.right);
              update_message('RIGHT', el, el@.contacts.right);
              end;
            end;
        total.cons := total.cons + 2;
        platapex := platapex@.cw;
        el := next;
        end;
    if gi.updating
      then
        writeln(debug_o, 'total number of contacts', total.cons);
    if gi.tracing
      then
        writeln(trace_o, '  EXIT procedure UPDATE_AREA');
    end {update_area};
```

```
procedure get_apex(var base, oater: corn_ptr; number: integer);

  var
    num: integer;

  begin
    if gi.tracing
      then
        writeln(trace_o, 'Entered procedure GET_APEX');
    if base = nil
      then begin
        new(base);
        base@.cw := base;
        base@.aw := base;
        oater := base;
        number := number - 1;
        end;
    for num := 1 to number do begin
      new(oater);
      oater@.aw := base@.aw;
      oater@.cw := base;
      base@.aw@.cw := oater;
      base@.aw := oater;
      end;
    if gi.tracing
      then
        writeln(trace_o, '  EXIT procedure GET_APEX');
    end {get_apex};


procedure mesh;


  procedure cre_platen(el: ptr_type);

    begin
      if gi.tracing
        then
          writeln(trace_o, 'Entered procedure CRE_PLATEN');
      new(platen);
      platen@.next := nil;
      platen@.apexes := nil;
      get_apex(platen@.apexes, apex, total.slices + 1);
      platapex := platen@.apexes;
      platapex@.c := el@.apexes@.c;
      while el ~ = nil do begin
        platapex := platapex@.cw;
        platapex@.c := el@.apexes@.aw@.c;
        el := el@.next
        end;
      with platen@, platen@.contacts, platen@.data do begin
        posn := nilc;
        force := nilv;
        s := nilv;
        mass := 0;
        cohes := 0;
        phi := 0;
```

```
      rho := 0;
      k := 0;
      sidec := 0;
      sphi := 0;
      pwp := 0;
      spwp := 0;
      e := 0;
      sliceno := 0;
      typ := free;
      right := nil;
      base := nil;
      end;
    if gi.tracing
      then
        writeln(trace_o, '  EXIT procedure CRE_PLATEN');
    end {cre_platen};


  procedure cre_slices;

    var
      sort: string(12);
      typs: com_type;
      surf, temp: real;
      x, y, xn, yn: real;

    begin {FIFO OF ELEMENTS}
      if gi.tracing
        then
          writeln(trace_o, 'Entered procedure CRE_SLICES');
      get_command(mesher, qdum, typs, '', cmd_i);
      while NOT eoln do begin
        case typs of
          track, free:
            if slice_list = nil
              then begin
                new(slice_list);
                eolist := slice_list;
                eolist@.next := nil;
                end
              else begin
                new(eolist@.next);
                eolist := eolist@.next;
                eolist@.next := nil;
                end;
          otherwise
            return;
          end;

        with eolist@, eolist@.data, opt.meshtbs do begin
          read(cmd_i, cohes, phi, rho, k, sidec, sphi, pwp, spwp, e);
          phi := phi * arctan(1) / 45;
          sphi := sphi * arctan(1) / 45;
          phi := sin(phi) / cos(phi);
          sphi := sin(sphi) / cos(sphi);
          total.slices := total.slices + 1;
          sliceno := total.slices;
          typ := typs;
          apexes := nil;
```

```
        posn := nilc;
        force := nilv;
        s := nilv;
        get_apex(apexes, apex, 4);
        apex := apexes;
        if total.slices = 1
          then
            read(cmd_i, xb, yb, xt, yt);
        apex@.c.xc := xb;
        apex@.c.yc := yb;
        apex := apex@.cw;
        apex@.c.xc := xt;
        apex@.c.yc := yt;
        apex := apexes@.aw;
        read(cmd_i, xb, yb, xt, yt);
        apex@.c.xc := xb;
        apex@.c.yc := yb;
        apex := apex@.aw;
        apex@.c.xc := xt;
        apex@.c.yc := yt;

{ area and centroid of block }
        surf := 0;
        apex := apexes;
        repeat
          with apex@.c, apex@.aw@ do begin
            surf := surf + (xc - c.xc) * (yc + c.yc);
            posn.yc := posn.yc + (xc - c.xc) * ((yc - c.yc) * (yc + 2 * c.yc
                ) + 3 * sqr(c.yc));
            posn.xc := posn.xc + (yc - c.yc) * ((xc - c.xc) * (xc + 2 * c.xc
                ) + 3 * sqr(c.xc));
            apex := apex@.cw;
            end;
          until apex = apexes;
        surf := surf * 0.5;
        posn.yc := posn.yc / (6 * surf);
        posn.xc := - posn.xc / (6 * surf);
        mass := surf * data.rho;
        if gi.updating
          then
            writeln(debug_o, 'mass,surf', mass, surf);
        end;
      end;
    if gi.tracing
      then
        writeln(trace_o, '  EXIT procedure CRE_SLICES');
    end {cre_slices};

  var
    meshquit: boolean;
    meshcom: com_type;


  begin {mesh}
    if gi.tracing
      then
        writeln(trace_o, 'Entered procedure MESH');

    repeat
```

```
    get_command(creater, meshquit, meshcom, '', cmd_i);
    case meshcom of
      create:
        cre_slices;
      meshend:
        meshquit := true otherwise;
      end
    until meshquit;
  cre_platen(slice_list);
  if gi.tracing
    then
      writeln(trace_o, '  EXIT procedure MESH');
  end {mesh};


procedure read_restart_file;

  var
    labl: char;
    ele: ptr_type;
    new_con: con_ptr;

  begin
    if gi.tracing
      then
        writeln(trace_o, 'Entered procedure READ_RESTART_FILE');
    reset(rest_i, 'unit=2');
    rewrite(cycmd_i, 'FILE=-sass.cmd.i');
    rewrite(repts_i, 'FILE=-sass.rep.i');
    while ~ eof(rest_i) do begin
      read(rest_i, buffer);
      labl := buffer.tag;
      case labl of
        'G': begin
          gi := buffer.gen_info_rep;
          read(rest_i, buffer);
          total := buffer.totals_rep;
          read(rest_i, buffer);
          sum := buffer.sum_rep;
          read(rest_i, buffer);
          opt := buffer.option_rep;
          read(rest_i, buffer);
          plspace := buffer.grid_rep;
          end;
        'c':
          writeln(cycmd_i, buffer.hed_rep);
        'r':
          writeln(repts_i, buffer.hed_rep);
        'F': begin
          new(new_slice);
          new_slice@ := buffer.element_rep;
          new_slice@.next := nil;
          new_slice@.apexes := nil;
          if slice_list = nil
            then
              slice_list := new_slice
            else
              ele@.next := new_slice;
          ele := new_slice;
```

```
        end;
       'R': begin
        new(new_con);
        new_con@ := buffer.con_rep;
        ele@.contacts.right := new_con;
        end;
       'B': begin
        new(new_con);
        new_con@ := buffer.con_rep;
        ele@.contacts.base := new_con;
        end;
       'P': begin
        new(new_slice);
        new_slice@ := buffer.element_rep;
        platen := new_slice;
        platen@.next := nil;
        platen@.apexes := nil;
        platen@.contacts.right := nil;
        platen@.contacts.base := nil;
        ele := platen;
        end;
       'a': begin
        get_apex(ele@.apexes, apex, 1);
        apex@.c := buffer.coord_rep;
        end;
       '*':;
       end;
     end;
   writeln(output, substr(pos_str, 1, mes_pos),
      ' A restart file has been read');
   if gi.tracing
     then
       writeln(trace_o, '  EXIT procedure READ_RESTART_FILE');
   end {read_restart_file};


procedure write_restart_file;


  procedure write_r_el(el: ptr_type; c: char);

    begin
      if gi.tracing
        then
          writeln(trace_o, 'Entered procedure WRITE_R_EL');
      while el ~ = nil do
        with el@ do begin
          buffer.tag := c;
          buffer.null_rep := nullrep;
          buffer.element_rep := el@;
          write(rest_o, buffer);
          if contacts.right ~ = nil
            then begin
              buffer.tag := 'R';
              buffer.null_rep := nullrep;
              buffer.con_rep := contacts.right@;
              write(rest_o, buffer)
              end;
          if contacts.base ~ = nil
```

```
          then begin
            buffer.tag := 'B';
            buffer.null_rep := nullrep;
            buffer.con_rep := contacts.base@;
            write(rest_o, buffer)
            end;
        apex := apexes;
        buffer.tag := 'a';
        repeat
          buffer.null_rep := nullrep;
          buffer.coord_rep := apex@.c;
          write(rest_o, buffer);
          apex := apex@.cw
          until apex = apexes;
        el := el@.next
        end;
    if gi.tracing
      then
        writeln(trace_o, '  EXIT procedure WRITE_R_EL');
    end {write_r_el};


  begin {write_restart_file}
    if gi.tracing
      then
        writeln(trace_o, 'Entered procedure WRITE_RESTART_FILE');

    if (opt.rf_over) OR (rf_first)
      then
        rewrite(rest_o, 'unit=1');
    rf_first := false;
    buffer.tag := 'G';
    buffer.null_rep := nullrep;
    buffer.gen_info_rep := gi;
    write(rest_o, buffer);
    buffer.null_rep := nullrep;
    buffer.totals_rep := total;
    write(rest_o, buffer);
    buffer.null_rep := nullrep;
    buffer.sum_rep := sum;
    write(rest_o, buffer);
    buffer.null_rep := nullrep;
    buffer.option_rep := opt;
    write(rest_o, buffer);
    buffer.null_rep := nullrep;
    buffer.grid_rep := plspace;
    write(rest_o, buffer);

    reset(cycmd_i, 'FILE=-sass.cmd.i');
    buffer.tag := 'c';
    while ~ eof(cycmd_i) do begin
      buffer.null_rep := nullrep;
      readln(cycmd_i, buffer.hed_rep);
      write(rest_o, buffer);
      end;
    reset(repts_i, 'FILE=-sass.rep.i');
    buffer.tag := 'r';
    while ~ eof(repts_i) do begin
      buffer.null_rep := nullrep;
```

```
    readln(repts_i, buffer.hed_rep);
    write(rest_o, buffer);
    end;
   write_r_el(slice_list, 'F');
   write_r_el(platen, 'P');
   buffer.tag := '*';
   buffer.null_rep := nullrep;
   buffer.hed_rep := 'END of RESTART FILE ';
   write(rest_o, buffer);
   writeln(output, substr(pos_str, 1, fil_pos),
      ' A restart file has been written');
   if gi.tracing
    then
     writeln(trace_o, ' EXIT procedure WRITE_RESTART_FILE');
   end {write_restart_file};


 procedure complete;

  begin
   if gi.tracing
    then
     writeln(trace_o, 'Entered procedure COMPLETE');
   with total do begin
    writeln(output, substr(pos_str, 1, tot_pos), '     total slices ',
       slices: 6, ' contacts  ', cons: 6);
    writeln(output, '     total cycles ', cycles: 6, ' restarts  ',
       restarts: 6);
    writeln(output, '     total frames ', pages: 6, ' plots    ', pics: 6)
     ;
    end;
   if gi.tracing
    then
     writeln(trace_o, ' EXIT procedure COMPLETE');
   end {complete};


 begin {start_shut}
  if gi.tracing
   then
    writeln(trace_o, 'Entered procedure START_SHUT');
  case starting of
   cold: begin
    if screen
     then
      writeln(output, substr(pos_str, 1, pro_pos),
         ' Enter heading ................');
    readln(cmd_i, gi.heading);
    writeln(output, substr(pos_str, 1, tit_pos), gi.heading);
    writeln(debug_o, gi.heading);
    plots(cmd_i, 'initialise');
    mesh;
    update_area(slice_list);
    plots(cmd_i, 'border');
    plots(cmd_i, 'slices');
    plots(cmd_i, 'page');
    end;
   shutdown: begin
    plots(input, 'endplot');
```

```
      complete;
      factors_of_safety(slice_list);
      writeln(debug_o, sum.sc);
      write_restart_file;
      writeln(output, substr(pos_str, 1, fil_pos), curson);
      halt;
      end;
    warm: begin
      read_restart_file;
      total.restarts := total.restarts + 1;
      plots(cmd_i, 'initialise');
      update_area(slice_list);
      plots(cmd_i, 'border');
      plots(cmd_i, 'slices');
      plots(cmd_i, 'page');
      end;
    keep:
      write_restart_file;
    end;
  if gi.tracing
    then
      writeln(trace_o, ' EXIT procedure START_SHUT');
  end {start_shut};
{**************************************** END STARTSHUT }

{**************************************** BEGIN DEBUG }


procedure debug_slice(var cmd_i: text);
{   debugging routine,   called from contrl,   calls dump }

  var
    debugend: boolean;
    deb_com: com_type;


  procedure write_con(el: ptr_type);


    procedure wr_con(el: ptr_type; con: con_ptr);

      begin
        if gi.tracing
          then
            writeln(trace_o, 'Entered procedure WR_CON');
        with con@ do begin
          writeln(debug_o, 'slice home, other, damp ', el@.data.sliceno: 3,
              other@.data.sliceno: 3, dampf: 6);
          writeln(debug_o, 'corner coordinates x, y ', corn@.c.xc: 6, corn@.c.yc
              : 6);
          writeln(debug_o, 'edge coordinate  s x, y ', edge@.c.xc: 6, edge@.c.yc
              : 6);
          write(debug_o, 'stresses  -  n, s, l, u ', consol.ns: 6, consol.ss: 6,
              consol.lims: 6);
          end;
        if gi.tracing
          then
            writeln(trace_o, ' EXIT procedure WR_CON');
        end {wr_con};
```

```
  begin {write_con}
    if gi.tracing
      then
        writeln(trace_o, 'Entered procedure WRITE_CON');
    writeln(debug_o, ' Contact information :');
    writeln(debug_o);
    writeln(debug_o, ' coords of    corn, edge');
    writeln(debug_o);
    while el ~ = nil do
      with el@.contacts do begin
        if right ~ = nil
          then begin
            wr_con(el, right);
            writeln(debug_o, el@.data.spwp: 6);
            end;
        if base ~ = nil
          then begin
            wr_con(el, base);
            writeln(debug_o, el@.data.pwp: 6);
            end;
        el := el@.next;
        end;
    if gi.tracing
      then
        writeln(trace_o, '  EXIT procedure WRITE_CON');
    end {write_con};


procedure write_sli(el: ptr_type);

  begin
    if gi.tracing
      then
        writeln(trace_o, 'Entered procedure WRITE_SLI');
    writeln(debug_o, ' Element data :');
    writeln(debug_o);
    writeln(debug_o, 'mass  force  x     y disp  x     y  n');
    writeln(debug_o);
    while el ~ = nil do
      with el@ do begin
        writeln(debug_o, data.mass: 6, force.x: 8, force.y: 8, s.x: 8, s.y: 8,
            data.sliceno: 3);
        el := el@.next;
        end;
    if gi.tracing
      then
        writeln(trace_o, '  EXIT procedure WRITE_SLI');
    end {write_sli};


begin {debug_slice}
  if gi.tracing
    then
      writeln(trace_o, 'Entered procedure DEBUG_SLICE');
  repeat
    get_command(debuger, debugend, deb_com, '', cmd_i);
    case deb_com of
```

```
slices:
  write_sli(slice_list);
con:
  write_con(slice_list);
gen: begin
  writeln(debug_o, gi.heading);
  writeln(debug_o);
  with plspace, opt, total do begin
    writeln(debug_o, ' mapping  xmin   ', xmin: 6, ' xmax   ', xmax: 6)
      ;
    writeln(debug_o, ' mapping  ymin   ', ymin: 6, ' ymax   ', ymax: 6)
      ;
    writeln(debug_o);
    writeln(debug_o, ' plot    interval', cycle_interval: 6);
    writeln(debug_o, ' gravity  x      ', grav.x: 6, ' y      ', grav.y
       : 6);
    writeln(debug_o, ' damping  base    ', damp: 6, ' side  ', damps: 6
       );
    writeln(debug_o);
    writeln(debug_o, ' totals  slices  ', slices: 6, ' contact', cons:
       6);
    writeln(debug_o, '         cycles  ', cycles: 6, ' restarts',
       restarts: 6);
    writeln(debug_o, '         frames  ', pages: 6, ' plots  ', pics:
       6);
    writeln(debug_o);
    end;
  end;
fon:
  with gi do begin
    motioning := true;
    updating := true;
    cycling := true;
    fording := true;
    oscing := true;
    tracing := true;
    end;
fof:
  with gi do begin
    motioning := false;
    updating := false;
    cycling := false;
    fording := false;
    oscing := false;
    tracing := false;
    end;
mot:
  gi.motioning := onoff(cmd_i);
csl:
  gi.consoling := onoff(cmd_i);
upd:
  gi.updating := onoff(cmd_i);
cyc:
  with gi do begin
    cycling := onoff(cmd_i);
    fording := cycling;
    motioning := cycling;
    end;
fod:
  gi.fording := onoff(cmd_i);
```

```
      tra:
        gi.tracing := onoff(cmd_i);
      osc:
        gi.oscing := onoff(cmd_i);
      otherwise;
      end;
    until debugend;
  if gi.tracing
    then
      writeln(trace_o, ' EXIT procedure DEBUG_SLICE');
  end {debug_slice};
{**************************************** END DEBUG }

{**************************************** BEGIN PARAMETERS }


procedure parameters(var cmd_i: text);


 procedure calculator;


   function intcalc(op: real): real;

     var
       result, v: real;
       oper: com_type;

     begin
       get_command(operter, qdum, oper, '', cmd_i);
       if oper ~ = enquiry
         then begin
           if screen
             then
               writeln(output, substr(pos_str, 1, pro_pos),
                   ' Enter value .................');
           read(cmd_i, v);
           end;
       case oper of
         equal:
           result := v;
         mult:
           result := op * v;
         divid:
           result := op / v;
         plus:
           result := op + v;
         minus:
           result := op - v;
         power:
           result := exp(ln(op) * v);
         otherwise
           result := op;
         end;
       if opt.echo
         then
           writeln(output, substr(pos_str, 1, mes_pos), ' The value is : ',
               result: 12: 7, '    ');
       intcalc := result;
```

```
      end {intcalc};

    var
      calquit: boolean;
      calcom: com_type;


    begin {calculator}
      repeat
        get_command(calcter, calquit, calcom, '', cmd_i);
        case calcom of
          cyclegp:
            opt.cyclegap := round(intcalc(opt.cyclegap));
          gravity:
            opt.grav.y := intcalc(opt.grav.y);
          ptime:
            gi.tstep := intcalc(gi.tstep);
          cmdint:
            opt.cycle_interval := round(intcalc(opt.cycle_interval));
          fdamp:
            opt.damp := intcalc(opt.damp);
          otherwise;
          end;
        until calquit;
      end {calculator};

  var
    parcom: com_type;
    parquit: boolean;
    flimit: integer;
    cmdlistword: string(12);


  begin {parameters}
    repeat
      get_command(paramer, parquit, parcom, '', cmd_i);
      case parcom of
        echo:
          opt.echo := onoff(cmd_i);
        framlim: begin
          if screen
            then
              writeln(output, substr(pos_str, 1, pro_pos),
                  ' Enter frame limit ............');
          read(cmd_i, flimit);
          gpstop(flimit);
          if opt.echo
            then
              writeln(output, substr(pos_str, 1, mes_pos),
                  ' Frame limit is now : ', flimit);
          end;
        cyclegp: begin
          if screen
            then
              writeln(output, substr(pos_str, 1, pro_pos),
                  ' Enter gap between writing.....');
          read(cmd_i, opt.cyclegap);
          if opt.echo
            then
```

```
        writeln(output, substr(pos_str, 1, mes_pos),
            ' Cycle gap is now  : ', opt.cyclegap);
    end;
  gravity: begin
    if screen
      then
        writeln(output, substr(pos_str, 1, pro_pos),
            ' Enter gravity values x, y ....');
    read(cmd_i, opt.grav.y);
    if opt.echo
      then
        writeln(output, substr(pos_str, 1, mes_pos),
            ' Gravity is now    : ', opt.grav.y: 6);
    end;
  ptime: begin
    if screen
      then
        writeln(output, substr(pos_str, 1, pro_pos),
            ' Enter time step increment ....');
    read(cmd_i, gi.tstep);
    if opt.echo
      then
        writeln(output, substr(pos_str, 1, mes_pos),
            ' Time increment is  : ', gi.tstep);
    end;
  fdamp: begin
    if screen
      then
        writeln(output, substr(pos_str, 1, pro_pos),
            ' Enter value for damping ......');
    read(cmd_i, opt.damp);
    read(cmd_i, opt.damps);
    if opt.echo
      then
        writeln(output, substr(pos_str, 1, mes_pos),
            ' Damping factor is  : ', gi.tstep);
    end;
  calc:
    calculator;
  cmdint: begin
    if screen
      then
        writeln(output, substr(pos_str, 1, pro_pos),
            ' Enter cmd process interval ...');
    read(cmd_i, opt.cycle_interval);
    if opt.echo
      then
        writeln(output, substr(pos_str, 1, mes_pos),
            ' Process interval is: ', opt.cycle_interval);
    end;
  cmdlist: begin
    rewrite(cycmd_i, 'FILE=-sass.cmd.i');
    repeat
      word_scan(cmd_i, cmdlistword);
      writeln(cycmd_i, cmdlistword)
      until cmdlistword = 'cend';
    end;
  listpr:
    opt.cmdprocessing := onoff(cmd_i);
  over_rf:
```

```
        opt.rf_over := onoff(cmd_i);
      otherwise;
      end;
    until parquit;
  end {parameters};
{****************************************** END PARAMETERS }

{****************************************** BEGIN REPEATER }


procedure repeater(var cmd_i: text);

  var
    cmdreptword: string(12);
    loopcntor, loopctr: integer;

  begin
    if gi.tracing
      then
        writeln(trace_o, 'Entered procedure REPEATER');
    rewrite(repts_i, 'FILE=-sass.rep.i');
    read(cmd_i, loopctr);
    repeat
      word_scan(cmd_i, cmdreptword);
      writeln(repts_i, cmdreptword)
      until cmdreptword = 'rend';
    for loopcntor := 1 to loopctr do begin
      reset(repts_i, 'FILE=-sass.rep.i');
      gi.reptend := false;
      repeat
        control(repts_i)
        until gi.
      reptend;
      end;
    if gi.tracing
      then
        writeln(trace_o, '  EXIT procedure REPEATER');
    end {repeater};
{****************************************** END REPEATER }

{****************************************** BEGIN CONTROL }


procedure control;
{   controls the execution of the datafile commands, called from main }

  var
    com: com_type;

  begin
    if gi.tracing
      then
        writeln(trace_o, 'Entered procedure CONTROL');
    get_command(contler, qdum, com, '', cmd_i);
    case com of
      sets:
        parameters(cmd_i);      { set parameter values }
      cend:
        gi.cmdend := true;      { end interrupt commands }
```

```
    rend:
      gi.reptend := true;      { end command stack }
    rest:
      start_shut(cmd_i, warm); { restart a previous run }
    save:
      start_shut(cmd_i, keep); { update restart file }
    star:
      start_shut(cmd_i, cold); { start a new run }
    cycl:
      cycle(cmd_i);            { calculation routines }
    plot:
      plots(cmd_i, '');        { plot routines }
    debg:
      debug_slice(cmd_i);      { debugging routine }
    rept:
      repeater(cmd_i);         { command stack }
    stop:
      start_shut(cmd_i, shutdown);{ stop command }
    retur:;
    end;
  if gi.tracing
    then
      writeln(trace_o, ' EXIT procedure CONTROL');
  end {control};
{***************************************** END CONTROL }

{***************************************** BEGIN MAIN }


begin {slices}
  initialise_globals;
  headers;
  repeat
    control(input);
    until quit;
  end {slices}.
```

APPENDIX E

PROGRAM CIRCLES

```
program circles(debug_o, sercom);
%include est8:u.ghost.lib
%include trap

const
  led_pos = 2;
  tit_pos = 4;
  req_pos = 6;
  fra_pos = 7;
  plo_pos = 8;
  upd_pos = 9;
  cra_pos = 10;
  cyc_pos = 11;
  com_pos = 12;
  mes_pos = 13;
  err_pos = 14;
  tot_pos = 14;
  fil_pos = 20;
  pro_pos = 16;
  pos_str = ' ';
  maxcycle = 1000000;
  blank = ' ';
  commands = {onoffer}
  'null       on         off        ' ||                        {onoffer}
  'picture    horizontal vertical   full      fullnoscales' || {map}
  'zoom       ' ||
  'initialise ballplot   dotplot    velocities displacement' || {plot}
  'conplot    failplot   graticule  standard   page       ' ||
  'border     map        endplot    ' ||
  'create     relative   absolute   dataset    for        ' ||
  'endfor     single     multiple   meshend    position   ' ||
  'move       angle      free       fixed      track      ' || {mesh}
  '                      ' ||                                  {spare}
  '=          *          /          +          -          ' ||
  '^          ?          ' ||                                  {operators}
  'echo       echodebug  cmdproc    overwrite  framelimit ' || {set}
  'writegap   interval   cmdlist    gravity    time       ' ||
  'calculate  soiltype   ' ||
  'damp       mass       cohesion   friction   density    ' || {parameter}
  'radius     stiffness  ' ||
  'datalist   blocks     areas      contacts   general    ' || {debug}
  'flagson    flagsoff   rearea     update     cycle      ' ||
  'motion     ford       consolidate trace     oscillate  ' ||
  'set        cend       rend       restart    save       ' || {control}
  'start      go         plot       repeat     debug      ' ||
  'settle     collapse   stop       return     ';

type
  com_type = (null, on, off, piccie, horiz, vertic, whole, fnosc, zoom, init,
          ballplot, dotplot, velplot, displot, conplot, failplot, graticule,
```

```
          standard, page, frames, maps, plotstop, create, relative,
          absolute, dataset, forloop, endfor, sing, multip, meshend,
          position, movepos, angle, free, fixed, track, cracked, both,
          equal, mult, divid, plus, minus, power, enquiry, echo, debech,
          listpr, over_rf, framlim, cyclegp, cmdint, cmdlist, gravity,
          ptime, calc, datype, dfact, dmass, dcohe , dfric, ddens, dradi,
          dstif, dat, blk, are, con, gen, fon, fof, reb, upd, mot, cyc, fod,
          sol, tra, osc, sets, cend, rend, rest, save, star, cycl, plot,
          rept, debg, sett, coll, stop, retur);

call_type = (errorer, onoffer, mapper, plotter, mesher, datert, operter,
          calcter, datalte, paramer, debuger, contler);
el_list_types = free .. both;
para_ptr = @parabk_type;
ptr_type = @element_type;
con_ptr = @con_type;
parabk_type = record
            damp, mass, cohes, phi, rho, rad, kn: real;
            preincarnate, flagno: integer;
            typ: el_list_types;
            next_data: para_ptr;
           end;
vector_type = record
            x, y: real;
           end;
coord_type = record
           xc, yc: real;
          end;
con_type = record
          gapsum, offs: real;
          other: ptr_type;
          next_con: con_ptr;
          c_force: vector_type;
          f_force, f_angle: real;
          failed: boolean;
         end;
element_type = record
            source, posn: coord_type;
            consol, force, v, a, s: vector_type;
            data: para_ptr;
            no_of_contacts: integer;
            con_list: con_ptr;
            next: ptr_type;
           end;
grid_type = record
           xmin, xmax, ymin, ymax: real;
          end;
rowcol_type = - 1 .. 100;
area_directions = (self, n, ne, e, se, s, sw, w, nw, nex);
area_ptr = @area_type;
area_type = record
           corners: grid_type;
           upd_min, upd_par: real;
           row, col: rowcol_type;
           n, e, s, w, next_area: area_ptr;
           fixed_list, free_list: ptr_type;
          end;
cycle_type = 0 .. maxcycle;
hed_type = string(80);
start_type = (cold, warm, shutdown, keep);
```

```
  gen_info_type = record
                  heading: hed_type;
                  nextword: string(12);
                  charsize, tfrac, tstep, max_rad: real;
                  settling, reptend, cmdend, jumping, single, reareaing,
                     motioning, updating, consoling, cycling, fording,
                     tracing, oscing, debecho: boolean
                end;
  area_i_type = record
                size: coord_type;
                xmax, ymax: 0 .. 100;
                nos: integer;
              end;
  option_type = record
                plot_lims: grid_type;
                meshtbs: record
                        xb, yb, xt, yt: real
                      end;
                grav: vector_type;
                cyclegap, cycle_interval: cycle_type;
                cmdprocessing, echo1, echo, rf_over: boolean;
              end;
  sum_type = record
             en, sc, scold: real
           end;
  totals_type = record
                cycles, updates, circles, fixed, cracked, cons, pics, pages,
                   datatypes: integer
              end;

const

  nilv = vector_type(0, 0);
  nilc = coord_type(0, 0);
  nilhed = ' ';
  tens_fuzz = 0.05;
{make this a parameter sometime as 0.05 * max_rad/datatype}

var

  repts_i, cycmd_i, oscil_o, debug_o, trace_o, sercom: text;
  screen, rf_first, cy_first, quit, qdum: boolean;
  gi: gen_info_type;
  opt: option_type;
  sum: sum_type;
  total: totals_type;
  sdl, cdp: para_ptr;
  plspace, plot_space, force_map: grid_type;
  this_area, area, spare_area, sal: area_ptr;
  area_i: area_i_type;
  re_area_list: ptr_type;
{**************************************** BEGIN GLOBAL ROUTINES }


procedure error_simple(ob, caller: string(40));

  begin
   if gi.tracing
    then
```

```
      writeln(trace_o, 'Entered procedure ERROR_SIMPLE');
    rewrite(sercom, 'UNIT=11');
    writeln(sercom, '! error ', '''', ob, '''', ' found in routine ', caller);
    halt;
    end {error_simple};


procedure word_scan(var cmds_in: text; var word: string(12));

  var
    ch: string(1);


  procedure skipblks(var ch: string(1));

    begin
      if gi.tracing
        then
          writeln(trace_o, 'Entered procedure SKIPBLKS');
      ch := ' ';
      while ch = blank do begin
        while (~ eoln(cmds_in)) AND (ch = blank) do
          read(cmds_in, ch);
        if (eoln(cmds_in)) AND (~ eof(cmds_in))
          then
            readln(cmds_in);
        if eof(cmds_in)
          then
            error_simple(' End of file causes return to mts', 'skipblks');
        end;
      if gi.tracing
        then
          writeln(trace_o, '  EXIT procedure SKIPBLKS');
      end {skipblks};


  procedure skipcomment(var ch: string(1));

    begin
      if gi.tracing
        then
          writeln(trace_o, 'Entered procedure SKIPCOMMENT');
      while ch ~ = '}' do begin
        while (~ eoln(cmds_in)) AND (ch ~ = '}') do
          read(cmds_in, ch);
        if (eoln(cmds_in)) AND (~ eof(cmds_in))
          then
            readln(cmds_in);
        if eof(cmds_in)
          then
            error_simple('end of file causes return to mts', 'skipcomment');
        end;
      skipblks(ch);
      if ch = '{'
        then
          skipcomment(ch);
      if gi.tracing
        then
          writeln(trace_o, '  EXIT procedure SKIPCOMMENT');
```

```
    end {skipcomment};

  begin {word_scan}
    if gi.tracing
      then
        writeln(trace_o, 'Entered procedure WORD_SCAN');
    word := '';
    if screen
      then begin
        writeln(output, substr(pos_str, 1, err_pos + 2),
            '     Input a command please ..........    ');
        reset(cmds_in, 'UNIT=11,INTERACTIVE');
        repeat
          read(cmds_in, ch)
          until ch ~ = '';
        end
      else
        skipblks(ch);
    while (~ eoln(cmds_in)) AND (ch ~ = blank) do begin
      if ch = '{'
        then
          skipcomment(ch);
      word := word || ch;
      read(cmds_in, ch);
      end;
    if ch ~ = blank
      then
        word := word || ch;
    if opt.echo
      then
        writeln(output, substr(pos_str, 1, com_pos), ' Command : ', word,
            '     ');
    if (eoln(cmds_in)) AND (~ eof(cmds_in))
      then
        readln(cmds_in);
    if gi.tracing
      then
        writeln(trace_o, '  EXIT procedure WORD_SCAN    ', word);
    end {word_scan};


procedure control(var cmd_i: text);
  forward;


procedure start_shut(var cmd_i: text; starting: start_type);
  forward;


procedure trapper;

  var
    ch: char;

  begin
    if gi.tracing
      then
        writeln(trace_o, 'Entered procedure TRAPPER');
```

```
  writeln(output, substr(pos_str, 1, err_pos),
     ' Attn!  : Do you want to stop ?');
  reset(sercom, 'UNIT=11');
  repeat
    read(sercom, ch);
    until (ch ~ = ' ');
  trpreset;
  if ch = 'y'
    then
      start_shut(input, shutdown);
  writeln(output, substr(pos_str, 1, err_pos),
     '                              ');
  if gi.tracing
    then
      writeln(trace_o, '  EXIT procedure TRAPPER');
  end {trapper};


procedure get_command(caller: call_type; var quiter: boolean; var retcom:
  com_type; intcall: string(12); var cmds_ig: text);

 const
  last = 1122;

 var
  ifail: boolean;
  beg, loca, indes: 0 .. 1200;
  this_com: string(12);

 begin
  if gi.tracing
    then
      writeln(trace_o, 'Entered procedure GET_COMMAND  ', intcall);
  this_com := intcall;
  if this_com = ''
    then begin
      this_com := gi.nextword;
      gi.nextword := '';
      end;
  if this_com = ''
    then
      word_scan(cmds_ig, this_com);
  if trap
    then
      trapper;

  beg := 1;
  repeat
    indes := index(substr(commands, beg, last - beg + 1), this_com);
    loca := indes + beg - 1;
    if loca MOD 12 = 1
      then
        indes := 0
      else
        beg := loca + 1;
    until (indes = 0) OR (last - beg < 12);
  retcom := com_type(loca DIV 12);
  case caller of
    errorer:
```

```
      ifail := NOT (retcom IN on .. retur );
    onoffer:
      ifail := NOT (retcom IN on .. off );
    mapper:
      ifail := NOT (retcom IN piccie .. zoom );
    plotter:
      ifail := NOT (retcom IN init .. plotstop, zoom );
    datert:
      ifail := NOT (retcom IN free .. track );
    mesher:
      ifail := NOT (retcom IN create .. angle );
    operter:
      ifail := NOT (retcom IN equal .. enquiry );
    datalte:
      ifail := NOT (retcom IN dfact .. dstif );
    calcter:
      ifail := NOT (retcom IN cyclegp, cmdint, gravity, ptime, datype );
    paramer:
      ifail := NOT (retcom IN echo .. calc );
    debuger:
      ifail := NOT (retcom IN dat .. osc );
    contler:
      ifail := NOT (retcom IN sets .. retur );
    end;
  if ifail
    then begin {some thing's wrong}
      if (retcom = null) OR (caller = contler)
        then begin {invalid command}
          screen := true;
          write(output, substr(pos_str, 1, err_pos));
          writeln(output, '! error ', '''', this_com, '''',
              ' found in routine ', 'get_command');
          writeln(output, 'Input corrected commands ... <RETURN> ...');
          get_command(errorer, ifail, retcom, '', sercom);
          while retcom ~ = retur do begin
            gi.nextword := substr(commands, ord(retcom) * 12 + 1, 12);
            control(sercom); {control returns with nextword = return|keyword}
            get_command(errorer, ifail, retcom, gi.nextword, sercom);
            end;
          screen := false;
          gi.nextword := 'return';
          quiter := false;
          end
        else begin {valid command wrong caller}
          quiter := true;
          gi.nextword := this_com;
          end;
      end
    else
      quiter := intcall ~ = '';  {alls ok}
  if gi.tracing
    then
      writeln(trace_o, ' EXIT procedure GET_COMMAND');
  end {get_command};


function onoff(var cmd_i: text): boolean;

 var
```

```
    onof: com_type;

  begin
    if gi.tracing
      then
        writeln(trace_o, 'Entered procedure ONOFF');
    get_command(onoffer, qdum, onof, '', cmd_i);
    case onof of
      on:
        onoff := true;
      off:
        onoff := false;
      otherwise;
      end;
    if gi.tracing
      then
        writeln(trace_o, '  EXIT procedure ONOFF');
    end {onoff};


procedure headers;

  begin
    writeln(output, ' .0');
    writeln(output, substr(pos_str, 1, led_pos),
        ' PROGRAM CIRCLES RUNNING COMMENTARY ON : ');
    writeln(output, substr(pos_str, 1, tit_pos), ' ', gi.heading);
    writeln(output, substr(pos_str, 1, req_pos), '     0  cycles requested');
    writeln(output, ' ', total.pages: 6, '  frames plotted');
    writeln(output, ' ', total.pics: 6, '  plots types drawn');
    writeln(output, ' ', total.updates: 6, '  updates executed');
    writeln(output, ' ', total.cracked: 6, '  cracking completed');
    writeln(output, ' ', total.cycles: 6, '  cycles and still counting!');
    end {headers};


procedure initialise_globals;

  begin
    rewrite(debug_o, 'UNIT=7');
    rewrite(trace_o, 'UNIT=8');
    rewrite(oscil_o, 'UNIT=10');
    rewrite(sercom, 'UNIT=11');
    rewrite(cycmd_i, 'FILE=-sass.cmd');
    new(sdl);
    cdp := sdl;
    quit := false;
    rf_first := true;
    screen := false;
    cy_first := true;
    with gi, plspace, opt, total, opt.meshtbs, sdl@ do begin
      damp := 0;
      mass := 0;
      cohes := 0;
      phi := 0;
      rho := 0;
      rad := 0;
      kn := 0;
      preincarnate := ord(sdl);
```

```
flagno := 0;
typ := free;
next_data := sdl;
reptend := false;
heading := nilhed;
nextword := '';
settling := false;
cmdend := false;
tfrac := 0;
tstep := 0;
max_rad := 0;
jumping := false;
single := false;
reareaing := false;
motioning := false;
updating := false;
cycling := false;
fording := false;
oscing := false;
tracing := false;
consoling := false;
charsize := 0.0;
debecho := false;
xmin := 0;
xmax := 0;
ymin := 0;
ymax := 0;
plot_space := plspace;
plot_lims := plspace;
force_map := plspace;
this_area := nil;
area := nil;
sal := nil;
spare_area := nil;
re_area_list := nil;
area_i.size := nilc;
area_i.xmax := 0;
area_i.ymax := 0;
grav := nilv;
cyclegap := 100;
cycle_interval := maxcycle;
cmdprocessing := false;
echo1 := true;
echo := true;
rf_over := true;
yb := 0;
yt := 0;
xb := 0;
xt := 0;
circles := 0;
fixed := 0;
cracked := 0;
cons := 0;
cycles := 0;
updates := 0;
pics := 0;
pages := 0;
datatypes := 0;
sum.en := 0.0;
sum.sc := 1E70;
```

```
    sum.scold := 0;
    end;
  end {initialise_globals};


function no_cols(xcoord: real): integer;

 begin
  if gi.tracing
   then
    writeln(trace_o, 'Entered procedure NO_COLS');
  no_cols := trunc(xcoord / area_i.size.xc);
  if gi.tracing
   then
    writeln(trace_o, ' EXIT procedure NO_COLS');
  end {no_cols};


function no_rows(ycoord: real): integer;

 begin
  if gi.tracing
   then
    writeln(trace_o, 'Entered procedure NO_ROWS');
  no_rows := trunc(ycoord / area_i.size.yc);
  if gi.tracing
   then
    writeln(trace_o, ' EXIT procedure NO_ROWS');
  end {no_rows};


function shift_area(el: area_ptr; num: integer; dir: area_directions): area_ptr;

 var
  shifts: 0 .. 100;

 begin
  if gi.tracing
   then
    writeln(trace_o, 'Entered procedure SHIFT_AREA');
  if num < 0
   then begin
    num := - num;
{area_directions = (self,n,ne,e,se,s,sw,w,nw,nex);}
    case dir of
     n .. se:
      dir := area_directions(ord(dir) + 4);
     s .. nw:
      dir := area_directions(ord(dir) - 4);
     otherwise;
     end;
    end;

  for shifts := num downto 1 do begin
   if el ~ = nil
    then begin
     if gi.reareaing
      then
       writeln(debug_o, 'Area ', el@.col: 6, el@.row: 6, ' ', num: 6);
```

```
        case dir of
          n:
            el := el@.n;
          ne:
            el := shift_area(el@.e, 1, n);
          e:
            el := el@.e;
          se:
            el := shift_area(el@.e, 1, s);
          s:
            el := el@.s;
          sw:
            el := shift_area(el@.w, 1, s);
          w:
            el := el@.w;
          nw:
            el := shift_area(el@.w, 1, n);
          self:;
          nex:
            el := el@.next_area;
          otherwise
            error_simple('illegal direction specification', 'shift_area');
          end;
        end;
    end;

  shift_area := el;
  if (gi.reareaing) AND (el ~ = nil)
    then
      writeln(debug_o, 'Area ', el@.col: 6, el@.row: 6);
  if gi.tracing
    then
      writeln(trace_o, ' EXIT procedure SHIFT_AREA');
  end {shift_area};


function sign(val, donor: real): real;

 begin
  if donor ~ = 0
    then
      sign := abs(val * donor) / donor
    else
      sign := val;
  end {sign};


function max(a, b, c: real): real;

 var
  v: real;

 begin
  v := abs(c);
  if abs(a) > v
    then
      v := abs(a);
  if abs(b) > v
    then
```

```
      v := abs(b);
   max := v;
   end ;
{**************************************** END GLOBALS }


procedure do_this(procedure proc_name(arg: ptr_type); curr_area: area_ptr;
{does for all elements} single: boolean; lists: el_list_types);

  begin
   if gi.tracing
    then
      writeln(trace_o, 'Entered procedure DO_THIS');

   while curr_area ~ = nil do
    with curr_area@ do begin
      this_area := curr_area;
      case lists of
       free:
        if free_list ~ = nil
         then
          proc_name(free_list);
       fixed:
        if fixed_list ~ = nil
         then
          proc_name(fixed_list);
       both: begin
        if free_list ~ = nil
         then
          proc_name(free_list);
        if fixed_list ~ = nil
         then
          proc_name(fixed_list);
        end;
       end;
      if ~ single
       then
        curr_area := next_area
       else
        curr_area := nil;
      end;
   if gi.tracing
    then
      writeln(trace_o, '  EXIT procedure DO_THIS');
   end {do_this};
{**************************************** BEGIN PLOTS }


procedure plots(var cmd_i: text; plot_command: string(12));

  var
   plotcom: com_type;
   plotquit, writing: boolean;
   plot_scale: real;


  procedure map_space(var cmd_i: text; sp_comst: string(12));

   const
```

```
    paph_space = grid_type(0.06, 0.96, 0.05, 0.65);
    papv_space = grid_type(0.16, 0.74, 0.14, 0.93);

  var
    eightht, htratio: real;
    sp_com: com_type;
    map_sp, plt_sp: grid_type;
    mapquit: boolean;

  begin
    if gi.tracing
      then
        writeln(trace_o, 'Entered procedure MAP_SPACE');

    repeat
      get_command(mapper, mapquit, sp_com, sp_comst, cmd_i);
      with plt_sp do begin
        plt_sp := plot_space;
        eightht := (ymax - ymin) / 6;
        case sp_com of
          horiz: begin
            plot_space := paph_space;
            plt_sp := plot_space;
            eightht := (ymax - ymin) / 6;
            end;
          vertic: begin
            plot_space := papv_space;
            plt_sp := plot_space;
            eightht := (ymax - ymin) / 6;
            end;
          fnosc, whole, zoom:;
          otherwise;
          end;
        pspace(xmin, xmax, ymin, ymax);
        if ymax - ymin < 1E-20
          then
            htratio := 1
          else
            htratio := (xmax - xmin) / (ymax - ymin);
        end;

      with map_sp do begin
        map_sp := plspace;
        case sp_com of
          piccie, horiz, vertic, whole, fnosc:
            ymax := ymin + (xmax - xmin) / htratio;
          zoom:
            with plspace do begin
              if screen
                then
                  writeln(output, substr(pos_str, 1, pro_pos),
                      ' Enter xmin, xmax, and ymin ...');
              read(cmd_i, xmin, xmax, ymin);
              ymax := ymin + (xmax - xmin) / htratio;
              map_sp := plspace;
              end;
          otherwise;
          end;
        gi.charsize := 0.012 * (ymax - ymin);
        ctrsiz(gi.charsize);
```

```
      map(xmin, xmax, ymin, ymax);
      end;
    if sp_com ~ = fnosc
      then
        scales;
    border;
    until mapquit;
  if gi.tracing
    then
      writeln(trace_o, '  EXIT procedure MAP_SPACE');
  end {map_space};


procedure setup_plot;
{   sets up  plotting parameters }
{   suitable for a4 size paper / laser printer}
{   called from either start or restar }
{   end of line }

  begin
  if gi.tracing
    then
      writeln(trace_o, 'Entered procedure SETUP_PLOT');
  paper(1);
  cspace(0.00, 1.00, 0.00, 0.80);
  pspace(0.06, 0.96, 0.05, 0.65);
  map_space(cmd_i, 'vertical');
  map_space(cmd_i, 'zoom');
  blkpen;
  if gi.tracing
    then
      writeln(trace_o, '  EXIT procedure SETUP_PLOT');
  end {setup_plot};


procedure grid_plot;
{   procedure to plot area called from plot end of line }

  var
   i, j, lines: 1 .. 100;
   xm, ym, x, y: real;

  begin
  if gi.tracing
    then
      writeln(trace_o, 'Entered procedure GRID_PLOT');
  xm := plspace.xmax;
  ym := plspace.ymax;
{   draw vertical lines }
  x := 0.0;
  y := 0.0;
  lines := area_i.xmax + 1;
  for i := 1 to lines do begin
    positn(x, y);
    join(x, ym);
    x := x + area_i.size.xc;
    end;
{   draw horizontal lines }
```

```
    x := 0.0;
    y := 0.0;
    lines := area_i.ymax + 1;
    for j := 1 to lines do begin
      positn(x, y);
      join(xm, y);
      y := y + area_i.size.yc;
      end;
    if gi.tracing
      then
        writeln(trace_o, '  EXIT procedure GRID_PLOT');
    end {grid_plot};


  procedure fram_plot;
{   *** sets up plotting frames ***}
{   suitable for a4 size paper / laser printer }
{   called from either stplot or plot }
{   may call grid_plot if areaing set }

   var
    time, yline: real;

   begin
    if gi.tracing
      then
        writeln(trace_o, 'Entered procedure FRAM_PLOT');
    with plspace do begin
      map_space(cmd_i, 'fullnoscales');
      time := gi.tstep * total.cycles;
      gi.charsize := 0.02;
      ctrsiz(gi.charsize);
      yline := ymax - 4 * gi.charsize;
      undlin(1);
      italic(1);
      plotcs(3 * gi.charsize, yline, gi.heading, 80);
      pcsend(xmax - 11 * gi.charsize, yline, 'TIME  ', 6);
      plotne(xmax - 9 * gi.charsize, yline, time, 4);
      italic(0);
      undlin(0);
      if gi.reareaing
        then begin
          grid_plot;
          end;
      writeln(debug_o);
      end;
    if gi.tracing
      then
        writeln(trace_o, '  EXIT procedure FRAM_PLOT');
    end {fram_plot};


  procedure circle_plot(el: ptr_type);
{ plot a snapshot of the geometry}

   begin
    if gi.tracing
      then
```

```
      writeln(trace_o, 'Entered procedure circle_PLOT');
    while el ~ = nil do
      with el@, data@ do begin
        gpoint(posn.xc, posn.yc);
        circle(rad);
        el := next;
        end;
    if gi.tracing
      then
        writeln(trace_o, ' EXIT procedure circle_PLOT');
    end {circle_plot};


 procedure dot_plot(el: ptr_type);

  begin
    if gi.tracing
      then
        writeln(trace_o, 'Entered procedure DOT_PLOT');
    while el ~ = nil do
      with el@ do begin
        gpoint(posn.xc, posn.yc);
        el := next;
        end;
    if gi.tracing
      then
        writeln(trace_o, ' EXIT procedure DOT_PLOT');
    end {dot_plot};


 procedure prof_plot;

  begin
    if gi.tracing
      then
        writeln(trace_o, 'Entered procedure PROF_PLOT');
    if gi.tracing
      then
        writeln(trace_o, ' EXIT procedure PROF_PLOT');
    end {prof_plot};


 procedure arrow(x, y, vx, vy, scale: real);
{ this routine plots an arrow
 x   - x - coordinate of arrow centre  y   - y - coordinate of arrow centre
 vx  -  x component of vector          vy  -  y component of vector      }

  var
    rdenom, alen, sina, cosa, ahlen, ahwid, alend2, alen2x, alen2y, ahwidx,
      ahwidy, xtip, ytip, xtipmh, ytipmh: real;

  begin
    alen := scale * sqrt(sqr(vx) + sqr(vy));
    if alen > 1.0E-50
      then begin
        ahlen := 0.15 * alen;
        ahwid := 0.04 * alen;

{   find angle shaft makes with horizontal direction}
```

```
        rdenom := scale / alen;
        sina := vy * rdenom;
        cosa := vx * rdenom;
        if abs(vy * 0.00001) - abs(vx) > 0
          then begin
            sina := sign(1.0, vy);
            cosa := 0.0;
            end;
        alend2 := 0.5 * alen;
        alen2x := alend2 * cosa;
        alen2y := alend2 * sina;
        ahwidx := ahwid * sina;
        ahwidy := ahwid * cosa;
        xtip := x + alen2x;
        ytip := y + alen2y;
        xtipmh := xtip - ahlen * cosa;
        ytipmh := ytip - ahlen * sina;

{   plot arrow starting at tail}

        positn(x - alen2x, y - alen2y);
        join(xtip, ytip);
        join(xtipmh - ahwidx, ytipmh + ahwidy);
        join(xtipmh + ahwidx, ytipmh - ahwidy);
        join(xtip, ytip);
        positn(x - ahwidx, y + ahwidy);
        join(x + ahwidx, y - ahwidy);
        end;
    end {arrow};


 procedure find_scale(el: ptr_type);

   var
     con_node: con_ptr;

   begin
     if gi.tracing
       then
         writeln(trace_o, 'Entered procedure FIND_SCALE');
     case plotcom of
       conplot:
         while el ~ = nil do
           with el@ do begin
             con_node := con_list;
             while con_node ~ = nil do
               with con_node@ do begin
                 if abs(c_force.x) > plot_scale
                   then
                     plot_scale := abs(c_force.x);
                 if abs(c_force.y) > plot_scale
                   then
                     plot_scale := abs(c_force.y);
                 con_node := next_con;
                 end;
             el := next;
             end;
       failplot:
         while el ~ = nil do
```

```
        with el@ do begin
          con_node := con_list;
          while con_node ~ = nil do
            with con_node@ do begin
              if (failed) AND (abs(f_force) > plot_scale)
                then
                  plot_scale := abs(f_force);
              con_node := next_con;
              end;
          el := next;
          end;
      velplot:
        while el ~ = nil do
          with el@, v do begin
            if abs(x) > plot_scale
              then
                plot_scale := abs(x);
            if abs(y) > plot_scale
              then
                plot_scale := abs(y);
            el := next;
            end;
      end;
    if gi.tracing
      then
        writeln(trace_o, '  EXIT procedure FIND_SCALE');
    end {find_scale};


  procedure vel_plot(el: ptr_type);
{   plot the velocities of the circles, called from plot, end of line }

  begin
    if gi.tracing
      then
        writeln(trace_o, 'Entered procedure VEL_PLOT');
    while el ~ = nil do
      with el@, posn, v do begin
        arrow(xc, yc, x, 0, plot_scale);
        arrow(xc, yc, 0, y, plot_scale);
        arrow(xc, yc, x, y, plot_scale);
        el := next;
        end;
    if gi.tracing
      then
        writeln(trace_o, '  EXIT procedure VEL_PLOT');
    end {vel_plot};


  procedure disp_plot(el: ptr_type);
{   plot of displacements, called from plot, end of line }

  begin
    if gi.tracing
      then
        writeln(trace_o, 'Entered procedure DISP_PLOT');
    while el ~ = nil do
      with el@ do begin
        with source do
```

```
        gpoint(xc, yc);
      with posn do
        join(xc, yc);
      source := posn;
      el := next;
      end;
  if gi.tracing
    then
      writeln(trace_o, '  EXIT procedure DISP_PLOT');
  end {disp_plot};


procedure cont_plot(el: ptr_type);

  var
    midx, midy: real;
    con_node: con_ptr;

  begin
    if gi.tracing
      then
        writeln(trace_o, 'Entered procedure CONT_PLOT');
    while el ~ = nil do
      with el@, el@.posn do begin
        con_node := con_list;
        while con_node ~ = nil do
          with con_node@ do begin
            positn(xc, yc);
            join(other@.posn.xc, other@.posn.yc);
            midx := (xc + other@.posn.xc) / 2;
            midy := (yc + other@.posn.yc) / 2;
            arrow(midx, midy, c_force.x, 0, plot_scale);
            arrow(midx, midy, 0, c_force.y, plot_scale);
            arrow(midx, midy, c_force.x, c_force.y, plot_scale);
            con_node := next_con;
            end;
        el := next;
        end;
    if gi.tracing
      then
        writeln(trace_o, '  EXIT procedure CONT_PLOT');
    end {cont_plot};


procedure draw_split(x, y, theta, scale: real);

  var
    xlen, ylen: real;

  begin
    ylen := scale * cos(theta);
    xlen := scale * sin(theta);
    positn(x + 2 * xlen, y - 2 * ylen);
    join(x - 2 * xlen, y + 2 * ylen);
    positn(x + xlen, y + ylen);
    join(x - xlen, y - ylen)
    end {draw_split};
```

```
procedure fail_plot(el: ptr_type);

  var
    midx, midy: real;
    con_node: con_ptr;

  begin
    if gi.tracing
      then
        writeln(trace_o, 'Entered procedure FAIL_PLOT');
    while el ~ = nil do
      with el@ do begin
        con_node := con_list;
        while con_node ~ = nil do
          with con_node@ do begin
            draw_split((posn.xc + other@.posn.xc) / 2, (posn.yc + other@.posn.
                yc) / 2, f_angle, f_force * plot_scale);
            con_node := next_con;
            end;
        el := next;
        end;
    if gi.tracing
      then
        writeln(trace_o, '  EXIT procedure FAIL_PLOT');
    end {fail_plot};


begin {plots}
  if gi.tracing
    then
      writeln(trace_o, 'Entered procedure PLOTS');
  writing := opt.echo;
  repeat
    get_command(plotter, plotquit, plotcom, plot_command, cmd_i);
    if plotcom IN ballplot .. failplot
      then
        total.pics := total.pics + 1;
    case plotcom of
      ballplot:
        do_this(circle_plot, sal, false, both);
      dotplot:
        do_this(dot_plot, sal, false, both);
      velplot: begin
        plot_scale := 0;
        do_this(find_scale, sal, false, free);
        if plot_scale ~ = 0
          then begin
            plot_scale := gi.max_rad / (total.datatypes * plot_scale);
            do_this(vel_plot, sal, false, free);
            end
          else
            writeln(output, substr(pos_str, 1, mes_pos),
                ' Warning : all velocities zero     ');
        end;
      displot:
        do_this(disp_plot, sal, false, free);
      conplot: begin
        plot_scale := 0;
        do_this(find_scale, sal, false, free);
```

```
        if plot_scale ~ = 0
          then begin
            plot_scale := gi.max_rad / (total.datatypes * plot_scale);
            do_this(cont_plot, sal, false, free);
            end
          else
            writeln(output, substr(pos_str, 1, mes_pos),
                ' Warning : all contact forces zero  ');
        end;
      failplot: begin
        plot_scale := 0;
        do_this(find_scale, sal, false, free);
        if plot_scale ~ = 0
          then begin
            plot_scale := gi.max_rad / (2 * total.datatypes * plot_scale);
            do_this(fail_plot, sal, false, free);
            end
          else
            writeln(output, substr(pos_str, 1, mes_pos),
                ' Warning : no failures : no plot    ');
        end;
      standard: begin
        ;
        do_this(circle_plot, sal, false, both);
        total.pics := total.pics + 1;
        fram_plot;
        frame;
        total.pages := total.pages + 1;
        end;
      frames:
        fram_plot;
      page: begin
        frame;
        total.pages := total.pages + 1;
        end;
      graticule:
        grid_plot;
      init: begin
        setup_plot;
        fram_plot;
        grid_plot;
        writing := false;
        end;
      plotstop: begin
        map_space(cmd_i, 'full');
        fram_plot;
        grid_plot;
        do_this(circle_plot, sal, false, both);
        grend;
        total.pics := total.pics + 1;
        writing := false;
        total.pages := total.pages + 1;
        end;
      zoom:
        map_space(cmd_i, 'zoom');
      maps:
        map_space(cmd_i, '');
      otherwise;
      end;
    if writing AND ~ cy_first
```

```
      then begin
        if opt.echo
          then
            writeln(output, substr(pos_str, 1, fra_pos), total.pages: 8,
                substr(pos_str, 1, plo_pos), total.pics: 8);
        end;
    until plotquit;
  if gi.tracing
    then
      writeln(trace_o, '  EXIT procedure PLOTS');
  end {plots};
{***************************************** END PLOTS }
{***************************************** BEGIN UPDATE }


procedure update_area(el: ptr_type);

  var
    con_lim, con_res: real;


  procedure update_el(el: ptr_type);

    var
      sibling: ptr_type;
      forf: el_list_types;
      dir_lim, sweep: area_directions;
      centre: area_ptr;
      gap: real;
      offset: real;


    procedure update_brain(elem, twin: ptr_type);

      var
        found: boolean;


      procedure destroy_contact(owner: ptr_type; pre_victim: con_ptr);

        var
          victim: con_ptr;

        begin
          if gi.tracing
            then
              writeln(trace_o, 'Entered procedure DESTROY_CONTACT');
          if pre_victim = nil
            then begin
              victim := owner@.con_list;
              owner@.con_list := owner@.con_list@.next_con;
              end
            else begin
              victim := pre_victim@.next_con;
              pre_victim@.next_con := victim@.next_con;
              end;
          dispose(victim);
          total.cons := total.cons - 1;
          if gi.updating
```

```
      then
        writeln(debug_o, '   Victim destroyed');
    if gi.tracing
      then
        writeln(trace_o, ' EXIT procedure DESTROY_CONTACT');
    end {destroy_contact};


procedure scan_con(home_el, away_el: ptr_type);

  var
    home_con, prev_con: con_ptr;

  begin
    if gi.tracing
      then
        writeln(trace_o, 'Entered procedure SCAN_CON');
    home_con := home_el@.con_list;
    prev_con := nil;
    found := false;
    while (NOT found) AND (home_con ~ = nil) do begin
      found := home_con@.other = away_el;
      if NOT found
        then begin
          prev_con := home_con;
          home_con := home_con@.next_con;
          end;
      end;
    if (found) AND (gap >= con_lim)
      then
        destroy_contact(home_el, prev_con);

    if gi.tracing
      then
        writeln(trace_o, ' EXIT procedure SCAN_CON');
    end {scan_con};


procedure create_contact(domicus, vagrantus: ptr_type);

  var
    newcon: con_ptr;

  begin
    if gi.tracing
      then
        writeln(trace_o, 'Entered procedure CREATE_CONTACT');
    new(newcon);
    with newcon@ do begin
      next_con := domicus@.con_list;
      domicus@.con_list := newcon;
      other := vagrantus;
      if gap > 0
        then {tensional}
          offs := domicus@.data@.rad + vagrantus@.data@.rad
        else {overlapping}
          offs := offset;
      gapsum := 0;
      c_force := nilv;
```

```
          f_force := 0;
          f_angle := 0;
          failed := false;
          end;
        total.cons := total.cons + 1;
        if gi.updating
          then
            writeln(debug_o, '   Contact created', gap: 6, domicus@.posn.xc:
                6, domicus@.posn.yc: 6, vagrantus@.posn.xc: 6, vagrantus@.posn
                .yc: 6);
        if gi.tracing
          then
            writeln(trace_o, ' EXIT procedure CREATE_CONTACT');
        end {create_contact};


    begin {update_brain}
      if gi.tracing
        then
          writeln(trace_o, 'Entered procedure UPDATE_BRAIN');

      scan_con(elem, twin);
      if NOT found
        then
          scan_con(twin, elem);
      if (NOT found) AND (gap < con_lim)
        then
          create_contact(elem, twin);
      if gi.tracing
        then
          writeln(trace_o, ' EXIT procedure UPDATE_BRAIN');
      end {update_brain};


  function central(elem: ptr_type): boolean;

    begin
      if gi.tracing
        then
          writeln(trace_o, 'Entered procedure CENTRAL');
      with centre@.corners do
        central := min(elem@.posn.yc - ymin, ymax - elem@.posn.yc, elem@.posn.
            xc - xmin, xmax - elem@.posn.xc) > 2 * gi.max_rad + con_res;
      if gi.tracing
        then
          writeln(trace_o, ' EXIT procedure CENTRAL');
      end {central};


  begin
{ it is fortunate that two fixed blocks are not allowed to have contacts}
    if gi.tracing
      then
        writeln(trace_o, 'Entered procedure UPDATE_EL');

    with el@ do begin
      centre := this_area;
      if central(el)
        then
```

```
          dir_lim := self
        else
          dir_lim := nw;
      for sweep := self to dir_lim do begin
        area := shift_area(centre, 1, sweep);
        if area = nil
          then
            continue;
        if sweep = self
          then
            sibling := el@.next
          else
            sibling := area@.free_list;
        for forf := free to fixed do begin
          while (sibling ~ = nil) do begin
            offset := sqrt(sqr(sibling@.posn.xc - posn.xc) + sqr(sibling@.posn
                .yc - posn.yc));
            gap := offset - (data@.rad + sibling@.data@.rad);
            if gap < con_res
              then
                update_brain(el, sibling);
            sibling := sibling@.next;
            end;
          sibling := area@.fixed_list;
          end;
        end;
      end;
    if gi.tracing
      then
        writeln(trace_o, '  EXIT procedure UPDATE_EL');
    end {update_el};


begin {update_area}
  if gi.tracing
    then
      writeln(trace_o, 'Entered procedure UPDATE_AREA');
  if this_area = spare_area
    then
      return;
  with this_area@ do begin
    upd_min := 0.5 * gi.max_rad / total.datatypes;
    con_lim := upd_min;       { gap > con_lim not a contact}
    con_res := 2.1 * con_lim; { gap < con_res check lists}
    upd_par := 0;
    end;
  while el ~ = nil do begin
    update_el(el);
    el := el@.next;
    if gi.updating
      then
        writeln(debug_o, 'total number of contacts', total.cons);
    end;
  total.updates := total.updates + 1;
  if gi.tracing
    then
      writeln(trace_o, '  EXIT procedure UPDATE_AREA');
  end {update_area};
{****************************************  END UPDATE  }
```

```
{****************************************  BEGIN RE_AREA  }


procedure re_area;

  var
    el: ptr_type;

  begin
    if gi.tracing
      then
        writeln(trace_o, 'Entered procedure RE_AREA');

    while re_area_list ~ = nil do
      with re_area_list@ do begin
        if gi.reareaing
          then
            writeln(debug_o, 'ori x,y, new x,y ', source.xc: 6, source.yc: 6,
                posn.xc: 6, posn.yc: 6);

        area := shift_area(shift_area(sal, no_cols(posn.xc), e), no_rows(posn.yc
            ), n);
        if area = nil
          then begin
            area := spare_area;
            total.circles := total.circles - 1;
            writeln(output, substr(pos_str, 1, mes_pos),
                ' Warning : circles leaving area ', el@.posn.xc: 8, ' ', el@.
                posn.yc: 8);
            end;
        el := re_area_list;
        re_area_list := el@.next;
        el@.next := area@.free_list;
        area@.free_list := el;
        end;
    if total.circles = 0
      then
        start_shut(input, shutdown);
    if gi.tracing
      then
        writeln(trace_o, ' EXIT procedure RE_AREA');
    end {re_area};
{****************************************  BEGIN CYCLE  }


procedure cycle(var cmd_i: text);
{   driver for iterations }
{   the calculation sequence module }
{   called from contrl }
{   may call ford, motion, updat and stop }
{   called via motion }

  var
    cycles, no_of_cycles, outcounter, cycle_lim: cycle_type;
    max_adisp, min_adisp: real;
```

```
procedure hide_el(var el: ptr_type);

  var
    elem, prev: ptr_type;

  begin
    if gi.tracing
      then
        writeln(trace_o, 'Entered procedure HIDE_EL');
    if this_area = spare_area
      then
        return;
    with this_area@ do begin
      elem := free_list;
      prev := nil;
      while el ~ = elem do begin
        prev := elem;
        elem := elem@.next;
        end;
      el := el@.next;
      if prev = nil
        then
          free_list := el
        else
          prev@.next := el;
      elem@.next := re_area_list;
      re_area_list := elem;
      end;
    if gi.tracing
      then
        writeln(trace_o, '  EXIT procedure HIDE_EL');
    end {hide_el};


procedure clear_forces(el: ptr_type);
{ set all forces to zero }

  begin
    if gi.tracing
      then
        writeln(trace_o, 'Entered procedure CLEAR_FORCES');
    while el ~ = nil do
      with el@ do begin
        force := nilv;
        el := next;
        end;
    if gi.tracing
      then
        writeln(trace_o, '  EXIT procedure CLEAR_FORCES');
    end {clear_forces};


procedure fordmot(el: ptr_type);

  var
    con_node: con_ptr;
    sine, cose, gap, dx, dy, con_force: real;
    stress, t_force: vector_type;
    n, s1, s3, si, sn, st: real;
```

```
begin
  if gi.tracing
    then
      writeln(trace_o, 'Entered procedure FORD');

  while el ~ = nil do
    with el@, el@.data@ do begin
      con_node := con_list;
      if gi.fording
        then
          writeln(debug_o,
              'deltagap   con_force  force  t_force  for x then y');
      while con_node ~ = nil do
        with con_node@ do begin
          dx := other@.posn.xc - posn.xc + (other@.s.x - s.x);
          dy := other@.posn.yc - posn.yc + (other@.s.y - s.y);
          gap := sqrt(sqr(dx) + sqr(dy));
          sine := dy / gap;
          cose := dx / gap;
          if gi.fording
            then
              write(debug_o, gap, sine, cose);
          gap := gap - offs;
          if gi.fording
            then
              writeln(debug_o, gap);
          if gap < this_area@.upd_min
            then begin
              con_force := kn * gap * damp;
              t_force.x := c_force.x + con_force * cose;
              t_force.y := c_force.y + con_force * sine;
              if gap * kn + gapsum > 0
                then begin
                  if gap * kn + gapsum > tens_fuzz
                    then begin
                      f_force := (t_force.x + t_force.y) / 2;
                      if abs(sine) < 1E-40
                        then
                          sine := 1e-40;
                      f_angle := arctan(- cose / sine);
                      t_force := nilv;
                      failed := true;
                      end;
                  end
                else begin
                  stress.x := abs(t_force.x / rad);
                  stress.y := abs(t_force.y / rad);
                  if stress.x < stress.y
                    then begin
                      s3 := stress.x;
                      si := stress.y;
                      end
                    else begin
                      s3 := stress.y;
                      si := stress.x;
                      end;
                  if phi = 0
                    then begin
                      s1 := s3 + 2 * cohes;
```

```
              sn := 1E7;
              st := cohes;
              end
            else begin
              n := sqrt(1 + sqr(phi));
              n := (1 + phi - n) / (phi - 1 + n);
              sn := (s3 + n * cohes) / (1 - n * phi);
              st := phi * sn + cohes;
              s1 := 2 * (sn + st * phi) - s3;
              end;
          if si > s1
            then begin
              if stress.x < stress.y
                then
                  t_force.y := sign(s1 * rad, t_force.y)
                else
                  t_force.x := sign(s1 * rad, t_force.x);
              if ~ failed
                then begin
                  failed := true;
                  f_force := st;
                  if abs(sn - s3) > 1e-20
                    then
                      f_angle := arctan(st / (sn - s3))
                    else
                      f_angle := 0;
                  total.cracked := total.cracked + 1;
                  if opt.echo
                    then
                      writeln(output, substr(pos_str, 1, cra_pos),
                          total.cracked: 8, substr(pos_str, 1,
                          cyc_pos), total.cycles: 8, substr(pos_str,
                          1, mes_pos), ' Sphere cracked at ', posn.
                          xc: 8: 3, posn.yc: 8: 3, stress.x: 8: 3,
                          stress.y: 8: 3);
                  end;
              end;
          other@.force.x := other@.force.x - t_force.x;
          other@.force.y := other@.force.y - t_force.y;
          force.x := force.x + t_force.x;
          force.y := force.y + t_force.y;
          if gi.fording
            then
              writeln(debug_o, con_force: 8, dx: 8, force.x: 8,
                  other@.force.x: 8, dy: 8, force.y: 8, other@.force
                  .y: 8);
          if gi.consoling
            then
              writeln(debug_o, t_force.x, t_force.y);
          end;
        end;
      con_node := next_con;
      end;
    el := next;
    end;
  if gi.tracing
    then
      writeln(trace_o, ' EXIT procedure FORD');
  end {fordmot};
```

```
procedure fordcon(el: ptr_type);

  var
    con_node: con_ptr;
    sine, cose, gap, dx, dy, con_force: real;
    stress, t_force: vector_type;
    n, s1, s3, si, sn, st: real;

  begin
    if gi.tracing
      then
        writeln(trace_o, 'Entered procedure FORDCON');

    while el ~ = nil do
      with el@, el@.data@ do begin
        con_node := con_list;
        if gi.fording
          then
            writeln(debug_o,
                'deltagap   con_force  force  t_force  for x then y');
        while con_node ~ = nil do
          with con_node@ do begin
            dx := other@.posn.xc - posn.xc;
            dy := other@.posn.yc - posn.yc;
            gap := sqrt(sqr(dx) + sqr(dy));
            sine := dy / gap;
            cose := dx / gap;
            dx := dx + (other@.s.x - s.x);
            dy := dy + (other@.s.y - s.y);
            gap := sqrt(sqr(dx) + sqr(dy));
            if gi.fording
              then
                write(debug_o, gap, sine, cose);
            gap := gap - offs;
            if gi.fording
              then
                writeln(debug_o, gap);
            if gap < this_area@.upd_min
              then begin
                gapsum := gapsum + gap;
                con_force := gap * damp;
                t_force.x := c_force.x + con_force * cose;
                t_force.y := c_force.y + con_force * sine;
                if gapsum > 0
                  then begin
                    if gapsum > cohes
                      then begin
                        f_force := (t_force.x + t_force.y) / 2;
                        if abs(sine) < 1E-40
                          then
                            sine := 1e-40;
                        f_angle := arctan(- cose / sine);
                        t_force := nilv;
                        failed := true;
                        end;
                    end
                  else begin
                    stress.x := abs(t_force.x / rad);
```

```
            stress.y := abs(t_force.y / rad);
            if stress.x < stress.y
              then begin
                s3 := stress.x;
                si := stress.y;
                end
              else begin
                s3 := stress.y;
                si := stress.x;
                end;
            if phi = 0
              then begin
                s1 := s3 + 2 * cohes;
                sn := 1E7;
                st := cohes;
                end
              else begin
                n := sqrt(1 + sqr(phi));
                n := (1 + phi - n) / (phi - 1 + n);
                sn := (s3 + n * cohes) / (1 - n * phi);
                st := phi * sn + cohes;
                s1 := 2 * (sn + st * phi) - s3;
                end;
            if si > s1
              then begin
                if stress.x < stress.y
                  then
                    t_force.y := sign(s1 * rad, t_force.y)
                  else
                    t_force.x := sign(s1 * rad, t_force.x);
                if ~ failed
                  then begin
                    failed := true;
                    f_force := st;
                    if abs(sn - s3) > 1e-20
                      then
                        f_angle := arctan(st / (sn - s3))
                      else
                        f_angle := 0;
                    total.cracked := total.cracked + 1;
                    end;
                end;
            end;
        c_force := t_force;
        other@.force.x := other@.force.x - c_force.x;
        other@.force.y := other@.force.y - c_force.y;
        force.x := force.x + c_force.x;
        force.y := force.y + c_force.y;
        if gi.fording
          then
            writeln(debug_o, con_force: 8, dx: 8, force.x: 8, other@.
                force.x: 8, dy: 8, force.y: 8, other@.force.y: 8);
        if gi.consoling
          then
            writeln(debug_o, c_force.x, c_force.y);
        end;
    con_node := next_con;
    end;
el := next;
end;
```

```
    if gi.tracing
      then
        writeln(trace_o, '  EXIT procedure FORDCON');
    end {fordcon};


 procedure premotion(el: ptr_type);

   var
     con_node: con_ptr;

   begin
     if gi.tracing
       then
         writeln(trace_o, 'Entered procedure PREMOTION');
     while el ~ = nil do
       with el@ do begin
         con_node := con_list;
         while con_node ~ = nil do
           with con_node@ do begin
             if other@.data@.typ = fixed
               then
                 no_of_contacts := no_of_contacts + 1
               else
                 if (abs(other@.s.y - s.y) > 0.0005) AND (abs(posn.yc - other@.
                     posn.yc) > 0.1)
                   then begin
                     no_of_contacts := no_of_contacts + 1;
                     other@.no_of_contacts := other@.no_of_contacts + 1;
                     end;
             con_node := next_con;
             end;
         el := next;
         end;
     if gi.tracing
       then
         writeln(trace_o, '  EXIT procedure PREMOTION');
     end {premotion};


 procedure fconsolxy(el: ptr_type);

   begin
     if gi.tracing
       then
         writeln(trace_o, 'Entered procedure FCONSOLXY');
     while el ~ = nil do
       with el@, el@.force, el@.data@, opt, gi do begin
         if no_of_contacts = 0
           then
             no_of_contacts := 1;
         s.x := (x / mass + grav.x) * sqr(gi.tstep) {/  no_of_contacts};
         s.y := (y / mass + grav.y) * sqr(gi.tstep) {/  no_of_contacts};
         sum.sc := max(s.x, s.y, sum.sc);
         if motioning
           then
             writeln(debug_o, flagno: 3, ' f ', x, y, ' s ', s.x, s.y);
         no_of_contacts := 0;
         force := nilv;
```

```
      el := next;
      end;
    if gi.tracing
      then
        writeln(trace_o, '  EXIT procedure FCONSOLXY');
    end {fconsolxy};


  procedure motionxy(el: ptr_type);

    var
      max_disp: real;

    begin
      if gi.tracing
        then
          writeln(trace_o, 'Entered procedure MOTION');

      max_disp := 0;
      while el ~ = nil do
        with el@ do begin
          posn.xc := posn.xc + s.x;
          posn.yc := posn.yc + s.y;
          a.x := force.x / data@.mass + opt.grav.x;
          a.y := force.y / data@.mass + opt.grav.y;
          v.x := v.x + a.x * gi.tstep;
          v.y := v.y + a.y * gi.tstep;
          s.x := v.x * gi.tstep;
          s.y := v.y * gi.tstep;
          sum.sc := max(s.x, s.y, sum.sc);
          max_disp := max(max_disp, s.x, s.y);
          if (gi.oscing) AND (data@.typ = track)
            then
              writeln(oscil_o, data@.flagno: 4, total.cycles: 6, force.x: 12, a.
                x: 12, v.x: 12, s.x: 12, posn.xc: 12, force.y: 12, a.y: 12, v.
                y: 12, s.y: 12, posn.yc: 12);
          if (trunc(posn.yc / area_i.size.yc) ~ = this_area@.row) OR (trunc(posn
              .xc / area_i.size.xc) ~ = this_area@.col)
            then
              hide_el(el)
            else
              el := next;  {el may be changed by hide_el}
          end;

      with this_area@ do begin
        upd_par := upd_par + max_disp;
        if upd_par > upd_min
          then begin
            if re_area_list ~ = nil
              then
                re_area;
            do_this(update_area, this_area, true, free);
            if opt.echo
              then
                writeln(output, substr(pos_str, 1, upd_pos), total.updates: 8);
            end;
        end;
      if gi.tracing
        then
```

```
      writeln(trace_o, '  EXIT procedure MOTION');
    end {motionxy};


  begin {cycle}
    if gi.tracing
      then
        writeln(trace_o, 'Entered procedure CYCLES');
    if screen
      then
        writeln(output, substr(pos_str, 1, pro_pos),
            ' Enter no of cycles required...');
    read(cmd_i, no_of_cycles);
    if opt.echo
      then
        writeln(output, substr(pos_str, 1, req_pos), no_of_cycles: 8);

    if total.circles = 0
      then begin
        writeln(output, substr(pos_str, 1, mes_pos),
            ' Warning no circles left ');
        return
        end;

    max_adisp := gi.max_rad / (200 * total.datatypes);
    min_adisp := max_adisp / 50;
    outcounter := total.cycles;
    cycles := 0;
    while (cycles < no_of_cycles) AND (~ quit) do begin
      if opt.cycle_interval < no_of_cycles - cycles
        then
          cycle_lim := total.cycles + opt.cycle_interval
        else
          cycle_lim := total.cycles + no_of_cycles - cycles;
      while (total.cycles < cycle_lim) AND (~ quit) do begin
        sum.scold := sum.sc;
        sum.sc := 0;
        if gi.settling
          then begin
            do_this(premotion, sal, false, free);
            do_this(fordcon, sal, false, free);
            do_this(fconsolxy, sal, false, free);
            end
          else begin
            do_this(fordmot, sal, false, free);
            do_this(motionxy, sal, false, free);
            if re_area_list ~ = nil
              then
                re_area;
            do_this(clear_forces, sal, false, both);
            end;
        total.cycles := total.cycles + 1;
        if (opt.echo) AND (total.cycles MOD opt.cyclegap = 0)
          then begin
            writeln(output, substr(pos_str, 1, cyc_pos), total.cycles: 8);
            if sum.scold < sum.sc
              then
                writeln(output, substr(pos_str, 1, mes_pos),
                    ' Decreasing stability ', sum.sc)
```

```
          else
            writeln(output, substr(pos_str, 1, mes_pos),
                ' Increasing stability ', sum.sc);
        end;
      if (opt.cmdprocessing) AND (total.cycles MOD opt.cycle_interval = 0)
        then begin
          reset(cycmd_i, 'FILE=-sass.cmd');
          gi.cmdend := false;
          while ~ gi.cmdend do
            control(cycmd_i);
          if opt.echo
            then
              writeln(output, substr(pos_str, 1, req_pos), no_of_cycles: 8);
          end;
      quit := (gi.settling) AND (sum.sc < 1e-14);
      if ~ gi.settling
        then begin
          if sum.sc > max_adisp
            then begin
              gi.tstep := gi.tstep / 2;
              writeln(output, substr(pos_str, 1, err_pos),
                  ' Current time step set to : ', gi.tstep: 12: 10);
              end
            else
              if sum.sc < min_adisp
                then begin
                  gi.tstep := gi.tstep * 1.05;
                  writeln(output, substr(pos_str, 1, err_pos),
                      ' Current time step set to : ', gi.tstep: 12: 10);
                  end;
          end;
      if trap
        then
          trapper;
      if gi.cycling
        then
          writeln(debug_o, 'max individual disp ', sum.sc);
      end;
    cycles := total.cycles - outcounter;
    end;
  if (quit) AND (sum.sc < 1e-14)
    then
      quit := false;
  if gi.tracing
    then
      writeln(trace_o, ' EXIT procedure CYCLES');
  end {cycle};
{**************************************** END CYCLE }


procedure start_shut;
{  initialises the run, called from control, initialisation modules }

  type
    records = (rpar, rvec, rcoo, rcon, rele, rgri, rare, rara, rgen, ropt, rtot,
          rsum, rhed);
    buffertype = record
              tag: char;
              case records of
```

```
                rpar: (parabk_rep: parabk_type);
                rvec: (vector_rep: vector_type);
                rcoo: (coord_rep: coord_type);
                rcon: (con_rep: con_type);
                rele: (element_rep: element_type);
                rgri: (grid_rep: grid_type);
                rare: (area_i_rep: area_i_type);
                rara: (area_rep: area_type);
                rgen: (gen_info_rep: gen_info_type);
                ropt: (option_rep: option_type);
                rtot: (totals_rep: totals_type);
                rsum: (sum_rep: sum_type);
                rhed: (hed_rep: hed_type);
            end;

var
  rest_o, rest_i: file of buffertype;
  buffer: buffertype;
  sd: para_ptr;
  new_circle: ptr_type;


procedure mesh_areas;


  procedure get_area(var n_a: area_ptr; cols, rows: rowcol_type);

    begin
      if gi.tracing
        then
          writeln(trace_o, 'Entered procedure GET_AREA');
      new(n_a);
      with n_a@ do begin
        corners.xmax := area_i.size.xc * cols;
        corners.xmin := corners.xmax - area_i.size.xc;
        corners.ymax := area_i.size.yc * rows;
        corners.ymin := corners.ymax - area_i.size.yc;
        row := rows - 1;
        col := cols - 1;
        upd_par := 0.0;
        upd_min := 0;
        free_list := nil;
        fixed_list := nil;
        n := nil;
        e := nil;
        s := nil;
        w := nil;
        next_area := nil;
        end;
      if gi.reareaing
        then
          with n_a@.corners do
            writeln(debug_o, 'x,x~ ,y,y~ ', xmin: 6, xmax: 6, ymin: 6, ymax);
      if gi.tracing
        then
          writeln(trace_o, ' EXIT procedure GET_AREA');
      end {get_area};

  var
```

```
    new_area: area_ptr;
    columns, layers: rowcol_type;


  begin {mesh_areas}
    if gi.tracing
      then
        writeln(trace_o, 'Entered procedure MESH_AREAS');
    get_area(area, 1, 1);
    sal := area;
    for columns := 1 to area_i.xmax do begin
      if gi.reareaing
        then
          writeln(debug_o, 'setup areas col number', columns);
      for layers := 2 to area_i.ymax do begin
        if gi.reareaing
          then
            writeln(debug_o, 'setup areas row number', layers);
        get_area(new_area, columns, layers);
        area@.next_area := new_area;
        area@.n := new_area;
        area@.n@.s := area;
        if columns = 1
          then
            area := area@.n
          else begin
            area := shift_area(area, 1, nw);
            area@.e := new_area;
            area@.e@.w := area;
            area := area@.e;
            end;
        end;

      get_area(new_area, columns + 1, 1);
      area@.next_area := new_area;
      area := shift_area(area, area_i.ymax - 1, s);
      area@.e := new_area;
      area@.e@.w := area;
      area := area@.e;
      end;

    spare_area := area;
    area := shift_area(area, 1, w);
    area@.e := nil;
    spare_area@.w := nil;
    area := shift_area(area, area_i.ymax - 1, n);
    area@.next_area := nil;
    area := sal;
    if gi.tracing
      then
        writeln(trace_o, '  EXIT procedure MESH_AREAS');
    end {mesh_areas};


 procedure setup_a_info;

  begin
    if gi.tracing
      then
```

```
      writeln(trace_o, 'Entered procedure SETUP_A_INFO');
    with area_i do begin
      if screen
        then
          writeln(output, substr(pos_str, 1, pro_pos),
              ' Enter no of areas in x and y..');
      read(cmd_i, xmax, ymax);
      size.xc := (plspace.xmax - plspace.xmin) / xmax;
      size.yc := (plspace.ymax - plspace.ymin) / ymax;
      nos := xmax * ymax;
      end;
    if gi.tracing
      then
        writeln(trace_o, '  EXIT procedure SETUP_A_INFO');
    end {setup_a_info};


  procedure mesh;
{procedure to create profiles, called from start }

    var
      oldx, oldy, cose, sine: real;
      numrep: integer;


    procedure cre_data;

      const
        sort_str = 'null free fixedtrack';

      var
        new_data: para_ptr;
        sort: string(12);
        sind: integer;

      begin
        if gi.tracing
          then
            writeln(trace_o, 'Entered procedure CRE_DATA');
        new(new_data);
        new_data@.preincarnate := ord(new_data);
        new_data@.next_data := cdp@.next_data;
        cdp@.next_data := new_data;
        cdp := new_data;
        with cdp@ do begin
          if screen
            then
              writeln(output, substr(pos_str, 1, pro_pos),
                  ' Enter data as fdmcprrk .......');
          read(cmd_i, flagno, damp, mass, cohes, phi, rho, rad, kn);
          damp := damp / mass;
          if ~ screen and opt.echo
            then
              writeln(output, substr(pos_str, 1, mes_pos), ' ', flagno: 6, damp:
                  6, mass: 6, cohes: 6, phi: 6, rho: 6, rad: 6, kn: 6, ' ');
          gi.max_rad := gi.max_rad + rad;
          total.datatypes := total.datatypes + 1;
          end;
        get_command(datert, qdum, cdp@.typ, '', cmd_i);
```

```
    if gi.tracing
      then
        writeln(trace_o, '  EXIT procedure CRE_DATA');
    end {cre_data};


  procedure cre_circles;

    var
      x, y: real;
      repind, numrept: integer;
      new_circle, area_l: ptr_type;

    begin
      if gi.tracing
        then
          writeln(trace_o, 'Entered procedure CRE_circleS');
      if screen
        then
          writeln(output, substr(pos_str, 1, pro_pos),
              ' Enter x, y ... coordinates..');
      while NOT eoln do begin
        read(cmd_i, x, y);
        if (~ gi.jumping) OR (gi.single)
          then
            numrept := 1
          else
            numrept := numrep;
        for repind := 1 to numrept do begin
          if gi.jumping
            then begin
              oldx := x * cose + oldx;
              oldy := y * sine + oldy;
              end
            else begin
              oldx := x;
              oldy := y;
              end;
          if opt.echo
            then
              write(output, substr(pos_str, 1, pro_pos), oldx: 9, oldy: 9);
          area := shift_area(shift_area(sal, no_cols(oldx), e), no_rows(oldy),
              n);
          if area = nil
            then
              error_simple('circle coordinates out of range ',
                  'create circles');
          if gi.reareaing
            then
              with area@.corners do
                writeln(debug_o, 'x,x~ ,y,y~ ', xmin: 6, xmax: 6, ymin: 6, ymax:
                    6, ' X, Y ', oldx: 6, oldy: 6);

          new(new_circle);
          case cdp@.typ of
            track, free: begin
              total.circles := total.circles + 1;
              new_circle@.next := area@.free_list;
              area@.free_list := new_circle
```

```
          end;
        fixed: begin
          total.fixed := total.fixed + 1;
          new_circle@.next := area@.fixed_list;
          area@.fixed_list := new_circle
          end;
        end;

      with new_circle@ do begin
        data := cdp;
        source.xc := oldx;
        source.yc := oldy;
        posn := source;
        force.x := 0;
        force.y := 0;
        s := force;
        v := force;
        a := force;
        consol := force;
        no_of_contacts := 0;
        con_list := nil;
        end;
      end;
    end;
  if gi.tracing
    then
      writeln(trace_o, '  EXIT procedure CRE_circleS');
  end {cre_circles};

var
  x, y: real;
  meshquit: boolean;
  meshcom: com_type;


begin {mesh}
  if gi.tracing
    then
      writeln(trace_o, 'Entered procedure MESH');

  numrep := 1;
  sine := 1;
  cose := 1;
  repeat
    get_command(mesher, meshquit, meshcom, '', cmd_i);
    case meshcom of
      forloop: begin
        if screen
          then
            writeln(output, substr(pos_str, 1, pro_pos),
                ' Enter no of repeats desired...');
        read(cmd_i, numrep);
        gi.single := false;
        end;
      endfor:
        gi.single := true;
      sing:
        gi.single := true;
      multip:
```

```
        gi.single := false;
       relative:
        gi.jumping := true;
       absolute:
        gi.jumping := false;
       dataset:
        cre_data;
       create:
        cre_circles;
       position: begin
        if screen
         then
          writeln(output, substr(pos_str, 1, pro_pos),
              ' Enter position to move to xy..');
        read(cmd_i, oldx, oldy);
        end;
       movepos: begin
        if screen
         then
          writeln(output, substr(pos_str, 1, pro_pos),
              ' Enter translate by x, y ......');
        read(cmd_i, x, y);
        oldx := oldx + x * cose;
        oldy := oldy + y * sine;
        end;
       angle: begin
        if screen
         then
          writeln(output, substr(pos_str, 1, pro_pos),
              ' Enter angle ( in degress ) ...');
        read(cmd_i, sine);
        sine := sine * arctan(1) / 45;
        cose := cos(sine);
        sine := sin(sine);
        end;
       meshend:
        meshquit := true otherwise;
       end
      until meshquit;
     if gi.tracing
      then
       writeln(trace_o, '  EXIT procedure MESH');
     end {mesh};


 procedure read_restart_file;

   var
    labl: char;
    no_areas: integer;
    cont: con_ptr;
    ele: ptr_type;
    old_fixe, old_free: ptr_type;


   procedure data_link(el: ptr_type);

    begin
     if gi.tracing
```

```
      then
        writeln(trace_o, 'Entered procedure DATA_LINK');
    if ord(el@.data) = sd@.preincarnate
      then
        el@.data := sd
      else begin
        sd := cdp;
        while ord(el@.data) ~ = sd@.preincarnate do
          sd := sd@.next_data;
        el@.data := sd
        end;
    if gi.tracing
      then
        writeln(trace_o, '  EXIT procedure DATA_LINK');
    end {data_link};


  procedure find_a_contact(var con: con_ptr);


    procedure find_an_element;

      begin
        if gi.tracing
          then
            writeln(trace_o, 'Entered procedure FIND_AN_ELEMENT');
        repeat
          if ele ~ = nil
            then
              ele := ele@.next
            else begin
              if area = nil
                then
                  area := sal
                else
                  area := area@.next_area;
              ele := area@.free_list
              end
          until ele ~ = nil;
        con := ele@.con_list;
        if gi.tracing
          then
            writeln(trace_o, '  EXIT procedure FIND_AN_ELEMENT');
        end {find_an_element};


    begin {find_a_contact}
      if gi.tracing
        then
          writeln(trace_o, 'Entered procedure FIND_A_CONTACT');
      repeat
        if con ~ = nil
          then
            con := con@.next_con
          else
            find_an_element;
        until con ~ = nil;
      if gi.tracing
        then
```

```
        writeln(trace_o, '  EXIT procedure FIND_A_CONTACT');
      end {find_a_contact};


  begin {read_restart_file}
    if gi.tracing
      then
        writeln(trace_o, 'Entered procedure READ_RESTART_FILE');
    no_areas := 0;
    old_fixe := nil;
    old_free := nil;
    reset(rest_i, 'unit=2');
    rewrite(cycmd_i, 'FILE=-sass.cmd');
    rewrite(repts_i, 'FILE=-sass.rep');
    writeln(output);
    while ~ eof(rest_i) do begin
      read(rest_i, buffer);
      labl := buffer.tag;
      if cols(output) = 1
        then
          write(output, substr(pos_str, 1, com_pos), ' Reading : ');
      write(output, labl);
      if cols(output) = 31
        then
          writeln(output);
      case labl of
        'G': begin
          gi := buffer.gen_info_rep;
          read(rest_i, buffer);
          total := buffer.totals_rep;
          read(rest_i, buffer);
          sum := buffer.sum_rep;
          read(rest_i, buffer);
          opt := buffer.option_rep;
          read(rest_i, buffer);
          area_i := buffer.area_i_rep;
          read(rest_i, buffer);
          plspace := buffer.grid_rep;
          mesh_areas;
          plots(input, 'init');
          end;
        'c':
          writeln(cycmd_i, buffer.hed_rep);
        'r':
          writeln(repts_i, buffer.hed_rep);
        'D': begin
          new(sd);
          with sd@ do begin
            sd@ := buffer.parabk_rep;
            next_data := cdp@.next_data;
            cdp@.next_data := sd;
            cdp := sd;
            end;
          end;
        'A': begin
          with area@ do begin
            corners := buffer.area_rep.corners;
            upd_par := buffer.area_rep.upd_par;
            upd_min := buffer.area_rep.upd_min;
```

```
      area := next_area;
      end;
    old_fixe := nil;
    old_free := nil;
    no_areas := no_areas + 1;
    if no_areas = area_i.nos
      then begin
        total.cons := 0;
        do_this(update_area, sal, false, free);
        area := nil;
        ele := nil;
        cont := nil
        end;
    end;
  'F': begin
    new(new_circle);
    new_circle@ := buffer.element_rep;
    new_circle@.con_list := nil;
    data_link(new_circle);
    new_circle@.next := nil;
    if old_fixe = nil
      then
        area@.fixed_list := new_circle
      else
        old_fixe@.next := new_circle;
    old_fixe := new_circle;
    end;
  'f': begin
    new(new_circle);
    new_circle@ := buffer.element_rep;
    new_circle@.con_list := nil;
    data_link(new_circle);
    new_circle@.next := nil;
    if old_free = nil
      then
        area@.free_list := new_circle
      else
        old_free@.next := new_circle;
    old_free := new_circle;
    end;
  'C': begin
    find_a_contact(cont);
    with cont@ do begin
      offs := buffer.con_rep.offs;
      c_force := buffer.con_rep.c_force;
      gapsum := buffer.con_rep.gapsum;
      f_force := buffer.con_rep.f_force;
      f_angle := buffer.con_rep.f_angle;
      failed := buffer.con_rep.failed;
      end;
    end;
  '*': begin
    sd := cdp;
    repeat
      cdp := cdp@.next_data;
      cdp@.preincarnate := ord(cdp);
      until sd = cdp;
    writeln(output, substr(pos_str, 1, mes_pos),
        ' A restart file has been read');
    end;
```

```
      end;
    end;
  if gi.tracing
    then
      writeln(trace_o, '  EXIT procedure READ_RESTART_FILE');
  end {read_restart_file};


procedure write_restart_file;


  procedure wr_con_rf(el: ptr_type);

    var
      con: con_ptr;

    begin
      if gi.tracing
        then
          writeln(trace_o, 'Entered procedure WR_CON_RF');
      while el ~ = nil do begin
        con := el@.con_list;
        while con ~ = nil do begin
          buffer.con_rep := con@;
          write(rest_o, buffer);
          con := con@.next_con;
          end;
        el := el@.next;
        end;
      if gi.tracing
        then
          writeln(trace_o, '  EXIT procedure WR_CON_RF');
      end {wr_con_rf};


  procedure rest_w_boxes(arel: area_ptr);


    procedure wr_blk_rf(el: ptr_type);

      begin
        if gi.tracing
          then
            writeln(trace_o, 'Entered procedure WR_BLK_RF');
        while el ~ = nil do begin
          buffer.element_rep := el@;
          write(rest_o, buffer);
          el := el@.next;
          end;
        if gi.tracing
          then
            writeln(trace_o, '  EXIT procedure WR_BLK_RF');
        end {wr_blk_rf};


    begin {rest_w_boxes}
      if gi.tracing
        then
          writeln(trace_o, 'Entered procedure REST_W_BOXES');
```

```
      while arel ~ = nil do
        with arel@ do begin
          buffer.tag := 'F';
          wr_blk_rf(fixed_list);
          buffer.tag := 'f';
          wr_blk_rf(free_list);
          buffer.tag := 'A';
          buffer.area_rep := arel@;
          write(rest_o, buffer);
          arel := next_area
          end;
      if gi.tracing
        then
          writeln(trace_o, '  EXIT procedure REST_W_BOXES');
      end {rest_w_boxes};


  begin {write_restart_file}
    if gi.tracing
      then
        writeln(trace_o, 'Entered procedure WRITE_RESTART_FILE');
    if re_area_list ~ = nil
      then
        re_area;
    do_this(update_area, sal, false, free);
    if (opt.rf_over) OR (rf_first)
      then
        rewrite(rest_o, 'unit=1');
    rf_first := false;
    buffer.tag := 'G';
    buffer.gen_info_rep := gi;
    write(rest_o, buffer);
    buffer.totals_rep := total;
    write(rest_o, buffer);
    buffer.sum_rep := sum;
    write(rest_o, buffer);
    buffer.option_rep := opt;
    write(rest_o, buffer);
    buffer.area_i_rep := area_i;
    write(rest_o, buffer);
    buffer.grid_rep := plspace;
    write(rest_o, buffer);

    reset(cycmd_i, 'FILE=-sass.cmd');
    buffer.tag := 'c';
    while ~ eof(cycmd_i) do begin
      buffer.hed_rep := nilhed;
      readln(cycmd_i, buffer.hed_rep);
      write(rest_o, buffer);
      end;
    reset(repts_i, 'FILE=-sass.rep');
    buffer.tag := 'r';
    while ~ eof(repts_i) do begin
      buffer.hed_rep := nilhed;
      readln(repts_i, buffer.hed_rep);
      write(rest_o, buffer);
      end;
    buffer.tag := 'D';
    cdp := sdl;
```

```
   repeat
     buffer.parabk_rep := cdp@;
     write(rest_o, buffer);
     cdp := cdp@.next_data;
     until cdp = sdl;
   rest_w_boxes(sal);
   buffer.tag := 'C';
   do_this(wr_con_rf, sal, false, both);
   buffer.tag := '*';
   buffer.hed_rep := 'END of RESTART FILE ';
   write(rest_o, buffer);
   writeln(output, substr(pos_str, 1, fil_pos),
       ' A restart file has been written');
   if gi.tracing
     then
       writeln(trace_o, ' EXIT procedure RESTART_FILE');
   end {write_restart_file};


 procedure complete;
{ tidy up and stop called from contrl or cycle calls bplot, check and rfile }

  begin
   if gi.tracing
     then
       writeln(trace_o, 'Entered procedure COMPLETE');
   plots(input, 'endplot');
   with total do begin
     writeln(output, substr(pos_str, 1, tot_pos), '     total circles  ',
        circles: 6, ' fixed    ', fixed: 6);
     writeln(output, '   total cracked', cracked: 6, ' contacts  ', cons: 6
        );
     writeln(output, '   total cycles ', cycles: 6, ' no.updats ', updates:
        6);
     writeln(output, '   total frames ', pages: 6, ' plots    ', pics: 6);
     end;
   if gi.tracing
     then
       writeln(trace_o, '  EXIT procedure COMPLETE');
   end {complete};


 begin {start_shut}
  if gi.tracing
   then
     writeln(trace_o, 'Entered procedure START_SHUT');
  case starting of
   cold: begin
     if screen
       then
         writeln(output, substr(pos_str, 1, pro_pos),
            ' Enter heading ................');
     readln(cmd_i, gi.heading);
     writeln(output, substr(pos_str, 1, tit_pos), gi.heading);
     plots(cmd_i, 'initialise');
     setup_a_info;
     mesh_areas;
     mesh;
     plots(cmd_i, 'ballplot');
```

```
      do_this(update_area, sal, false, free);
      end;
    shutdown: begin
      complete;
      write_restart_file;
      halt;
      end;
    warm: begin
      read_restart_file;
      headers;
      end;
    keep:
      write_restart_file;
    end;
  if gi.tracing
    then
      writeln(trace_o, ' EXIT procedure START_SHUT');
  end {start_shut};
{****************************************** END STARTSHUT }

{****************************************** BEGIN DEBUG }


procedure debug_circle(var cmd_i: text);
{   debugging routine,   called from contrl,   calls dump }

 var
  debugend: boolean;
  deb_com: com_type;
  sd: para_ptr;


 procedure write_con(el: ptr_type);


  procedure write_scan_con(home_el: ptr_type);

    var
      home_con: con_ptr;

    begin
      if gi.tracing
        then
          writeln(trace_o, 'Entered procedure WRITE_SCAN');
      home_con := home_el@.con_list;
      while home_con ~ = nil do begin
        with home_con@ do
          write(debug_o, c_force.x: 6, c_force.y: 6);
        with home_con@ do
          write(debug_o, offs: 6);
        with home_con@.other@ do
          write(debug_o, posn.xc: 6, posn.yc: 6);
        with home_el@ do
          write(debug_o, posn.xc: 6, posn.yc: 6);
        writeln(debug_o);
        home_con := home_con@.next_con;
        end;
      if gi.tracing
        then
```

```
        writeln(trace_o, '  EXIT procedure WRITE_SCAN');
      end {write_scan_con};


    begin {write_con}
      if gi.tracing
        then
          writeln(trace_o, 'Entered procedure WRITE_CON');
      if gi.debecho
        then
          writeln(debug_o, ' Contact information :');
      writeln(debug_o);
      while el ~ = nil do
        with el@ do begin
          if gi.debecho
            then begin
              writeln(debug_o, ' forces  of   contact, sibling, owner');
              writeln(debug_o);
              end;
          write_scan_con(el);
          el := next;
          end;
      if gi.tracing
        then
          writeln(trace_o, '  EXIT procedure WRITE_CON');
      end {write_con};


  procedure write_are(el: area_ptr);

    begin
      if gi.tracing
        then
          writeln(trace_o, 'Entered procedure WRITE_ARE');
      if gi.debecho
        then
          writeln(debug_o, ' Area data :');
      writeln(debug_o);
      if gi.debecho
        then
          writeln(debug_o, ' xmin,xmax,ymin,ymax, upd_par');
      writeln(debug_o);
      while el ~ = nil do
        with el@ do begin
          with corners do
            write(debug_o, xmin: 6, xmax: 6, ymin: 6, ymax: 6);
          writeln(debug_o, col: 6, row: 6, upd_par: 6);
          el := el@.next_area
          end;
      if gi.tracing
        then
          writeln(trace_o, '  EXIT procedure WRITE_ARE');
      end {write_are};


  procedure write_blk(el: ptr_type);

    begin
      if gi.tracing
```

```
      then
        writeln(trace_o, 'Entered procedure WRITE_BLK');
    if gi.debecho
      then begin
        writeln(debug_o, ' Element data :');
        writeln(debug_o);
        writeln(debug_o, ' offs posn force velocity accleration datatype');
        writeln(debug_o);
        end;
    while el ~ = nil do
      with el@ do begin
        writeln(debug_o, source.xc: 6, source.yc: 6, posn.xc: 6, posn.yc: 6,
            consol.x: 8, consol.y: 8, force.x: 8, force.y: 8, v.x: 8, v.y: 8,
            a.x: 8, a.y: 8, data@.flagno: 3);
        el := el@.next;
        end;
    if gi.tracing
      then
        writeln(trace_o, '  EXIT procedure WRITE_BLK');
    end {write_blk};


  begin {debug_circle}
    if gi.tracing
      then
        writeln(trace_o, 'Entered procedure DEBUG_circle');
    repeat
      get_command(debuger, debugend, deb_com, '', cmd_i);
      case deb_com of
        dat: begin
          cdp := sdl;
          if gi.debecho
            then begin
              writeln(debug_o);
              writeln(debug_o, 'flag  damp   inert  mass    c     phi    '
                  || 'rho    rad    kn      typ');
              end;
          repeat
            with cdp@ do begin
              writeln(debug_o, flagno: 6, damp: 8, mass: 8, cohes: 8, phi: 8,
                  rho: 8, rad: 8, kn: 8, ord(typ): 6);
              cdp := next_data
              end
            until cdp = sdl
          end;
        blk:
          do_this(write_blk, sal, false, both);
        con:
          do_this(write_con, sal, false, free);
        are:
          write_are(sal);
        gen: begin
          writeln(debug_o, gi.heading);
          writeln(debug_o);
          with area_i do begin
            writeln(debug_o, ' xareas   number ', xmax: 6, ' length ', size.xc:
                6);
            writeln(debug_o, ' yareas   number ', ymax: 6, ' length ', size.yc:
                10);
```

```
      writeln(debug_o, ' total   number ', nos: 6);
      end;
    with plspace do
      writeln(debug_o, ' mapping xmax  ', xmax: 6, ' ymax  ', ymax: 6)          ;
    writeln(debug_o);
    with opt do begin
      writeln(debug_o, ' plot   interval', cycle_interval: 6);
      writeln(debug_o, ' gravity x      ', grav.x: 6, ' y      ', grav.y
        : 6);
      writeln(debug_o, ' timing delay ', gi.tfrac: 6);
      writeln(debug_o);
      end;
    with total do begin
      writeln(debug_o, ' totals circles ', circles: 6, ' fixed ',
        fixed: 6);
      writeln(debug_o, '        cracked ', cracked: 6, ' types  ',
        datatypes: 6);
      writeln(debug_o, '        contact ', cons: 6, ' cycles  ', cycles:
        6);
      writeln(debug_o, '        updates ', updates: 6, ' frames  ',
        pages: 6);
      writeln(debug_o, '        plots  ', pics: 6);
      writeln(debug_o);
      end;
    end;
  fon:
    with gi do begin
      reareaing := true;
      motioning := true;
      updating := true;
      cycling := true;
      fording := true;
      oscing := true;
      tracing := true;
      consoling := true;
      end;
  fof:
    with gi do begin
      reareaing := false;
      motioning := false;
      updating := false;
      cycling := false;
      fording := false;
      oscing := false;
      tracing := false;
      consoling := false;
      end;
  reb:
    gi.reareaing := onoff(cmd_i);
  mot:
    gi.motioning := onoff(cmd_i);
  upd:
    gi.updating := onoff(cmd_i);
  cyc:
    with gi do begin
      cycling := onoff(cmd_i);
      fording := cycling;
      motioning := cycling;
      reareaing := cycling;
      consoling := cycling;
```

```
          end;
        fod:
          gi.fording := onoff(cmd_i);
        sol:
          gi.consoling := onoff(cmd_i);
        tra:
          gi.tracing := onoff(cmd_i);
        osc:
          gi.oscing := onoff(cmd_i);
        otherwise;
        end;
      until debugend;
    if gi.tracing
      then
        writeln(trace_o, '  EXIT procedure DEBUG_circle');
    end {debug_circle};
{**************************************** END DEBUG }
{**************************************** BEGIN PARAMETERS }


procedure parameters(var cmd_i: text);


  procedure calculator;


    function intcalc(op: real): real;

      var
        result, v: real;
        oper: com_type;

      begin
        get_command(operter, qdum, oper, '', cmd_i);
        if oper ~ = enquiry
          then begin
            if screen
              then
                writeln(output, substr(pos_str, 1, pro_pos),
                    ' Enter value .................');
            read(cmd_i, v);
            end;
        case oper of
          equal:
            result := v;
          mult:
            result := op * v;
          divid:
            result := op / v;
          plus:
            result := op + v;
          minus:
            result := op - v;
          power:
            result := exp(ln(op) * v);
          otherwise
            result := op;
          end;
        if opt.echo
```

```
      then
        if oper = enquiry
          then
            writeln(output, substr(pos_str, 1, mes_pos), ' The value is : ',
                result)
          else
            writeln(output, substr(pos_str, 1, mes_pos), ' ', v, ' : ',
                result);
    intcalc := result;
    end {intcalc};

var
  datquit, calquit: boolean;
  datcom, calcom: com_type;
  flag: integer;


begin {calculator}
  repeat
    get_command(calcter, calquit, calcom, '', cmd_i);
    case calcom of
      cyclegp:
        opt.cyclegap := round(intcalc(opt.cyclegap));
      gravity:
        opt.grav.y := intcalc(opt.grav.y);
      ptime:
        gi.tstep := intcalc(gi.tstep);
      cmdint:
        opt.cycle_interval := round(intcalc(opt.cycle_interval));
      datype: begin
        if screen
          then
            writeln(output, substr(pos_str, 1, pro_pos),
                ' Enter data type flag .........');
        read(flag);
        repeat
          get_command(datalte, datquit, datcom, '', cmd_i);
          cdp := sdl;
          repeat
            cdp := cdp@.next_data;
            with cdp@ do
              if flagno = flag
                then
                  case datcom of
                    dfact:
                      damp := intcalc(damp);
                    dmass:
                      mass := intcalc(mass);
                    dcohe:
                      cohes := intcalc(cohes);
                    dfric:
                      phi := intcalc(phi);
                    ddens:
                      rho := intcalc(rho);
                    dradi:
                      rad := intcalc(rad);
                    dstif:
                      kn := intcalc(kn);
                    otherwise;
                    end;
```

```
          until cdp = sdl;
        until datquit;
      end;
    otherwise;
    end;
  until calquit;
end {calculator};

var
  parcom: com_type;
  parquit: boolean;
  flimit: integer;
  cmdlistword: string(12);


begin {parameters}
  repeat
    get_command(paramer, parquit, parcom, '', cmd_i);
    case parcom of
      echo:
        opt.echo := onoff(cmd_i);
      debech:
        gi.debecho := onoff(cmd_i);
      framlim: begin
        if screen
          then
            writeln(output, substr(pos_str, 1, pro_pos),
                ' Enter frame limit ...........');
        read(cmd_i, flimit);
        gpstop(flimit);
        if opt.echo
          then
            write(output, substr(pos_str, 1, mes_pos),
                ' Frame limit is now : ', flimit);
        end;
      cyclegp: begin
        if screen
          then
            writeln(output, substr(pos_str, 1, pro_pos),
                ' Enter gap between writing.....');
        read(cmd_i, opt.cyclegap);
        if opt.echo
          then
            write(output, substr(pos_str, 1, mes_pos),
                ' Cycle gap is now  : ', opt.cyclegap);
        end;
      gravity: begin
        if screen
          then
            writeln(output, substr(pos_str, 1, pro_pos),
                ' Enter gravity values x, y ....');
        read(cmd_i, opt.grav.x, opt.grav.y);
        if opt.echo
          then
            write(output, substr(pos_str, 1, mes_pos),
                ' Gravity is now    : ', opt.grav.x: 6, opt.grav.y: 6);
        end;
      ptime: begin
        if screen
```

```
          then
            writeln(output, substr(pos_str, 1, pro_pos),
                ' Enter time step increment ....');
        read(cmd_i, gi.tstep);
        if opt.echo
          then
            write(output, substr(pos_str, 1, mes_pos),
                ' Time increment is  :  ', gi.tstep);
        end;
      calc:
        calculator;
      cmdint: begin
        if screen
          then
            writeln(output, substr(pos_str, 1, pro_pos),
                ' Enter cmd process interval ...');
        read(cmd_i, opt.cycle_interval);
        if opt.echo
          then
            write(output, substr(pos_str, 1, mes_pos),
                ' Process interval is: ', opt.cycle_interval);
        end;
      cmdlist: begin
        rewrite(cycmd_i, 'FILE=-sass.cmd');
        repeat
          word_scan(cmd_i, cmdlistword);
          writeln(cycmd_i, cmdlistword);
          until cmdlistword = 'cend';
        end;
      listpr:
        opt.cmdprocessing := onoff(cmd_i);
      over_rf:
        opt.rf_over := onoff(cmd_i);
      otherwise;
      end;
    until parquit;
  end {parameters};
{***************************************** END PARAMETERS }

{***************************************** BEGIN REPEATER }


procedure repeater(var cmd_i: text);

  var
    cmdreptword: string(12);
    loopcntor, loopctr: integer;

  begin
    if gi.tracing
      then
        writeln(trace_o, 'Entered procedure REPEATER');
    rewrite(repts_i, 'FILE=-sass.rep');
    read(cmd_i, loopctr);
    repeat
      word_scan(cmd_i, cmdreptword);
      writeln(repts_i, cmdreptword)
      until cmdreptword = 'rend';
    for loopcntor := 1 to loopctr do begin
```

```
    reset(repts_i, 'FILE=-sass.rep');
    gi.reptend := false;
    while NOT gi.reptend do
      control(repts_i);
    end;
  if gi.tracing
    then
      writeln(trace_o, '  EXIT procedure REPEATER');
  end {repeater};
{****************************************** END REPEATER }

{****************************************** BEGIN CONTROL }


procedure control;
{   controls the execution of the datafile commands, called from main }

 var
  com: com_type;

 begin
  if gi.tracing
    then
      writeln(trace_o, 'Entered procedure CONTROL');
  get_command(contler, qdum, com, '', cmd_i);
  case com of
    sets:
      parameters(cmd_i);      { set parameter values }
    cend:
      gi.cmdend := true;      { end interrupt commands }
    rend:
      gi.reptend := true;    { end command stack }
    rest:
      start_shut(cmd_i, warm);{ restart a previous run }
    save:
      start_shut(cmd_i, keep);{ update restart file }
    star:
      start_shut(cmd_i, cold);{ start a new run }
    cycl:
      cycle(cmd_i);         { calculation routines }
    sett:
      gi.settling := true;    { settlement of elements }
    coll:
      gi.settling := false;  { collapse of elements }
    plot:
      plots(cmd_i, '');      { plot routines }
    debg:
      debug_circle(cmd_i);    { debugging routine }
    rept:
      repeater(cmd_i);       { command stack }
    stop:
      quit := true;         { stop command }
    retur:;
    end;
  if quit
    then
      start_shut(cmd_i, shutdown);
  if gi.tracing
```

```
    then
      writeln(trace_o, '  EXIT procedure CONTROL');
   end {control};
{*************************************** END CONTROL  }

{*************************************** BEGIN MAIN  }


begin {circles}
  initialise_globals;
  headers;
  repeat
    control(input);
    until quit;
  start_shut(input, shutdown);
  if gi.tracing
    then
      writeln(trace_o, '  EXIT procedure MAIN');
  end {circles}.
```