REPORT NO.
UCB/SEMM-98/06

STRUCTURAL ENGINEERING
MECHANICS AND MATERIALS

A SORTING CONTACT DETECTION ALGORITHM: FORMULATION AND FINITE ELEMENT IMPLEMENTATION

BY

E. G. PETÖCZ

AND

F. ARMERO

APRIL 1998

DEPARTMENT OF CIVIL ENGINEERING
UNIVERSITY OF CALIFORNIA
BERKELEY, CALIFORNIA

A Sorting Contact Detection Algorithm: Formulation and Finite Element Implementation

by

E. G. Petőcz & F. Armero

Structural Engineering, Mechanics and Materials
Department of Civil and Environmental Engineering
University of California at Berkeley
Berkeley, CA 94720
e-mail: armero@ce.berkeley.edu

Abstract

This report describes the formulation and numerical implementation of a contact detection algorithm for multi-body contact problems in the context of the finite element method. In particular, a contact algorithm for two dimensional contact/impact problems is enhanced by the addition of a sorting algorithm. The binary space partitioning method combined with a binary tree database is used to perform a geometric sorting of all the bodies involved in the contact problem by detecting possible contacting pairs. These pairs consist of a particle (a surface node of a body in the finite element sense) and a body. Ultimately, contact is determined by means of a closest point procedure among the possible contacting pairs yielded by the sorting scheme. Detailed information is given related to the implementation of such a scheme in FORTRAN 90. Numerical experimentation confirms the expected $\mathcal{O}(N \log N)$ behavior of binary space partitioning based methods as compared with an all-to-all methodology with a worse-case cost of $\mathcal{O}(N^2)$, for a problem involving N bodies.

1 Introduction

We can distinguish two conceptually different parts in the numerical algorithms employed in the simulation of contact problems. The first part of the scheme deals with the contact detection whereas the second part involves different techniques and approaches to enforce the impenetrability constraints between two surfaces in contact. The role of the contact detection algorithms become particularly critical when simulating multi-body systems, in terms of speed and memory usage. The contact detection procedure may become cumbersome as the system of bodies becomes large. In a standard scheme, this task would include performing N^2 contact detection procedures between any two bodies in a problem involving N bodies. Therefore, the need arises for a more efficient way of determining the geometric relationships among objects within a working space when N is large, since this process accounts for a significant portion of the computational effort of solving contact/impact problems among many bodies.

Contact detection can be defined as finding the members of a set of points that lie inside a sub-region of an n_{dim} dimensional space. In the finite element context, the set of points could be interpreted as the set of nodes which lie on the surface of one body and, in turn, each body defines a sub-region. The two dimensional case is considered in the actual numerical implementation described in this report. In a multi-body system, it is appropriate to distinguish two phases in the contact detection procedure: a spatial sorting phase and a contact resolution (or searching) phase; see e.g. WILLIAMS & O'CONNOR [7]. The spatial sorting enables to find all the pairs of bodies which could be potential contactors, and the contact resolution phase finds the actual two points in contact on the surface of each pair of bodies.

We describe in this report several issues involved in the formulation and numerical implementation of a spatial sorting phase of the contact detection algorithm within a finite element formulation. The described sorting/resolution scheme has been implemented in the Finite Element Program Analysis FEAP (see [8]). The numerical simulations presented herein employ the energy-momentum conserving contact algorithms proposed recently in ([1, 2]) by the authors.

An outline of the rest of this report is as follows. Section 2 presents a brief summary of the available techniques to perform spatial sorting procedures. In Section 3, we elaborate on the concept of binary space partitioning and its role in a sorting algorithm, which is presented in Section 4. We develop in Section 5 the notion of the resolution phase in the contact

detection algorithm. Different details of the implementation are presented in Section 6 and we asses in Section 7 the performance of the scheme in a series of representative simulations.

2 Overview of various sorting techniques

Consider a general system of bodies whose configuration changes in time. The efficiency of the contact detection algorithm can be greatly improved if one can assume *a priori* knowledge of how this system will evolve; however the range of problems that can be tackled may be limited. Following verbatim WILLIAMS & O'CONNOR [7], the kind of *a priori* knowledge that can be used to produce an efficient detection algorithm can be classified as follows:

1. **Fixed topology**
   Fixed topology can be found in finite element algorithms when the relative position of the elements remains unchanged during the simulation of a particular problem.

2. **Slowly varying topology**
   The objects move around only a small amount, so that each object only interacts with its neighbors and we only keep track of a small number of possible contactors per body. If one can keep track of the characteristic velocities in a problem, then it is also possible to check for contact after only a certain number of time steps instead of after every one.

3. **Spatially sparse systems**
   If the system is very sparse, it makes sense to project trajectories in such a way that we only check for the intersection of cones or cylinders in a space-time system.

4. **Exhaustive spatial schemes**
   In this case, the scheme makes no *a priori* assumptions about the evolution of the problem and it is based only on the present geometric configuration. These schemes are more general and complete, but are potentially slower than non-exhaustive schemes.

5. **Spatial sorting algorithms**
   Spatial sorting gives a valuable tool to decide which bodies should be considered for a more detailed contact resolution. It seeks to avoid the all-to-all body search for contact at each time step. For a small quantity of objects, this all-to-all method is acceptable, but it can become computationally prohibitive as the number of objects increases.

In this work, we use a *spatial sorting technique* that does not make use of any *a priori* knowledge of the system for the sorting phase. The more traditional closest-point projection is used for the contact resolution phase. Below, we present a review several of the most common methods to perform spatial sorting, following again WILLIAMS & O'CONNOR [7] (we refer to this reference for a complete discussion of these different methods):

1. **Grid subdivision**
   The grid subdivision method divides in a uniform way the simulation volume or area into rectilinear cells, each cell enclosing one or more objects. Objects are associated with a cell. Neighboring objects are detected by their cell assignment. The performance of this method depends greatly on the homogeneity of the spatial distribution of the objects within the working space and is not a good overall methodology for a wide range of problems.

2. **Adaptive grid methods**
   Adaptive cell methods are used to avoid the problems associated with simple *grid subdivision*, though at the cost of managing multiple cell dimensions. With this approach, the scheme suffers when the distribution of objects becomes homogeneous.

3. **Body based cells**
   In this case, the cell surrounding an object is based on its centroid. Objects lying within this cell are considered to be a potential contactor and added to a *neighbor list.*

4. **Spatial heap-sort**
   Heap-sort is one of several algorithms used to sort the objects into an ordered list which at the same time gives an indication of the location of each particular object. In this case, the key to the ordered list is the object's coordinate along one or more global axes. A heap-sort algorithm called DFR has been developed by WILLIAMS & O'CONNOR [7] and applied to baseline granular simulation problems.

5. **Tree methods**
   Binary sort/search algorithms provide a flexible and general methodology for contact detection in two dimensional problems. The octree sort/search algorithm was derived from the binary one to handle problems in three dimensions. The method considers objects as associated with rectilinear cells and ordered into a tree data structure. We refer to the classical work of KNUTH [5] for a complete account on the creation and handling of trees. In particular, the time required to create a binary tree is of the order $\mathcal{O}(N \log(N))$, where N is the total number of objects in the problem; the same applies

for the traversing of the tree. In short, such an algorithm is a valuable improvement over an all-to-all search, which is $\mathcal{O}(N^2)$.

In this work, we choose the *binary space partitioning* implemented with a binary tree database structure to perform the sorting for the two dimensional problems under consideration (see e.g. BONET & PERAIRE[3] and MUNJIZA ET AL[6]). It has proven to be a versatile tool that performs efficiently when applied to a wide range of problems in the area of contact/impact simulations, without making use of any *a priori* knowledge of the problem at hand.

3 Binary space partitioning

We describe in this section several fundamental concepts used in a binary space partitioning (BSP) scheme implemented in this work. For completeness, primary computational tools, like a binary tree, and their role in the BSP scheme are defined and presented in detail.

3.1 Binary tree structure

A tree structure stores in a systematic way a collection of data in order to enable a quick access and retrieval of the information. A tree consists of *nodes* where the actual data is stored. Each node is extended by the addition of two links to two other nodes known as the *left child* and the *right child.* The node from which a particular node springs is called the *parent* node. Each tree has a starting node which we name *root.* Also, at each node, there is a *subtree* originating from it so that the node becomes the root of this subtree.

This definition establishes a hierarchy of nodes: the root at the top level; 0, 1 or 2 nodes at the next level, each of which in turn has 0, 1 or 2 nodes at the next level of hierarchy; and so forth. The quantity of levels indicate the depth of the tree. A node without children is said to be a *leaf.* This hierarchical structure inspires the graphical representation shown in Figure 1.

3.2 Space partitioning using a binary tree

We describe below the procedure by which we construct the binary tree. Consider a problem with N bodies within an area which is called the *searching space* or *working space.* The

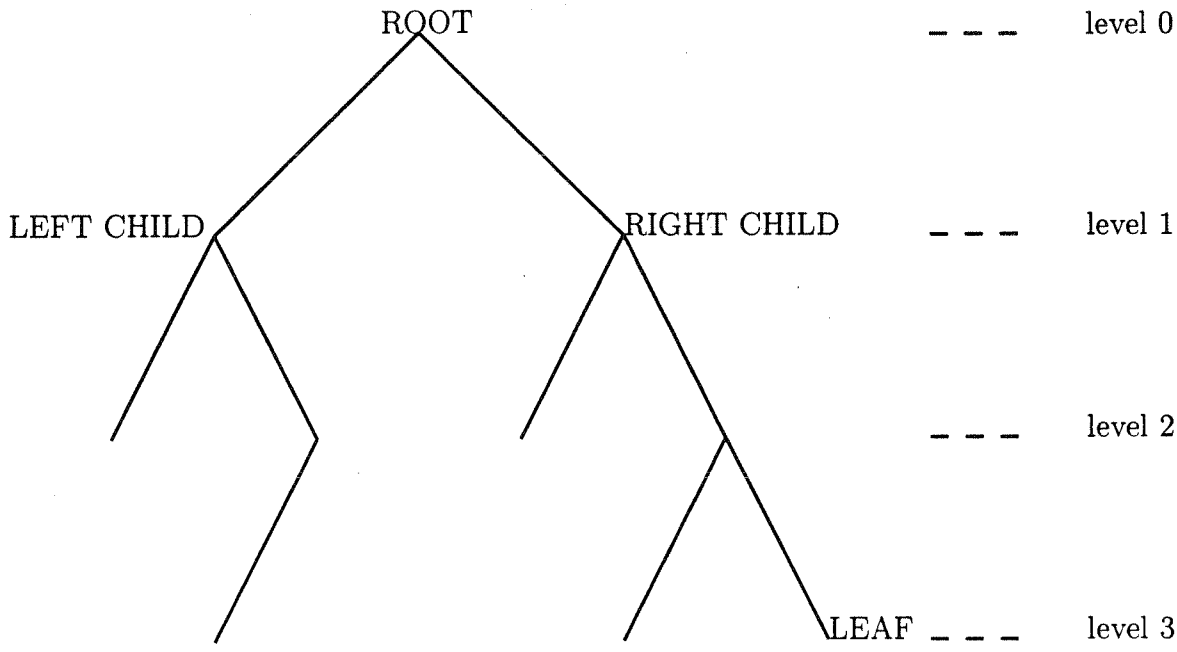

Figure 1: Schematic drawing of a binary tree

searching space may be redefined for each sorting procedure. If this space is not redefined for an interval of time, during which we may perform any number of sorting procedures, its position should be chosen such that during this time interval of interest the bodies in question never leave the searching space. To maximize the efficiency of a binary tree based sorting scheme, one can redefine the working space at each instant one performs the sorting procedure. This last consideration tends to maximize the homogeneity of the distribution of the bodies in the problem; thus the depth of the tree decreases making the subsequent retrieval of information much faster.

Consider the set of surface nodes corresponding to the surfaces of the N bodies and denote by n_{total} the size of this set of nodes. From the point of view of the widely used slave/master methodology for contact problems (HALLQUIST ET AL [4]), this set of nodes corresponds to the surface nodes of the slave and master bodies which participate in the contact problem prior to any closest point projection procedure. No such distinction is needed in the discussion that follows. Figure 2 shows a typical configuration of two bodies. Figure 3 shows the corresponding system of particles in a chosen working space.

Remark 3.1 In the rest of this paper, we refer as *particles* to the set of n_{total} nodes belonging to the surfaces of the bodies participating in the contact problem, and we give the name *nodes* to the nodes of the binary tree used in storing the data.

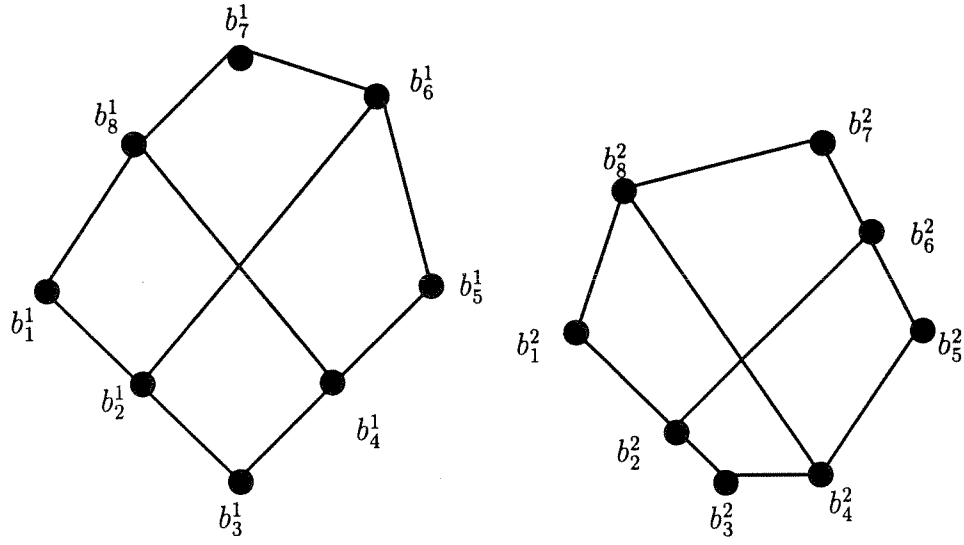

Figure 2: Schematic drawing of a two body configuration.

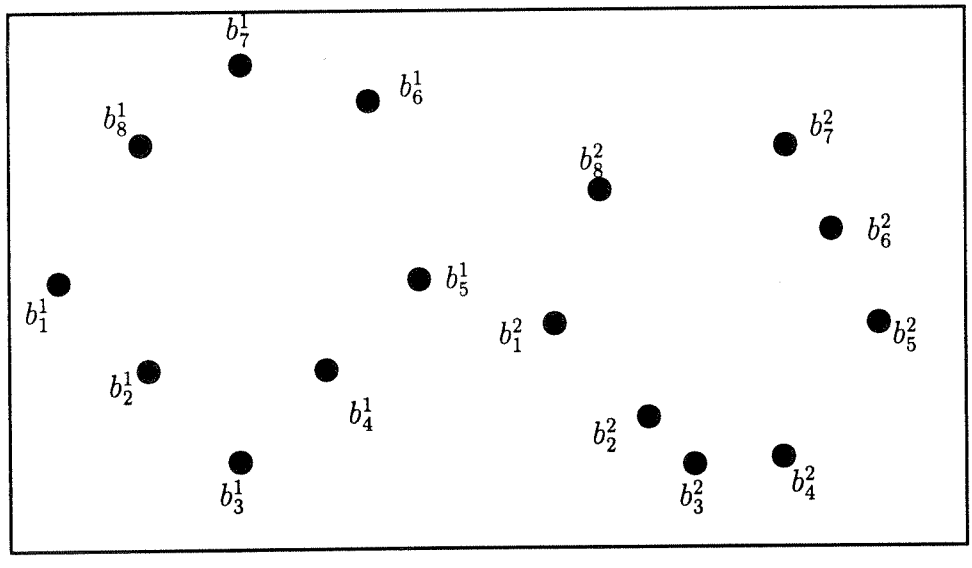

Figure 3: Corresponding distribution of particles in the working space.

The root of the binary tree is associated with the chosen searching space. Consider a particle with its corresponding current coordinates. We divide the working space in half in a particular direction, say vertically, and then determine on which side this particle falls; then, we create a child which we associate with that half space. We think of this procedure as *inserting* the particle in a node. Next, we consider another particle. If this new particle falls in the same half space as the previous one, we subdivide the half space and we move the previously inserted particle into the corresponding quarter space, and insert the new particle into the quarter space where it belongs. The procedure is repeated until each particle resides on a leaf. On the other hand, if the new particle falls on the opposite half space as the previous one, we just create the second child in the tree and continue the process by considering the next particle in the queue. When we finish inserting all the particles in the tree, they all reside on leaves, i.e. there is a unique particle residing in each cell.

Remark 3.2 Notice that a child node is not created unless there is a particle occupying it. This is a modified way of building a tree and avoids the creation of nodes which later would have to be deleted because they are empty. Other authors use different versions of tree building algorithms which are more suitable for their purposes (see e.g. BONET & PERAIRE[3]).

To clarify the particle insertion procedure, we present in Figures 4 and 5 a straightforward example of five particles within a square working space. Figure 4 shows the configuration of the particles and Figure 5 depicts the corresponding binary tree when all the particles have been inserted.

3.2.1 Binary tree construction program

Given a list of particle coordinates, the binary tree can be constructed recursively as follows:

```
Procedure BintreeBuild
  Bintree = {empty}
  do i = 1:n_total
    Call BinInsert(i,root)
  end do
```

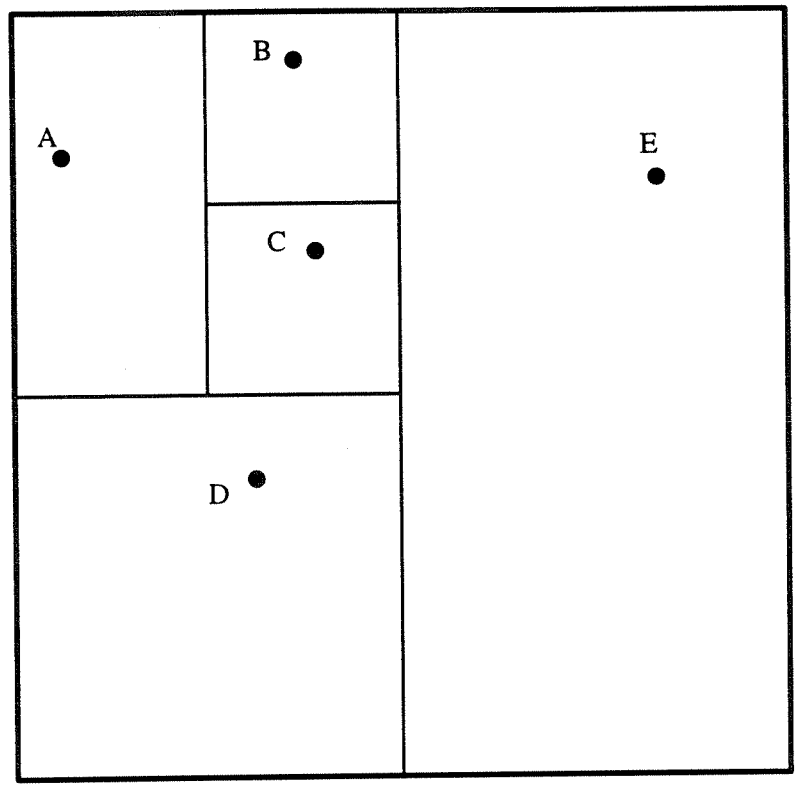

Figure 4: Schematic drawing of a set of particles within a square working space

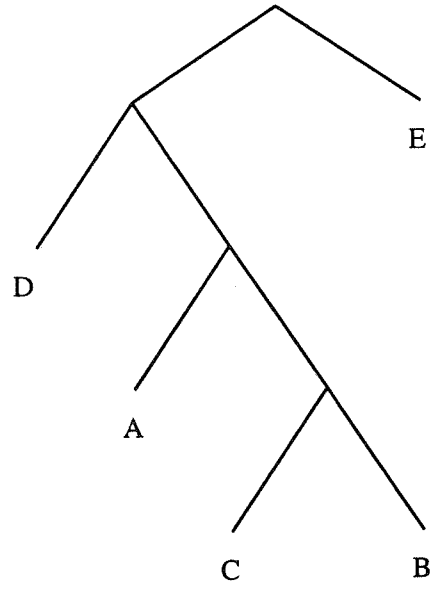

Figure 5: Binary tree structure corresponding to the example shown in Figure 4

```
Procedure BinInsert(i,n)...insert particle i in node n
  if the subtree rooted at n contains more than 1 particle
    Determine which child c of node n particle i lies in
    Call BinInsert(i,c)
  else if the subtree rooted at n contains 1 particle, n is a leaf
    Consider n's 2 children, create the child in which
    the particle already in n lies and move particle i into it.
    Let c be the child in which particle i lies
    Call BinInsert(i,c)
  else if the subtree rooted at n is empty, n is a leaf
    Store particle i in node n
  end if
```

As may be observed from the previous pseudo-code, the algorithm is recursive. This aspect of the algorithm may be managed by using a programming language that permits recursive subroutines or, alternatively, by the use of stacks; the latter is explained extensively in KNUTH[5]. As described in more detail below, we use the first option, taking advantage of the recursive access of routines in FORTRAN 90.

4 Spatial sorting using a binary tree

Once the binary tree has been constructed, as described in the previous sections, we proceed to the actual sorting phase of the algorithm. We describe in this section a general sorting algorithm; see [6] for a similar procedure.

The sorting task consists in looking for all possible contact pairs between any of the n_{total} particles and the N bodies. Recall that the particles are the actual surface nodes (in the finite element sense) of these bodies; therefore we retain the information of the provenance of each particle. To simplify the description of the implementation herein, and with no loss of generality, we assume that a node cannot penetrate the body it belongs to, thus eliminating self contact cases from the algorithm.

We construct a *buffer zone* around each body to simplify contact detection [6]. The simplification comes from the fact that a body with a complicated shape can be bounded by a simple geometric shape, like a rectangle. In a two dimensional space, an example of

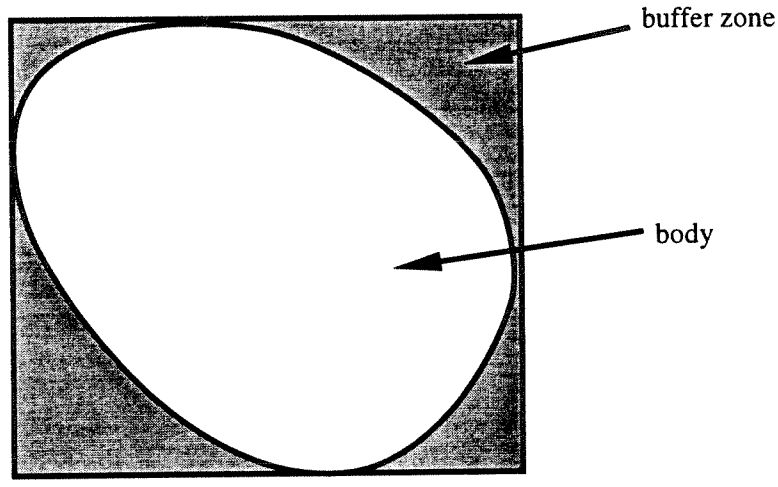

Figure 6: Schematic drawing of a body and its corresponding buffer zone

a buffer zone is the circumscribing rectangle which is aligned with the reference coordinate system (see Figure 6). The coordinates of this rectangle are the minimum and maximum coordinates in each space dimension that bound the object. In this work, we use this type of buffer zone, which does not assume any *a priori* knowledge of the problem.

Remark 4.1 The calculation of the buffer zone is related to the sorting scheme, since it depends on how often one performs a sorting procedure. If a sorting procedure is performed every n time steps, the buffer zone may be chosen so that its thickness b is at least $b > nv_{max}\Delta t$, where v_{max} is the maximum velocity among all the bodies.

4.1 Traversing the binary tree

The contact search is performed by *traversing* the tree. The binary tree is traversed N times (once for each body). We *visit* each node on the tree, which has a subpartition or *cell* of rectangular shape. The task called *visiting* a tree node involves checking for superposition between the target body with its designated buffer zone and the corresponding cell of this tree node. The key advantage of using the binary tree formulation is that if no superposition with a node is detected, then no further contact detection need be performed for the corresponding subtree rooted at that node, thus generating a quite efficient methodology for contact detection. When a *leaf* of the tree which contains an actual particle is reached, the particle is checked for inclusion within the body in question.

To traverse the tree we use the recursive *preorder scheme.* See KNUTH[5] for the

basics on traversing trees and preorder schemes. The traversing scheme employed in the actual numerical implementation involves traversing the tree from the root.

4.1.1 Preorder scheme

We present the pseudo-code for the preorder scheme used in the actual implementation.

```
subroutine preorder(current-address)

call visit  (Visit node stored at  current-address)
if  (cont = .true.) then
  if  (current-left-child exists) then
    call  preorder(current-left-child-address)
  end if
  if  (current-right-child exists) then
    call preorder(current-right-child-address)
  end if
end if
stop
end
```

With the previous interpretation, *visiting* a node involves checking for superposition between two rectangles (the subpartition associated with the tree node and the buffer zone rectangle) or checking whether a particle lies within a rectangle (the finite element node and the buffer zone rectangle). During the "visit", then, we recognize a *pair* consisting of a particle and a body as being in possible contact. This pair, which we call *bp-pair* for body-particle-pair, is then added to the list of possible pairs in contact.

Remark 4.2 Even though we do not assign *slave* and *master* categories to the particles before the sorting phase, the categorization comes up automatically. The closest-point projection phase to follow considers that the particle in the bp-pair is the *slave node* to be checked for penetration into the body belonging to the bp-pair.

Remark 4.3 In addition, as every particle is checked for inclusion within every body, the set of bp-pairs corresponds effectively to a *double pass* contact detection procedure in the context of a typical master/slave methodology HALLQUIST ET AL[4] where only slave nodes

are checked for penetration. Recall that in the double-pass methodology, the set of slave nodes becomes the set of master nodes and viceversa.

5 Contact resolution phase

After the sorting phase is completed, that is, the binary tree is constructed and traversed the contact resolution phase is performed. As noted above, the resolution phase is accomplished in the present work by the closest-point procedure which defines a *gap function.* The gap function measures the penetration of the particle into the body, both belonging to the bp-pair, and is given by

$$g(\boldsymbol{X}): \quad = \min_{\boldsymbol{Y}\in\Gamma^{(2)}} \left\{ \|\boldsymbol{\varphi}^{(2)}(\boldsymbol{Y}) - \boldsymbol{\varphi}^{(1)}(\boldsymbol{X})\| \right\} \tag{1}$$

$$= \boldsymbol{\nu} \cdot \left[\boldsymbol{\varphi}^{(1)}(\boldsymbol{X}) - \boldsymbol{\varphi}^{(2)}(\tilde{\boldsymbol{Y}}(\boldsymbol{X})) \right] \geq 0 \,, \tag{2}$$

in terms of the closest-point projection mapping $\tilde{\boldsymbol{Y}}(\boldsymbol{X})$ of a material point $\boldsymbol{X} \in \Gamma^{(1)}$ (boundary of solid (1)), the deformations (current positions) $\boldsymbol{\varphi}^{(i)}$ $i = 1, 2$ of the two solids in contact, and the resulting "unit normal" $\boldsymbol{\nu}$ to $\Gamma^{(2)}$. If there were no sorting, the closest point projection would have to be performed for every particle with respect to $N - 1$ bodies.

The closest-point projection involves the following procedure for bilinear elements (involving linear boounding segments, 2D):

1. Finding the closest node to the particle S (for slave), denoted by M_C.
2. Find which neighboring node to M_C, M_L or M_R, forms the master segment whose distance to M_C is minimal.

The concept of the closest-point projection is illustrated in Figure 7.

6 Implementation of a sorting algorithm

The type of database involved in the sort/search procedure requires the use of data structures and pointers. FORTRAN 90 provides the necessary tools to define *structures* or *data-types* that are a collection of previously defined numbers, arrays and pointers. The design and definition of the database structure developed in this work are of the main importance, as they affect the efficiency and memory usage of the algorithm. We present below some of the most important structures and related issues involved in the implementation.

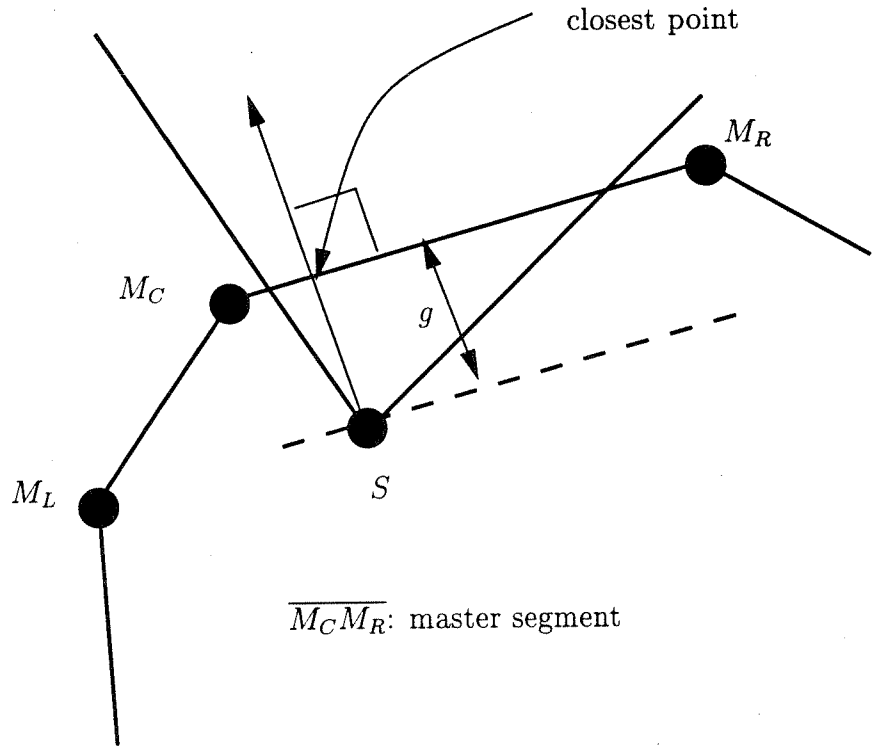

Figure 7: Schematic drawing of the closest point procedure on a discrete space setting in two dimensions. In the case shown, contact is detected for the particle S and the surface element $\overline{M_C M_R}$.

6.1 Binary tree

A node in the tree is defined as follows:

```
type node
  type(particle),pointer :: part-in-node
  real(8), dimension(2) :: xmin,xmax
  type(node),pointer :: parent,left-child,right-child
end type node
```

Pointers are used to point to other data, thus eliminating the need to make multiple copies of the data. For example, when traversing the binary tree, we need to go down to the children of a certain node, so we include in the node data structure two pointers, one to each child.

The first part of the **node** data structure is the *part-in-node*, which is a pointer to a **particle** type structure. If the node is not a leaf then this pointer is *nullified* (does not point to any data). When this happens, the pointer is said to be *disassociated.*

6.2 Body, surface and particle

A body is formed by a set of surfaces and each one of these surfaces consists of a set of particles. Following this reasoning, we can think of a particle as being a substructure of a surface and a surface as a substructure of a body.

Keeping this in mind, we define the following structures:

```
type body
  integer :: num-surf
  type(surface),dimension(:),pointer :: id-surf
  real(8), dimension(2) :: lim-min,lim-max
end type body
```

The integer *num-surf* indicates the number of surfaces that define the body. The real numbers *lim-min* and *lim-max* contain the data of the buffer zone rectangle. The pointer array *id-surf* points to each surface data structure, as described.

Remark 6.1 The capability of defining a body by means of multiple surfaces enables the

scheme to simulate a body with different surface properties. For example, a body may have different surface finishes, each with a different friction coefficient.

```
type surface
  integer :: num-part
  type(surface),dimension(:),pointer :: id-part
end type surface
```

The integer *num-part* indicates the number of particles that define the surface. The pointer array *id-part* points to each particle data structure as described.

```
type particle
  integer :: glob-num,b-id,s-id
  real(8) :: x,y
end type particle
```

The integers *glob-num*, *b-id* and *s-id* represent the global node number (in the finite element sense), the body number and surface number which this particle belongs to, respectively.

Remark 6.2 The inclusion of the body identifier *b-id* is useful to eliminate detecting false contacts; that is, a node cannot be in contact with the body to which it belongs. Other means can be similarly developed when considering problems where the deformations may involve self-contact situations in a body.

6.3 Linked lists

The output of the sorting phase, the set of *bp-pairs*, is stored in a linked list. The data structure *bp-pair* is defined as follows:

```
type bp-pair
  type(body),pointer :: bp-pair-body
  type(particle),pointer :: bp-pair-part
  type(bp-history),pointer :: bp-pair-hist1, bp-pair-hist2
end type bp-pair
```

The pointer *bp-pair-body* points to the body data that is in contact with the particle whose data is pointed to in turn by *bp-pair-part*. The data *bp-pair-hist1* and *bp-pair-hist2* correspond to two history arrays belonging to data at t_n and t_{n+1}, respectively. These arrays make the storage easy for calculations that involve data from previous time steps. The history arrays have the following data structure:

```
type bp-history
  logical :: cont
  integer,dimension(3) :: ixl
  real(8) :: norm(2), gap
  ... other data needed for the calculation of the force and tangent matrix contribution
end type bp-history
```

The logical variable *cont* is the contact flag for the considered time step considered. The integer array *ixl* contains the global node numbers (in the finite element sense) of the three nodes that form the three-node contact element (which is the result of the closest-point projection). The real variable *gap* is the gap which is generally needed for most contacting schemes to enforce the impenetrability constraint. The real vector *norm* contains the normal to the master segment which is generally used to determine the direction of the normal contact force.

When the sorting scheme detects a possible contact, the bp-pair is added to a linked list. When no contact is detected by the closest point projection procedure, these bp-pairs are taken out from the linked list and the space allocated for them is then deallocated. This procedure has the advantage of using memory in a dynamic way thus saving memory storage.

A linked list is composed of *list-nodes*. The structure of a list-node is as follows:

```
type list-node
  type(bp-pair),pointer :: elem
  type(list-node),pointer :: prev,next
end type list-node
```

The pointer *elem* points to the information of the bp-pair that is contained in the given *list-node* and the pointers *prev* and *next* point to the previous and next list-nodes in the linked list, respectively.

6.4 Auxiliary tools

In addition to all the previously defined data structures, we use some auxiliary arrays. One of the most important is an array of pointers called *pairs-table*, of dimension $n_{total} \times N$. These pointers are initially nullified, but if a particular bp-pair (with particle number np and body number nb), was active at the converged state of the previous time step, then pairs-table(nb,np) points to that particular bp-pair, and the algorithm neither allocates more memory nor generates a second copy of the same bp-pair. On the other hand, if pairs-table(nb,np) is nullified then it is evident that this particular bp-pair was not active at the previous time step, and the algorithm must allocate memory to create it.

6.5 Basic algorithm

With the salient points of the proposed database explained, we detail in Table 1 the basic sorting/search algorithm used in the implementation of the considered contact/impact scheme. We denote by $a\%b$ the element b of a certain type, where a is of that type. For example, using FORTRAN 90, we can declare

$$type(node) :: a \quad (a \text{ is a node in the binary tree}) ,$$

so that we have access to the left or right child of a, respectively by

$$l = a\%left - child \quad \text{or} \quad r = a\%right - child$$

where l and r are declared as

$$type(node) :: l, r \quad (l \text{ and } r \text{ are nodes in the binary tree}) ,$$

Remark 6.3 Notice that the binary tree is constructed for each sorting phase and deleted as soon as we form the linked list containing all the bp-pairs in possible contact. In this way, the coordinates of the working space can be recalculated every time.

Remark 6.4 The numerical simulations presented in Section 7 have been obtained using the energy-momentum conserving schemes for the enforcement of contact constraints presented recently by the authors in [1, 2] for general dynamic contact problems. Briefly for the normal contact component (e.g. frictionless case), these schemes involves two basic ingredients.

BASIC ALGORITHM FOR THE SORT/SEARCH PROCEDURE

- Create root
- Update particle coordinates
- Calculate buffer boxes

DO $i = 1 : n_{total}$

- Insert particle i in the binary tree

END DO

DO $i = 1 : N$

- Traverse the tree and check for contact between body i and the particles in each leaf of the tree

END DO

- Delete binary tree

DO WHILE list%bp-pair **exists**

IF (list%bp-pair existed at time t_n(check pairs-table)) **THEN**

- Perform closest point projection
- For EM scheme: calculate the dynamic gap g^d_{n+1}; check for contact based on g^d_{n+1}.

ELSE

- Perform closest point projection
- For EM scheme: follow closest-point at t_n, calculate g_n, and the dynamic gap g^d_{n+1}; check for contact based on g^d_{n+1}.

END IF

IF (cont = .true.) **THEN**

- Calculate the contact force

END IF

list $\Rightarrow$ list%next

END DO

Table 1: Basic algorithm for the sort/search procedure: EM = energy-momentum scheme presented in [1, 2]; see also Remark 6.4. In this case, the geometric sort and closest-point projection are performed at $t_{n+1/2}$.

First, one defines the so-called *"dynamic gap"* g^d_{n+1} for a typical time step $[t_n, t_{n+1}]$

$$g^d_{s,n+1} = g^d_{s,n} + \nu_{n+1/2} \cdot \Big[\Big(\varphi^{(1)}_{n+1}(\boldsymbol{X}_s) - \varphi^{(2)}_{n+1}(\tilde{\boldsymbol{Y}}_{n+1/2}(\boldsymbol{X}_s)) \Big) - \tag{3}$$

$$\Big(\varphi^{(1)}_{n}(\boldsymbol{X}_s) - \varphi^{(2)}_{n}(\tilde{\boldsymbol{Y}}_{n+1/2}(\boldsymbol{X}_s)) \Big) \Big] , \tag{4}$$

for the closest-point projection map $\tilde{\boldsymbol{Y}}_{n+1/2}(\boldsymbol{X}_s)$, defined by 1, at the midpoint $t_{n+1/2}$ with the unit normal direction $\nu_{n+1/2}$ to the corresponding position of the solids (defined by the configurations $\varphi^{(i)}$ at times t_n and t_{n+1}). In this last relation, the gap g^d_n is initialized with the real gap g_n in the first time increment in contact. Then, the contact pressure is obtained from a penalty regularization as

$$p_s = -\frac{U(g^d_{s,n+1}) - U(g^d_{s,n})}{g^d_{s,n+1} - g^d_{s,n}} , \qquad \text{with} \qquad U(g^d) := \begin{cases} \frac{1}{2}\, \kappa_N\, g^{d2} & \text{if } g^d \leq 0, \\ 0 & \text{otherwise,} \end{cases} \tag{5}$$

for a penalty parameter $\kappa_N > 0$. We refer to [1] for additional extensions, and to [2] for the development of dissipative schemes for the frictional case

7 Numerical assessment

As an illustrative example of the type of problems most appropriately handled by the proposed contact detection scheme, we show the simulation of the impact of 49 elastic disks enclosed within four rigid walls. The disks have Lamé constants $\lambda = 2000.0$ and $G = 1000.0$, and density $\rho = 1.0$. The left-most column of disks is given an initial velocity $v_0 = (0.5, 0.1)$. Figures 8 and 9 show the evolution of the system. Frictionless contacts are assumed and the newly proposed energy-momentum conserving scheme presented in [1] is considered. Notice that clusters of bodies are formed at various instants, i.e. a particular body may be in contact with many others at certain time steps.

As stated above, the addition of a sorting phase decreases the computational effort involved in the contact detection part of the contact algorithm in multi-body problems. In this section, we measure (1) the average CPU time for sorting/searching in each time step, and (2) the average CPU time for the entire step, for a sample problem while we vary the number of bodies involved. These measurements give an estimate of the speed increase provided by the sorting procedure as the number of bodies increases. It is also of interest to see the fraction of time the contact detection part takes within an overall time step.

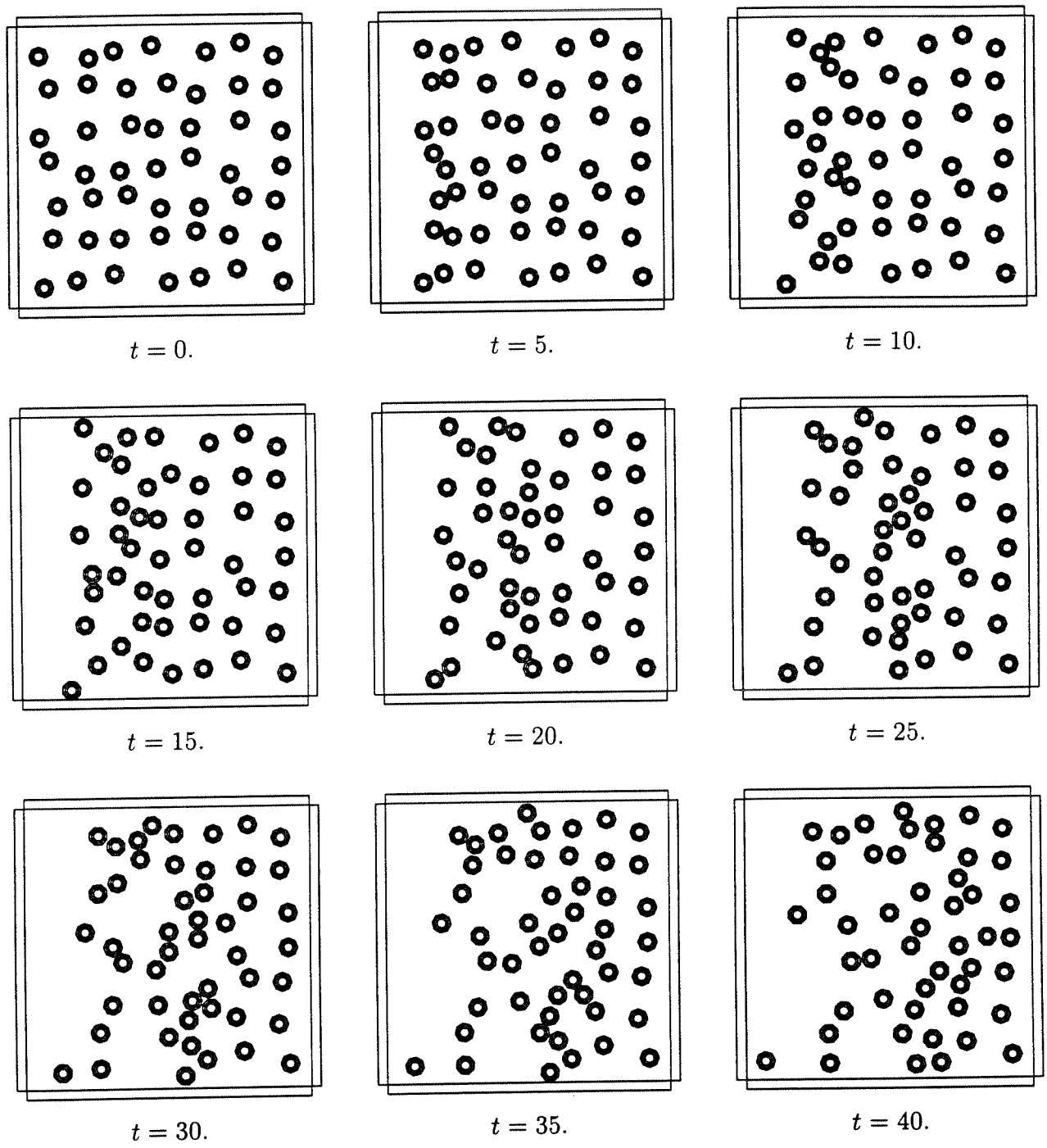

Figure 8: Impact of 49 quasi-rigid disks. Evolution of the system.

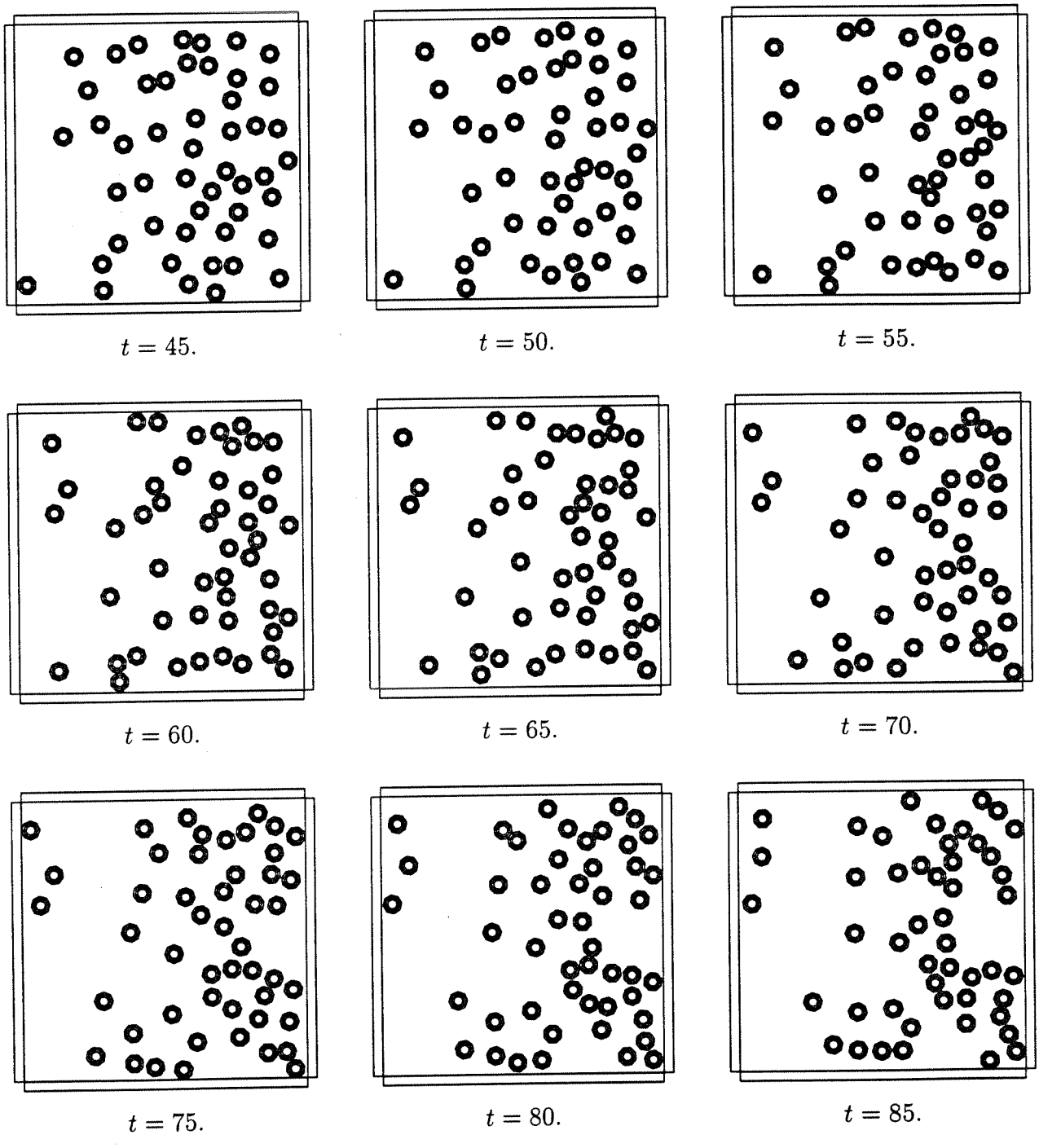

Figure 9: Impact of 49 quasi-rigid disks. Evolution of the system (continued).

N	Sorting		No sorting	
	$\text{CPU}_{detection}$	CPU_{total}	$\text{CPU}_{detection}$	CPU_{total}
13	0.0085	0.07	0.1244	0.29
53	0.0361	0.90	2.7050	3.60
104	0.0732	1.80	10.8600	12.80
196	0.1386	3.60	40.3800	44.23
400	0.3030	7.64	-	-
900	0.7540	16.60	-	-

Table 2: Average computation times in seconds per iteration for various numbers of bodies.

Theoretical calculations show that for a homogeneous concentration of particles, which should produce a well-balanced tree, the times for building and traversing the tree should behave as $\mathcal{O}(N\log(N))$, where N is the number of particles. See [5] for details. The detection part of the algorithm with no sorting phase, i.e. an exhaustive search, behaves as $\mathcal{O}(N^2)$. For simplicity and without loss of generality, all the bodies are assumed to have the same order of possible contacting particles in these considerations and in the numerical examples presented in this section. In Table 2, we show the CPU time in seconds for different problems. All the simulations have been run with a DEC Alpha 3000 Model 700 with 128MB RAM.

A regression analysis has been performed with the times from Table 2 to evaluate the behavior of each algorithm. With the sorting algorithm, the tested times for detection followed the expected logarithmic dependence on N with a correlation coefficient of $R = 0.99985$, while the times required without sorting matched a quadratic curve with a correlation coefficient of $R = 0.999991$. Figures 10 and 11 show the curves obtained through regression analysis of the data in Table 2.

The ratio between the CPU time for the detection phase and the CPU time for the solver procedure is also of interest, since without the addition of a sorting phase the detection algorithm would dominate the CPU time for large N. We denote this ratio by Λ, i.e.,

$$\Lambda_{case} = \frac{\text{CPU}^{case}_{detection}}{\text{CPU}^{case}_{total} - \text{CPU}^{case}_{detection}} \left(\simeq \frac{\text{CPU}^{case}_{detection}}{\text{CPU}^{case}_{solver}} \right) , \tag{6}$$

where *case* refers to *sorting* or *no sorting*. In Table 3 we show these values for different number of bodies. One may observe that the ratio Λ increases linearly with N for the non-sorting algorithm, while the value is independent of N when sorting is added to the search algorithm.

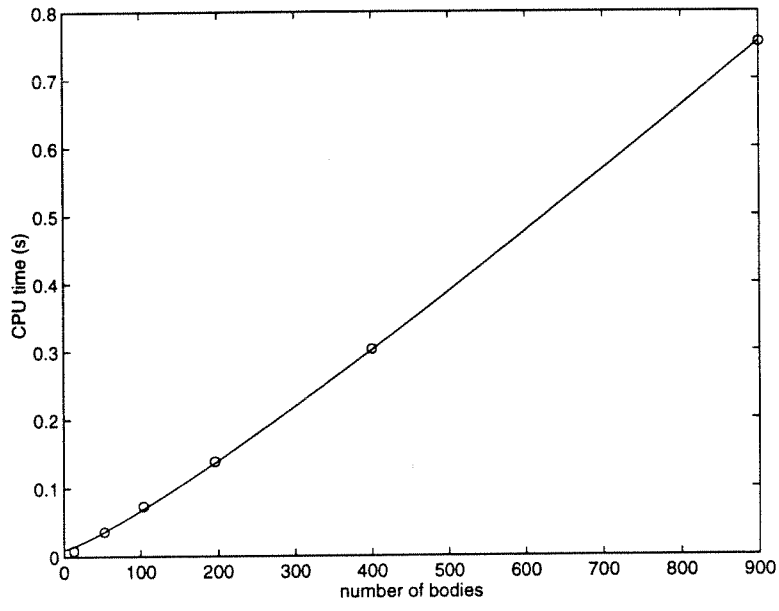

Figure 10: Contact detection CPU time for the algorithm with sorting procedure. Computational data (o); regression analysis (—).

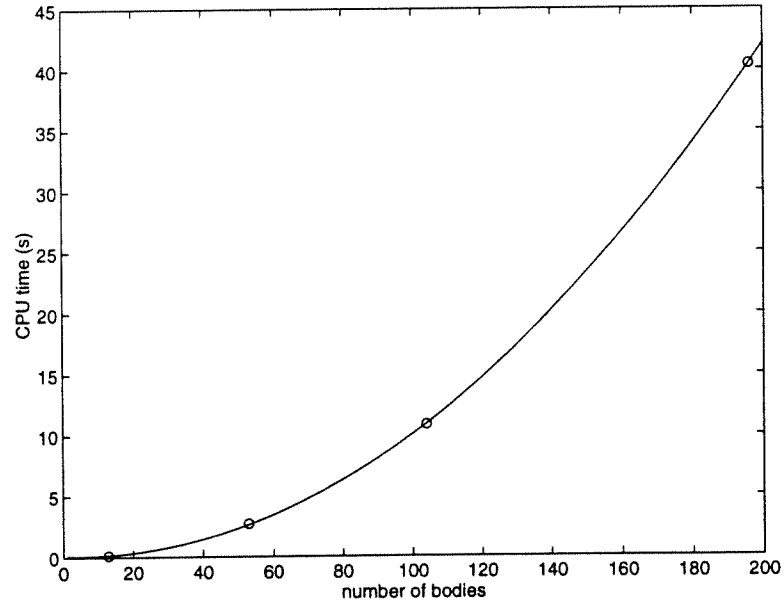

Figure 11: Contact detection CPU time for the algorithm with no sorting procedure. Computational data (o); regression analysis (—).

N	$\Lambda_{sorting}$	$\Lambda_{no\ sorting}$
13	0.0012	0.74
53	0.0418	2.91
104	0.0424	5.60
196	0.0400	10.49
400	0.0413	-
900	0.0476	-

Table 3: CPU ratios for various numbers of bodies.

Remark 7.1 A skyline direct solver has been used in the preceding simulations. Therefore, the cost associated with the solver can be estimated as $\mathcal{O}(N_{band}^2\, N_{eq})$ for N_{eq} equations with bandwith N_{band}. For the case of interest considered in this section with N bodies, we can then write the estimates

$$\Lambda_{sorting} \propto \frac{N \log N}{N\ N_{band}^2} = \frac{\log N}{N_{band}^2} \quad (\simeq \text{constant}), \tag{7}$$

and

$$\Lambda_{no\ sorting} \propto \frac{N^2}{N\ N_{band}^2} = \frac{N}{N_{band}^2} \quad (\text{linear in } N), \tag{8}$$

assuming $N_{band} \simeq$ constant (best possible case for the solver), and thus explaining the results reflected in Table 3.

8 Concluding remarks

We have presented in this report the formulation and finite element implementation of a sorting/searching scheme for multi-body contact problems. The scheme involves a sorting phase based on a binary space partitioning (BSP) strategy, leading in the considered two dimensional setting to a storage of the finite element nodes candidates for contact in a binary tree structure. The sorting of the possible body/particle contact pairs is then efficiently accomplished, leading to a linked-list of such possible contact pairs. Important specific issues, like the handling of the history arrays associated to these pairs, have been addressed in detail. A second resolution (or searching) phase based on the traditional closest-point projection identifies the actual active contact points at the local level of the body/particle pairs.

Special attention has been given to the implementation issues that need to be considered when designing an all-purpose contact detection algorithm. In particular, we have described the different database structures employed in the actual implementation. Details of the different routines have been presented in FORTRAN 90, making use of several new features of this most recent version of this programming language common in finite element codes (namely, pointers, structures or types, and recursive routine access, among others).

The need for the consideration of this type of contact detection sorting scheme to simulate contact problems in many body systems has been illustrated in representative dynamic simulations of these systems. Actual CPU times of the presented implementation shows the theoretically optimal cost of $\mathcal{O}(N \log N)$ versus the worse case of $\mathcal{O}(N^2)$ for a system with N bodies. In addition, these long-term simulations have shown also the improved numerical stability properties of the conserving contact algorithms presented recently by the authors (see [1, 2]). Current efforts include the extension of the contact detection scheme presented herein to the three dimensional case.

Acknowledgments

The contact detection scheme presented in this report has been implemented in FEAP, courtesy of Prof. R.L. Taylor. Financial support for this research has been provided by the AFOSR under contract no. F49620-97-1-0196 with the University of California at Berkeley. This support is gratefully acknowledged.

References

[1] F. Armero and E. Petocz. Formulation and analysis of conserving algorithms for frictionless dynamic contact/impact problems. *Comp. Methods in Applied Mech. Engrg.*, in press, 1996.

[2] F. Armero and E. Petocz. A new dissipative time-stepping algorithm for frictional contact problems: Formulation and analysis. *Comp. Methods in Applied Mech. Engrg.*, in press, 1997.

[3] J. Bonet and J. Peraire. An alternating digital tree (ADT) algorithm for 3D geometric searching and intersection problems. *Int. J. Numer. Methods Engrg.*, 31:1–17, 1991.

[4] J.O. Hallquist, G.L. Goudreau, and D.J. Benson. Sliding interfaces with contact-impact in large-scale Lagrangian computations. *Comp. Methods in Applied Mech. Engrg.*, 51:107–137, 1985.

[5] D.E. Knuth. *The Art of Computer Programming, Fundamental Algorithms, Volume 1.* Addison-Wesley, Massachusetts, 1973.

[6] A. Munjiza, D.R.J. Owen, and N. Bićanić. A combined finite-discrete element method in transient dynamics of fracturing solids. *Eng. Comput.*, 12:145–174, 1995.

[7] J.R. Williams and R. O'Connor. A linear complexity intersection algorithm for discrete element simulation of arbitrary geometries. *Eng. Comput.*, 12:185–201, 1995.

[8] O. C. Zienkiewicz and R. L. Taylor. *The Finite Element Method, Vol-I,II.* McGraw Hill, U. K., 1989.