Stress Analysis of Highly Jointed Rock Using Parallel Processing

Technical Report 31-011

G.K. Egan
*M.A. Coulthard**
W.Heath

*CSIRO
Division of Geomechanics
Mount Waverley 3149

Version 2.0 Original Document 1/3/90

Key Words: *parallel processing, seismic modelling, distinct element method*

Abstract:

Computational stress analysis is now widely used in geomechanics for back analysis of observed rock mass behaviour around surface and underground excavations and as a tool for excavation design in mining and civil engineering. The distinct element (DE) method, represents a rock mass as a discontinuum, and has been shown to be more realistic than finite element (FE) or boundary element (BE) (continuum) methods for modelling systems such as subsiding strata over underground coal mine excavations. In this initial study a DE code been implemented in the applicative parallel processing language SISAL, explicit parallel Pascal and compared with the performance of the original FORTRAN implementation The work presented here is part of a larger study into the implementation of DE methods on parallel computer systems.

1. Introduction

Computational stress analysis is now widely used in geomechanics for back analysis of observed rock mass behaviour around surface and underground excavations and as a tool for excavation design in mining and civil engineering. The distinct element (DE) method, which represents a rock mass as a discontinuum, has been shown to be more realistic than finite element (FE) or boundary element (BE) (continuum) methods for modelling systems such as subsiding strata over underground coal mine excavations. However, whereas even 3D FE and BE analyses can now be performed readily on engineering workstations or the more powerful personal computers, the DE method generally requires orders of magnitude more computer processing time for analyses of comparable complexity. This has so far prevented the DE method from being applied widely in excavation design in industry.

Most DE codes are based upon an explicit time integration of Newton's laws of motion for each DE, usually involving many thousands of time steps or solution cycles in a full analysis. The explicit numerical method implies that, within each cycle, calculations for each DE can be carried out in parallel. With the growing availability of moderately priced medium range multiprocessors, multiprocessor workstations and parallel language systems there is the potential to obtain satisfactory performance at a reasonable cost.

In this paper we develop parallel processing versions of a relatively simple 2D DE stress analysis code. This code, originally implemented in FORTRAN, has been implemented in SISAL a modern applicative language, and Pascal an imperative language.

The changes made to the program and its translation into SISAL and Pascal will be outlined, times for the original SISAL and Pascal versions reported, and some preliminary conclusions drawn regarding the usefulness of various parallel processing options for DE stress analysis codes.

2. The Distinct Element Method

The DE method of stress analysis was introduced in [1] to deal with problems in rock mechanics which could not be treated adequately by the conventional continuum methods. The earliest DE programs (e.g. program RBM in [2]) assumed that the DEs were rigid, so that all deformations within the system took place at the DE interfaces. A second program described in [2], SDEM, allowed modelling of three simple modes of deformation of each DE - two compressive and one shear mode. The DE programs which are most widely used at present are UDEC [3] and 3DEC [4]; the DEs in each of these may be modelled as fully deformable via internal finite difference zoning.

2.1 Theoretical basis

Most DE programs are based on force-displacement relations describing DE interactions and Newton's second law of motion for the response of each DE to the unbalanced forces and moments acting on it.

The normal forces developed at a point of contact between DEs are calculated from the notional overlap of those DEs and the specified normal stiffness of the inter-DE joints. Tensile normal forces are usually not permitted, i.e. there is no restraint placed upon opening of a contact between DEs.

Shear interactions are load-path dependent, so incremental shear forces are calculated from the increments in shear displacement, in terms of the shear stiffness of the joints. The maximum shear force is usually limited by a Mohr-Coulomb or similar strength criterion.

The motion of each DE under the action of gravity, external loadings and the forces arising from contact with other DEs is determined from Newton's second law. A damping mechanism is also included in the model to account for dissipation of vibrational energy in the system.

The equations of motion may be integrated with respect to time using a central difference scheme to yield velocities and then integrated again to yield displacements. The velocity dependent damping terms have been omitted here for simplicity, but the same form of equations hold even when damping is included.

$$\dot{u}_i(t+\Delta t) = \dot{u}_i(t) + (\sum F_i(t)/m + g_i) \cdot \Delta t \qquad (1)$$

$$u_i(t+\Delta t) = u_i(t-\Delta t) + \dot{u}_i(t+\Delta t) \cdot \Delta t \qquad (2)$$

where $i = 1,2$ correspond to x and y directions respectively;
u_i are the components of displacement of the DE centroid;
F_i are the components of non-gravitational forces acting on the DE;
g_i are the components of gravitational acceleration;
m is the mass of the particular DE.

The equation of rotation for each DE can be integrated similarly. Note that, in the integrated equations, DE velocities and displacements are expressed explicitly in term of values at a previous time and so may each be calculated independently.

The calculation cycle proceeds, with the calculated displacements being used to update the geometry of the system, and thence to determine new DE interaction forces. These, in turn, are used in the next stage of the explicit integration of Newton's equations.

This explicit time integration scheme is only conditionally stable. Physically, the time step must be small enough that information cannot pass between neighbouring DEs in one step, thus justifying the independence of the integrated equations of motion.

2.2 DECYL

DECYL is a simple program which analyses 2 dimensional systems of interacting, rigid circular DEs of equal radius [5]. It is based on the same explicit integration algorithm used in DE programs for practical stress analysis of highly jointed rock. DECYL assumes that any of the circular DEs comprising the system being modelled may be in contact with any other DE; contact lists are not maintained and physical locality is not exploited. The diagramatic representation of a DECYL system is is shown in Figure 1. The general flow of computation in DECYL is shown in Figure 2.

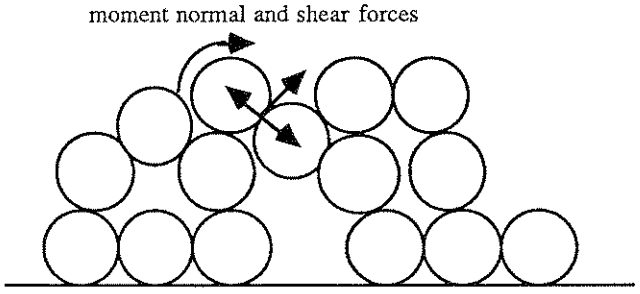

Figure 1. a DECYL system

```
while not stable do
    for j in 1 to no_DEs+1 do
        for i in j+1 to no_DEs+1 do
          if touching DE[i] and DE[j] then
              compute normal force
              if normal force > 0 then
                  compute moment
                  compute shear force
                  if slipping then
                      adjust shear force
              resolve force components
              accumulate forces and moments acting on DE[j] and DE[i]
        integrate accelerations on DE[j]
        compute new position of DE[j]
```

Figure 2. DECYL computational flow

3. Machines and languages

3.1 Machines

The machines used in this study were two Encore Multimax multiprocessor (one with four XPC processors at ~4MIP each and the other with 20 slower APC processors at ~2MIP each), an IBM RS6000/530 uniprocessor workstation and a Cray YMP. The IBM workstation was chosen as its CPU performance will be representative of CPUs in future medium cost multiprocessors. The Cray YMP single processor performance provides a reference point.

3.2 Languages

The languages used in the study were SISAL, explicit parallel Pascal and, as a language reference point, sequential FORTRAN.

SISAL [9] is an applicative language which has been targetted at a wide variety of systems including uniprocessors, current generation multiprocessors such as the Encore Multimax and research dataflow machines [10][11][12]. The textual form of SISAL, in terms of control structures and array representations, (Appendix) provides a relatively easy transition for those familiar with imperative languages and the optimising SISAL compiler (osc) from Colorado has yielded performance competitive with FORTRAN [13][14]. SISAL requires no directives or annotation at the source level.

The SISAL compiler exploits loop concurrency. Loops with no dependencies between cycles are 'sliced' into several loops each over some some number of cycles of the original loop. The number of slices is determined at runtime with the slices being executed concurrently.

For the Pascal implementation we used the standard UNIX fork primitive and the parallel programming primitives available in the Encore Multimax.

5. Results

The system used for the studies was a stack of 500 DEs for ten iterations. A 'movie of a smaller system of 100 DEs over 8000 iterations is shown in Figure 3.

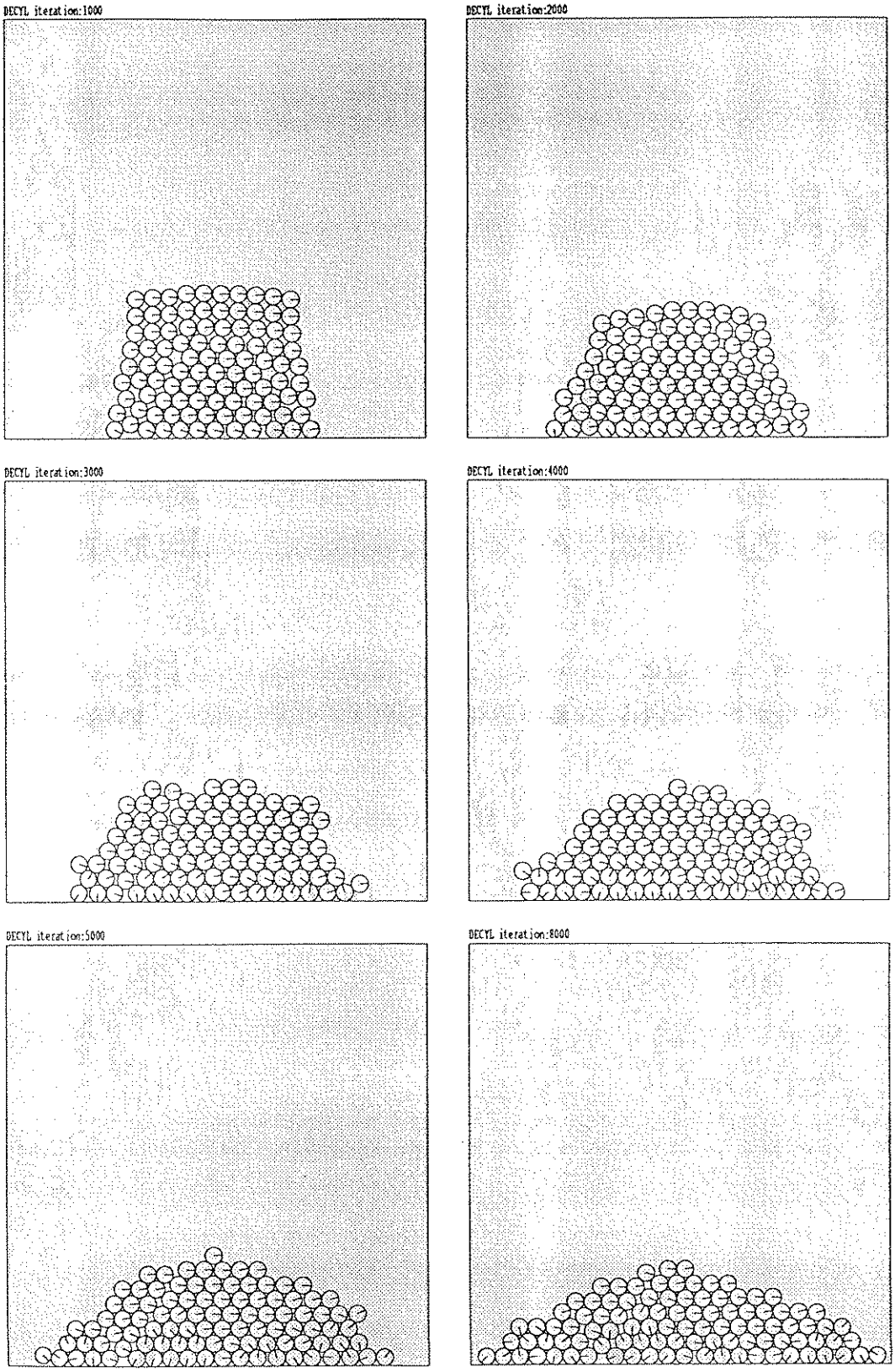

Figure 3. DECYL Movie

5.1 FORTRAN

Some elementary optimisations were performed on the FORTRAN including the elimination of division operations (Appendix A). The times (user+system) for unoptimised and optimised FORTRAN versions on the Encore Multimax and RS6000 are shown in table 1. These optimisations were carried over to the SISAL and Pascal versions. The run times for FORTRAN are used for reference against the SISAL and Pascal results which follow.

System	Compiler	original	optimised	difference
Encore Multimax (XPC)	f77 -O	73.8	31.3	-58%
IBM RS6000/530	xlf -O	6.9	4.3	-38%

Table 1. unoptimised and optimised FORTRAN times

5.2 SISAL

DECYL was translated into SISAL (Appendix B). No particular difficulty was encountered in this process as the original formulation in FORTRAN was well written with no side effects through COMMON and EQUIVALENCE statements. These FORTRAN statements can cause significant difficulties when recasting FORTRAN application codes into SISAL [16]. The speedup obtained for SISAL is presented in Figure 4.

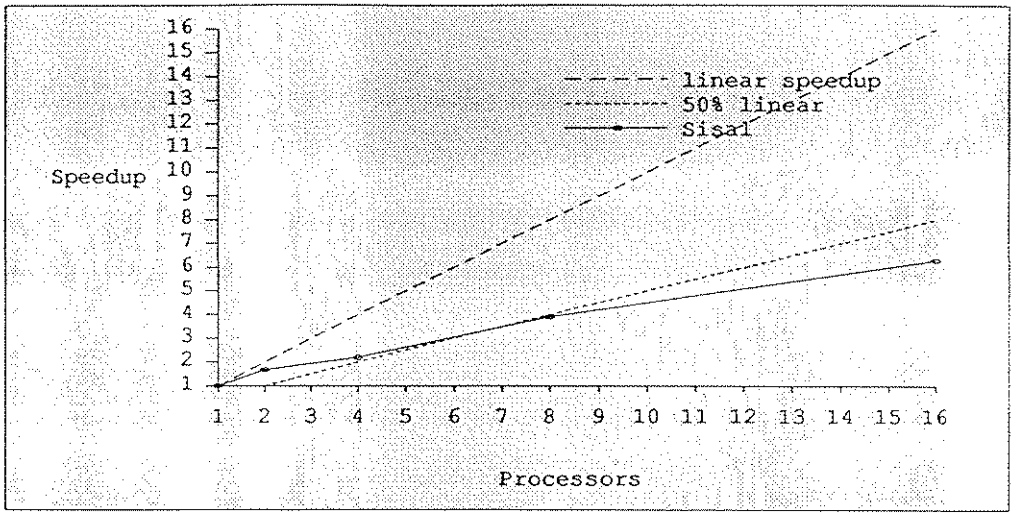

Figure 4. Speedup of DECYL in SISAL (Encore)

Further improvement in SISAL performance is expected, reducing overheads due mainly to the array construction and access mechanisms of the current implementation of SISAL. SISAL permits structures to change size at runtime involving indirect access to matix elements via a vector of pointers to each matrix row. There are also consequential memory allocation overheads as structures are progressively constructed. The potential gains from static allocation of structures have been acknowledged by the developers of SISAL and will be seen in later SISAL versions [15].

5.2 Pascal

To amortise the startup cost of loop slices it is best to maximise the work done for each slice. The strategy then is to parallelise outer loops where possible; inner loops need only be parallelised if the bounds of the outer loop are such as to not provide sufficient slices to load the machine.

The speedup for DECYL with the inner loop annotated explicitily (appendix C) is shown in Figure 5a and with the outer loop annotated (appendix D) in Figure 5b.

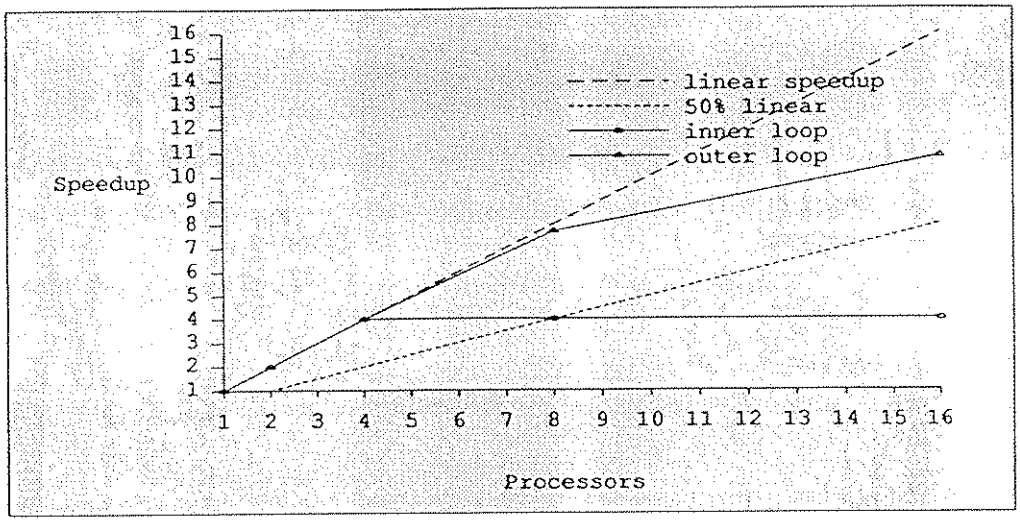

Figure 5. Speedup of DECYL in Pascal (Encore)

The collected times for SISAL, Pascal and FORTRAN on the YMP, IBM RS6000 and faster four processor Encore Multimax are given in table 2a and times for the slower Encore in table 2b.

	User + Sytem (seconds) speedup in ()			
	YMP 1 cpu	RS6000/530	Encore (XPC) 1 cpu	Encore (XPC) 4 cpus
FORTRAN	0.52	4.25	31.3	-
Sisal	5.15	11.78	57.9	25.7 (2.3)
Pascal *seq*	4.66	2.02	34.0	-
Pascal *inner*	-	-	40.3	23.6 (1.7)*
Pascal *outer*	-	-	34.6	9.0 (3.9)*

	Wallclock (seconds) speedup in ()			
	YMP 1 cpu	RS6000/530	Encore (XPC) cpu	Encore (XPC) 4 cpus
FORTRAN	0.956 (97%)	4.3	31.7	-
Sisal	33.4*	11.8	58.2	28.9
Pascal *seq*	34.0*	2.1	34.3	-
Pascal *inner*	-	-	40.8	26.0 (1.6)*
Pascal *outer*	-	-	34.9	11.1 (3.1)*

(a)

	User + Sytem (seconds) speedup in ()							
cpus	1	2		4		8		16
FORTRAN	?	-		-		-	-	
Sisal	163.0	97.0	(1.68)	73.0	(2.2)	42.0	(3.9)	26.0 (6.3)
Pascal *seq*	75.1	-		-		-	-	
Pascal *inner*	104.3	50.4	(2.0)	25.8	(4.0)	26.0	(4.0)	-
Pascal *outer*	85.3	42.7	(2.0)	21.6	(4.0)	11.1	(7.7)	7.9 (10.8)*

	Wallclock (seconds) speedup in ()							
cpus	1		2		4		8		16
FORTRAN	?							
Sisal	164.18	98.24	(1.7)	74.1	(2.2)	43.3	(3.8)	26.04	(6.3)
Pascal *seq*	75.1							
Pascal *inner*	96.9	53.2	(1.8)	29.2	(3.3)	29.9	(3.2)		
Pascal *outer*	86.7	44.3	(2.0)	23.7	(3.7)	13.7	(6.33)	11.7	(7.41)*

** systems heavily loaded*
YMP FORTRAN compiler (cf77 -Zp -Wf)

(b)

Table 1. Times and speedup () for 500 DEs and 10 time steps

6. Conclusions

This initial study has shown that it is possible to obtain good speedup on current multiprocessors. These processors have relatively poor scientific performance but it is certain that next generation processors will be greatly improved in this respect.

SISAL has performance competitive with FORTRAN and shows promise for implicit parallel programming. Its portability over a wide range of current parallel and high performace computer systems and next generation systems makes it attractive for studies in parallel programming.

Acknowledgements

The authors thank the other members of the Laboratory for Concurrent Computing Systems, at the Swinburne Institute of Technology, for their contributions to the work presented in this report.

References

[1] Cundall, P.A., A computer model for simulating progressive large-scale movements in DEy rock systems. Proc. ISRM Symp. on Rock Fracture, Nancy, vol. 1, paper II-8. 1971.

[2] Cundall, P.A. et al., Computer modeling of jointed rock masses. Technical Report N-78-4, U.S. Army Engineer Waterways Experiment Station, Vicksburg, Miss., 1978.

[3] Itasca. UDEC - Universal distinct element code, version ICG1.6; User's manual. Itasca Consulting Group, Inc., Minneapolis, 1990.

[4] Itasca. 3DEC - 3-D distinct element code, version 1.2; User's manual. Itasca Consulting Group, Inc., Minneapolis, 1990.

[5] Lorig, L.J., Private communication with M.A. Coulthard, 1985.

[6] Lemos, J.V., A hybrid distinct element - boundary element computational model for the half-plane. M.S. Thesis, Dept. of Civil and Mineral Engng., University of Minnesota, Minneapolis, 1983.

[7] Choi, S.K. and M.A. Coulthard, Mechanics of jointed rock masses using the distinct element method. Int. Conf. on Mechanics of Jointed and Faulted Rock, Vienna, 1990, (in press).

[8] Taylor, L.M., BLOCKS: A Block motion code for geomechanics studies, Sandia National Laboratories, Albuquerque, Report SAND-82-2373, 1983.

[9] McGraw et al, SISAL: Streams and Iteration in a Single Assignment Language, Language Reference Manual, Lawrence Livermore National Laboratories, M146.

[10] Skedzielewski S. and J. Glauert , IF1 An Intermediate Form for Applicative Languages, Lawrence Livermore National Laboratories, 1985.

[11] Bohm A.P.W and J. Sargeant, Efficient Dataflow Code Generation for SISAL, Technical Report UMCS-85-10-2, Department of Computer Science, University of Manchester, 1985.

[12] Webb N.J., Implementing an Applicative Language for the RMIT/CSIRO Dataflow Machine, Department of Computer Science, Royal Melbourne Institute of Technology, *M.App.Sci. Thesis in preparation*, 1990.

[13] Egan G.K., N.J. Webb and A.P.W. Bohm, Some Features of the CSIRAC II Dataflow Machine Architecture, in Advanced Topics in Data-Flow Computing, Prentice-Hall 1990, *in print.*

[14] Cann D.C. and R.R. Oldehoeft, Compilation Techniques for High Performance Applicative Computation, Technical Report CS-89-108, Colorado State University, May 1989.

[15] Cann D.C., High Performance Parallel Applicative Computation, Technical Report CS-89-104, Colorado State University, Feb.1989.

[16] Chang P.S. and G.K. Egan, An Implementation of a Barotropic Numerical Weather Prediction Model in the Functional Language SISAL, Second ACM Sigplan Symposium on Parallel Programming (PPoPP), Seattle, March, 1990.

APPENDIX A: Optimised DECYL in FORTRAN

```
      program decyl
c
c     This program was used by Dr. Loren Lorig at an internal workshop
c       on distinct element modelling at the Division of Geomechanics
c       in September 1985, just before he returned to the U.S. to join
c       Itasca Consulting Group, Inc., in Minneapolis
c
c     It is a very simple distinct element code, for 2-D analysis of the
c       mechanical interactions of a number of cylinders with each other
c       and a fixed horizontal plane.  (2-D implies that the axes of all
c       the cylinders are parallel with each other and with the fixed
c       plane; analysis is on a perpendicular plane through the system.)
c
c     The cylinders all have the same radius and are rigid.  Their
c       elastic interactions are governed by normal and shear stiffnesses
c       (which yield interaction forces according to the calculated
c       'overlap' at their points of contact).  If the calculated shear
c       force exceeds the specified frictional strength, the cylinders
c       will slip at that point of contact; if a tensile normal force is
c       calculated at a contact, that contact will be 'broken', i.e.
c       assumed zero tensile strength.
c
c     The equations of motion for the system are solved by explicit
c       numerical integration, with some imposed artificial damping to
c       represent loss of energy within the system and to prevent it
c       'ringing' interminably.
c
c     In the case which is hard-coded into this version of the program,
c       two cylinders are resting on the horizontal plane, touching each
c       other.  The third cylinder is dropped from a point on the plane
c       of symmetry.  After about 2000 cycles, the cylinders come to
c       rest, with the third one sitting between the other two on the
c       plane.
c
c     We will amend the code shortly to input the geometrical and other
c       data, and to draw a simple plot of the system of cylinders, as
c       the job runs or at the end of an analysis.
c
c     Variables:
c       ncyl = number of cylinders
c                ('cylinder' ncyl+1 is the horizontal plane)
c       fn(i,j) = normal force between cylinders i and j
c       m(i,j)  = moment due to shear interaction between i and j
c       du(i) = u-coordinate of centroid of cylinder i
c       dv(i) = v-    "       "    "      "    "     "
c       da(i) = a-angle of cylinder from starting position
c       u(i)  = u-component of velocity of     "     "
c       v(i)  = v-    "      "    "      "     "     "
c       w(i)  = angular velocity of cylinder i
c       ddu(i)= increment in u-component of displacement of cylinder i
c       ddv(i)=     "       " v-    "      "       "        "    "     "
c       gamma(i)=   "       " rotation of cylinder i
c       fxsum(i)= u(x)-component of total force acting on cylinder i
c       fysum(i)= v(y)-   "       "   "     "     "     "    "     "
c       msum(i) = total moment acting on cylinder i
```

```
c       Optimisation G.K. Egan
c       Laboratory for Concurrent Computing Systems
c       Swinburne Institute of Technology
c
c       rho = density
c       d = diameter of cylinders;  r = radius
c       g = acceleration due to gravity
c       mu = coefficient of friction at contacts
c       mass = mass of cyclinders;  moi = moment of inertia
c       akn = normal stiffness at contacts;  akn = shear stiffness
c       alpha = damping constant
c       tdel = time step
c
      parameter (ncyl=500)
      parameter (iside=100)
      parameter (nn=ncyl+1)
      dimension fn(nn,nn), m(nn,nn), fxsum(nn), fysum(nn), msum(nn),
     .          u(nn), v(nn), w(nn), du(nn), dv(nn), gamma(nn),
     .          ddu(nn), ddv(nn),
     .          da(nn)
      real m, msum, mass, mu, moi, rmoi, rmass
cgke
      mod(i,j)= i-(i/j)*j
      pi=4.*atan(1.)
      rho=2000.
      d=100.
      rd=1.0/d
      dsqr=d*d
      r=d/2.
      rr=2./d
      g=-10.
      mu=0.1
      mass=rho*pi*r*r
      rmass=1.0/mass
      moi=mass*r*r/2.
      rmoi=1.0/moi
      akn=1.e9
      aks=1.e9
      frac=1.00
      freq=0.200
      alpha=2.*pi*frac*freq
      tdel=2.*sqrt(mass/akn)/10.
      con1=alpha*tdel/2.
      rplus1con1 = 1.0/(1.0+con1)
c
c     Initial coordinates of centroids of cylinders
c
c     centroid of (ncyc+1)th cylinder located at -r,
c           to give effect of fixed horizontal plane at v=0
cgke stack of cylinders
      do 222 i=1,ncyl
          du(i)=mod(i,iside)*d
          dv(i)=(i/iside)*d+r
222   continue
      du(nn)=0.0
      dv(nn)=-r
c
```

```
c     Initialise forces, velocities etc.
c
      do 7 j=1,nn
      do 8 i=j+1,nn
      fn(j,i)=0.
      m(j,i)=0.
    8 continue
      fxsum(j)=0.
      fysum(j)=0.
      msum(j)=0.
      u(j)=0.
      v(j)=0.
      w(j)=0.
      ddu(j)=0.
      ddv(j)=0.
      gamma(j)=0.
    7 continue
c
c     Set up counters for time integration stepping
c         - maximum no. of steps = its
c           kount used to print progressive results every 50 steps
c
      its=10
      iframes =1
      ikount=its/iframes
      kount=ikount
      write(6,*)ncyl,its,iframes
c      write(6,102) alpha,mu
c  102 format(' Program decyl (L. Lorig 1985) with alpha =',f6.2,
c     .       '  and mu =',f6.2)
      do 4 n=1,its
      if(kount.eq.ikount) then
c      do 999 ii=1,ncyl
c999   write(6,*) n-1, r,du(ii),dv(ii),da(ii)
        kount=0
        endif
      kount=kount+1
c     Loop over cylinders
      do 3 j=1,ncyl
c     Place (ncyc+1)th cylinder 'opposite' jth
      du(nn)=du(j)
c     Set gamma for (ncyc+1)th so get no shear interaction between
c       jth cylinder and flat plane if former is rolling rather than
c       sliding
      gamma(nn)=-ddu(j)*rr
c     Look for interactions with cylinders with higher numbers
      k=j+1
      do 2 i=k,nn
      dudif=du(j)-du(i)
      dvdif=dv(j)-dv(i)
c     is the cylinder (i) touching cyclinder (j) ?
      z=dudif*dudif+dvdif*dvdif
      if(z.le.dsqr) then
      rz=1./sqrt(z)
      dudif=dudif*rz
      dvdif=dvdif*rz
      ddudif=ddu(i)-ddu(j)
```

```
      ddvdif=ddv(i)-ddv(j)
c     calculate the normal force fn(j,i)
      dfn=akn*(ddvdif*dvdif+ddudif*dudif)
      fn(j,i)=fn(j,i)+dfn
      if(fn(j,i).ge.0.) then
c     calculate moment
      theta=(ddvdif*dudif-ddudif*dvdif)*rd
      dm=-aks*(gamma(j)+gamma(i)-theta)
      m(j,i)=m(j,i)+dm
c     calculate shear force
      ft=m(j,i)*rr
c     is slip occurring?
      ff=mu*fn(j,i)
      abft=abs(ft)
      if(abft.gt.ff) then
        ft=ff*ft/abft
        endif
      m(j,i)=r*ft
c     calculate force components (fx,fy)
      fx=fn(j,i)*dudif-ft*dvdif
      fy=fn(j,i)*dvdif+ft*dudif
c     forces on cylinder j
      fxsum(j)=fxsum(j)+fx
      fysum(j)=fysum(j)+fy
      msum(j)=msum(j)+m(j,i)
c     reactions back on cylinder j
      fxsum(i)=fxsum(i)-fx
      fysum(i)=fysum(i)-fy
      msum(i)=msum(i)+m(j,i)
      endif
      else
      fn(j,i)=0.
      m(j,i)=0.
      endif
    2 continue
c
c     integrate accelerations to find displacements
      u(j)=(u(j)*(1.-con1)+((fxsum(j)*rmass)*tdel))*r1plusconl
      v(j)=(v(j)*(1.-con1)+((fysum(j)*rmass+g)*tdel))*r1plusconl
      w(j)=(w(j)*(1.-con1)+((msum(j)*rmoi)*tdel))*r1plusconl
      ddu(j)=u(j)*tdel
      ddv(j)=v(j)*tdel
      gamma(j)=w(j)*tdel
      du(j)=du(j)+ddu(j)
      dv(j)=dv(j)+ddv(j)
      da(j)=da(j)+gamma(j)
      fxsum(j)=0.
      fysum(j)=0.
      msum(j)=0.
    3 continue
    4 continue
      stop
      end
```

APPENDIX B: DECYL in SISAL

```
% Original program decyl by Dr. Loren Lorig in FORTRAN.
% Sisal  G.K. Egan Laboratory for Concurrent Computing Systems 1990
% Swinburne Institute of Technology loosely based on a version by Warwick
% Heath.

define main

type    vector = array [real];              % Data rep. copied from orig.
type  dudv_rec = record[du, dv : real]      % Just for output.

global sqrt(x: real returns real)


function main( returns array[array[dudv_rec]])

for initial
    % auto generate a stack of cylinders
    ncyl:integer := 500;
    iside:integer := 100;
    i,j:integer;
    its:integer := 10;
    n : integer := ncyl+1;
    pi : real := 3.141592654;
    rho : real := 2000.0;
    d : real := 100.0;
    dd :real := d*d;
    rd :real := 1.0/d;
    r : real := d/2.0;
    rr: real := 1.0/r;
    initialdu: vector := for i in 1,ncyl
       returns array of real(i-(i/iside)*iside)*d end for;
    initialdv: vector := for i in 1,ncyl
      returns array of real(i/iside)*d+r end for;
    g : real := -10.0;
    mu :real := 0.1;
    mass : real := rho*pi*r*r;
    rmass : real := 1.0/mass;
    moi : real := mass*r*r/2.0;
    rmoi: real := 1.0/moi;
    akn : real := 1.0e9;
    aks : real := 1.0e9;
    frac : real := 1.0;
    freq : real := 0.20;
    alpha : real := 2.0*pi*frac*freq;
    tdel : real := 2.0*sqrt(mass/akn)/10.0;
    con1 : real := alpha*tdel/2.0;
    r1pluscon1:real := 1.0/(1.0+con1);

    fn : array[vector] :=
             for j in 1,ncyl returns array of
               for i in j+1,n
                 returns array of 0.0
               end for
             end for;
    m : array[vector] := fn;
```

```
    u : vector := array_fill(1,ncyl,0.0);
    v : vector := u;
    w : vector := u;
    ddu : vector := u;
    ddv : vector := u;
    gamma : vector := u;
    du : vector := initialdu;
    dv : vector := initialdv;

    iter : integer := 1;
    cylinders : array[dudv_rec] :=
      array_fill(1,ncyl,record dudv_rec[du:0.0;dv:0.0]);

while (iter <= its) repeat  % for number of iterations

  fxsumj, fysumj, msumj, fn, m := for j in 1,ncyl
    alldu : vector := array_addh(old du,old du[j]);
    alldv : vector := array_addh(old dv,-r);
    allddu : vector := array_addh(old ddu,0.0);
    allddv : vector := array_addh(old ddv,0.0);
    allgamma : vector := array_addh(old gamma,(-old ddu[j])/r);

    fxsumji, fysumji, msumji, fnji, mji := for i in j+1,n
        dudif : real := alldu[j] - alldu[i];
        dvdif : real := alldv[j] - alldv[i];
        z : real := dudif*dudif + dvdif*dvdif;

        % Interactions with other cylinders???
        fxsumji, fysumji, msumji, fnji, mji : real :=
          if (z > dd | j = i) then
            0.0, 0.0, 0.0, 0.0, 0.0
          else
            let
            rz := 1.0/sqrt(z);
            zdudif :real := dudif*rz;
            zdvdif :real := dvdif*rz;
            ddudif :real := allddu[i] - allddu[j];
            ddvdif :real := allddv[i] - allddv[j];
            % calculate the normal force between i and j
            dfn : real := akn * (ddvdif*zdvdif + ddudif*zdudif);
            testfn : real := (old fn[j,i]) + dfn;
            retfxsum, retfysum, retmsum, retfn, retm : real :=
              if (testfn < 0.0) then
                0.0, 0.0, 0.0, 0.0, 0.0
              else
                let
                % Calculate moment
                theta : real :=
                  (ddvdif*zdudif - ddudif*zdvdif)*rd;
                dm : real := -aks *(allgamma[j] + allgamma[i] - theta);
                mji : real := old m[j,i] + dm;
                % Calculate shear force
                testft : real := mji*rr;
                % Is slip occurring
                ff : real := mu * testfn;
                abft : real := abs(testft);
                ft : real :=
```

```
                if (abft > ff) then ff * testft / abft
                else testft
                end if;
              retmji := r*ft;
              % Calculate force components (fx,fy)
              fx : real := (testfn*zdudif) - (ft*zdvdif);
              fy : real := (testfn*zdvdif) + (ft*zdudif);
              in
                fx, fy, retmji, testfn, retmji
              end let
            end if;
          in
            retfxsum, retfysum, retmsum, retfn, retm
          end let
        end if;
    returns
      value of sum fxsumji
      value of sum fysumji
      value of sum msumji
      array of fnji
      array of mji
    end for;
  returns
    array of fxsumji
    array of fysumji
    array of msumji
    array of fnji
    array of mji
  end for;

%
% Now integrate accelerations to find displacements
%
u,v,w,ddu,ddv,gamma,du,dv:=
 for j in 1, ncyl
uj,vj,wj,dduj,ddvj,gammaj,duj,dvj:=
 if j=1 then
((fxsumj[j]*rmass)*tdel) * r1plusconl,
  ((fysumj[j]*rmass+g)*tdel) * r1plusconl,
  ((msumj[j]*rmoi)*tdel) * r1plusconl,
  0.0,
  0.0,
  ((msumj[j]*rmoi)*tdel) * r1plusconl,
  0.0,
  0.0
else
  let
    tuj:real:=(old u[j]*(1.0-conl) + ((fxsumj[j]*rmass)*tdel))
                                       *r1plusconl;
    tvj:real:=(old v[j]*(1.0-conl) + ((fysumj[j]*rmass+g)*tdel))
                                       *r1plusconl;
    twj:real:=(old w[j]*(1.0-conl) + ((msumj[j]*rmoi)*tdel))
                                       *r1plusconl;
    tdduj:=tuj*tdel;
    tddvj:=tvj*tdel;
    tgammaj:=twj*tdel
  in
```

```
      tuj,
      tvj,
      twj,
      tdduj,
      tddvj,
      tgammaj,
      old du[j] + tdduj,
      old dv[j] + tddvj
    end let
  end if;
returns
array of uj
array of vj
array of wj
array of dduj
array of ddvj
array of gammaj
array of duj
array of dvj
end for;
    iter := old iter + 1;
  cylinders := for j in 1,ncyl
  returns
    array of record dudv_rec[du:du[j]; dv:dv[j]]
  end for;
returns
    array of cylinders when (mod(iter,its) = 0)
end for % number of iterations
end function % main
```

APPENDIX C: Explicitily annotated DECYL in Pascal - Version 1

```
program decyl(input, output);
{Original by L.Lorig 1985 (visitor to CSIRO Geomechanics Melb.Australia)}
{Pascal version by G.K.Egan 1990
 Laboratory for Concurrent Computing Systems
 Swinburne Institute of Technology }

label
               999;
const
               ncyl = 500;
              iside = 100;
                 nn = 501 {ncyl+1};
            frames = 1;

type
               lock = volatile char;
        barrierrec = volatile array[0..31] of integer;
        ptrbarrier = ^barrierrec;
var
       ip,procs,p,i,j,n:integer;
                  rz,
                   z,
                 dfn,
               theta,
                  dm,
                  ft,
                abft,
                  ff,
                  fx,
                  fy,
               dudif,
              dvdif,
             ddudif,
             ddvdif:real;
shared
               klock: lock;
                   k: integer;
        init_barrier,
         acc_barrier:barrierrec;
                  fn,
                   m: array [1..nn, 1..nn] of real;
              fxsum,
              fysum,
               msum,
                  u,
                  v,
                  w,
                 du,
                 dv,
                 da,
              gamma,
                ddu,
                ddv: array [1..nn] of real;
               mass,
              rmass,
```

```
                mu,
               moi,
              rmoi,
                pi,
               rho,
                 d,
              dsqr,
                rd,
                 r,
                rr,
                 g,
               akn,
               aks,
              frac,
              freq,
             alpha,
              con1,
         r1plusco1,
                dn,
              tdel:real;
             kount,
            ikount,
                ii,
               its: integer;


{DECLARE lock=volatile char; ptrbarrier=^integer}


procedure spinlock(var l:lock); nonpascal;

procedure spinunlock(var l:lock); nonpascal;

procedure fbarrier_init(var b:barrierrec;count:integer;var p:integer);
nonpascal;

procedure fbarrier(var b: barrierrec); nonpascal;

function fork:integer; nonpascal;

procedure wait(i:integer); nonpascal;

procedure exit(i:integer); nonpascal;

procedure new_barrier(var b:barrierrec;count:integer; var pid:integer);

begin
  fbarrier_init(b,count,pid);
end;

function nfork(nprocs:integer):integer;
var proc,child:integer;
begin
  child:=-1;
  proc:= nprocs - 1;
  while (child <> 0) and (proc > 0) do
    begin
```

```
      child:=fork;
      if child <>  0 then
        proc:=pred(proc);
    end;
  nfork:=proc
end;

procedure njoin(id,nprocs:integer);
var j:integer;

begin
  if id = 0 then
    for j := 1 to (nprocs - 1) do
      wait(0)
  else
    exit(0)
end;

begin
pi := 4.0 * arctan(1.0);
rho := 2000.0;
d := 100.0;
rd:= 1.0/d;
dsqr:=sqr(d);
r := d / 2.0;
rr:=2.0/d;
g := - 10.0;
mu := 0.1;
mass := rho * pi * r * r;
rmass:=1.0/mass;
moi := mass * sqr(r) / 2.0;
rmoi:=1.0/moi;
akn := 1.0e9;
aks := 1.0e9;
frac := 1.00;
freq := 0.200;
alpha := 2.0 * pi * frac * freq;
tdel := 2.0 * sqrt(mass / akn) / 10.0;
con1 := alpha * tdel / 2.0;
r1plusconl:=1.0/(1.0+con1);
{stack of cylinders}
for i:=1 to ncyl do begin
          du[i]:=(i mod iside)*d;
          dv[i]:=(i/iside)*d+r;
  end;
dv[nn] := - r;
{Initialise forces, velocities etc.}
for j := 1 to nn do
  begin
    for i := 1 to nn do
      begin
        fn[j, i] := 0.0;
        m[j, i] := 0.0;
      end;
    da[j] :=0.0;
    fxsum[j] := 0.0;
    fysum[j] := 0.0;
```

```
    msum[j] := 0.0;
    u[j] := 0.0;
    v[j] := 0.0;
    w[j] := 0.0;
    ddu[j] := 0.0;
    ddv[j] := 0.0;
    gamma[j] := 0.0;
  end;
its := 10;
ikount:=its div frames;
kount := ikount;
writeln(ncyl,' ',its,' ',frames);

writeln('procs ? ');
readln(procs);
new_barrier(init_barrier,procs,p);
new_barrier(acc_barrier,procs,p);

spinunlock(klock);

p:=nfork(procs);

for n := 1 to its do
  begin {Loop over cylinders}
    if p = 0 then begin
    if kount = ikount then
      begin
        {for j:= 1 to ncyl do
          writeln(n-1,' ', r,' ',du[j],' ',dv[j],' ',da[j]);}
        kount := 0;
      end;
    kount := kount + 1;
    end;
    for j:= 1 to ncyl do
      begin {Place (ncyc+1)th cylinder 'opposite' jth}
        if p=0 then begin
        du[nn] := du[j];
        {Set gamma for (ncyc+1)th so get no shear interaction between
         jth cylinder and flat plane if former is rolling rather than
         sliding}
        gamma[nn] := - ddu[j] / r;
        {Look for interactions with cylinders with higher numbers}
        k := j + 1;
        end;

        fbarrier(init_barrier) {to protect k} ;

        repeat
        spinlock(klock);
          ip:=k;
          k:=k+1;
        spinunlock(klock);

        if ip <= nn then begin
            dudif := du[j] - du[ip];
            dvdif := dv[j] - dv[ip];
           {is the cylinder [ip] touching cyclinder [j] ?}
```

```
      z := dudif * dudif + dvdif * dvdif;
      if z <= dsqr
      then
        begin
          rz := 1.0/sqrt(z);
          dudif := dudif * rz;
          dvdif := dvdif * rz;
          ddudif := ddu[ip] - ddu[j];
          ddvdif := ddv[ip] - ddv[j];
          {calculate the normal force fn[j,ip]}
          dfn := akn * (ddvdif * dvdif + ddudif * dudif);
          fn[j, ip] := fn[j, ip] + dfn;
          if fn[j, ip] >= 0.0
          then
            begin
              {calculate moment}
              theta := (ddvdif * dudif - ddudif * dvdif) * rd;
              dm := - aks * (gamma[j] + gamma[ip] - theta);
              m[j, ip] := m[j, ip] + dm;
              {calculate shear force}
              ft := m[j, ip] * rr;
              {is slip occurring?}
              abft := abs(ft);
              ff := mu * fn[j, ip];
              if abft > ff then
                ft := ff * ft / abft;
              m[j, ip] := r * ft;
              {calculate force components (fx,fy)}
              fx := fn[j, ip] * dudif - ft * dvdif;
              fy := fn[j, ip] * dvdif + ft * dudif;
              {forces on cylinder j}
              fxsum[j] := fxsum[j] + fx;
              fysum[j] := fysum[j] + fy;
              msum[j] := msum[j] + m[j, ip];
              {reactions back on cylinder j}
              fxsum[ip] := fxsum[ip] - fx;
              fysum[ip] := fysum[ip] - fy;
              msum[ip] := msum[ip] + m[j, ip];
            end
          else
            begin
              fn[j, ip] := 0.0;
              m[j, ip] := 0.0;
            end
        end
      else
        begin
          fn[j, ip] := 0.0;
          m[j, ip] := 0.0;
        end;
    end;
  until ip>= nn;
  {integrate accelerations to find displacements}
  fbarrier(acc_barrier);
  if p = 0 then begin
  u[j] := (u[j] * (1.0 - con1) + ((fxsum[j] * rmass) * tdel)) *
                                    r1plusconl;
```

```
        v[j] := (v[j] * (1.0 - con1) + ((fysum[j] * rmass + g) * tdel)) *
                                        rlplusconl;
        w[j] := (w[j] * (1.0 - con1) + ((msum[j] * rmoi) * tdel)) *
                                        rlplusconl;
        ddu[j] := u[j] * tdel;
        ddv[j] := v[j] * tdel;
        gamma[j] := w[j] * tdel;
        du[j] := du[j] + ddu[j];
        dv[j] := dv[j] + ddv[j];
        da[j]:=da[j]+gamma[j];
        fxsum[j] := 0.0;
        fysum[j] := 0.0;
        msum[j] := 0.0;
        end;
      end;
  end;
  writeln('finished ',p:1);
  njoin(p,procs);
  999:
end.
```

APPENDIX D: Explicitily annotated DECYL in Pascal - Version 2

```
program decyl(input, output);
{original by L.Lorig 1985 (visitor to CSIRO Geomechanics Melb.Australia)}
{translated to Pascal by G.K.Egan 1989
 Laboratory for Concurrent Computing Systems
 Swinburne Institute of Technology }

label
               999;
const
               ncyl = 500;
              iside = 100;
                 nn = 501 {ncyl+1};
            frames = 1;

type
               lock = volatile char;
         barrierrec = volatile array[0..31] of integer;
         ptrbarrier = ^barrierrec;
var
       procs,p,i,j,k,n:integer;
                  rz,
                   z,
                 dfn,
              gammann,
                dunn,
               theta,
                  dm,
                  ft,
                abft,
                  ff,
                  fx,
                  fy,
               dudif,
              dvdif,
             ddudif,
             ddvdif:real;
shared
               acclock,jlock: lock;
            jj_global,
            j_global: integer;
         init_barrier,
         end_acc_barrier,
          acc_barrier:barrierrec;
                  fn,
                   m: array [1..nn, 1..nn] of real;
              fxsum,
              fysum,
               msum,
                  u,
                  v,
                  w,
                 du,
                 dv,
                 da,
              gamma,
```

```
                ddu,
                ddv: array [1..nn] of real;
               mass,
              rmass,
                 mu,
                moi,
               rmoi,
                 pi,
                rho,
                  d,
               dsqr,
                 rd,
                  r,
                 rr,
                  g,
                akn,
                aks,
               frac,
               freq,
              alpha,
               con1,
          r1pluscon1,
                 dn,
               tdel:real;
             kount,
            ikount,
                ii,
              its: integer;

{DECLARE lock=volatile char; ptrbarrier=^integer}


procedure spinlock(var l:lock); nonpascal;

procedure spinunlock(var l:lock); nonpascal;

procedure fbarrier_init(var b:barrierrec;count:integer;var p:integer);
nonpascal;

procedure fbarrier(var b: barrierrec); nonpascal;

function fork:integer; nonpascal;

procedure wait(i:integer); nonpascal;

procedure exit(i:integer); nonpascal;

procedure new_barrier(var b:barrierrec;count:integer; var pid:integer);

begin
  fbarrier_init(b,count,pid);
end;

function nfork(nprocs:integer):integer;
var proc,child:integer;
begin
```

```
  child:=-1;
  proc:= nprocs - 1;
  while (child <> 0) and (proc > 0) do
    begin
      child:=fork;
      if child <>  0 then
        proc:=pred(proc);
    end;
  nfork:=proc
end;

procedure njoin(id,nprocs:integer);
var j:integer;

begin
  if id = 0 then
    for j := 1 to (nprocs - 1) do
      wait(0)
  else
    exit(0)
end;

begin
pi := 4.0 * arctan(1.0);
rho := 2000.0;
d := 100.0;
rd:= 1.0/d;
dsqr:=sqr(d);
r := d / 2.0;
rr:=2.0/d;
g := - 10.0;
mu := 0.1;
mass := rho * pi * r * r;
rmass:=1.0/mass;
moi := mass * sqr(r) / 2.0;
rmoi:=1.0/moi;
akn := 1.0e9;
aks := 1.0e9;
frac := 1.00;
freq := 0.200;
alpha := 2.0 * pi * frac * freq;
tdel := 2.0 * sqrt(mass / akn) / 10.0;
con1 := alpha * tdel / 2.0;
r1plusco1:=1.0/(1.0+con1);
{stack of cylinders}
for i:=1 to ncyl do begin
          du[i]:=(i mod iside)*d;
          dv[i]:=(i/iside)*d+r;
  end;
dv[nn] := - r;
{Initialise forces, velocities etc.}
for j := 1 to nn do
  begin
    for i := 1 to nn do
      begin
        fn[j, i] := 0.0;
        m[j, i] := 0.0;
```

```
      end;
    da[j] :=0.0;
    fxsum[j] := 0.0;
    fysum[j] := 0.0;
    msum[j] := 0.0;
    u[j] := 0.0;
    v[j] := 0.0;
    w[j] := 0.0;
    ddu[j] := 0.0;
    ddv[j] := 0.0;
    gamma[j] := 0.0;
  end;
its := 10;
ikount:=its div frames;
kount := ikount;
writeln(ncyl,' ',its,' ',frames);

writeln('procs ? ');
readln(procs);
new_barrier(init_barrier,procs,p);
new_barrier(acc_barrier,procs,p);
new_barrier(end_acc_barrier,procs,p);

spinunlock(acclock);
spinunlock(jlock);

p:=nfork(procs);

for n := 1 to its do
  begin {Loop over cylinders}
    if p = 0 then begin
    if kount = ikount then
      begin
          {for k:= 1 to ncyl do
            writeln(n-1,' ', r,' ',du[k],' ',dv[k],' ',da[k]);}
        kount := 0;
      end;
    kount := kount + 1;
    j_global:=1;
    jj_global:=1
    end;

    fbarrier(init_barrier);

    repeat
      spinlock(jlock);
      j:=j_global;
      j_global:=j_global+1;
      spinunlock(jlock);

    if j <= ncyl then
      begin {Place (ncyc+1)th cylinder 'opposite' jth}
        dunn := du[j];
        {Set gamma for (ncyc+1)th so get no shear interaction between
         jth cylinder and flat plane if former is rolling rather than
         sliding}
        gammann := - ddu[j] / r;
```

```
{Look for interactions with cylinders with higher numbers}
k := j + 1;

for i:= k to nn do begin
    if i = nn then
      dudif := du[j] - dunn
    else
      dudif := du[j] - du[i];
    dvdif := dv[j] - dv[i];
   {is the cylinder [i] touching cyclinder [j] ?}
    z := dudif * dudif + dvdif * dvdif;
    if z <= dsqr
    then
      begin
        rz := 1.0/sqrt(z);
        dudif := dudif * rz;
        dvdif := dvdif * rz;
        ddudif := ddu[i] - ddu[j];
        ddvdif := ddv[i] - ddv[j];
        {calculate the normal force fn[j,i]}
        dfn := akn * (ddvdif * dvdif + ddudif * dudif);
        fn[j, i] := fn[j, i] + dfn;
        if fn[j, i] >= 0.0
        then
          begin
            {calculate moment}
            theta := (ddvdif * dudif - ddudif * dvdif) * rd;
            if i = nn then
              dm := - aks * (gamma[j] + gammann - theta)
            else
              dm := - aks * (gamma[j] + gamma[i] - theta);
            m[j, i] := m[j, i] + dm;
            {calculate shear force}
            ft := m[j, i] * rr;
            {is sli occurring?}
            abft := abs(ft);
            ff := mu * fn[j, i];
            if abft > ff then
              ft := ff * ft / abft;
            m[j, i] := r * ft;
            {calculate force components (fx,fy)}
            fx := fn[j, i] * dudif - ft * dvdif;
            fy := fn[j, i] * dvdif + ft * dudif;
            {forces on cylinder j}
            fxsum[j] := fxsum[j] + fx;
            fysum[j] := fysum[j] + fy;
            msum[j] := msum[j] + m[j, i];
            {reactions back on cylinder j}
            spinlock(acclock);
              fxsum[i] := fxsum[i] - fx;
              fysum[i] := fysum[i] - fy;
              msum[i] := msum[i] + m[j, i];
            spinunlock(acclock);
          end
        else
          begin
            fn[j, i] := 0.0;
```

```
                    m[j, i] := 0.0;
                  end
              end
            else
              begin
                fn[j, i] := 0.0;
                m[j, i] := 0.0;
              end;
          end;
          end;
        {integrate accelerations to find displacements}
        until j>= nn;

        fbarrier(acc_barrier);

        repeat
          spinlock(jlock);
            j:=jj_global;
            jj_global:=jj_global+1;
          spinunlock(jlock);

        if j <= ncyl then begin
        u[j] := (u[j] * (1.0 - con1) + ((fxsum[j] * rmass) * tdel)) *
r1plusconl;
        v[j] := (v[j] * (1.0 - con1) + ((fysum[j] * rmass + g) * tdel)) *
r1plusc
on1;
        w[j] := (w[j] * (1.0 - con1) + ((msum[j] * rmoi) * tdel)) *
r1plusconl;
        ddu[j] := u[j] * tdel;
        ddv[j] := v[j] * tdel;
        gamma[j] := w[j] * tdel;
        du[j] := du[j] + ddu[j];
        dv[j] := dv[j] + ddv[j];
        da[j]:=da[j]+gamma[j];
        fxsum[j] := 0.0;
        fysum[j] := 0.0;
        msum[j] := 0.0;
        end;
      until j>= ncyl;
      fbarrier(end_acc_barrier);
  end;
  writeln('finished ',p:1);
  njoin(p,procs);
  999:
end.
```